Read the Docs Template
Documentation

Read the Docs

Sep 22, 2023
## CONTENTS

1 Get Started ................................................................. 3
   1.1 Development Board Overview ........................................ 3
   1.2 About ESP-ADF ..................................................... 3
   1.3 Quick Start ............................................................ 4
   1.4 Installation Step by Step .............................................. 5
   1.5 Step 1. Set up ESP-IDF ............................................... 5
   1.6 Step 2. Get ESP-ADF ................................................... 6
   1.7 Step 3. Set up Path to ESP-ADF ...................................... 6
   1.8 Step 4. Set up the environment variables ............................ 7
   1.9 Step 5. Start a Project ............................................... 8
   1.10 Step 6. Connect Your Device ...................................... 8
   1.11 Step 7. Configure .................................................... 8
   1.12 Step 8. Build the Project ......................................... 11
   1.13 Step 9. Flash onto the Device ..................................... 11
   1.14 Step 10. Monitor .................................................... 12
   1.15 VS Code Extension ................................................ 13
   1.16 IDF Eclipse Plugin and Espressif IDE .............................. 13
   1.17 Update ESP-ADF ................................................... 16

2 API Reference .......................................................... 17
   2.1 Audio Framework ..................................................... 18
   2.2 Audio Streams ......................................................... 70
   2.3 Playlist ................................................................. 95
   2.4 Codecs ................................................................. 115
   2.5 Audio Processing ..................................................... 128
   2.6 Services ............................................................... 140
   2.7 Speech Recognition .................................................. 185
   2.8 Peripherals ............................................................. 190
   2.9 Abstraction Layer .................................................... 220
   2.10 Configuration Options .............................................. 242

3 Design Guide ............................................................ 247
   3.1 Project Design ....................................................... 247
   3.2 Design Considerations ............................................... 249
   3.3 Software Design ....................................................... 253
   3.4 Development Boards .................................................. 254
   3.5 Audio Samples ........................................................ 340

4 Resources ................................................................. 343
This is the documentation for Espressif Audio Development Framework (ADF).

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

| CONTENTS |
|----------|---------------|
| 1        |               |
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 (v3.3, v4.0, v4.1, v4.2, v4.3 and v4.4). 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.
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 10. 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 10. 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:

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

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

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

2. Set the $ESP-ADF path by running:

```bash
cd esp-adf
export ADF_PATH=$PWD
```

3. Configure the $ESP-IDF compilation environment by running:

```bash
cd $ADF_PATH/esp-idf
./install.sh
./export.sh
```

4. 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.

```bash
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:

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

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

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

2. Download the full ESP-IDF Windows Installer from ESP-IDF Windows Installer.
3. 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-V4.4 CMD shortcut icon on the desktop, the script will automatically help you download submodules, and set environment variables such as IDF_PATH.

4. Set the $ESP-ADF path by running the following commands. Note that %userprofile%\esp is used as an installation folder for ESP-ADF. You can use any directory, but you will need to adjust paths for the commands accordingly.

   set ADF_PATH=%userprofile%\espesp-ADF echo %ADF_PATH%

5. 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 Path to ESP-ADF**
- **Step 4. Set up the environment variables**

#### 1.4.2 Creating Your First Project

- **Step 5. Start a Project**
- **Step 6. Connect Your Device**
- **Step 7. Configure**
- **Step 8. Build the Project**
- **Step 9. Flash onto the Device**
- **Step 10. 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:

```plaintext
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:

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

### 1.7 Step 3. Set up Path to ESP-ADF

The toolchain programs access ESP-ADF using `ADF_PATH` environment variable. This variable should be set up on your PC, otherwise the projects will not build.

#### 1.7.1 Windows

Open Command Prompt and run the following command:

```plaintext
set ADF_PATH=%userprofile%\esp\esp-adf
```

You need to enter this command each time you start your PC. To avoid retyping you can add it to “ESP-IDF Command Prompt”, batch or Power Shell scripts described in Step 4 below.

To make sure that `ADF_PATH` has been set up properly, run:

```plaintext
echo %ADF_PATH%
```

It should return the path to your ESP-ADF directory.
1.7.2 Linux and macOS

Open Terminal, and run the following commands:

```
export ADF_PATH=~/esp/esp-adf
```

You need to enter this command each time you open a Terminal. To make this setting permanent follow similar instructions for configuration of IDF_PATH in ESP-IDF Programming Guide.

Check if ADF_PATH has been set up to point to directory with ESP-ADF:

```
printenv ADF_PATH
```

1.8 Step 4. Set up the environment variables

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-IDF provides a script which does that.

1.8.1 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-idf\export.bat
```

or with Windows PowerShell

```
.$HOME/esp/esp-idf/export.ps1
```

1.8.2 Linux and macOS

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

```
. $HOME/esp/esp-idf/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-idf/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.
1.9 Step 5. 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.

1.9.1 Windows

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

1.9.2 Linux and macOS

```bash
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.

1.10 Step 6. 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.

1.11 Step 7. Configure

Navigate to your play_mp3_control directory from Step 5. Start a Project and configure the project:
1.11.1 ESP-IDF v3.3.2 and v4.0 releases

Windows

```bash
cd %userprofile%\esp\play_mp3_control
idf.py menuconfig
```

Linux and macOS

```bash
cd ~/esp/play_mp3_control
idf.py menuconfig
```

1.11.2 ESP-IDF v4.1 and master releases

Windows

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

Linux and macOS

```bash
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` for ESP-IDF master release or `idf.py set-target esp32s2beta` for ESP-IDF v4.1 release.

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
1.12 Step 8. 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.

1.13 Step 9. 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 6. 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)
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.

1.14 Step 10. 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.

### 1.15 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:

   ![VS Code Extension Toolbar](image)

   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.

### 1.16 IDF Eclipse Plugin and Espressif IDE

#### 1.16.1 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, 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.

1.16.2 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.
1.16.3 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).

1.16.4 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.
1.17 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.
API REFERENCE

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

![Elements of the Audio Development Framework](image)

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.

![Sample Organization of Elements in Audio Pipeline](image)

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

- `audio_pipeline/include/audio_element.h`

Functions

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

**Return**

- `audio_element` handle object
- `NULL`

**Parameters**

- `config`: The configuration

```c
esp_err_t audio_element_deinit(audio_element_handle_t el)
Destroy audio element handle object, stop, clear, delete all.
```

**Return**

- `ESP_OK`
- `ESP_FAIL`

**Parameters**

- `[in] el`: The audio element handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] el: The audio element handle
- data: The data pointer

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

Return  data pointer

Parameters

- [in] el: The audio element handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] el: The audio element handle
- [in] tag: The tag name pointer

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

Return  Element tag name pointer

Parameters

- [in] el: The audio element handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] el: The audio element handle
- info: The information pointer

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

- ESP_OK
- ESP_FAIL

Parameters

- [in] el: The audio element handle
- info: The information pointer

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

Set audio element URI.

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] el: The audio element handle
- [in] uri: The uri pointer

```c
char *audio_element_get_uri(audio_element_handle_t el)
```

Get audio element URI.

Return  URI pointer

Parameters

- [in] el: The audio element handle

```c
esp_err_t audio_element_run(audio_element_handle_t el)
```

Start Audio Element. With this function, audio_element will start as freeRTOS task, and put the task into 'PAUSED' state. Note: Element does not actually start when this function returns.

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] el: The audio element handle

```c
esp_err_t audio_element_terminate(audio_element_handle_t el)
```

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] el: The audio element handle
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.

Return

• ESP_OK
• ESP_FAIL

Parameters

• [in] el: The audio element handle
• [in] ticks_to_wait: The maximum amount of time to blocking

esp_err_t audio_element_stop(audio_element_handle_t el)

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

Return

• ESP_OK
• ESP_FAIL

Parameters

• [in] el: The audio element handle

esp_err_t audio_element_wait_for_stop(audio_element_handle_t el)

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

Return

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

Parameters

• [in] el: The audio element handle

esp_err_t audio_element_wait_for_stop_ms(audio_element_handle_t el, TickType_t ticks_to_wait)

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

Return

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

Parameters

2.1. Audio Framework
• [in] el: The audio element handle
• [in] ticks_to_wait: The maximum amount of time to wait for stop

esp_err_t audio_element_pause(audio_element_handle_t el)
Request audio Element enter ‘PAUSE’ state. In this state, the task will wait for any event.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle
• [in] wait_for_rb_threshold: The wait for rb threshold (0 .. 1)
• [in] timeout: The timeout

esp_err_t audio_element_msg_set_listener(audio_element_handle_t el, audio_event_iface_handle_t listener)
This function will add a listener to listen to all events from audio element el. Any event from el->external_event will be send to the listener.

Return
• ESP_OK
• ESP_FAIL

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

esp_err_t audio_element_set_event_callback(audio_element_handle_t el, event_cb_func cb_func, void *ctx)
This function will add a callback to be called from audio element el. Any event to caller will cause to call callback function.

Return
• ESP_OK
• ESP_FAIL

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

esp_err_t audio_element_msg_remove_listener(audio_element_handle_t el, audio_event_iface_handle_t listener)
Remove listener out of el. No new events will be sent to the listener.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle
• listener: The listener

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle
• [in] rb: The ringbuffer handle

ringbuf_handle_t audio_element_get_input_ringbuf(audio_element_handle_t el)
Get Element input ringbuffer.

Return ringbuf_handle_t

Parameters
• [in] el: The audio element handle

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle
• [in] rb: The ringbuffer handle
**ringbuf_handle_t audio_element_get_output_ringbuf** *(audio_element_handle_t el)*

Get Element output ringbuffer.

**Return** ringbuf_handle_t

**Parameters**
- [in] el: The audio element handle

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

Get current Element state.

**Return** audio_element_state_t

**Parameters**
- [in] el: The audio element handle

**esp_err_t audio_element_abort_input_ringbuf** *(audio_element_handle_t el)*

If the element is requesting data from the input ringbuffer, this function forces it to abort.

**Return**
- ESP_OK
- ESP_FAIL

**Parameters**
- [in] el: The audio element handle

**esp_err_t audio_element_abort_output_ringbuf** *(audio_element_handle_t el)*

If the element is waiting to write data to the ringbuffer output, this function forces it to abort.

**Return**
- ESP_OK
- ESP_FAIL

**Parameters**
- [in] el: The audio element handle

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

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

**Return**
- ESP_OK
- ESP_FAIL

**Parameters**
- [in] el: The audio element handle
- [in] size_expect: The size expect
- [in] timeout: The timeout
esp_err_t audio_element_report_status(audio_element_handle_t el, audio_element_status_t status)
Element will sendout event (status) to event by this function.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle
• [in] status: The status

esp_err_t audio_element_report_info(audio_element_handle_t el)
Element will sendout event (information) to event by this function.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle

esp_err_t audio_element_report_codec_fmt(audio_element_handle_t el)
Element will sendout event (codec format) to event by this function.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle

esp_err_t audio_element_report_pos(audio_element_handle_t el)
Element will sendout event with a duplicate information by this function.

Return
• ESP_OK
• ESP_FAIL
• ESP_ERR_NO_MEM

Parameters
• [in] el: The audio element handle

esp_err_t audio_element_set_input_timeout(audio_element_handle_t el, TickType_t timeout)
Set input read timeout (default is portMAX_DELAY).

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle
• [in] timeout: The timeout

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle
• [in] timeout: The timeout

esp_err_t audio_element_reset_input_ringbuf (audio_element_handle_t el)
Reset input buffer.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle

esp_err_t audio_element_finish_state (audio_element_handle_t el)
Set element finish state.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle

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.

Return
• ESP_OK
• ESP_FAIL
• ESP_ERR_INVALID_ARG Element handle is null

Parameters
• [in] el: The audio element handle
• [in] cmd: Specific command from audio_element_msg_cmd_t
esp_err_t audio_element_reset_output_ringbuf(audio_element_handle_t el)
Reset output buffer.

Return
- ESP_OK
- ESP_FAIL

Parameters
- [in] el: The audio element handle

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

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

Parameters
- [in] el: The audio element handle
- buffer: The buffer pointer
- [in] wanted_size: The wanted size

audio_element_err_t audio_element_output(audio_element_handle_t el, char *buffer, int write_size)
Call this function to sendout Element output data. Depending on setup using ringbuffer or function callback,
Element will invoke write to ringbuffer, or call write callback function.

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

Parameters
- [in] el: The audio element handle
- buffer: The buffer pointer
- [in] write_size: The write size

esp_err_t audio_element_set_read_cb(audio_element_handle_t el, stream_func fn, void *context)
This API allows the application to set a read callback for the first audio_element in the pipeline for allowing the
pipeline to interface with other systems. The callback is invoked every time the audio element requires data to be processed.

Return
- ESP_OK
- ESP_FAIL

Parameters
- [in] el: The audio element handle
• [in] fn: Callback read function. The callback function should return number of bytes read or -1 in case of error in reading. Note that the callback function may decide to block and that may block the entire pipeline.

• [in] context: An optional context which will be passed to callback function on every invocation

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

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

Return

• ESP_OK
• ESP_FAIL

Parameters

• [in] el: The audio element

```c
stream_func audio_element_get_write_cb(audio_element_handle_t el)
```

Get callback write function that register to the element.

Return

• Callback write function pointer
• NULL Failed

Parameters

• [in] el: The audio element

```c
stream_func audio_element_get_read_cb(audio_element_handle_t el)
```

Get callback read function that register to the element.

Return

• Callback read function pointer
• NULL Failed

Parameters

• [in] el: The audio element

```c
QueueHandle_t audio_element_get_event_queue(audio_element_handle_t el)
```

Get External queue of Emitter. We can read any event that has been send out of Element from this QueueHandle_t.

Return

QueueHandle_t

Parameters

• [in] el: The audio element handle
esp_err_t audio_element_set_ringbuf_done(audio_element_handle_t el)
Set input buffer and output buffer have finished.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle

esp_err_t audio_element_reset_state(audio_element_handle_t el)
Enforce ‘AEL_STATE_INIT’ state.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle

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

Return
• =0: Parameter NULL
• >0: Size of ringbuffer

Parameters
• [in] el: The audio element handle

esp_err_t audio_element_set_output_ringbuf_size(audio_element_handle_t el, int rb_size)
Set Element output ringbuffer size.

Return
• ESP_OK
• ESP_FAIL

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

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

Return
• ESP_OK
• ESP_ERR_INVALID_ARG
Parameters

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

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

Call this function write data by multi output ringbuffer.

Return

- `ESP_OK`
- `ESP_FAIL`

Parameters

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

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

Set multi input ringbuffer Element.

Return

- `ESP_OK`
- `ESP_FAIL`

Parameters

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

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

Set multi output ringbuffer Element.

Return

- `ESP_OK`
- `ESP_ERR_INVALID_ARG`

Parameters

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

\texttt{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.

\textbf{Return}

• NULL Error
• Others ringbuf\_handle\_t

\textbf{Parameters}

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

\texttt{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.

\textbf{Return}

• NULL Error
• Others ringbuf\_handle\_t

\textbf{Parameters}

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

\texttt{esp\_err\_t audio\_element\_process\_init(audio\_element\_handle\_t el)}

Provides a way to call element’s open

\textbf{Return}

• ESP_OK
• ESP_FAIL

\textbf{Parameters}

• [in] el: The audio element handle

\texttt{esp\_err\_t audio\_element\_process\_deinit(audio\_element\_handle\_t el)}

Provides a way to call element’s close

\textbf{Return}

• ESP_OK
• ESP_FAIL

\textbf{Parameters}

• [in] el: The audio element handle
esp_err_t audio_element_seek (audio_element_handle_t el, void *in_data, int in_size, void *out_data, int *out_size)

Call element’s seek

Return

• ESP_OK
• ESP_FAIL
• ESP_ERR_NOT_SUPPORTED

Parameters

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

bool audio_element_is_stopping (audio_element_handle_t el)

Get Element stopping flag.

Return element’s stopping flag

Parameters

• [in] el: The audio element handle

esp_err_t audio_element_update_byte_pos (audio_element_handle_t el, int pos)

Update the byte position of element information.

Return

• ESP_OK
• ESP_FAIL

Parameters

• [in] el: The audio element handle
• [in] pos: The byte_pos accumulated by this value

esp_err_t audio_element_set_byte_pos (audio_element_handle_t el, int pos)

Set the byte position of element information.

Return

• ESP_OK
• ESP_FAIL

Parameters

• [in] el: The audio element handle
• [in] pos: This value is assigned to byte_pos

esp_err_t audio_element_update_total_bytes (audio_element_handle_t el, int total_bytes)

Update the total bytes of element information.
esp_err_t audio_element_set_total_bytes(audio_element_handle_t el, int total_bytes)
Set the total bytes of element information.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle
• [in] total_bytes: The total_bytes accumulated by this value

esp_err_t audio_element_set_bps(audio_element_handle_t el, int bit_rate)
Set the bps of element information.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle
• [in] bit_rate: This value is assigned to bps

esp_err_t audio_element_set_codec_fmt(audio_element_handle_t el, int format)
Set the codec format of element information.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle
• [in] format: This value is assigned to codec_fmt

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.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle
• [in] sample_rates: Sample_rates of music information
• [in] channels: Channels of music information
• [in] bits: Bits of music information

esp_err_t audio_element_set_duration(audio_element_handle_t el, int duration)
Set the duration of element information.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle
• [in] duration: This value is assigned to duration

esp_err_t audio_element_set_reserve_user0(audio_element_handle_t el, int user_data0)
Set the user_data_0 of element information.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle
• [in] user_data0: This value is assigned to user_data_0

esp_err_t audio_element_set_reserve_user1(audio_element_handle_t el, int user_data1)
Set the user_data_1 of element information.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] el: The audio element handle
• [in] user_data1: This value is assigned to user_data_1

esp_err_t audio_element_set_reserve_user2(audio_element_handle_t el, int user_data2)
Set the user_data_2 of element information.

Return
• ESP_OK
• ESP_FAIL
Parameters

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

`esp_err_t audio_element_set_reserve_user3(audio_element_handle_t el, int user_data3)`
Set the user_data_3 of element information.

Return

- ESP_OK
- ESP_FAIL

Parameters

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

`esp_err_t audio_element_set_reserve_user4(audio_element_handle_t el, int user_data4)`
Set the user_data_4 of element information.

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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)

espCodecType_t codec_fmt
    Music format (optional)

audioElementReserveData_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)

2.1. Audio Framework
Enumerations

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

enum audio_element_state_t
   Audio element state.
   Values:
   AEL_STATE_NONE = 0
   AEL_STATE_INIT = 1
   AEL_STATE_INITIALIZING = 2
   AEL_STATE_RUNNING = 3
   AEL_STATE_PAUSED = 4
   AEL_STATE_STOPPED = 5
   AEL_STATE_FINISHED = 6
   AEL_STATE_ERROR = 7

enum audio_element_msg_cmd_t
   Audio element action command, process on dispatcher
   Values:
   AEL_MSG_CMD_NONE = 0
   AEL_MSG_CMD_FINISH = 2
   AEL_MSG_CMD_STOP = 3
   AEL_MSG_CMD_PAUSE = 4
   AEL_MSG_CMD_RESUME = 5
   AEL_MSG_CMD_DESTROY = 6
   AEL_MSG_CMD_REPORT_STATUS = 8
   AEL_MSG_CMD_REPORT_MUSIC_INFO = 9
   AEL_MSG_CMD_REPORT_CODEC_FMT = 10
   AEL_MSG_CMD_REPORT_POSITION = 11

enum audio_element_status_t
   Audio element status report
   Values:
   AEL_STATUS_NONE = 0
AEL_STATUS_ERROR_OPEN = 1
AEL_STATUS_ERROR_INPUT = 2
AEL_STATUS_ERROR_PROCESS = 3
AEL_STATUS_ERROR_OUTPUT = 4
AEL_STATUS_ERROR_CLOSE = 5
AEL_STATUS_ERROR_TIMEOUT = 6
AEL_STATUS_ERROR_UNKNOWN = 7
AEL_STATUS_INPUT_DONE = 8
AEL_STATUS_INPUTBUFFERING = 9
AEL_STATUS_OUTPUT_DONE = 10
AEL_STATUS_OUTPUT_BUFFERING = 11
AEL_STATUS_STATE_RUNNING = 12
AEL_STATUS_STATE_PAUSED = 13
AEL_STATUS_STATE_STOPPED = 14
AEL_STATUS_STATE_FINISHED = 15
AEL_STATUS_MOUNTED = 16
AEL_STATUS_UNMOUNTED = 17

2.1.2 Audio Pipeline

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

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

![Diagram of Audio Pipeline]

Fig. 3: Sample Organization of Elements in Audio Pipeline

2.1. Audio Framework
API Reference

Header File

- audio_pipeline/include/audio_pipeline.h

Functions

`audio_pipeline_handle_t audio_pipeline_init (audio_pipeline_cfg_t *config)`

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

Return

- audio_pipeline_handle_t on success
- NULL when any errors

Parameters

- `config`: The configuration - audio_pipeline_cfg_t

`esp_err_t audio_pipeline_deinit (audio_pipeline_handle_t pipeline)`

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

Return ESP_OK

Parameters

- `[in] pipeline`: The Audio Pipeline Handle

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

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

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

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `[in] pipeline`: The Audio Pipeline Handle
- `[in] el`: The Audio Element Handle
- `[in] name`: The name identifier of the audio_element in this audio_pipeline
esp_err_t audio_pipeline_unregister (audio_pipeline_handle_t pipeline, audio_element_handle_t el)

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

Return

• ESP_OK on success
• ESP_FAIL when any errors

Parameters

• [in] pipeline: The Audio Pipeline Handle
• [in] el: The Audio Element Handle

est_err_t audio_pipeline_run (audio_pipeline_handle_t pipeline)

Start Audio Pipeline.

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

Return

• ESP_OK on success
• ESP_FAIL when any errors

Parameters

• [in] pipeline: The Audio Pipeline Handle

est_err_t audio_pipeline_terminate (audio_pipeline_handle_t pipeline)

Stop Audio Pipeline.

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

Return

• ESP_OK on success
• ESP_FAIL when any errors

Parameters

• [in] pipeline: The Audio Pipeline Handle

est_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.

Return

• ESP_OK
• ESP_FAIL

Parameters

• [in] pipeline: The Audio Pipeline Handle
esp_err_t audio_pipeline_resume(audio_pipeline_handle_t pipeline)
This function will set all the elements to the `RUNNING` state and process the audio data as an inherent feature of audio_pipeline.

Return

• ESP_OK on success
• ESP_FAIL when any errors

Parameters

• [in] pipeline: The Audio Pipeline Handle

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

Return

• ESP_OK on success
• ESP_FAIL when any errors

Parameters

• [in] pipeline: The Audio Pipeline Handle

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

Return

• ESP_OK on success
• ESP_FAIL when any errors

Parameters

• [in] pipeline: The Audio Pipeline Handle

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

Return

• ESP_OK on success
• ESP_FAIL when any errors

Parameters

• [in] pipeline: The Audio Pipeline Handle
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.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- [in] pipeline: The Audio Pipeline Handle
- [in] ticks_to_wait: The maximum amount of time to block wait for stop

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

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

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

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

esp_err_t audio_pipeline_unlink (audio_pipeline_handle_t pipeline)

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

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- [in] pipeline: The Audio Pipeline Handle

audio_element_handle_t audio_pipeline_get_el_by_tag (audio_pipeline_handle_t pipeline, const char *tag)

Find un-kept element from registered pipeline by tag.

Return

- NULL when any errors
- Others on success

Parameters

2.1. Audio Framework
• [in] pipeline: The Audio Pipeline Handle
• [in] tag: A char pointer

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

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

**Return**

• NULL when any errors
• Others on success

**Parameters**

• [in] pipeline: The Audio Pipeline Handle
• [in] start_el: Specific beginning element
• [in] tag: A char pointer

```c
esp_err_t audio_pipeline_remove_listener (audio_pipeline_handle_t pipeline)
```

Remove event listener from this audio_pipeline.

**Return**

• ESP_OK on success
• ESP_FAIL when any errors

**Parameters**

• [in] pipeline: The Audio Pipeline Handle

```c
esp_err_t audio_pipeline_set_listener (audio_pipeline_handle_t pipeline, audio_event_iface_handle_t evt)
```

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

**Return**

• ESP_OK on success
• ESP_FAIL when any errors

**Parameters**

• [in] pipeline: The Audio Pipeline Handle
• [in] evt: The Event Handle

```c
audio_event_iface_handle_t audio_pipeline_get_event_iface (audio_pipeline_handle_t pipeline)
```

Get the event iface using by this pipeline.

**Return** The Event Handle

**Parameters**

• [in] pipeline: The pipeline
esp_err_t audio_pipeline_link_insert (audio_pipeline_handle_t pipeline, bool first, audio_element_handle_t prev, ringbuf_handle_t conect_rb, audio_element_handle_t next)
Insert the specific audio_element to audio_pipeline, previous element connect to the next element by ring buffer.

Return
• ESP_OK
• ESP_FAIL

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

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] pipeline: The audio pipeline handle
• [in] element_1: The element to add to the audio_pipeline.
• [in] ...: Additional elements to add to the audio_pipeline.

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] pipeline: The audio pipeline handle
• [in] element_1: The element to add to the audio_pipeline.
• [in] ...: Additional elements to add to the audio_pipeline.

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

Return

2.1. Audio Framework
• ESP_OK
• ESP_FAIL

Parameters
• [in] pipeline: The audio pipeline handle
• [in] element_1: The element to add to the audio_pipeline.
• [in] ...: Additional elements to add to the audio_pipeline.

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] pipeline: The audio pipeline handle
• [in] element_1: The element event to subscribe to the audio_pipeline.
• [in] ...: Additional elements event to subscribe to the audio_pipeline.

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

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

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

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

Return
• ESP_OK on success
• ESP_FAIL when any errors

Parameters
• [in] pipeline: The Audio Pipeline Handle

esp_err_t audio_pipeline_reset_ringbuffer(audio_pipeline_handle_t pipeline)  
Reset pipeline element ringbuffer.
Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- [in] pipeline: The Audio Pipeline Handle

esp_err_t audio_pipeline_reset_elements(audio_pipeline_handle_t pipeline)

Reset Pipeline linked elements state.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- [in] pipeline: The Audio Pipeline Handle

esp_err_t audio_pipeline_reset_kept_state(audio_pipeline_handle_t pipeline, audio_element_handle_t el)

Reset the specific element kept state.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- [in] pipeline: The Audio Pipeline Handle
- [in] el: The Audio element Handle

esp_err_t audio_pipeline_breakup_elements(audio_pipeline_handle_t pipeline, audio_element_handle_t kept_ctx_el)

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

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

Return

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

Parameters

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

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

Basing on element’s name already registered by audio_pipeline_register, relink the pipeline following the order of names in the *link_tag.
Note If the ringbuffer is not enough to connect the new pipeline will create new ringbuffer.

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

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

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

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

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

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

esp_err_t audio_pipeline_change_state(audio_pipeline_handle_t pipeline, audio_element_state_t new_state)
Set the pipeline state.

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

Parameters
- [in] pipeline: The Audio Pipeline Handle
- [in] new_state: The new state will be set
Structures

struct audio_pipeline_cfg
Audio Pipeline configurations.

Public Members

int rb_size
Audio Pipeline ringbuffer size

Macros

#define DEFAULT_PIPELINE_RINGBUF_SIZE
#define 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

- audio_pipeline/include/audio_event_iface.h

Functions

audio_event_iface_handle_t audio_event_iface_init (audio_event_iface_cfg_t *config)
Initialize audio event.

Return

- ESP_OK
- ESP_FAIL
Parameters

- **config**: The configurations

```c
esp_err_t audio_event iface_destroy(audio_event iface handle_t evt)
```
Cleanup event, it doesn’t free evt pointer.

**Return**

- ESP_OK
- ESP_FAIL

Parameters

- **evt**: The event

```c
esp_err_t audio_event iface set listener(audio_event iface handle_t evt, audio_event iface handle_t listener)
```
Add audio event `evt` to the listener, then we can listen `evt` event from `listener`.

**Return**

- ESP_OK
- ESP_FAIL

Parameters

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

```c
esp_err_t audio_event iface remove listener(audio_event iface handle_t listener, audio_event iface handle_t evt)
```
Remove audio event `evt` from the listener.

**Return**

- ESP_OK
- ESP_FAIL

Parameters

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

```c
esp_err_t audio_event iface set cmd waiting timeout(audio_event iface handle_t evt, TickType_t wait_time)
```
Set current queue wait time for the event.

**Return**

- ESP_OK
- ESP_FAIL

Parameters

- **evt**: The event
- **wait_time**: The wait time
esp_err_t audio_event_iface_waiting_cmd_msg(audio_event_iface_handle_t evt)

Waiting internal queue message.

Return

• ESP_OK
• ESP_FAIL

Parameters

• evt: The event

esp_err_t audio_event_iface_cmd(audio_event_iface_handle_t evt, audio_event_iface_msg_t *msg)

Trigger an event for internal queue with a message.

Return

• ESP_OK
• ESP_FAIL

Parameters

• evt: The event
• msg: The message

esp_err_t audio_event_iface_cmd_from_isr(audio_event_iface_handle_t evt, audio_event_iface_msg_t *msg)

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

Return

• ESP_OK
• ESP_FAIL

Parameters

• [in] evt: The event
• msg: The message

esp_err_t audio_event_iface_sendout(audio_event_iface_handle_t evt, audio_event_iface_msg_t *msg)

Trigger and event out with a message.

Return

• ESP_OK
• ESP_FAIL

Parameters

• evt: The event
• msg: The message

esp_err_t audio_event_iface_discard(audio_event_iface_handle_t evt)

Discard all ongoing event message.

Return
- ESP_OK
- ESP_FAIL

Parameters
- `evt`: The event

`esp_err_t audio_event_iface_listen(audio_event_iface_handle_t evt, audio_event_iface_msg_t *msg, TickType_t wait_time)`
Listening and invoke callback function if there are any event are coming.

Return
- ESP_OK
- ESP_FAIL

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

QueueHandle_t `audio_event_iface_get_queue_handle(audio_event_iface_handle_t evt)`
Get External queue handle of Emmitter.

Return
External QueueHandle_t

Parameters
- `[in]` `evt`: The external queue

`esp_err_t audio_event_iface_read(audio_event_iface_handle_t evt, audio_event_iface_msg_t *msg, TickType_t wait_time)`
Read the event from all the registered event emitters in the queue set of the interface.

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

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

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

Return
Internal QueueHandle_t

Parameters
- `[in]` `evt`: The Internal queue
esp_err_t audio_event_iface_set_msg_listener(audio_event_iface_handle_t evt, audio_event_iface_handle_t listener)

Add audio internal event \( \text{evt} \) to the listener, then we can listen \( \text{evt} \) event from \( \text{listener} \)

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**

- \( \text{listener} \): The event can listen another event
- \( \text{evt} \): The event to be added to

**Structures**

**struct audio_event_iface_msg_t**

Event message

**Public Members**

- int \( \text{cmd} \)
  Command id
- void *\( \text{data} \)
  Data pointer
- int \( \text{data\_len} \)
  Data length
- void *\( \text{source} \)
  Source event
- int \( \text{source\_type} \)
  Source type (To know where it came from)
- bool \( \text{need\_free\_data} \)
  Need to free data pointer after the event has been processed

**struct audio_event_iface_cfg_t**

Event interface configurations

**Public Members**

- int \( \text{internal\_queue\_size} \)
  It’s optional, Queue size for event \( \text{internal\_queue} \)
- int \( \text{external\_queue\_size} \)
  It’s optional, Queue size for event \( \text{external\_queue} \)
- int \( \text{queue\_set\_size} \)
  It’s optional, QueueSet size for event \( \text{queue\_set} \)
- on_event_iface_func \( \text{on\_cmd} \)
  Function callback for listener when any event arrived

2.1. Audio Framework
void *context
    Context will pass to callback function

TickType_t wait_time
    Timeout to check for event queue

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

Macros

DEFAULT_AUDIO_EVENT_IFACE_SIZE
AUDIO_EVENT_IFACE_DEFAULT_CFG()

Type Definitions

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

typedef struct audio_event_iface *audio_event_iface_handle_t

2.1.4 Audio Common

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

API Reference

Header File

- audio_pipeline/include/audio_common.h

Macros

ELEMENT_SUB_TYPE_OFFSET
mem_assert(x)

Enumerations

enum audio_element_type_t
    Values:
        AUDIO_ELEMENT_TYPE_UNKNOW = 0x01 << ELEMENT_SUB_TYPE_OFFSET
        AUDIO_ELEMENT_TYPE_ELEMENT = 0x01 << (ELEMENT_SUB_TYPE_OFFSET + 1)
        AUDIO_ELEMENT_TYPE_PLAYER = 0x01 << (ELEMENT_SUB_TYPE_OFFSET + 2)
        AUDIO_ELEMENT_TYPE_SERVICE = 0x01 << (ELEMENT_SUB_TYPE_OFFSET + 3)
        AUDIO_ELEMENT_TYPE_PERIPH = 0x01 << (ELEMENT_SUB_TYPE_OFFSET + 4)

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

## 2.1.5 ESP Audio

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

### API Reference

**Header File**

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

### Structures

**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:
    ESP_ERR_AUDIO_NO_ERROR = ESP_OK
    ESP_ERR_AUDIO_FAIL = ESP_FAIL
    ESP_ERR_AUDIO_NO_INPUT_STREAM = ESP_ERR_AUDIO_BASE + 1
    ESP_ERR_AUDIO_NO_OUTPUT_STREAM = ESP_ERR_AUDIO_BASE + 2
    ESP_ERR_AUDIO_NO_CODEC = ESP_ERR_AUDIO_BASE + 3
    ESP_ERR_AUDIO_HAL_FAIL = ESP_ERR_AUDIO_BASE + 4
    ESP_ERR_AUDIO_MEMORY_LACK = ESP_ERR_AUDIO_BASE + 5
    ESP_ERR_AUDIO_INVALID_URI = ESP_ERR_AUDIO_BASE + 6
    ESP_ERR_AUDIO_INVALID_PATH = ESP_ERR_AUDIO_BASE + 7
    ESP_ERR_AUDIO_INVALID_PARAMETER = ESP_ERR_AUDIO_BASE + 8
    ESP_ERR_AUDIO_NOT_READY = ESP_ERR_AUDIO_BASE + 9
    ESP_ERR_AUDIO_NOT_SUPPORT = ESP_ERR_AUDIO_BASE + 10
    ESP_ERR_AUDIO_TIME_OUT = ESP_ERR_AUDIO_BASE + 11
    ESP_ERR_AUDIO_ALREADY_EXISTS = ESP_ERR_AUDIO_BASE + 12
    ESP_ERR_AUDIO_LINK_FAIL = ESP_ERR_AUDIO_BASE + 13
    ESP_ERR_AUDIO_UNKNOWN = ESP_ERR_AUDIO_BASE + 14
    ESP_ERR_AUDIO_OUT_OF_RANGE = ESP_ERR_AUDIO_BASE + 15
    ESP_ERR_AUDIO_STOP_BY_USER = ESP_ERR_AUDIO_BASE + 16
    ESP_ERR_AUDIO_OPEN = ESP_ERR_AUDIO_BASE + 0x100
    ESP_ERR_AUDIO_INPUT = ESP_ERR_AUDIO_BASE + 0x101
    ESP_ERR_AUDIO_PROCESS = ESP_ERR_AUDIO_BASE + 0x102
    ESP_ERR_AUDIO_OUTPUT = ESP_ERR_AUDIO_BASE + 0x103
    ESP_ERR_AUDIO_CLOSE = ESP_ERR_AUDIO_BASE + 0x104

enum esp_audio_status_t

Values:
    AUDIO_STATUS_UNKNOWN = 0
    AUDIO_STATUS_RUNNING = 1
    AUDIO_STATUS_PAUSED = 2
enum audio_termination_type_t

Values:

TERMINATION_TYPE_NOW = 0
Audio operation will be terminated immediately

TERMINATION_TYPE_DONE = 1
Audio operation will be stopped when finished

TERMINATION_TYPE_MAX

enum esp_audio_prefer_t

Values:

ESP_AUDIO_PREFER_MEM = 0
ESP_AUDIO_PREFER_SPEED = 1

enum media_source_type_t

Values:

MEDIA_SRC_TYPE_NULL = 0
MEDIA_SRC_TYPE_MUSIC_BASE = 0x100
MEDIA_SRC_TYPE_MUSIC_SD = MEDIA_SRC_TYPE_MUSIC_BASE + 1
MEDIA_SRC_TYPE_MUSIC_HTTP = MEDIA_SRC_TYPE_MUSIC_BASE + 2
MEDIA_SRC_TYPE_MUSIC_FLASH = MEDIA_SRC_TYPE_MUSIC_BASE + 3
MEDIA_SRC_TYPE_MUSIC_A2DP = MEDIA_SRC_TYPE_MUSIC_BASE + 4
MEDIA_SRC_TYPE_MUSIC_DLNA = MEDIA_SRC_TYPE_MUSIC_BASE + 5
MEDIA_SRC_TYPE_MUSIC_RAW = MEDIA_SRC_TYPE_MUSIC_BASE + 6
MEDIA_SRC_TYPE_MUSIC_MAX = 0x1FF
MEDIA_SRC_TYPE_TONE_BASE = 0x200
MEDIA_SRC_TYPE_TONE_SD = MEDIA_SRC_TYPE_TONE_BASE + 1
MEDIA_SRC_TYPE_TONE_HTTP = MEDIA_SRC_TYPE_TONE_BASE + 2
MEDIA_SRC_TYPE_TONE_FLASH = MEDIA_SRC_TYPE_TONE_BASE + 3
MEDIA_SRC_TYPE_TONE_MAX = 0x2FF
MEDIA_SRC_TYPE_RESERVE_BASE = 0x800
MEDIA_SRC_TYPE_RESERVE_MAX = 0xFFF

2.1. Audio Framework
Header File

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

Functions

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

Create esp_audio instance according to ‘cfg’ parameter.

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

**Return**

- NULL: Error
- Others: esp_audio instance fully certifying

**Parameters**

- [in] cfg: Provide esp_audio initialization configuration

```c
audio_err_t esp_audio_destroy(esp_audio_handle_t handle)
```

Specific esp_audio instance will be destroyed.

**Return**

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

**Parameters**

- [in] handle: The esp_audio instance

```c
audio_err_t esp_audio_input_stream_add(esp_audio_handle_t handle, audio_element_handle_t in_stream)
```

Add audio input stream to specific esp_audio instance.

**Return**

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

**Parameters**

- [in] handle: The esp_audio instance
- [in] in_stream: Audio stream instance

```c
audio_err_t esp_audio_output_stream_add(esp_audio_handle_t handle, audio_element_handle_t out_stream)
```

Add audio output stream to specific esp_audio instance.

**Return**

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

Parameters

• [in] handle: The esp_audio instance
• [in] out_stream: The audio stream element instance

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

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

Return

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

Parameters

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

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

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

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

Return

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

Parameters

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

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

Play the given uri.

The esp_audio_play have follow activity, setup inputstream, outputstream and codec by uri, start all of them.

There is a rule that esp_audio will select input stream, codec and output stream by URI field.

Rule of URI field are as follow.

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

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

Now esp_audio_play support follow URIs.

• “https://dl.espressif.com/dl/audio/mp3_music.mp3”
• “http://media-ice.musicradio.com/ClassicFMMP3”
• “file://sdcard/test.mp3”
• “iis://16000:2@from.pcm/rec.wav#file”
• “iis://16000:1@record.pcm/record.wav#raw”
• “aadp://44100:2@bt/sink/stream.pcm”
• “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.

Return

• 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.

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

```
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

Return
• 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

Parameters
• handle: The esp_audio_handle_t instance
• uri: Such as “file://sdcard/test.wav” or “http://iot.espressif.com/file/example.mp3”,
• pos: Specific starting position by bytes

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

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

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

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

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

Parameters
• [in] handle: The esp_audio instance
• [in] type: Stop immediately or done

`audio_err_t esp_audio_pause(esp_audio_handle_t handle)`
Pause the esp_audio.

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

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

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

Parameters
• [in] handle: The esp_audio instance

audio_err_t esp_audio_resume (esp_audio_handle_t handle)
Resume the music paused.

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

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

Parameters
• [in] handle: The esp_audio instance

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.

Return
• ESP_ERR_AUDIO_NO_ERROR: on success
• ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

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

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.

Return
• ESP_ERR_AUDIO_NO_ERROR: on success
• ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

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

```c
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.

**Return**
- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

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

```c
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.

**Return**
- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

**Parameters**
- **[in]** handle: The esp_audio instance
- **[in]** speed_index: Value from “esp_audio_speed_t” enum.

```c
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.

**Return**
- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

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

```c
audio_err_t esp_audio_vol_set(esp_audio_handle_t handle, int vol)
```

Setting esp_audio volume.

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

---

2.1. Audio Framework
Parameters

- \([\text{in}]\) handle: The esp_audio instance
- \([\text{in}]\) vol: Specific volume will be set. 0-100 is legal. 0 will be mute.

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

Return

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

Parameters

- \([\text{in}]\) handle: The esp_audio instance
- \([\text{out}]\) vol: A pointer to int that indicates esp_audio volume.

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

Return

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

Parameters

- \([\text{in}]\) handle: The esp_audio instance
- \([\text{out}]\) state: A pointer to \text{esp_audio_state_t} that indicates esp_audio status.

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

\textbf{Note} This function works only with decoding music.

Return

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

Parameters

- \([\text{in}]\) handle: The esp_audio instance
- \([\text{out}]\) pos: A pointer to int that indicates esp_audio decoding position.

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

\textbf{Note} This function works only with decoding music.

Return

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

Parameters
• [in] handle: The esp_audio instance
• [out] time: A pointer to int that indicates esp_audio decoding position.

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

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

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

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

Parameters
• [in] handle: The esp_audio instance
• [in] sets: A pointer to esp_audio_setup_t.

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

**audio_err_t esp_audio_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.

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

Parameters
• [in] handle: The esp_audio instance
• [out] seek_time_sec: A pointer to int that indicates esp_audio decoding position.
audio_err_t esp_audio_duration_get (esp_audio_handle_t handle, int *duration)
Get the duration in microseconds of playing music.

**Note** This function works only with decoding music.

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

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

audio_err_t esp_audio_play_timeout_set (esp_audio_handle_t handle, int time_ms)
Setting the maximum amount of time to waiting for esp_audio_play only.

**Return**
- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

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

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

**Return**
- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: no esp_audio instance

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

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.

**Return**
- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: no esp_audio instance

**Parameters**
- [in] handle: The esp_audio instance
- [out] que: A pointer to QueueHandle_t
Structures

```c
struct esp_audio_cfg_t

    esp_audio configuration parameters
```

Public Members

```c
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 specific 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
```

```c
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:

        ESP_AUDIO_PLAY_SPEED_UNKNOW = -1
        ESP_AUDIO_PLAY_SPEED_0_50 = 0
        ESP_AUDIO_PLAY_SPEED_0_75 = 1
        ESP_AUDIO_PLAY_SPEED_1_00 = 2
        ESP_AUDIO_PLAY_SPEED_1_25 = 3
        ESP_AUDIO_PLAY_SPEED_1_50 = 4
        ESP_AUDIO_PLAY_SPEED_1_75 = 5
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

- audio_stream/include/algorithm_stream.h

Functions

audio_element_handle_t algo_stream_init(algorithm_stream_cfg_t *config)

Initialize algorithm stream.

Return The audio element handle

Parameters

- config: The algorithm Stream configuration

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.

Return

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

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.

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

Return ESP_OK

Parameters

- sbuff: I2S data buffer
- len: I2S data len

2.2. Audio Streams
Structures

```c
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 period
- `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 amplification factor of record signal
- `int ref_linear_factor`
  - The linear amplification 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
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:

ALGORITHM_STREAM_INPUT_TYPE1 = 0
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

ALGORITHM_STREAM_INPUT_TYPE2 = 1
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:

ALGORITHM_STREAM_USE_AEC = (0x1 << 0)
Use AEC

ALGORITHM_STREAM_USE_AGC = (0x1 << 1)
Use AGC

ALGORITHM_STREAM_USE_NS = (0x1 << 2)
Use NS

ALGORITHM_STREAM_USE_VAD = (0x1 << 3)
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

- `audio_stream/include/fatfs_stream.h`

Functions

```c
audio_element_handle_t fatfs_stream_init (fatfs_stream_cfg_t *config)
```

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

**Return** The Audio Element handle

**Parameters**

- `config`: The configuration

Structures

```c
struct fatfs_stream_cfg_t
```

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

**Public Members**

```c
audio_stream_type_t type
```

Stream type

```c
int buf_sz
```

Audio Element Buffer size

```c
int out_rb_size
```

Size of output ringbuffer

```c
int task_stack
```

Task stack size

```c
int task_core
```

Task running in core (0 or 1)

```c
int task_prio
```

Task priority (based on freeRTOS priority)

```c
bool 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

• audio_stream/include/http_stream.h

Functions

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

Return The Audio Element handle

Parameters

• config: The configuration
esp_err_t http_stream_next_track (audio_element_handle_t el)
  Connect to next track in the playlist.
  This function can be used in event_handler of http_stream. User can call this function to connect to next track in playlist when he/she gets HTTP_STREAM_FINISH_TRACK event

  **Return**
  * ESP_OK on success
  * ESP_FAIL on errors

  **Parameters**
  * el: The http_stream element handle

esp_err_t http_stream_restart (audio_element_handle_t el)

esp_err_t http_stream_fetch_again (audio_element_handle_t el)
  Try to fetch the tracks again.
  If this is live stream we will need to keep fetching URIs.

  **Return**
  * ESP_OK on success
  * ESP_ERR_NOT_SUPPORTED if playlist is finished

  **Parameters**
  * el: The http_stream element handle

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

  **Return**
  * ESP_OK on success

  **Parameters**
  * el: The http_stream element handle
  * cert: server certification

**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

Macros

HTTP_STREAM_TASK_STACK
HTTP_STREAM_TASK_CORE
HTTP_STREAM_TASK_PRIO
HTTP_STREAM_RINGBUFFER_SIZE
HTTP_STREAM_CFG_DEFAULT()

Type Definitions
define http_stream_event_handle_t (http_stream_event_msg_t *msg)

Enumerations
define http_stream_event_id_t
HTTP Stream hook type.

Values:

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

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

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

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

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

HTTP_STREAM_RESOLVE_ALL_TRACKS
HTTP_STREAM_FINISH_TRACK
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 \texttt{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 \texttt{i2s\_stream\_cfg\_t}.

Application Example

- Reader example: \texttt{recorder/pipeline\_wav\_amr\_sdcard}
- Writer example: \texttt{get-started/play\_mp3\_control}

Header File

- \texttt{audio\_stream/include/i2s\_stream.h}

Functions

\texttt{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 \texttt{AUDIO\_STREAM\_READER} or \texttt{AUDIO\_STREAM\_WRITER}.

\textbf{Note} If I2S stream is enabled with built-in DAC mode, please don’t use \texttt{I2S\_NUM\_1}. The built-in DAC functions are only supported on I2S0 for the current ESP32 chip.

\textbf{Return} The Audio Element handle

\textbf{Parameters}

- \texttt{config}: The configuration

\texttt{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 \texttt{i2s\_stream\_init}

\textbf{Return}

- \texttt{ESP\_OK}
- \texttt{ESP\_FAIL}

\textbf{Parameters}

- \texttt{[in] i2s\_stream}: The i2s element handle
- \texttt{[in] rate}: Clock rate (in Hz)
- \texttt{[in] bits}: Audio bit width (8, 16, 24, 32)
- \texttt{[in] ch}: Number of Audio channels (1: Mono, 2: Stereo)

\texttt{esp\_err\_t i2s\_alc\_volume\_set (audio\_element\_handle\_t i2s\_stream, int volume)}

Setup volume of stream by using ALC.
Return

- ESP_OK
- ESP_FAIL

Parameters

- \([\text{in}]\) \text{i2s\_stream}: The i2s element handle
- \([\text{in}]\) \text{volume}: The volume of stream will be set.

```c
esp_err_t i2s\_alc\_volume\_get(audio\_element\_handle\_t i2s\_stream, int *volume)
```

Get volume of stream.

Return

- ESP_OK
- ESP_FAIL

Parameters

- \([\text{in}]\) \text{i2s\_stream}: The i2s element handle
- \([\text{in}]\) \text{volume}: The volume of stream

```c
esp_err_t i2s\_stream\_sync\_delay(audio\_element\_handle\_t i2s\_stream, int delay\_ms)
```

Set sync delay of stream.

Return

- ESP_OK
- ESP_FAIL

Parameters

- \([\text{in}]\) \text{i2s\_stream}: The i2s element handle
- \([\text{in}]\) \text{delay\_ms}: The delay of stream

Structures

```
struct i2s\_stream\_cfg\_t
```

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

Public Members

```
audio\_stream\_type\_t type
```

Type of stream

```
i2s\_config\_t i2s\_config
```

I2S driver configurations

```
i2s\_port\_t i2s\_port
```

I2S driver hardware port

```
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

i2s_bits_per_sample_t expand_src_bits
The source bits per sample when data expand

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_INTERNAL_DAC_CFG_DEFAULT()
I2S_STREAM_TX_PDM_CFG_DEFAULT()
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

- 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.

Return The audio element handle

Parameters

- config: The PWM Stream configuration

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

2.2. Audio Streams
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

• audio_stream/include/raw_stream.h

Functions

audio_element_handle_t raw_stream_init (raw_stream_cfg_t *cfg)
Initialize RAW stream.

Return The audio element handle

Parameters

• cfg: The RAW Stream configuration

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

Return Number of bytes actually read.

Parameters

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

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

Write data to Stream.

**Return**  Number of bytes written

**Parameters**
- `pipeline`: The audio pipeline handle
- `buffer`: The buffer
- `buf_size`: Number of bytes to write

**Structures**

```c
struct raw_stream_cfg_t
```

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

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

**Public Members**

```c
audio_stream_type_t type
```

Type of stream

```c
int out_rb_size
```

Size of output ringbuffer

**Macros**

```c
RAW_STREAM_RINGBUFFER_SIZE
RAW_STREAM_CFG_DEFAULT()
```

### 2.2.7 SPIFFS Stream

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

**Application Example**

- `player/pipeline_spiffs_mp3`
Header File

- audio_stream/include/spiffs_stream.h

Functions

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

Return The Audio Element handle
Parameters
- config: The configuration

Structures

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

Public Members

audio_stream_type_t type
Stream type

int buf_sz
Audio Element Buffer size

int out_rb_size
Size of output ringbuffer

int task_stack
Task stack size

int task_core
Task running in core (0 or 1)

int task_prio
Task priority (based on freeRTOS priority)

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

- 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.

Return The audio element handle

Parameters

- config: The configuration

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:
   TCP_STREAM_STATE_NONE
   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

  • 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.

  Return The Audio Element handle

  Parameters

  • config: The configuration
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

- 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.

Return The Audio Element handle

Parameters

- config: The configuration

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

```c
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
```

```c
struct embed_item_info
    Embed tone information in flash.

Public Members

```c
const uint8_t *address
    The corresponding address in flash
int size
    Size of corresponding data
```

Macros

```c
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

```c
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

- 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.

Return The Audio Element handle

Parameters

- config: The configuration

esp_err_t tts_stream_set_strings (audio_element_handle_t el, const char *strings)
Set tts stream strings.

Return

- ESP_OK
- ESP_FAIL

Parameters

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

esp_err_t tts_stream_set_speed (audio_element_handle_t el, tts_voice_speed_t speed)
Setting tts stream voice speed.

Return

- ESP_OK
- ESP_FAIL

Parameters

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

esp_err_t tts_stream_get_speed (audio_element_handle_t el, tts_voice_speed_t *speed)
Get tts stream voice speed.
Return

- ESP_OK
- ESP_FAIL

Parameters

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

Structures

```c
struct tts_stream_cfg_t
```

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

Public Members

```c
audio_stream_type_t type
```

Stream type

```c
int buf_siz
```

Audio Element Buffer size

```c
int out_rb_siz
```

Size of output ringbuffer

```c
int task_stack
```

Task stack size

```c
int task_core
```

Task running in core (0 or 1)

```c
int task_prio
```

Task priority (based on freeRTOS priority)

```c
bool 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

```c
enum tts_voice_speed_t
{
    TTS_VOICE_SPEED_0,
    TTS_VOICE_SPEED_1,
    TTS_VOICE_SPEED_2,
    TTS_VOICE_SPEED_3,
    TTS_VOICE_SPEED_4,
    TTS_VOICE_SPEED_5,
    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

- playlist/include/sdcard_scan.h

Functions

```c
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.

**Return**

- ESP_OK success
- ESP_FAIL failed

**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

Type Definitions

```c
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

- playlist/include/sdcard_list.h

Functions

```c
esp_err_t sdcard_list_create(playlist_operator_handle_t *handle)

Create a playlist in sdcard by list id.

Return

- ESP_OK success
- ESP_FAIL failed

Parameters

- [out] handle: The playlist handle from application layer

esp_err_t sdcard_list_show(playlist_operator_handle_t handle)

Show all the URLs in sdcard playlist.

Return

- ESP_OK success
- ESP_FAIL failed

Parameters

- handle: Playlist handle

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

The following URLs in sdcard playlist.

Return

- ESP_OK success
- ESP_FAIL failed

Parameters

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

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

The previous URLs in sdcard playlist.

Return
```
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle
• step: The offset of URL from current URL
• [out] url_buff: A second rank pointer to get a address of URL

bool sdcard_list_exist (playlist_operator_handle_t handle, const char *url)
Judge whether the url exists in sdcard playlist.

Return
• true existence
• false Non-existent

Parameters
• handle: Playlist handle
• url: The url to be checked

esp_err_t sdcard_list_reset (playlist_operator_handle_t handle)
Reset sdcard playlist.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle

esp_err_t sdcard_list_current (playlist_operator_handle_t handle, char **url_buff)
Get current URL in sdcard playlist.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle
• [out] url_buff: A second rank pointer to get a address of URL

esp_err_t sdcard_list_choose (playlist_operator_handle_t handle, int url_id, char **url_buff)
Choose a url by url id.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle
• url_id: The id of url in sdcard list
• [out] url_buff: A second rank pointer to get a address of URL

int sdcards_list_get_url_num (playlist_operator_handle_t handle)
Get URLs number in sdcard playlist.

Return
• URLs number in sdcard playlist
• ESP_FAIL Fail to get number of urls

Parameters
• handle: Playlist handle

int sdcards_list_get_url_id (playlist_operator_handle_t handle)
Get current url id in the sdcard playlist.

Return
• Current url id in partition playlist
• ESP_FAIL Fail to get url id

Parameters
• handle: Playlist handle

esp_err_t sdcards_list_destroy (playlist_operator_handle_t handle)
Destroy sdcard playlist.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle

esp_err_t sdcards_list_save (playlist_operator_handle_t handle, const char *url)
Save URL to sdcard playlist.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle
• url: URL to be saved
**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**

- `playlist/include/dram_list.h`

**Functions**

```c
esp_err_t dram_list_create(playlist_operator_handle_t *handle)
```

Create a playlist in dram.

**Return**

- ESP_OK success
- ESP_FAIL failed

**Parameters**

- `[out]` handle: The playlist handle from application layer

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

Save URL to dram playlist.

**Return**

- ESP_OK success
- ESP_FAIL failed

**Parameters**

- handle: Playlist handle
- url: URL to be saved

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

The following URLs in dram playlist.

**Return**

- ESP_OK success
- ESP_FAIL failed

**Parameters**

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

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

The previous URLs in dram playlist.

**Return**
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle
• step: The offset of URL from current URL
• [out] url_buff: A second rank pointer to get a address of URL

bool dram_list_exist (playlist_operator_handle_t handle, const char *url)
Judge whether the url exists in dram playlist.

Return
• true existence
• false Non-existent

Parameters
• handle: Playlist handle
• url: The url to be checked

esp_err_t dram_list_reset (playlist_operator_handle_t handle)
Reset dram playlist.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle

esp_err_t dram_list_current (playlist_operator_handle_t handle, char **url_buff)
The current URL in current playlist.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle
• [out] url_buff: A second rank pointer to get a address of URL

esp_err_t dram_list_choose (playlist_operator_handle_t handle, int url_id, char **url_buff)
Choose a url by url id.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters

2.3. Playlist
### dram_list_get_url_num

```c
int dram_list_get_url_num(playlist_operator_handle_t handle)
```

Get URLs number in the dram playlist.

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

**Parameters**
- handle: Playlist handle

### dram_list_get_url_id

```c
int dram_list_get_url_id(playlist_operator_handle_t handle)
```

Get current url id in the dram playlist.

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

**Parameters**
- handle: Playlist handle

### dram_list_show

```c
esp_err_t dram_list_show(playlist_operator_handle_t handle)
```

Show all the URLs in the dram playlist.

**Return**
- ESP_OK success
- ESP_FAIL failed

**Parameters**
- handle: Playlist handle

### dram_list_remove_by_url

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

Remove corresponding url in dram list.

**Return**
- ESP_OK success
- ESP_FAIL failed

**Parameters**
- handle: Playlist handle
- url: The url to be removed

### dram_list_remove_by_url_id

```c
esp_err_t dram_list_remove_by_url_id(playlist_operator_handle_t handle, uint16_t url_id)
```

Remove url by id.
Return

• ESP_OK success
• ESP_FAIL failed

Parameters

• handle: Playlist handle
• url_id: The url id to be removed

`esp_err_t dram_list_destroy (playlist_operator_handle_t handle)`

Destroy the dram playlist.

Return

• ESP_OK success
• ESP_FAIL failed

Parameters

• handle: Playlist handle

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

• playlist/include/flash_list.h

Functions

`esp_err_t flash_list_create (playlist_operator_handle_t *handle)`

Create a playlist in nvs flash.

Return

• ESP_OK success
• ESP_FAIL failed

Parameters

• [out] handle: Playlist handle

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

Save URL to nvs flash list.

Return

• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle
• url: URL to be saved

esp_err_t flash_list_show (playlist_operator_handle_t handle)
Show all the URLs in nvs flash list.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle

esp_err_t flash_list_next (playlist_operator_handle_t handle, int step, char **url_buff)
The following URLs in nvs flash playlist.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle
• step: The offset of URL from current URL
• [out] url_buff: A second rank pointer to get a address of URL

esp_err_t flash_list_prev (playlist_operator_handle_t handle, int step, char **url_buff)
The previous URLs in nvs flash playlist.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle
• step: The offset of URL from current URL
• [out] url_buff: A second rank pointer to get a address of URL

esp_err_t flash_list_current (playlist_operator_handle_t handle, char **url_buff)
The current URL in nvs flash playlist.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle
• [out] url_buff: A second rank pointer to get a address of URL
bool **flash_list_exist** *(playlist_operator_handle_t handle, const char *url)*
Judge whether the url exists in flash playlist.

**Return**
- true existence
- false Non-existent

**Parameters**
- **handle**: Playlist handle
- **url**: The url to be checked

**esp_err_t flash_list_reset** *(playlist_operator_handle_t handle)*
Reset flash playlist.

**Return**
- ESP_OK success
- ESP_FAIL failed

**Parameters**
- **handle**: Playlist handle

**esp_err_t flash_list_choose** *(playlist_operator_handle_t handle, int url_id, char **url_buff)*
Choose a url by url id.

**Return**
- ESP_OK success
- ESP_FAIL failed

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

**int flash_list_get_url_num** *(playlist_operator_handle_t handle)*
Get URLs number in the flash playlist.

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

**Parameters**
- **handle**: Playlist handle

**int flash_list_get_url_id** *(playlist_operator_handle_t handle)*
Get current url id in the flash playlist.

**Return**
- Curernt url id in flash playlist

2.3. Playlist
• ESP_FAIL Fail to get url id

Parameters
• handle: Playlist handle

esp_err_t flash_list_destroy (playlist_operator_handle_t handle)
Destroy the nvs flash playlist.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle

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
• 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

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• [out] handle: The playlist handle from application layer

esp_err_t partition_list_save (playlist_operator_handle_t handle, const char *url)
Save URL to partition playlist.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle
• url: URL to be saved
esp_err_t partition_list_next (playlist_operator_handle_t handle, int step, char **url_buff)
The following URLs in partition playlist.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle
• step: The offset of URL from current URL
• [out] url_buff: A second rank pointer to get a address of URL

esp_err_t partition_list_prev (playlist_operator_handle_t handle, int step, char **url_buff)
The previous URLs in partition playlist.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle
• step: The offset of URL from current URL
• [out] url_buff: A second rank pointer to get a address of URL

bool partition_list_exist (playlist_operator_handle_t handle, const char *url)
Judge whether the url exists in partition playlist.

Return
• true existence
• false Non-existent

Parameters
• handle: Playlist handle
• url: The url to be checked

esp_err_t partition_list_reset (playlist_operator_handle_t handle)
Reset partition playlist.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle

esp_err_t partition_list_current (playlist_operator_handle_t handle, char **url_buff)
Get current URL in the partition playlist.
Return

- ESP_OK success
- ESP_FAIL failed

Parameters

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

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

Choose a url by url id.

Return

- ESP_OK success
- ESP_FAIL failed

Parameters

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

```c
int partition_list_get_url_num (playlist_operator_handle_t handle)
```

Get URLs number in the partition playlist.

Return

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

Parameters

- handle: Playlist handle

```c
int partition_list_get_url_id (playlist_operator_handle_t handle)
```

Get current url id in the partition playlist.

Return

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

Parameters

- handle: Playlist handle

```c
esp_err_t partition_list_show (playlist_operator_handle_t handle)
```

Show all the URLs in the partition playlist.

Return

- ESP_OK success
- ESP_FAIL failed

Parameters

```c
```
• handle: Playlist handle

esp_err_t `partition_list_destroy`(playlist_operator_handle_t handle)
Destroy the partition playlist.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist handle

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
• playlist/include/playlist.h

Functions

`playlist_handle_t playlist_create`(void)
Create a playlist manager handle.

Return
• 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 overwritten by the newest partition 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.

Return
• ESP_OK success
• ESP_FAIL failed

Parameters
• handle: Playlist manager handle
• list_handle: The playlist handle to be added
• list_id: The playlist id to be registered

esp_err_t `playlist_checkout_by_id`(playlist_handle_t handle, uint8_t id)
Playlist checkout by list id.

2.3. Playlist
Return

• ESP_OK success
• ESP_FAIL failed

Parameters

• handle: Playlist handle
• id: Specified list id

```c
int playlist_get_list_num (playlist_handle_t handle)
```
Get number of playlists in the handle.

Return

• success Number of playlists in handle
• failed -1

Parameters

• handle: Playlist handle

```c
playlist_type_t playlist_get_current_list_type (playlist_handle_t handle)
```
Get current playlist type.

Return

• success Type of current playlist
• failed -1

Parameters

• handle: Playlist handle

```c
int playlist_get_current_list_id (playlist_handle_t handle)
```
Get current playlist id.

Return

• success Current playlist id
• failed -1

Parameters

• handle: Playlist handle

```c
esp_err_t playlist_get_current_list_url (playlist_handle_t handle, char **url_buff)
```
Get current URL in current playlist.

Return

• ESP_OK success
• ESP_FAIL failed

Parameters

• handle: Playlist handle
• [out] url_buff: A second rank pointer to get a address of URL
int playlist_get_current_list_url_num (playlist_handle_t handle)
Get number of URLs in current playlist.

Return Number of URLs in current playlist

Parameters
- handle: Playlist handle

int playlist_get_current_list_url_id (playlist_handle_t handle)
Get current url id in current playlist.

Return Current url’s id in current playlist

Parameters
- handle: Playlist handle

esp_err_t playlist_save (playlist_handle_t handle, const char *url)
Save a URL to the current playlist.

Return
- ESP_OK success
- ESP_FAIL failed

Parameters
- handle: Playlist handle
- url: The URL to be saved on sdcard

esp_err_t playlist_next (playlist_handle_t handle, int step, char **url_buff)
Next URL in current playlist.

Return
- ESP_OK success
- ESP_FAIL failed

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

esp_err_t playlist_prev (playlist_handle_t handle, int step, char **url_buff)
Previous URL in current playlist.

Return
- ESP_OK success
- ESP_FAIL failed

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

```
esp_err_t playlist_choose (playlist_handle_t handle, int url_id, char **url_buff)
Choose a url by url id.
```

**Return**

• ESP_OK success
• ESP_FAIL failed

**Parameters**

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

```
esp_err_t playlist_show (playlist_handle_t handle)
Show URLs in current playlist.
```

**Return**

• ESP_OK success
• ESP_FAIL failed

**Parameters**

• handle: Playlist handle

```
esp_err_t playlist_reset (playlist_handle_t handle)
Reset current playlist.
```

**Return**

• ESP_OK success
• ESP_FAIL failed

**Parameters**

• handle: Playlist handle

```
esp_err_t playlist_remove_by_url (playlist_handle_t handle, const char *url)
Remove corresponding url.
```

**Return**

• ESP_OK success
• ESP_FAIL failed

**Parameters**

• handle: Playlist handle
• url: The url to be removed

```
esp_err_t playlist_remove_by_url_id (playlist_handle_t handle, uint16_t url_id)
Remove url by url id.
```
Return

- ESP_OK success
- ESP_FAIL failed

Parameters

- handle: Playlist handle
- url_id: The id of url to be removed

bool playlist_exist (playlist_handle_t handle, const char *url)
Judge whether the url exists in current playlist.

Return

- true existence
- false Non-existent

Parameters

- handle: Playlist handle
- url: The url to be checked

esp_err_t playlist_destroy (playlist_handle_t handle)
Destroy all playlists in the handle.

Return

- ESP_OK success
- ESP_FAIL failed

Parameters

- handle: Playlist handle

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:

  PLAYLIST_UNKNOWN = -1
      Unknown type

  PLAYLIST_SDCARD
      Playlist in sdcard

  PLAYLIST_FLASH
      Playlist in nvs
2.4 Codecs

2.4.1 AAC Decoder

Decode an audio data stream provided in AAC format.

API Reference

Header File

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

Functions

\texttt{audio_element_handle_t aac_decoder_init(aac_decoder_cfg_t *\textit{config})}

Create an Audio Element handle to decode incoming AAC data.

\textbf{Return} The audio element handle

\textbf{Parameters}

- \textit{config}: The configuration

Structures

\textbf{struct aac_decoder_cfg_t}

AAC Decoder configuration.

\textbf{Public Members}

- \texttt{int out\_rb\_size}
  Size of output ringbuffer

- \texttt{int task\_stack}
  Task stack size

- \texttt{int task\_core}
  CPU core number (0 or 1) where decoder task is running

- \texttt{int task\_prio}
  Task priority (based on freeRTOS priority)

- \texttt{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

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

Functions

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

    Return  The audio element handle

    Parameters

        • config: The configuration
Structures

```c
struct amr_decoder_cfg_t
    AMR Decoder configuration.
```

Public Members

```c
int out_rb_size
    Size of output ringbuffer

int task_stack
    Task stack size

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

int task_prio
    Task priority (based on freeRTOS priority)

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

Macros

```c
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

```c
• esp-adf-libs/esp_codec/include/codecs/amrnb_encoder.h
```

Functions

```c
esp_err_t amrnb_encoder_set_bitrate(audio_element_handle_t self, amrnb_encoder_bitrate_t bitrate_mode)
    Set AMRNB encoder bitrate.

Return ESP_OK ESP_FAIL

Parameters
```

```c
• self: Audio element handle

• bitrate_mode: Bitrate choose, value from amrnb_encoder_bitrate_t
```

```c
audio_element_handle_t amrnb_encoder_init (amrnb_encoder_cfg_t *config)
    Create an Audio Element handle to encode incoming AMRNB data.
```
Return  The audio element handle

Parameters

- config: The configuration

Structures

struct amrnb_encoder_cfg_t
AMRNB Encoder configurations.

Public Members

- int out_rb_size
  Size of output ringbuffer
- int task_stack
  Task stack size
- int task_core
  Task running in core (0 or 1)
- int task_prio
  Task priority (based on freeRTOS priority)
- 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

```c
enum amrnb_encoder_bitrate_t
    Enum of AMRNB Encoder bitrate choose.
    Values:
    AMRNB_ENC_BITRATE_UNKNOW = -1
        Invalid mode
    AMRNB_ENC_BITRATE_MR475 = 0
    AMRNB_ENC_BITRATE_MR515 = 1
    AMRNB_ENC_BITRATE_MR59 = 2
    AMRNB_ENC_BITRATE_MR67 = 3
    AMRNB_ENC_BITRATE_MR74 = 4
    AMRNB_ENC_BITRATE_MR795 = 5
    AMRNB_ENC_BITRATE_MR102 = 6
    AMRNB_ENC_BITRATE_MR122 = 7
    AMRNB_ENC_BITRATE_MRDTX = 8
    AMRNB_ENC_BITRATE_N_MODES = 9
```

API Reference - AMR-WB Encoder

Header File

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

Functions

```c
esp_err_t amrwb_encoder_set_bitrate(audio_element_handle_t self, amrwb_encoder_bitrate_t bitrate_mode)
    Set AMRWB encoder bitrate.
    Return ESP_OK ESP_FAIL
    Parameters
        • self: Audio element handle
        • bitrate_mode: Bitrate choose, value from amrwb_encoder_bitrate_t

audio_element_handle_t amrwb_encoder_init(amrwb_encoder_cfg_t *config)
    Create an Audio Element handle to encode incoming amrwb data.
    Return The audio element handle
    Parameters
        • config: The configuration
```
Structures

```c
struct amrwb_encoder_cfg_t
    AMRWB Encoder configurations.
```

Public Members

```c
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)
```

```c
amrwb_encoder_bitrate_t bitrate_mode
    AMRWB Encoder bitrate choose
```

```c
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)
```

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

Macros

```c
AMRWB_ENCODER_TASK_STACK
AMRWB_ENCODER_TASK_CORE
AMRWB_ENCODER_TASK_PRIO
AMRWB_ENCODER_RINGBUFFER_SIZE
DEFAULT_AMRWB_ENCODER_CONFIG()
```

Enumerations

```c
enum amrwb_encoder_bitrate_t
    Enum of AMRWB Encoder bitrate choose.
```

```c
    Values:
    AMRWB_ENC_BITRATE_MDNONE = -1
        Invalid mode
    AMRWB_ENC_BITRATE_MD66 = 0
        6.60kbps
    AMRWB_ENC_BITRATE_MD885 = 1
        8.85kbps
```
AMRWB_ENC_BITRATE_MD1265 = 2
12.65kbps

AMRWB_ENC_BITRATE_MD1425 = 3
14.25kbps

AMRWB_ENC_BITRATE_MD1585 = 4
15.85kbps

AMRWB_ENC_BITRATE_MD1825 = 5
18.25kbps

AMRWB_ENC_BITRATE_MD1985 = 6
19.85kbps

AMRWB_ENC_BITRATE_MD2305 = 7
23.05kbps

AMRWB_ENC_BITRATE_MD2385 = 8
23.85kbps

AMRWB_ENC_BITRATE_N_MODES = 9
Invalid mode

2.4.3 FLAC Decoder

Decode an audio data stream provided in FLAC format.

API Reference

Header File

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

Functions

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

Return The audio element handle

Parameters

• config: The configuration

Structures

struct flac_decoder_cfg_t
FLAC Decoder configuration.
Public Members

int out_rb_size  
   Size of output ringbuffer

int task_stack  
   Task stack size

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

int task_prio  
   Task priority (based on freeRTOS priority)

bool stack_in_ext  
   Try to allocate stack in external memory

Macros

FLAC_DECODER_TASK_STACK_SIZE
FLAC_DECODER_TASK_CORE
FLAC_DECODER_TASK_PRIO
FLAC_DECODER_RINGBUFFER_SIZE
DEFAULT_FLAC_DECODER_CONFIG()

2.4.4 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

- esp-adf-libs/esp_codec/include/codecs/mp3_decoder.h
Functions

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

Return The audio element handle
Parameters
- config: The configuration

Structures

struct mp3_decoder_cfg_t
Mp3 Decoder configuration.

Public Members

int out_rb_size
Size of output ringbuffer

int task_stack
Task stack size

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

int task_prio
Task priority (based on freeRTOS priority)

bool stack_in_ext
Try to allocate stack in external memory

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

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

Functions

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

Return The audio element handle

Parameters

- config: The configuration

Structures

struct ogg_decoder_cfg_t
OGG Decoder configuration.

Public Members

int out_rb_size
Size of output ringbuffer

int task_stack
Task stack size

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

int task_prio
Task priority (based on freeRTOS priority)

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

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

Functions

`audio_element_handle_t decoder_opus_init (opus_decoder_cfg_t *config)`

Create an Audio Element handle to decode incoming OPUS data.

Return  The audio element handle

Parameters

- config: The configuration

Structures

`struct opus_decoder_cfg_t`

OPUS Decoder configuration.

Public Members

- `int out_rb_size`
  Size of output ringbuffer
- `int task_stack`
  Task stack size
- `int task_core`
  CPU core number (0 or 1) where decoder task in running
- `int task_prio`
  Task priority (based on freeRTOS priority)
- `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

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

Functions

\texttt{audio_element_handle_t wav_decoder_init(wav_decoder_cfg_t \ast config)}

Create an Audio Element handle to decode incoming WAV data.

\textbf{Return}  The audio element handle

\textbf{Parameters}

- \texttt{config}: The configuration

Structures

\texttt{struct wav_decoder_cfg_t}

brief WAV Decoder configurations
Public Members

int out_rb_size
Size of output ringbuffer

int task_stack
Task stack size

int task_core
Task running in core (0 or 1)

int task_prio
Task priority (based on freeRTOS priority)

bool stack_in_ext
Try to allocate stack in external memory

Macros

WAV_DECODER_TASK_STACK
WAV_DECODER_TASK_CORE
WAV_DECODER_TASK_PRIO
WAV_DECODER_RINGBUFFER_SIZE
DEFAULT_WAV_DECODER_CONFIG()

API Reference - Encoder

Header File

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

Functions

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

Return The audio element handle

Parameters

• config: The configuration
Structures

```c
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

```c
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 newcomer audio file, into one output audio file. The newcomer audio file will be downmixed into the base audio file with individual gains applied to each file.

![Illustration of Downmixing Process](image)

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

The downmix process has 3 states:

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

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

Application Example

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

API Reference

Header File

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

void downmix_set_input_rb_timeout (audio_element_handle_t self, int ticks_to_wait, 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)
Passes number of channels for output stream. Only supported mono and dual.

Return ESP_OK ESP_FAIL

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

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

Return ESP_OK ESP_FAIL

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

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

Return ESP_OK ESP_FAIL

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

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

Parameters

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

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

Return ESP_OK ESP_FAIL

Parameters

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

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

Return ESP_OK ESP_FAIL

Parameters

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

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

Return ESP_OK ESP_FAIL

Parameters

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

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

Return The initialized Audio Element handle

Parameters

• config: the configuration

2.5. Audio Processing
Structures

```
struct downmix_cfg_t
    Downmix configuration.
```

**Public Members**

```
enum 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.

<table>
<thead>
<tr>
<th>Band Index</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31 Hz</td>
</tr>
<tr>
<td>1</td>
<td>62 Hz</td>
</tr>
<tr>
<td>2</td>
<td>125 Hz</td>
</tr>
<tr>
<td>3</td>
<td>250 Hz</td>
</tr>
<tr>
<td>4</td>
<td>500 Hz</td>
</tr>
<tr>
<td>5</td>
<td>1 kHz</td>
</tr>
<tr>
<td>6</td>
<td>2 kHz</td>
</tr>
<tr>
<td>7</td>
<td>4 kHz</td>
</tr>
<tr>
<td>8</td>
<td>8 kHz</td>
</tr>
<tr>
<td>9</td>
<td>16 kHz</td>
</tr>
</tbody>
</table>

Default gain of each band is -13 dB. To set the gains of all bands use structure `equalizer_cfg`. To set the gain of individual band use function `equalizer_set_gain_info()`.
Application Example

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

API Reference

Header File

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

Functions

```c
esp_err_t equalizer_set_info (audio_element_handle_t self, int rate, int ch)
```

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

**Return** ESP_OK ESP_FAIL

**Parameters**

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

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

Set the audio gain to be processed by the equalizer.

**Return** ESP_OK ESP_FAIL

**Parameters**

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

```c
audio_element_handle_t equalizer_init (equalizer_cfg_t *config)
```

Create an Audio Element handle that equalizes incoming data.

**Return** The created audio element handle

**Parameters**

- `config`: The configuration
## Structures

**struct equalizer_cfg**

Equalizer Configuration.

### Public Members

- **int samplerate**
  
  Audio sample rate (in Hz)

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

- **int *set_gain**
  
  Equalizer gain

- **int out_rb_size**
  
  Size of output ring buffer

- **int task_stack**
  
  Task stack size

- **int task_core**
  
  Task running in core...

- **int task_prio**
  
  Task priority

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

### Macros

- **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.
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

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

Functions

`esp_err_t esp_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 don’t consure source data infomation, please use rsp_filter_change_src_info ti instead this function. In future, the function will be forbidden.

**Return** ESP_OK ESP_FAIL

**Parameters**

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

`esp_err_t esp_filter_change_src_info(audio_element_handle_t self, int src_rate, int src_ch, int src_bit)`
Set the source audio sample rate and the number of channels, bit per sample to be processed by the resample.

**Return** ESP_OK ESP_FAIL

**Parameters**

- `self`: Audio element handle
- `src_rate`: The sample rate of stream data
- `src_ch`: The number of channels of stream data
- `src_bit`: The bit per sample of stream data
audio_element_handle_t rsp_filter_init (rsp_filter_cfg_t *config)

Create an Audio Element handle to resample incoming data.

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

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

**Return** The audio element handler

**Parameters**

- config: The configuration

**Structures**

**struct rsp_filter_cfg_t**

Resample Filter Configuration.

**Public Members**

- int src_rate
  - The sampling rate of the source PCM file (in Hz)
- int src_ch
  - The number of channel(s) of the source PCM file (Mono=1, Dual=2)
- int dest_rate
  - The sampling rate of the destination PCM file (in Hz)
- int dest_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, 16bits, 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.

```c
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.
```

```c
esp_rsp_prefer_type_t prefer_flag
The select flag about lesser CPU usage or lower INRAM usage, refer to esp_resample.h
```

```c
int out_rb_size
Output ringbuffer size
```

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

```c
int task_core
Task running on core
```

```c
int task_prio
Task priority
```

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

### Macros

```c
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

- 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.

Return ESP_OK ESP_FAIL

Parameters

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

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

Return ESP_OK ESP_FAIL

Parameters

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

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

Return The sonic audio element handle

Parameters

- config: The sonic configuration
Structures

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

**Public Members**

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

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

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

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

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

**struct sonic_cfg_t**
Sonic configuration.

**Public Members**

`sonic_info_t sonic_info`
Information of sonic

int `out_rb_size`
Size of output ring buffer

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
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

- bluetooth_service/include/bluetooth_service.h

Functions

```c
esp_err_t bluetooth_service_start (bluetooth_service_cfg_t *config)
```

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

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**

- config: The configuration

```c
audio_element_handle_t bluetooth_service_create_stream()
```

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

**Return** The Audio Element handle

```c
esp_periph_handle_t bluetooth_service_create_periph()
```

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

**Return** The Peripheral handle

```c
esp_err_t periph_bluetooth_play (esp_periph_handle_t periph)
```

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

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**

- [in] periph: The periph

```c
esp_err_t periph_bluetooth_pause (esp_periph_handle_t periph)
```

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

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**

- [in] periph: The periph
esp_err_t periph_bluetooth_stop(esp_periph_handle_t periph)
Send the AVRC passthrough command (STOP) to the Bluetooth device.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] periph: The periph

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] periph: The periph

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] periph: The periph

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] periph: The periph

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] periph: The periph
esp_err_t **periph_bluetooth_discover** (*esp_periph_handle_t periph*)
Start device discovery.

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

**Parameters**
- [in] periph: The periph

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

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

**Parameters**
- [in] periph: The periph

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

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

**Parameters**
- [in] periph: The periph
- [in] remote_bda: remote Bluetooth device address

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

**Return**
- ESP_OK
- ESP_FAIL

int **periph_bluetooth_get_a2dp_sample_rate ()**
Get a2dp sample rate.
Return

- 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:

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

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

Header File

  • bluetooth_service/include/bt_keycontrol.h

Header File

  • bluetooth_service/include/hfp_stream.h

Header File

  • 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

- 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.

**Return**  NULL failed others input key service handle

**Parameters**

- `input_key_config`: Configuration of input key service

`periph_service_state_t get_input_key_service_state(periph_service_handle_t input_handle)`

Get the state of input key service.

**Return**  state of input key service

**Parameters**

- `input_handle`: The handle 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.

**Return**  ESP_OK success ESP_FAIL failed

**Parameters**

- `input_key_handle`: handle of service
- `input_key_info`: input key’s information
- `add_key_num`: number of keys

Structures

`struct input_key_service_info_t`

input key’s information

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

```c
periph_service_config_t based_cfg
Peripheral service configuration
```

```c
esp_periph_set_handle_t handle
Peripheral set handle
```

Macros

```c
INPUT_KEY_SERVICE_TASK_STACK_SIZE
INPUT_KEY_SERVICE_TASK_PRIORITY
INPUT_KEY_SERVICE_TASK_ON_CORE
INPUT_KEY_SERVICE_DEFAULT_CONFIG()
```

Enumerations

```c
enum input_key_service_action_id_t
input key action id

Values:

INPUT_KEY_SERVICE_ACTION_UNKNOWN = 0
unknown action id

INPUT_KEY_SERVICE_ACTION_CLICK
click action id

INPUT_KEY_SERVICE_ACTION_CLICK_RELEASE
click release action id

INPUT_KEY_SERVICE_ACTION_PRESS
long press action id

INPUT_KEY_SERVICE_ACTION_PRESS_RELEASE
long press release id
```

Header File

- `input_key_service/include/input_key_com_user_id.h`

Enumerations

```c
enum input_key_user_id_t
input key user user-defined id

Values:

INPUT_KEY_USER_ID_UNKNOWN = -1
unknown user id

INPUT_KEY_USER_ID_REC = 0x01
user id for recording
```
INPUT_KEY_USER_ID_SET = 0x02
    user id for settings

INPUT_KEY_USER_ID_PLAY = 0x03
    user id for playing

INPUT_KEY_USER_ID_MODE = 0x04
    user id for mode

INPUT_KEY_USER_ID_VOLDOWN = 0x05
    user id for volume down

INPUT_KEY_USER_ID_VOLUP = 0x06
    user id for volume up

INPUT_KEY_USER_ID_MUTE = 0x07
    user id for mute

INPUT_KEY_USER_ID_CAPTURE = 0x08
    user id for capture photo

INPUT_KEY_USER_ID_MSG = 0x09
    user id for message

INPUT_KEY_USER_ID_BATTERY_CHARGING = 0x0A
    user id for battery charging

INPUT_KEY_USER_ID_WAKEUP = 0x0B
    user id for GPIO wakeup

INPUT_KEY_USER_ID_COLOR = 0x0C
    user id for color

INPUT_KEY_USER_ID_MAX = 0x101

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
- advanced_examples/esp_dispatcher_dueros
Header File

- wifi_service/include/wifi_service.h

Functions

```c
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

```c
struct wifi_service_config_t
    WiFi service configurations.
```

Public Members

```c
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:
    WIFI_SERV_EVENT_UNKNOWN
    WIFI_SERV_EVENT_CONNECTING
    WIFI_SERV_EVENT_CONNECTED
    WIFI_SERV_EVENT_DISCONNECTED
    WIFI_SERV_EVENT_SETTING_TIMEOUT
    WIFI_SERV_EVENT_SETTING_FAILED
    WIFI_SERV_EVENT_SETTING_FINISHED

enum wifi_service_disconnect_reason_t
    WiFi STA disconnection reasons.
    Values:
    WIFI_SERV_STA_UNKNOWN
    WIFI_SERV_STA_COM_ERROR
    WIFI_SERV_STA_AUTH_ERROR
    WIFI_SERV_STA_AP_NOT_FOUND
    WIFI_SERV_STA_BY_USER
    WIFI_SERV_STA_SET_INFO
Header File

- wifi_service/include/esp_wifi_setting.h

Functions

```c
esp_wifi_setting_handle_t esp_wifi_setting_create(const char *tag)
```

brief Create wifi setting handle instance

Return

- NULL, Fail
- Others, Success

Parameters

- `tag`: Tag of the wifi setting handle

```c
esp_err_t esp_wifi_setting_destroy(esp_wifi_setting_handle_t handle)
```

brief Destroy wifi setting handle instance

Return

- ESP_OK
- ESP_FAIL

Parameters

- `handle`: The wifi setting handle instance

```c
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.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `handle`: The wifi setting handle instance
- `start`: The start wifi setting
- `stop`: The stop
- `teardown`: The destroy

```c
esp_err_t esp_wifi_setting_register_notify_handle(esp_wifi_setting_handle_t handle, void *on_handle)
```

Register the notify execute handle.

Return

- ESP_OK
• ESP_FAIL

Parameters

• handle: The peripheral handle

• on_handle: The notify execute handle

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

Return

• ESP_OK

• ESP_FAIL

Parameters

• handle: The wifi setting handle instance

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

Return

• ESP_OK

• ESP_FAIL

Parameters

• [in] handle: The wifi setting handle instance

• data: The user data

void *esp_wifi_setting_get_data(esp_wifi_setting_handle_t handle)
Get the user data stored on handle

Return user data pointer

Parameters

• [in] handle: The wifi setting handle instance

esp_err_t esp_wifi_setting_start(esp_wifi_setting_handle_t handle)
Call this to execute start function of wifi setting instance.

Return

• ESP_OK

• ESP_FAIL

Parameters

• handle: The wifi setting handle instance
esp_err_t \textbf{esp_wifi_setting_stop} (esp_wifi_setting_handle_t handle)
Call this to execute \texttt{stop} function of wifi setting instance.

\textbf{Return}
- ESP_OK
- ESP_FAIL

\textbf{Parameters}
- \texttt{handle}: The wifi setting handle instance

esp_err_t \textbf{esp_wifi_setting_teardown} (esp_wifi_setting_handle_t handle, wifi_config_t *info)
Call this to execute \texttt{teardown} function of wifi setting instance.

\textbf{Return}
- ESP_OK
- ESP_FAIL

\textbf{Parameters}
- \texttt{handle}: The wifi setting handle instance
- \texttt{info}: The \texttt{wifi_config_t}

\textbf{Type Definitions}

\begin{verbatim}
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)
\end{verbatim}

\textbf{Header File}

- wifi_service/include/smart_config.h

\textbf{Functions}

\begin{verbatim}
esp_wifi_setting_handle_t \textbf{smart_config_create} (smart_config_info_t *info)
brief Create smartconfig setting handle instance

\textbf{Return}
- NULL, Fail
- Others, Success

\textbf{Parameters}
- \texttt{info}: Configuration of the smartconfig
\end{verbatim}
Structures

```c
struct smart_config_info_t
  esp smartconfig configuration
```

Public Members

```c
smartconfig_type_t type
  Type of smartconfig
```

Macros

```c
SMART_CONFIG_INFO_DEFAULT()
```

Header File

- wifi_service/include/blufi_config.h

Functions

```c
esp_wifi_setting_handle_t blufi_config_create (void *info)
  Create blufi setting handle instance.

  Return
  • NULL, Fail
  • Others, Success

  Parameters
  • [in] info: A pointer to void
```

```c
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.

  Return
  • NULL, Fail
  • Others, Success

  Parameters
  • [in] handle: Wifi setting handle
  • [in] flag: bool type of station connection state
```

```c
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.

  Return
  • ESP_FAIL, Fail
```
• ESP_OK, Success

Parameters
• [in] handle: Wifi setting handle
• [in] data: Customized data
• [in] data_len: Customized data length

`esp_err_t blufi_send_customized_data (esp_wifi_setting_handle_t handle)`
Send customized data that be set before.

Return
• ESP_FAIL, Fail
• ESP_OK, Success

Parameters
• [in] handle: Wifi setting handle

Header File

• 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

Return
• NULL, Fail
• Others, Success

Parameters
• info: Configuration of the airkiss

Structures

`struct airkiss_lan_pack_param_t`
airkiss lan data pack
Public Members

void *appid
    APP identifier data

void *deviceid
    Device identifier 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()
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

- ota_service/include/ota_service.h

Functions

periph_service_handle_t ota_service_create(ota_service_config_t *config)

Create the OTA service instance.

Return

- NULL: Failed
- Others: Success

Parameters

- config: configuration of the OTA service

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.

Return

- ESP_OK: Success
- Others: Failed

Parameters

- [in] handle: pointer to ‘periph_service_handle_t’ structure
- [in] list: pointer to ‘ota_upgrade_ops_t’ structure
- [in] list_len: length of the ‘list’
**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:

   OTA_SERV_EVENT_TYPE_RESULT
   OTA_SERV_EVENT_TYPE_FINISH

enum ota_service_err_reason_t
   The OTA service error reasons.
   Values:

   OTA_SERV_ERR_REASON_UNKNOWN = ESP_FAIL
   OTA_SERV_ERR_REASON_SUCCESS = ESP_OK
   OTA_SERV_ERR_REASON_NULL_POINTER = OTA_SERVICE_ERR_REASON_BASE + 1
   OTA_SERV_ERR_REASON_URL_PARSE_FAIL = OTA_SERVICE_ERR_REASON_BASE + 2
   OTA_SERV_ERR_REASON_ERROR_VERSION = OTA_SERVICE_ERR_REASON_BASE + 3
   OTA_SERV_ERR_REASON_NO_HIGHER_VERSION = OTA_SERVICE_ERR_REASON_BASE + 4
   OTA_SERV_ERR_REASON_ERROR_MAGIC_WORD = OTA_SERVICE_ERR_REASON_BASE + 5
   OTA_SERV_ERR_REASON_ERROR_PROJECT_NAME = OTA_SERVICE_ERR_REASON_BASE + 6
   OTA_SERV_ERR_REASON_FILE_NOT_FOUND = OTA_SERVICE_ERR_REASON_BASE + 7

2.6. Services
OTA_SERV_ERR_REASON_PARTITION_NOT_FOUND = OTA_SERVICE_ERR_REASON_BASE + 8
OTA_SERV_ERR_REASON_PARTITION_WT_FAIL = OTA_SERVICE_ERR_REASON_BASE + 9
OTA_SERV_ERR_REASON_PARTITION_RD_FAIL = OTA_SERVICE_ERR_REASON_BASE + 10
OTA_SERV_ERR_REASON_STREAM_INIT_FAIL = OTA_SERVICE_ERR_REASON_BASE + 11
OTA_SERV_ERR_REASON_STREAM_RD_FAIL = OTA_SERVICE_ERR_REASON_BASE + 12
OTA_SERV_ERR_REASON_GET_NEW_APP_DESC_FAIL = OTA_SERVICE_ERR_REASON_BASE + 13

Header File

• 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

Return

• void

Parameters

• [in] ops: 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

Return

• void

Parameters

• [in] ops: 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

Return

• OTA_SERV_ERR_REASON_SUCCESS: Success
• Others: Failed

Parameters

• [in] handle: pointer to upgrade handle
• [in] buf: pointer to receive buffer
• [in] wanted_size: bytes to read

ota_service_err_reason_t ota_data_partition_write(void *handle, char *buf, int size)

write to the data partition under upgrading
Return

- OTA_SERV_ERR_REASON_SUCCESS: Success
- Others: Failed

Parameters

- [in] handle: pointer to upgrade handle
- [in] buf: pointer to data buffer
- [in] size: bytes to write

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 application, this API needs to be called.

Return

- void

Parameters

- [in] handle: 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)

Return

- -1: Failed
- Others: version number

Parameters

- [in] version: pointer to the string of version

Header File

- ota_service/include/esp_fs_ota.h

Functions

esp_err_t esp_fs_ota (esp_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.

Return

- ESP_OK: OTA data updated, next reboot will use specified partition.
- ESP_FAIL: For generic failure.
- ESP_ERR_INVALID_ARG: Invalid argument

2.6. Services
• 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.

Parameters
• [in] ota_config: pointer to esp_fs_ota_config_t structure.

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.

Return
• 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.

Parameters
• [in] ota_config: pointer to esp_fs_ota_config_t structure
• [out] handle: pointer to an allocated data of type esp_fs_ota_handle_t which will be initialised in this function

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.

Return
• 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.

Parameters
• [in] fs_ota_handle: pointer to esp_fs_ota_handle_t structure
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

**Return**
- ESP_OK: Clean-up successful
- ESP_ERR_INVALID_STATE
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_OTA_VALIDATE_FAILED: Invalid app image

**Parameters**
- [in] fs_ota_handle: pointer to esp_fs_ota_handle_t structure

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.

**Return**
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_FAIL: Failed to read image descriptor
- ESP_OK: Successfully read image descriptor

**Parameters**
- [in] fs_ota_handle: pointer to esp_fs_ota_handle_t structure
- [out] new_app_info: pointer to an allocated esp_app_desc_t structure

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

```c
const char *path
    file path
const int buffer_size
    Size of buffer
```

Macros

```c
ESP_ERR_FS_OTA_BASE
ESP_ERR_FS_OTA_IN_PROGRESS
```

Type Definitions

```c
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

- dueros_service/include/dueros_service.h

Functions

```c
audio_service_handle_t dueros_service_create()
Create the dueros service.
```

**Return**
- NULL, Fail
- Others, Success

```c
service_state_t dueros_service_state_get()
Get dueros service state.
```

**Return** The state of service
esp_err_t dueros_voice_upload(audio_service_handle_t handle, void *buf, int len)
Upload voice to backend server.

Return ESP_OK ESP_FAIL
Parameters
  • handle: dueros service handle
  • buf: Data buffer
  • len: Size of buffer

esp_err_t dueros_voice_cancel(audio_service_handle_t handle)
Cancel the current session.

Return ESP_OK ESP_FAIL
Parameters
  • handle: dueros service handle

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

  • 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

```c
struct display_service_config_t
{
    Display service configurations.
}
```

**Public Members**

```c
periph_service_config_t based_cfg
    Peripheral service configuration
void *instance
    Sub-instance
```

Type Definitions

```c
typedef struct display_service_impl *display_service_handle_t
```

Enumerations

```c
enum display_pattern_t
{
    Values:
    DISPLAY_PATTERN_UNKNOWN = 0
    DISPLAY_PATTERN_WIFI_SETTING = 1
    DISPLAY_PATTERN_WIFI_CONNECTTING = 2
    DISPLAY_PATTERN_WIFI_CONNECTED = 3
    DISPLAY_PATTERN_WIFI_DISCONNECTED = 4
    DISPLAY_PATTERN_WIFI_SETTING_FINISHED = 5
    DISPLAY_PATTERN_BT_CONNECTTING = 6
    DISPLAY_PATTERN_BT_CONNECTED = 7
    DISPLAY_PATTERN_BT_DISCONNECTED = 8
    DISPLAY_PATTERN_RECORDING_START = 9
    DISPLAY_PATTERN_RECORDING_STOP = 10
    DISPLAY_PATTERN_RECOGNITION_START = 11
    DISPLAY_PATTERN_RECOGNITION_STOP = 12
    DISPLAY_PATTERN_WAKEUP_ON = 13
    DISPLAY_PATTERN_WAKEUP_FINISHED = 14
    DISPLAY_PATTERN_MUSIC_ON = 15
    DISPLAY_PATTERN_MUSIC_FINISHED = 16
    DISPLAY_PATTERN_VOLUME = 17
    DISPLAY_PATTERN_MUTE_ON = 18
    DISPLAY_PATTERN_MUTE_OFF = 19
```
DISPLAY_PATTERN_TURN_ON = 20
DISPLAY_PATTERN_TURN_OFF = 21
DISPLAY_PATTERN_BATTERY_LOW = 22
DISPLAY_PATTERN_BATTERY_CHARGING = 23
DISPLAY_PATTERN_BATTERY_FULL = 24
DISPLAY_PATTERN_POWERON_INIT = 25
DISPLAY_PATTERN_WIFI_NO_CFG = 26
DISPLAY_PATTERN_SPEECH_BEGIN = 27
DISPLAY_PATTERN_SPEECH_OVER = 28
DISPLAY_PATTERN_MAX

Header File

- 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.

Return

- NULL Error
- others Success

esp_err_t led_bar_aw2013_pattern (void *handle, int pat, int value)

Set led bar display pattern.

Return

- ESP_OK
- ESP_FAIL

Parameters

- handle: led bar instance
- pat: display pattern
- value: value of pattern

esp_err_t led_bar_aw2013_set_blink_time (void *handle, uint8_t time, int period)

Set blinking period and times.

Return

- ESP_OK
- ESP_FAIL

2.6. Services
Parameters

- handle: led bar instance
- time: times of blink
- period: period of blink

void led_bar_aw2013_deinit (esp_periph_handle_t handle)
Destroy esp_periph_handle_t instance.

Parameters

- handle: led bar instance

Header File

- display_service/led_bar/include/led_bar_is31x.h

Functions

`esp_periph_handle_t led_bar_is31x_init ()`
Initialize esp_periph_handle_t instance.

Return

- NULL, Fail
- Others, Success

`esp_err_t led_bar_is31x_pattern (void *handle, int pat, int value)`
Set led bar display pattern.

Return

- ESP_OK
- ESP_FAIL

Parameters

- handle: led bar instance
- pat: display pattern
- value: value of pattern

void led_bar_is31x_deinit (esp_periph_handle_t handle)
Destroy esp_periph_handle_t instance.

Return

- ESP_OK
- ESP_FAIL

Parameters

- handle: led bar instance
Header File

- 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.

**Return**

- led_bar_ws2812_handle_t

**Parameters**

- gpio_num: The GPIO number of ws2812
- led_num: The number of all ws2812

`esp_err_t led_bar_ws2812_pattern (void *handle, int pat, int value)`
Set ws2812 pattern.

**Return**

- ESP_OK, success
- Others, fail

**Parameters**

- handle: ws2812 indicator instance
- pat: display pattern
- value: value of pattern

`esp_err_t led_bar_ws2812_deinit (led_bar_ws2812_handle_t handle)`
Destroy led_bar_ws2812_handle_t instance.

**Return**

**Parameters**

- handle: ws2812 indicator instance
Type Definitions

typedef struct led_bar_ws2812_impl *led_bar_ws2812_handle_t

Header File

- 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.

**Return**
- NULL, Fail
- Others, Success

**Parameters**
- num: led gpio number

**esp_err_t led_indicator_pattern (void *handle, int pat, int value)**
Set led indicator display pattern.

**Return**
- ESP_OK
- ESP_FAIL

**Parameters**
- handle: led indicator instance
- pat: display pattern
- value: value of pattern

**void led_indicator_deinit (led_indicator_handle_t handle)**
Destroy led_indicator_handle_t instance.

**Return**
- ESP_OK
- ESP_FAIL

**Parameters**
- handle: led indicator instance
Type Definitions

```c
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

- `battery_service/include/battery_service.h`

Functions

```c
periph_service_handle_t battery_service_create (battery_service_config_t *config)
```

Create the battery service instance.

**Return**

- NULL: Failed
- Others: Success

**Parameters**

- `config`: configuration of the battery service

```c
esp_err_t battery_service_vol_report_switch (periph_service_handle_t handle, bool on_off)
```

Start or stop the battery voltage report.

**Return**

- ESP_OK: Success
- ESP_ERR_INVALID_ARG handle is NULL

**Parameters**

- [in] `handle`: pointer to `periph_service_handle_t` structure
- [in] `on_off`: ‘true’ to start, ‘false’ to stop

```c
esp_err_t battery_service_set_vol_report_freq (periph_service_handle_t handle, int freq)
```

Set voltage report frequency.

**Return**
• ESP_OK: Success
• ESP_ERR_INVALID_ARG handle is NULL

Parameters
• [in] handle: pointer to 'periph_service_handle_t' structure
• [in] freq: voltage report frequency

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

dataclass battery_service_event_t
Battery service event.

Values:

BAT_SERV_EVENT_UNKNOWN
BAT_SERV_EVENT_VOL_REPORT = 1
BAT_SERVEVENTBAT_FULL
BAT_SERV_EVENT_BAT_LOW
BAT_SERV_EVENT_CHARGING_BEGIN = 100
BAT_SERV_EVENT_CHARGING_STOP

Header File

- battery_service/monitors/include/voltage_monitor.h

Functions

```c
vol_monitor_handle_t vol_monitor_create (vol_monitor_param_t *config)
Create the voltage monitor instance.

Parameters
- [in] config: pointer to ‘vol_monitor_param_t’ structure
```

```c
esp_err_t vol_monitor_destroy (vol_monitor_handle_t handle)
Destroy the voltage monitor.

Parameters
- [in] handle: pointer to ‘vol_monitor_handle_t’ structure
```

```c
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
- [in] handle: pointer to ‘vol_monitor_handle_t’ structure
- [in] event_cb: callback used to handle the events of voltage monitor
- [in] user_ctx: user’s data
```

```c
esp_err_t vol_monitor_start_freq_report (vol_monitor_handle_t handle)
Start the voltage report with the configured frequency.

Parameters
```

2.6. Services
• ESP_OK: Success
• Others: Failed

Parameters
• [in] handle: pointer to ‘vol_monitor_handle_t’ structure

esp_err_t vol_monitor_stop_freq_report (vol_monitor_handle_t handle)
Stop the voltage frequency report.

Return
• ESP_OK: Success
• Others: Failed

Parameters
• [in] handle: pointer to ‘vol_monitor_handle_t’ structure

esp_err_t vol_monitor_set_report_freq (vol_monitor_handle_t handle, int freq)
Set the voltage report frequency.

Return
• ESP_OK: Success
• Others: Failed

Parameters
• [in] handle: pointer to ‘vol_monitor_handle_t’ structure
• [in] freq: voltage report frequency

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 \text{read_freq} \times \text{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:
      VOL_MONITOR_EVENT_FREQ_REPORT
      VOL_MONITOR_EVENT_BAT_FULL
      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

   • 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.

**Return**

- 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.

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**

- [in] handle: the `periph_service_handle_t`
- [in] url: server addr

**periph_service_handle_t coredump_upload_service_create (coredump_upload_service_config_t *config)**

Create the core dump upload service instance.

**Return**

- NULL: Failed
- Others: Success

**Parameters**

- config: configuration of the OTA service

### 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

- 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

Return
- NULL, Fail
- Others, Success

Parameters
- [in] config: Configuration of the peripheral service instance

`esp_err_t periph_service_destroy(periph_service_handle_t handle)`
brief Destroy peripheral service instance

Return
- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

Parameters
- [in] handle: The peripheral service instance

`esp_err_t periph_service_start(periph_service_handle_t handle)`
brief Start the specific peripheral service

Return
- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

Parameters
- [in] handle: The peripheral service instance
```c
esp_err_t periph_service_stop(periph_service_handle_t handle)

brief Stop the specific peripheral service

Return

• ESP_OK
• ESP_FAIL
• ESP_ERR_INVALID_ARG

Parameters

• [in] handle: The peripheral service instance
```

```c
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

Return

• ESP_OK
• ESP_FAIL
• ESP_ERR_INVALID_ARG

Parameters

• [in] handle: The peripheral service instance
• [in] cb: A pointer to service_callback
• [in] ctx: A pointer to user context
```

```c
esp_err_t periph_service_callback(periph_service_handle_t handle, periph_service_event_t *evt)

brief Called peripheral service by configurations

Return

• ESP_OK
• ESP_FAIL
• ESP_ERR_INVALID_ARG

Parameters

• [in] handle: The peripheral service instance
• [in] evt: A pointer to periph_service_event_t
```

```c
esp_err_t periph_service_set_data(periph_service_handle_t handle, void *data)

brief Set user data to specific peripheral service instance

Return

• ESP_OK
• ESP_FAIL
• ESP_ERR_INVALID_ARG

Parameters

• [in] handle: The peripheral service instance
```
• [in] data: A pointer to user data

```c
void *periph_service_get_data(periph_service_handle_t handle)
```

brief Get user data by specific peripheral service instance

**Return** A pointer to user data

**Parameters**

• [in] handle: The peripheral service instance

```c
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

**Return**

• ESP_OK
• ESP_FAIL
• ESP_ERR_INVALID_ARG

**Parameters**

• [in] handle: The peripheral service instance
• [in] ioctl_handle: Sub-instance handle
• [in] cmd: Command of value
• [in] value: Data of the command

### 2.6.11 Structures

**struct periph_service_event_t**

Peripheral service event informations.

**Public Members**

```c
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

cchar *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

erenum periph_service_state_t
Peripheral service state.

Values:

PERIPH_SERVICE_STATE_UNKNOWN
PERIPH_SERVICE_STATE_IDLE
PERIPH_SERVICE_STATE_RUNNING
PERIPH_SERVICE_STATE_STOPPED
2.6.14 Header File

• 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

Return
  • NULL, Fail
  • Others, Success

Parameters
  • [in] config: Configuration of the audio service instance

esp_err_t audio_service_destroy(audio_service_handle_t handle)
  brief Destroy audio service instance

Return
  • ESP_OK
  • ESP_FAIL
  • ESP_ERR_INVALID_ARG

Parameters
  • [in] handle: The audio service instance

esp_err_t audio_service_start(audio_service_handle_t handle)
  brief Start the specific audio service

Return
  • ESP_OK
  • ESP_FAIL
  • ESP_ERR_INVALID_ARG

Parameters
  • [in] handle: The audio service instance

esp_err_t audio_service_stop(audio_service_handle_t handle)
  brief Stop the specific audio service

Return
  • ESP_OK
  • ESP_FAIL
  • ESP_ERR_INVALID_ARG

Parameters
• [in] handle: The audio service instance

```c
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.

**Return**

• ESP_OK
• ESP_FAIL
• ESP_ERR_INVALID_ARG

**Parameters**

• [in] handle: The audio service instance
• [in] cb: A pointer to service_callback
• [in] ctx: A pointer to user context

```c
esp_err_t audio_service_callback(audio_service_handle_t handle, service_event_t *evt)
```

brief Called audio service by configurations

**Return**

• ESP_OK
• ESP_FAIL
• ESP_ERR_INVALID_ARG

**Parameters**

• [in] handle: The audio service instance
• [in] evt: A pointer to service_event_t

```c
esp_err_t audio_service_connect(audio_service_handle_t handle)
```

brief Connect the specific audio service

**Return**

• ESP_OK
• ESP_FAIL
• ESP_ERR_INVALID_ARG

**Parameters**

• [in] handle: The audio service instance

```c
esp_err_t audio_service_disconnect(audio_service_handle_t handle)
```

brief Disconnect the specific audio service

**Return**

• ESP_OK
• ESP_FAIL
• ESP_ERR_INVALID_ARG
Parameters

- **[in]** handle: The audio service instance

`esp_err_t audio_service_set_data(audio_service_handle_t handle, void *data)`

brief Set user data to specific audio service instance

**Return**

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

Parameters

- **[in]** handle: The audio service instance
- **[in]** data: A pointer to user data

`void *audio_service_get_data(audio_service_handle_t handle)`

brief Get user data by specific audio service instance

**Return** A pointer to user data

**Parameters**

- **[in]** handle: The audio service instance

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:
      SERVICE_STATE_UNKNOWN
      SERVICE_STATE_IDLE
      SERVICE_STATE_CONNECTING
      SERVICE_STATE_CONNECTED
      SERVICE_STATE_RUNNING
      SERVICE_STATE_STOPPED
2.7 Speech Recognition

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

Provided in this section functions also include automatic speech detection, also known as *voice activity detection (VAD)*, and *speech recording engine*.

The Speech Recognition API is designed to easy integrate with existing *Audio Framework* to retrieve the audio stream from a microphone connected to the audio chip.

### 2.7.1 Speech Recognition Interface

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

![Sample Speech Recognition Pipeline](image)

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

**Application Example**

The `speech_recognition/wwe/main/main.c` example demonstrates how to initialize the model, determine the number of samples and the sample rate of voice data to feed to the model, and detect the wakeup word.

Implementation of the speech recognition API is demonstrated in that example.

**API Reference**

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

### 2.7.2 Voice Activity Detection

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

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

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

API Reference

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

2.7.3 Recorder Engine

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

API Reference

Header File

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

Functions

esp_err_t rec_engine_create (rec_config_t *cfg)
Create recorder engine according to parameters.

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

Return
- 0: Success
- -1: Error

Parameters
- cfg: See rec_config_t structure for additional details

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

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

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

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

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

**Parameters**
- `flag`: REC_VOICE_SUSPEND_ON: Voice detection is suspended
  REC_VOICE_SUSPEND_OFF: Voice detection is not suspended

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

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

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

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

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

**Note** Upon completion of this function rec_close callback will be triggered.

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

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

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

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

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

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

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

**Return**

- 0: Success
- -1: Error

Parameters

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

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

**Note** support_encoding and rec_engine_enc_enable must be set.

**Return**

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

Parameters

- **buffer**: data pointer
- **buffer_size**: Size of buffer, must be equal to REC_ONE_BLOCK_SIZE.
- **waiting_time**: Timeout for reading data.
- **out_size**: Valid size of buffer.

```c
esp_err_t rec_engine_mute_enable (bool mute_enable)
```
Enable the recoder mute, or not.

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

**Return**

- 0: Success
- -1: Error

Parameters

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

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

- 0: Success
- -1: Error

Parameters

- \texttt{wakeup\_start\_t}: true is WAKEUP\_START, false is not.

Structures

\begin{verbatim}
struct rec_config_t
    recorder configuration parameters

Public Members

int \texttt{one\_frame\_duration\_ms}
    Duration of one frame (optional)

int \texttt{sensitivity}
    For response accuracy rate sensitivity. Default 0: 90%, 1: 95%

int \texttt{vad\_off\_delay\_ms}
    Vad off delay to stop if no voice is detected

int \texttt{wakeup\_time\_ms}
    Time of wakeup

bool \texttt{support\_encoding}
    Support encoding data

const char *\texttt{extension}
    Encoding format.”amr” or “amrwb” support

int \texttt{task\_core}
    Recorder task running in core (0 or 1)

bool \texttt{enable\_wwe}
    Enable Wake Word Engine or not

rec_open \texttt{open}
    Recorder open callback function

rec_fetch \texttt{fetch}
    Recorder fetch data callback function

rec_close \texttt{close}
    Recorder close callback function

rec_callback \texttt{evt\_cb}
    Recorder event callback function

void *\texttt{user\_data}
    Pointer to user data (optional)
\end{verbatim}
Macros

REC_ONE_BLOCK_SIZE
DEFAULT_REC_ENGINE_CONFIG()

Type Definitions

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

Enumerations

enum rec_event_type_t
    Values:
    REC_EVENT_WAKEUP_START
    REC_EVENT_WAKEUP_END
    REC_EVENT_VAD_START
    REC_EVENT_VAD_STOP

enum rec_voice_suspend_t
    Values:
    REC_VOICE_SUSPEND_OFF
    REC_VOICE_SUSPEND_ON

2.8 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

- 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.

**Return** The peripheral sets instance

**Parameters**

- [in] `config`: The configurations

`esp_err_t esp_periph_set_destroy (esp_periph_set_handle_t periph_set_handle)`

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

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**

- `periph_set_handle`: The esp_periph_set_handle_t instance

`esp_err_t esp_periph_set_stop_all (esp_periph_set_handle_t periph_set_handle)`

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

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**

- `periph_set_handle`: The esp_periph_set_handle_t instance
esp_periph_handle_t esp_periph_set_get_by_id(esp_periph_set_handle_t periph_set_handle, int periph_id)

Get the peripheral handle by Peripheral ID.

Return The esp_periph_handle_t

Parameters

• periph_set_handle: The esp_periph_set_handle_t instance
• [in] periph_id: as esp_periph_id_t, or any ID you use when calling esp_periph_create

audio_event_iface_handle_t esp_periph_set_get_event_iface(esp_periph_set_handle_t periph_set_handle)

Return the event_iface used by this esp_peripherals.

Return The audio event iface handle

Parameters

• periph_set_handle: The esp_periph_set_handle_t instance

esp_err_t esp_periph_set_register_callback(esp_periph_set_handle_t periph_set_handle, esp_periph_event_handle_t cb, void *user_context)

Register peripheral sets event callback function.

Return

• ESP_OK
• ESP_FAIL

Parameters

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

QueueHandle_t esp_periph_set_get_queue(esp_periph_set_handle_t periph_set_handle)

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

Return The queue handle

Parameters

• periph_set_handle: The esp_periph_set_handle_t instance

esp_err_t esp_periph_set_list_init(esp_periph_set_handle_t periph_set_handle)

Call this function to initialize all the listed peripherals.

Note Work with no task peripheral set only

Return

• ESP_OK
• ESP_FAIL

Parameters
• periph_set_handle: The esp_periph_set_handle_t instance

```c
esp_err_t esp_periph_set_list_run(esp_periph_set_handle_t periph_set_handle, audio_event_iface_msg_t msg)
```

Call this function to run all the listed peripherals.

**Note** Work with no task peripheral set only

**Return**
• ESP_OK
• ESP_FAIL

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

```c
esp_err_t esp_periph_set_list_destroy(esp_periph_set_handle_t periph_set_handle)
```

Call this function to destroy all the listed peripherals.

**Note** Work with no task peripheral set only

**Return**
• ESP_OK
• ESP_FAIL

**Parameters**
• periph_set_handle: The esp_periph_set_handle_t instance

```c
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.

**Return**
• ESP_OK
• ESP_FAIL

**Parameters**
• periph_set_handle: The esp_periph_set_handle_t instance
• periph: The esp_periph_handle_t instance

```c
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.

**Return**
• ESP_OK
• ESP_FAIL

**Parameters**
• periph_set_handle: The esp_periph_set_handle_t instance
• time_ms: The waiting time

`esp_periph_handle_t esp_periph_create(int periph_id, const char *tag)`

Call this function to initialize a new peripheral.

**Return** The peripheral handle

**Parameters**

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

`esp_err_t esp_periph_set_function(esp_periph_handle_t periph, esp_periph_func init, esp_periph_run_func run, esp_periph_func destroy)`

Each peripheral has a cycle of sequential operations from initialization, execution of commands to destroying the peripheral. These operations are represented by functions passed as call parameters to this function.

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**

- `[in]` periph: The periph
- `[in]` init: The initialize
- `[in]` run: The run
- `[in]` destroy: The destroy

`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`

**Return**

- ESP_OK on success
- ESP_FAIL when any errors

**Parameters**

- `[in]` periph_set_handle: The esp_periph_set_handle_t instance
- `[in]` periph: The peripheral instance

`esp_err_t esp_periph_stop(esp_periph_handle_t periph)`

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

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**

- `[in]` periph: The periph
esp_err_t esp_periph_send_cmd(esp_periph_handle_t periph, int cmd, void *data, int data_len)

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] periph: The periph
- [in] cmd: The command
- [in] data: The data
- [in] data_len: The data length

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] periph: The periph
- [in] cmd: The command
- [in] data: The data
- [in] data_len: The data length

esp_err_t esp_periph_send_event(esp_periph_handle_t periph, int event_id, void *data, int data_len)

In addition to sending an event via event_iface, this function will dispatch the event_handle callback if the event_handle callback is provided at esp_periph_init

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] periph: The peripheral
- [in] event_id: The event identifier
- [in] data: The data
- [in] data_len: The data length

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

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

• You do not need to stop or destroy the timer, when the `esp_periph_destroy` function is called, it will stop and destroy all
• This timer using FreeRTOS Timer, with autoreload = true

Return

• ESP_OK
• ESP_FAIL

Parameters

• [in] `periph`: The peripheral
• [in] `interval_tick`: The interval tick
• [in] `callback`: The callback

```c
esp_err_t esp_periph_stop_timer (esp_periph_handle_t periph)
```
Stop peripheral timer.

Return

• ESP_OK
• ESP_FAIL

Parameters

• [in] `periph`: The peripheral

```c
esp_err_t esp_periph_set_data (esp_periph_handle_t periph, void *data)
```
Set the user data.

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

Return

• ESP_OK
• ESP_FAIL

Parameters

• [in] `periph`: The peripheral
• `data`: The data

```c
void *esp_periph_get_data (esp_periph_handle_t periph)
```
Get the user data stored in the peripheral.

Return Peripheral data pointer

Parameters

• [in] `periph`: The peripheral

```c
esp_periph_state_t esp_periph_get_state (esp_periph_handle_t periph)
```
Get the current state of peripheral.
**Return** The peripheral working state

**Parameters**
- [in] **periph**: The handle of peripheral

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

**Return** The peripheral identifier

**Parameters**
- [in] **periph**: The peripheral

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

**Return**
- ESP_OK
- ESP_FAIL

**Parameters**
- [in] **periph**: The peripheral
- [in] **periph_id**: The peripheral identifier

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

**Return**
- ESP_OK
- ESP_FAIL

**Parameters**
- **periph**: The peripheral handle

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

**Return**
- ESP_OK
- ESP_FAIL

**Parameters**
- **periph**: The peripheral handle

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

**Return**
- ESP_OK
esp_err_t esp_periph_register_on_events (esp_periph_handle_t periph, esp_periph_event_t *evts)

Register peripheral on event handle.

Return

• ESP_OK
• ESP_FAIL

Parameters

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

Structures

struct esp_periph_config_t

Common peripherals configurations.

Public Members

int task_stack

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

int task_prio

Service task priority (based on freeRTOS priority)

int task_core

Service task running in core (0 or 1)

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:

PERIPH_ID_BUTTON = AUDIO_ELEMENT_TYPE_PERIPH + 1
PERIPH_ID_TOUCH = AUDIO_ELEMENT_TYPE_PERIPH + 2
PERIPH_ID_SDCARD = AUDIO_ELEMENT_TYPE_PERIPH + 3
PERIPH_ID_WIFI = AUDIO_ELEMENT_TYPE_PERIPH + 4
PERIPH_ID_FLASH = AUDIO_ELEMENT_TYPE_PERIPH + 5
PERIPH_ID_AUXIN = AUDIO_ELEMENT_TYPE_PERIPH + 6
PERIPH_ID_ADC = AUDIO_ELEMENT_TYPE_PERIPH + 7
PERIPH_ID_CONSOLE = AUDIO_ELEMENT_TYPE_PERIPH + 8
PERIPH_ID_BLUETOOTH = AUDIO_ELEMENT_TYPE_PERIPH + 9
PERIPH_ID_LEDE = AUDIO_ELEMENT_TYPE_PERIPH + 10
PERIPH_ID_SPIFFS = AUDIO_ELEMENT_TYPE_PERIPH + 11
PERIPH_ID_ADC BTN = AUDIO_ELEMENT_TYPE_PERIPH + 12
PERIPH_ID_IS31FL3216 = AUDIO_ELEMENT_TYPE_PERIPH + 13
PERIPH_ID_GPIO_ISR = AUDIO_ELEMENT_TYPE_PERIPH + 14
PERIPH_ID_WS2812 = AUDIO_ELEMENT_TYPE_PERIPH + 15
PERIPH_ID_AW2013 = AUDIO_ELEMENT_TYPE_PERIPH + 16
PERIPH_ID_LCD = AUDIO_ELEMENT_TYPE_PERIPH + 17

enum esp_periph_state_t
Peripheral working state.

Values:
PERIPH_STATE_NULL
PERIPH_STATE_INIT
PERIPH_STATE_RUNNING
PERIPH_STATE_PAUSE
PERIPH_STATE_STOPPING
PERIPH_STATE_ERROR
PERIPH_STATE_STATUS_MAX

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

### 2.8.2 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**

- esp_peripherals/include/periph_wifi.h

**Functions**

```c
esp_periph_handle_t periph_wifi_init (periph_wifi_cfg_t *config)
```
Create the wifi peripheral handle for esp_peripherals.

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

**Return** The esp peripheral handle

**Parameters**

- config: The configuration
esp_err_t esp_wifi_set_listen_interval(esp_periph_handle_t periph, int interval)
Set Wi-Fi listen interval for ESP32 station to receive beacon.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] periph: The wifi periph
• [in] interval: listen interval. units: AP beacon intervals(see BcnInt, default: 100ms)

desp_err_t periph_wifi_config_start(esp_periph_handle_t periph, periph_wifi_config_mode_t mode)
Start Wi-Fi network setup in mode

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] periph: The mode

esp_err_t periph_wifi_config_wait_done(esp_periph_handle_t periph, TickType_t tick_to_wait)
Wait for Wi-Fi setup done.

Return
• ESP_OK

2.8. Peripherals
• ESP_FAIL

Parameters
• [in] periph: The periph
• [in] tick_to_wait: The tick to wait

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

const char *ssid
   SSID of target AP

const char *password
   password of target AP

periph_wpa2_enterprise_cfg_t wpa2_e_cfg
   wpa2 enterprise config

Enumerations

enum periph_wifi_state_t
   Peripheral Wi-Fi event id.
   Values:
   PERIPH_WIFI_UNCHANGE = 0
   PERIPH_WIFI_CONNECTING
   PERIPH_WIFI_CONNECTED
   PERIPH_WIFI_DISCONNECTED
   PERIPH_WIFI_SETTING
   PERIPH_WIFI_CONFIG_DONE
   PERIPH_WIFI_CONFIG_ERROR
   PERIPH_WIFI_ERROR

enum periph_wifi_config_mode_t
   Wi-Fi setup mode type.
   Values:
   WIFI_CONFIG_ESPTOUCH
      Using smartconfig with ESPTOUCH protocol
   WIFI_CONFIG_AIRKISS
      Using smartconfig with AIRKISS protocol
   WIFI_CONFIG_ESPTOUCH_AIRKISS
      Using smartconfig with ESPTOUCH_AIRKISS protocol
   WIFI_CONFIG_WPS
      Using WPS (not support)
   WIFI_CONFIG_BLUEFI
      Using BLUEFI
   WIFI_CONFIG_WEB
      Using HTTP Server (not support)
2.8.3 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_sdcardUnmount()`. The data reading / writing is implemented in a separate API described in *FatFs Stream*.

Application Examples

Implementation of this API is demonstrated in couple of examples:

- player/pipeline_play_sdcard_music
- player/pipeline_sdcard_mp3_control
- recorder/pipeline_recording_to_sdcard
- recorder/pipeline_wav_amr_sdcard

API Reference

Header File

- esp_peripherals/include/periph_sdcard.h

Functions

`esp_periph_handle_t periph_sdcard_init (periph_sdcard_cfg_t *sdcard_config)`

Create the sdcard peripheral handle for esp_peripherals.

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

**Return** The esp peripheral handle

**Parameters**

- `sdcard_config`: The sdcard configuration

`bool periph_sdcard_is_mounted (esp_periph_handle_t periph)`

Check the sdcard is mounted or not.

**Return** SDCARD mounted state

**Parameters**

- **[in]** `periph`: The peripheral
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:

SDCARD_STATUS_UNKNOWN
    No event

SDCARD_STATUS_CARD_DETECT_CHANGE
    Detect changes in the card_detect pin

SDCARD_STATUS_MOUNTED
    SDCARD mounted successfully

SDCARD_STATUS_UNMOUNTED
    SDCARD unmounted successfully

SDCARD_STATUS_MOUNT_ERROR
    SDCARD mount error

SDCARD_STATUS_UNMOUNT_ERROR
    SDCARD unmount error

enum periph_sdcard_mode_t
    SD card mode, SPI, 1-line SD mode, 4-line SD mode.

Values:

SD_MODE_SPI = 0x0
    sd_card SPI

SD_MODE_1_LINE = 0x1
    sd_card 1-line SD mode

SD_MODE_4_LINE = 0x4
    sd_card 4-line SD mode

SD_MODE_8_LINE = 0x8
    sd_card 8-line SD mode

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_spiffsUnmount()`. 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

- esp_peripherals/include/periph_spiffs.h

Functions

`esp_periph_handle_t periph_spiffs_init(periph_spiffs_cfg_t *spiffs_config)`

Create the spiffs peripheral handle for esp_peripherals.

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

**Return** The esp peripheral handle

**Parameters**

- `spiffs_config`: The spiffs configuration

`bool periph_spiffs_is_mounted(esp_periph_handle_t periph)`

Check if the SPIFFS is mounted or not.

**Return** SPIFFS mounted state

**Parameters**

- `[in]` `periph`: The periph

Structures

`struct periph_spiffs_cfg_t`

The SPIFFS Peripheral configuration.
Public Members

const char *root
Base path for vfs

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

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

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

Enumerations

enum periph_spiffs_event_id_t
Peripheral spiffs event id.

Values:

SPIFFS_STATUS_UNKNOWN
No event

SPIFFS_STATUS_MOUNTED
SPIFFS mounted successfully

SPIFFS_STATUS_UNMOUNTED
SPIFFS unmounted successfully

SPIFFS_STATUS_MOUNT_ERROR
SPIFFS mount error

SPIFFS_STATUS_UNMOUNT_ERROR
SPIFFS unmount error

2.8.5 Console Peripheral

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

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

Code example

Please refer to cli/main/console_example.c.
### API Reference

#### Header File

- esp_peripherals/include/periph_console.h

#### Functions

```c
esp_periph_handle_t esp_periph_console_init(periph_console_cfg_t *config)
```

Initialize Console Peripheral.

**Return** The esp peripheral handle

**Parameters**

- config: The configuration

#### Structures

```c
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

```c
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

- esp_peripherals/include/periph_touch.h

Functions

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

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

   Return  The esp peripheral handle

   Parameters

   - config: The configuration
Structures

```c
struct periph_touch_cfg_t

    The Touch peripheral configuration.
```

**Public Members**

```c
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**

```c
enum esp_touch_pad_sel_t

    Touch pad selection.

    Values:

    TOUCH_PAD_SEL0 = BIT(0)
    TOUCH_PAD_SEL1 = BIT(1)
    TOUCH_PAD_SEL2 = BIT(2)
    TOUCH_PAD_SEL3 = BIT(3)
    TOUCH_PAD_SEL4 = BIT(4)
    TOUCH_PAD_SEL5 = BIT(5)
    TOUCH_PAD_SEL6 = BIT(6)
    TOUCH_PAD_SEL7 = BIT(7)
    TOUCH_PAD_SEL8 = BIT(8)
    TOUCH_PAD_SEL9 = BIT(9)
```

```c
enum periph_touch_event_id_t

    Peripheral touch event id.

    Values:

    PERIPH_TOUCH_UNCHANGE = 0
        No event

    PERIPH_TOUCH_TAP
        When touch pad is tapped

    PERIPH_TOUCH_RELEASE
        When touch pad is released after tap

    PERIPH_TOUCH_LONG_TAP
        When touch pad is tapped and held after long_tap_time_ms time
```
PERIPH TOUCH LONG RELEASE
When touch pad is released after long tap

2.8.7 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

- esp_peripherals/include/periph_button.h

Functions

```c
esp_periph_handle_t periph_button_init (periph_button_cfg_t *but_cfg)
```

Create the button peripheral handle for esp_peripherals.

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

**Return** The esp peripheral handle

**Parameters**

- but_cfg: The but configuration

Structures

```c
struct periph_button_cfg_t
```

The Button peripheral configuration.

**Public Members**

```c
uint64_t gpio_mask
```

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

```c
int long_press_time_ms
```

Long press duration in milliseconds, default is 2000ms
Enumerations

```c
enum periph_button_event_id_t
    Peripheral button event id.
    
    Values:
    
    PERIPH_BUTTON_UNCHANGE = 0
        No event
    
    PERIPH_BUTTON_PRESSED
        When button is pressed
    
    PERIPH_BUTTON_RELEASE
        When button is released
    
    PERIPH_BUTTON_LONG_PRESSED
        When button is pressed and kept for more than long_press_time_ms
    
    PERIPH_BUTTON_LONG_RELEASE
        When button is released and event PERIPH_BUTTON_LONG_PRESSED happened
```

2.8.8 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

- esp_peripherals/include/periph_led.h

Functions

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

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

    Return  The esp peripheral handle

    Parameters
        • config: The configuration
```
esp_err_t periph_led_blink(esp_periph_handle_t periph, int gpio_num, int time_on_ms, int time_off_ms, bool fade, int loop, periph_led_idle_level_t level)
Blink LED Peripheral, this function will automatically configure the gpio_num to control the LED, with time_on_ms as the time (in milliseconds) switch from OFF to ON (or ON if fade is disabled), and time_off_ms as the time (in milliseconds) switch from ON to OFF (or OFF if fade is disabled). When switching from ON -> OFF and vice versa, the loop decreases once, and will turn off the effect when the loop is 0. With a loop value less than 0, the LED effect will loop endlessly. PERIPH_LED_BLINK_FINISH events will be sent at each end of loop.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] periph: The LED periph
• [in] gpio_num: The gpio number
• [in] time_on_ms: The time on milliseconds
• [in] time_off_ms: The time off milliseconds
• [in] fade: Fading enabled
• [in] loop: Loop
• [in] level: idle level

esp_err_t periph_led_stop(esp_periph_handle_t periph, int gpio_num)
Stop Blink the LED.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [in] periph: The periph
• [in] gpio_num: The gpio number

Structures

struct periph_led_cfg_t
The LED peripheral configuration.
Public Members

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

ledc_timer_bit_t led_duty_resolution
    LEDC channel duty resolution

ledc_timer_t led_timer_num
    Select the timer source of channel (0 - 3)

uint32_t led_freq_hz
    LEDC timer frequency (Hz)

int gpio_num
    Optional, < 0 invalid gpio number

Enumerations

enum periph_led_event_id_t
    Peripheral LED event id.

Values:

PERIPH_LED_UNCHANGE = 0
    No event

PERIPH_LED_BLINK_FINISH
    When LED blink is finished

enum periph_led_idle_level_t
    Peripheral LED idle output level.

Values:

PERIPH_LED_IDLE_LEVEL_LOW
    Low level output

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 single ADC input to read several sets of buttons. For an example hardware implementation please refer to schematic of ESP32-LyraTD-MSC V2.2 Upper Board (PDF).
Application Examples

Implementation of this API is demonstrated in the following example:

• checks/check_board_buttons

API Reference

Header File

• esp_peripherals/include/periph_adc_button.h

Functions

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

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

Return: The esp peripheral handle.

Parameters

• `btn_cfg`: The button configuration.

Structures

```c
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

```c
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:

PERIPH_ADC_BUTTON_IDOLE = 0
PERIPH_ADC_BUTTON_PRESSED
PERIPH_ADC_BUTTON_RELEASE
PERIPH_ADC_BUTTON_LONG_PRESSED
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

- esp_peripherals/include/periph_is31f3216.h

Functions

esp_periph_handle_t periph_is31f3216_init(periph_is31f3216_cfg_t *is31f3216_config)
Initialize the is31f3216.

Return

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- is31f3216_config:
esp_err_t **periph_is31fl3216_set_state** (esp_periph_handle_t **periph**, periph_is31fl3216_state_t **state**)

Set the state of all the channels.

**Return**

- ESP_OK Success
- ESP_FAIL Fail

**Parameters**

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

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

Set the current enable channels.

**Return**

- ESP_OK Success
- ESP_FAIL Fail

**Parameters**

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

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

Set the duty of the channel.

**Return**

- ESP_OK Success
- ESP_FAIL Fail

**Parameters**

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

esp_err_t **periph_is31fl3216_set_duty_step** (esp_periph_handle_t **periph**, uint8_t **step**)

Set the duty step of flash.

**Return**

- ESP_OK Success
- ESP_FAIL Fail

**Parameters**

- **periph**: The is31fl3216 handle
- **step**: The step of flash
esp_err_t \texttt{periph\_is31fl3216\_set\_interval}(\texttt{esp\_periph\_handle\_t periph}, \texttt{uint16\_t interval\_ms})

Set the interval time.

Return

• ESP_OK Success
• ESP_FAIL Fail

Parameters

• \texttt{periph}: The is31fl3216 handle
• \texttt{interval\_ms}: Time of interval

esp_err_t \texttt{periph\_is31fl3216\_set\_shift\_mode}(\texttt{esp\_periph\_handle\_t periph}, \texttt{periph\_is31\_shift\_mode\_t mode})

Set the shift mode.

Return

• ESP_OK Success
• ESP_FAIL Fail

Parameters

• \texttt{periph}: The is31fl3216 handle
• \texttt{mode}: Mode of \texttt{periph\_is31\_shift\_mode\_t}

esp_err_t \texttt{periph\_is31fl3216\_set\_light\_on\_num}(\texttt{esp\_periph\_handle\_t periph}, \texttt{uint16\_t light\_on\_num}, \texttt{uint16\_t max\_light\_num})

Set the light on numbers.

Return

• ESP_OK Success
• ESP_FAIL Fail

Parameters

• \texttt{periph}: The is31fl3216 handle
• \texttt{light\_on\_num}: Enabled led number
• \texttt{max\_light\_num}: Maximum led number

esp_err_t \texttt{periph\_is31fl3216\_set\_act\_time}(\texttt{esp\_periph\_handle\_t periph}, \texttt{uint16\_t act\_ms})

Set the action time.

Return

• ESP_OK Success
• ESP_FAIL Fail

Parameters

• \texttt{periph}: The is31fl3216 handle
• \texttt{act\_ms}: Action time, unit is millisecond, 0 is infinite
Structures

```c
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

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

```c
enum periph_is31_shift_mode_t
    Values:
    PERIPH_IS31_SHIFT_MODE_UNKNOWN
    PERIPH_IS31_SHIFT_MODE_ACC
        accumulation mode
    PERIPH_IS31_SHIFT_MODE_SINGLE
```
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. 6: Ring Buffer used in Audio Pipeline](image)
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

- `audio_pipeline/include/ringbuf.h`

Functions

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

Return `ringbuf_handle_t`

Parameters

- `[in] block_size`: Size of each block
- `[in] n_blocks`: Number of blocks

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `[in] rb`: The Ringbuffer handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `[in] rb`: The Ringbuffer handle

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

Return

- ESP_OK
- ESP_FAIL
Parameters

- [in] rb: The Ringbuffer handle

```c
esp_err_t rb_reset_is_done_write (ringbuf_handle_t rb)
```
Reset is_done_write flag.

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] rb: The Ringbuffer handle

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

Return total bytes available

Parameters

- [in] rb: The Ringbuffer handle

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

Return The number of bytes that have filled the ringbuffer

Parameters

- [in] rb: The Ringbuffer handle

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

Return total size of Ringbuffer

Parameters

- [in] rb: The Ringbuffer handle

```c
int rb_read (ringbuf_handle_t rb, char *buf, int len, TickType_t ticks_to_wait)
```
Read from Ringbuffer to buf with len and wait tick_to_wait ticks until enough bytes to read if the ringbuffer bytes available is less than len. If buf argument provided is NULL, then ringbuffer do pseudo reads by simply advancing pointers.

Return Number of bytes read

Parameters

- [in] rb: The Ringbuffer handle
- buf: The buffer pointer to read out data
- [in] len: The length request
- [in] ticks_to_wait: The ticks to wait
int rb_write(ringbuf_handle_t rb, char *buf, int len, TickType_t ticks_to_wait)
  Write to Ringbuffer from buf with len and wait tick_to_wait ticks until enough space to write if the
  ringbuffer space available is less than len

  Return Number of bytes written

  Parameters
      • [in] rb: The Ringbuffer handle
      • buf: The buffer
      • [in] len: The length
      • [in] ticks_to_wait: The ticks to wait

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

  Return
      • ESP_OK
      • ESP_FAIL

  Parameters
      • [in] rb: The Ringbuffer handle

esp_err_t rb_unblock_reader(ringbuf_handle_t rb)
  Unblock from rb_read.

  Return
      • ESP_OK
      • ESP_FAIL

  Parameters
      • [in] rb: The Ringbuffer handle

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

- audio_hal/include/audio_hal.h

Functions

audio_hal_handle_t audio_hal_init (audio_hal_codec_config_t *audio_hal_conf, audio_hal_func_t *audio_hal_func)

Initialize media codec driver.

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

**Return** int, 0 success, others fail

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

esp_err_t audio_hal_deinit (audio_hal_handle_t audio_hal)

Uninitialize media codec driver.

**Return** int, 0 success, others fail

**Parameters**
- audio_hal: reference function pointer for selected audio codec

esp_err_t audio_hal_ctrl_codec (audio_hal_handle_t audio_hal, audio_hal_codec_mode_t mode, audio_hal_ctrl_t audio_hal_ctrl)

Start/stop codec driver.

**Return** int, 0 success, others fail

**Parameters**
- audio_hal: reference function pointer for selected audio codec
• **mode**: select media hal codec mode either encode/decode/or both to start from audio_hal_codec_mode_t

• **audio_hal_ctrl**: select start stop state for specific mode

```c
esp_err_t audio_hal_codec_iface_config(audio_hal_handle_t audio_hal, audio_hal_codec_mode_t mode, audio_hal_codec_i2s_iface_t *iface)
```

Set codec I2S interface samples rate & bit width and format either I2S or PCM/DSP.

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

**Parameters**
- **audio_hal**: reference function pointer for selected audio codec
- **mode**: select media hal codec mode either encode/decode/or both to start from audio_hal_codec_mode_t
- **iface**: I2S sample rate (ex: 16000, 44100), I2S bit width (16, 24, 32), I2s format (I2S, PCM, DSP).

```c
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.

**Return** int, 0 success, others fail

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

```c
esp_err_t audio_hal_set_volume(audio_hal_handle_t audio_hal, int volume)
```

Set voice volume.

**Note** if volume is 0, mute is enabled, range is 0-100.

**Return** int, 0 success, others fail

**Parameters**
- **audio_hal**: reference function pointer for selected audio codec
- **volume**: value of volume in percent(

```c
esp_err_t audio_hal_get_volume(audio_hal_handle_t audio_hal, int *volume)
```

Get voice volume.

**Note** if volume is 0, mute is enabled, range is 0-100.

**Return** int, 0 success, others fail

**Parameters**
- **audio_hal**: reference function pointer for selected audio codec
- **volume**: value of volume in percent returned(%)
esp_err_t audio_hal_enable_pa(audio_hal_handle_t audio_hal, bool enable)

Enables or disables PA.

Return int, 0 success, others fail

Parameters

• audio_hal: reference function pointer for selected audio codec
• enable: true/false.

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

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

```c
esp_err_t (*audio_codec_deinitialize)(void)
deinitialize codec
```

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

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

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

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

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

```c
esp_err_t (*audio_codec_enable_pa)(bool enable)
enable pa
```

```c
xSemaphoreHandle audio_hal_lock
semaphore of codec
```

```c
void *handle
handle of audio codec
```

Macros

```c
#define AUDIO_HAL_VOL_DEFAULT
```

Type Definitions

```c
typedef struct audio_hal *audio_hal_handle_t
```

```c
typedef struct audio_hal audio_hal_func_t
```
Configuration of functions and variables used to operate audio codec chip.

Enumerations

```c
typedef enum audio_hal_codec_mode_t
```
Select media hal codec mode.

```c
Values:
```

```c
# define AUDIO_HAL_CODEC_MODE_ENCODE = 1
select adc
```

```c
# define AUDIO_HAL_CODEC_MODE_DECODE
select dac
```
 AUDIO_HAL_CODEC_MODE_BOTH
  select both adc and dac
 AUDIO_HAL_CODEC_MODE_LINE_IN
  set adc channel

enum audio_hal_adc_input_t
  Select adc channel for input mic signal.
  Values:
  AUDIO_HAL_ADC_INPUT_LINE1 = 0x00
      mic input to adc channel 1
  AUDIO_HAL_ADC_INPUT_LINE2
      mic input to adc channel 2
  AUDIO_HAL_ADC_INPUT_ALL
      mic input to both channels of adc
  AUDIO_HAL_ADC_INPUT_DIFFERENCE
      mic input to adc difference channel

enum audio_hal_dac_output_t
  Select channel for dac output.
  Values:
  AUDIO_HAL_DAC_OUTPUT_LINE1 = 0x00
      dac output signal to channel 1
  AUDIO_HAL_DAC_OUTPUT_LINE2
      dac output signal to channel 2
  AUDIO_HAL_DAC_OUTPUT_ALL
      dac output signal to both channels

enum audio_hal_ctrl_t
  Select operating mode i.e. start or stop for audio codec chip.
  Values:
  AUDIO_HAL_CTRL_STOP = 0x00
      set stop mode
  AUDIO_HAL_CTRL_START = 0x01
      set start mode

enum audio_hal_iface_mode_t
  Select I2S interface operating mode i.e. master or slave for audio codec chip.
  Values:
  AUDIO_HAL_MODE_SLAVE = 0x00
      set slave mode
  AUDIO_HAL_MODE_MASTER = 0x01
      set master mode

enum audio_hal_iface_samples_t
  Select I2S interface samples per second.
  Values:
\textbf{audio_hal_iface_bits_t}  
Select I2S interface number of bits per sample.  
\textit{Values:}

\begin{verbatim}
 AUDIO_HAL_BIT_LENGTH_16BITS = 1  
    set 16 bits per sample
 AUDIO_HAL_BIT_LENGTH_24BITS  
    set 24 bits per sample
 AUDIO_HAL_BIT_LENGTH_32BITS  
    set 32 bits per sample
\end{verbatim}

\textbf{audio_hal_iface_format_t}  
Select I2S interface format for audio codec chip.  
\textit{Values:}

\begin{verbatim}
 AUDIO_HAL_I2S_NORMAL = 0  
    set normal I2S format
 AUDIO_HAL_I2S_LEFT  
    set all left format
 AUDIO_HAL_I2S_RIGHT  
    set all right format
 AUDIO_HAL_I2S_DSP  
    set dsp/pcm format
\end{verbatim}
2.9.3 ES8388 Driver

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

API Reference

Header File

- audio_hal/driver/es8388/es8388.h

Functions

```c
esp_err_t es8388_init (audio_hal_codec_config_t *cfg)

Initialize ES8388 codec chip.
```

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**

- `cfg`: configuration of ES8388

```c
esp_err_t es8388_deinit (void)

Deinitialize ES8388 codec chip.
```

**Return**

- ESP_OK
- ESP_FAIL

```c
esp_err_t es8388_config_fmt (es_module_t mod, es_i2s_fmt_t cfg)

Configure ES8388 I2S format.
```

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**

- `mod`: set ADC or DAC or both
- `cfg`: ES8388 I2S format

```c
esp_err_t es8388_i2s_config_clock (es_i2s_clock_t cfg)

Configure I2s clock in MSATER mode.
```

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**
- cfg: set bits clock and WS clock

```c
esp_err_t es8388_set_bits_per_sample(es_module_t mode, es_bits_length_t bit_per_sample)
Configure ES8388 data sample bits.
```

**Return**
- ESP_OK
- ESP_FAIL

**Parameters**
- mode: set ADC or DAC or both
- bit_per_sample: bit number of per sample

```c
esp_err_t es8388_start(es_module_t mode)
Start ES8388 codec chip.
```

**Return**
- ESP_OK
- ESP_FAIL

**Parameters**
- mode: set ADC or DAC or both

```c
esp_err_t es8388_stop(es_module_t mode)
Stop ES8388 codec chip.
```

**Return**
- ESP_OK
- ESP_FAIL

**Parameters**
- mode: set ADC or DAC or both

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

**Return**
- ESP_OK
- ESP_FAIL

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

```c
esp_err_t es8388_get_voice_volume(int *volume)
Get voice volume.
```

**Return**
- ESP_OK
• ESP_FAIL

Parameters
• [out] *volume: voice volume (0~100)

esp_err_t **es8388_set_voice_mute**(bool enable)
Configure ES8388 DAC mute or not. Basically you can use this function to mute the output or unmute.

Return
• ESP_FAIL Parameter error
• ESP_OK Success

Parameters
• enable: enable(1) or disable(0)

esp_err_t **es8388_get_voice_mute**(void)
Get ES8388 DAC mute status.

Return
• ESP_FAIL Parameter error
• ESP_OK Success

esp_err_t **es8388_set_mic_gain**(es_mic_gain_t gain)
Set ES8388 mic gain.

Return
• ESP_FAIL Parameter error
• ESP_OK Success

Parameters
• gain: db of mic gain

esp_err_t **es8388_config_adc_input**(es_adc_input_t input)
Set ES8388 adc input mode.

Return
• ESP_FAIL Parameter error
• ESP_OK Success

Parameters
• input: adc input mode

esp_err_t **es8388_config_dac_output**(es_dac_output_t output)
Set ES8388 dac output mode.

Return
• ESP_FAIL Parameter error
• ESP_OK Success
Parameters

- **output**: dac output mode

```c
esp_err_t es8388_write_reg(uint8_t reg_add, uint8_t data)
```
Write ES8388 register.

**Return**

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- **reg_add**: address of register
- **data**: data of register

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

**Return**

- void

```c
esp_err_t es8388_config_i2s(audio_hal_codec_mode_t mode, audio_hal_codec_i2s_iface_t *iface)
```
Configure ES8388 codec mode and I2S interface.

**Return**

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- **mode**: codec mode
- **iface**: I2S config

```c
esp_err_t es8388_ctrl_state(audio_hal_codec_mode_t mode, audio_hal_ctrl_t ctrl_state)
```
Control ES8388 codec chip.

**Return**

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- **mode**: codec mode
- **ctrl_state**: start or stop decode or encode progress

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

**Return**

- ESP_ERR_INVALID_ARG
- ESP_OK

**Parameters**

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

**Macros**

```c
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

- audio_hal/driver/es8374/es8374.h
Functions

```c
esp_err_t esp374_codec_init(audio_hal_codec_config_t *cfg)
Initialize ES8374 codec chip.
```

Return

- ESP_OK
- ESP_FAIL

Parameters

- cfg: configuration of ES8374

```c
esp_err_t esp374_codec_deinit(void)
Deinitialize ES8374 codec chip.
```

Return

- ESP_OK
- ESP_FAIL

```c
esp_err_t esp374_config_fmt(es_module_t mode, es_i2s_fmt_t fmt)
Configure ES8374 I2S format.
```

Return

- ESP_OK
- ESP_FAIL

Parameters

- mode: set ADC or DAC or both
- fmt: ES8374 I2S format

```c
esp_err_t esp374_i2s_config_clock(es_i2s_clock_t cfg)
Configure I2S clock in MSATER mode.
```

Return

- ESP_OK
- ESP_FAIL

Parameters

- cfg: set bits clock and WS clock

```c
esp_err_t esp374_set_bits_per_sample(es_module_t mode, es_bits_length_t bit_per_sample)
Configure ES8374 data sample bits.
```

Return

- ESP_OK
- ESP_FAIL

Parameters
Read the Docs Template Documentation

- **mode**: set ADC or DAC or both
- **bit_per_sample**: bit number of per sample

```c
esp_err_t es8374_start(es_module_t mode)
    Start ES8374 codec chip.

Return
- ESP_OK
- ESP_FAIL

Parameters
- **mode**: set ADC or DAC or both

esp_err_t es8374_stop(es_module_t mode)
Stop ES8374 codec chip.

Return
- ESP_OK
- ESP_FAIL

Parameters
- **mode**: set ADC or DAC or both

esp_err_t es8374_codec_set_voice_volume(int volume)
Set voice volume.

Return
- ESP_OK
- ESP_FAIL

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

esp_err_t es8374_codec_get_voice_volume(int *volume)
Get voice volume.

Return
- ESP_OK
- ESP_FAIL

Parameters
- [out] *volume: voice volume (0~100)

esp_err_t es8374_set_voice_mute(bool enable)
Mute or unmute ES8374 DAC. Basically you can use this function to mute or unmute the output.

Return
- ESP_FAIL Parameter error

2.9. Abstraction Layer
• ESP_OK Success

Parameters
• enable: mute(1) or unmute(0)

esp_err_t es8374_get_voice_mute (void)
Get ES8374 DAC mute status.

Return
• ESP_FAIL
• ESP_OK

esp_err_t es8374_set_mic_gain (es_mic_gain_t gain)
Set ES8374 mic gain.

Return
• ESP_FAIL Parameter error
• ESP_OK Success

Parameters
• gain: db of mic gain

esp_err_t es8374_config_adc_input (es_adc_input_t input)
Set ES8374 ADC input mode.

Return
• ESP_FAIL Parameter error
• ESP_OK Success

Parameters
• input: adc input mode

esp_err_t es8374_config_dac_output (es_dac_output_t output)
Set ES8374 DAC output mode.

Return
• ESP_FAIL Parameter error
• ESP_OK Success

Parameters
• output: dac output mode

esp_err_t es8374_write_reg (uint8_t reg_add, uint8_t data)
Write ES8374 register.

Return
• ESP_FAIL Parameter error
• ESP_OK Success
Parameters

- `reg_add`: address of register
- `data`: data of register

```c
void es8374_read_all()
{
    Print all ES8374 registers.
}
```

Return

- `void`

```c
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.
}
```

Return

- `ESP_FAIL` Parameter error
- `ESP_OK` Success

Parameters

- `mode`: codec mode
- `iface`: I2S config

```c
esp_err_t es8374_codec_ctrl_state(audio_hal_codec_mode_t mode, audio_hal_ctrl_t ctrl_state)
{
    Control ES8374 codec chip.
}
```

Return

- `ESP_FAIL` Parameter error
- `ESP_OK` Success

Parameters

- `mode`: codec mode
- `ctrl_state`: start or stop decode or encode progress

```c
esp_err_t es8374_pa_power(bool enable)
{
    Set ES8374 PA power.
}
```

Return

- `ESP_ERR_INVALID_ARG`
- `ESP_OK`

Parameters

- `enable`: true for enable PA power, false for disable PA power
2.9.5 ZL38063 Driver

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

API Reference

Header File

- audio_hal/driver/zl38063/zl38063.h

Functions

`esp_err_t zl38063_codec_init (audio_hal_codec_config_t *cfg)`

Initialize ZL38063 chip.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `cfg`: configuration of ZL38063

`esp_err_t zl38063_codec_deinit (void)`

Deinitialize ZL38063 chip.

Return

- ESP_OK
- ESP_FAIL

`esp_err_t zl38063_codec_ctrl_state (audio_hal_codec_mode_t mode, audio_hal_ctrl_t ctrl_state)`

Control ZL38063 chip.

The functions zl38063_ctrl_state and zl38063_config_i2s are not used by this driver. They are kept here to maintain the uniformity and convenience of the interface of the ADF project. These settings for zl38063 are burned in firmware and configuration files. Default i2s configuration: 48000Hz, 16bit, Left-Right channels. Use resampling to be compatible with different file types.

Return

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- `mode`: codec mode
- `ctrl_state`: start or stop decode or encode progress
esp_err_t zl38063_codec_config_i2s(audio_hal_codec_mode_t mode, audio_hal_codec_i2s_iface_t *iface)
Configure ZL38063 codec mode and I2S interface.

Return
• ESP_FAIL Parameter error
• ESP_OK Success

Parameters
• mode: codec mode
• iface: I2S config

esp_err_t zl38063_codec_set_voice_mute(bool mute)
mute or unmute the codec

Return
• ESP_OK
• ESP_FAIL

Parameters
• mute: true, false

esp_err_t zl38063_codec_set_voice_volume(int volume)
Set voice volume.

Return
• ESP_OK
• ESP_FAIL

Parameters
• volume: voice volume (0~100)

esp_err_t zl38063_codec_get_voice_volume(int *volume)
Get voice volume.

Return
• ESP_OK
• ESP_FAIL

Parameters
• [out] *volume: voice volume (0~100)
2.10 Configuration Options

Compile-time configuration options specific to ESP-ADF.

2.10.1 Audio Recorder

**AFE_MIC_NUM**

Number of microphones used for AFE

*Found in: Audio Recorder*

Now only support 1 or 2 mic

2.10.2 Audio Codec Device Configuration

**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_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.

2.10.3 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_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

2.10.4 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.

2.10.5 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

RECORD_ENGINE_MODE

Choose recorder engine functionality

Found in: ADF Library Configuration

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

Available options:

- REC_ENG_ENABLE_VAD_ONLY
- REC_ENG_ENABLE_VAD_WWE
- REC_ENG_ENABLE_VAD_WWE_AMR
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 / decode) 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 quality audio on the output is required.
- **Typical** - with an external codec chip and a power amplifier, for high quality output audio and multiple input / output options.

There may be several variation between the above projects, by adding or removing features/components. Below are 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](image)

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

When looking for better audio quality and more interfacing options we would use an external I2S codec to do all the analog input and output signal processing. The codec chip, depending on type, may provide additional functionality like audio input signal preamplifier, headphone output amplifier, multiple analog input and outputs, sound effects, etc. The I2S is considered as the industry standard for interfacing with audio codec chips, or in general for a high speed, continuous transfer of the audio data. To optimize performance of audio data processing, 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 documentation for details, especially pay attention to its Restrictions section which is very important.

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

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

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

- If PSRAM is in use, set all the static buffer to minimum value in Component config > Wi-Fi; if PSRAM is not used then dynamic buffer should be selected to save memory. Refer to Wi-Fi Buffer Usage section in IDF documentation for details.

- If PSRAM and BT are used, then `CONFIG_BT_ALLOCATION_FROM_SPIRAM_FIRST` and `CONFIG_BT_BLE_DYNAMIC_ENV_MEMORY` should be set as “yes” under Component config > Bluetooth > Bluedroid Enable, to allocate more of 40kB memory to PSRAM

- If PSRAM and Wi-Fi are used, then `CONFIG_WIFI_LWIP_ALLOCATION_FROM_SPIRAM_FIRST` should be set as “yes” under Component config > ESP32-specific > SPI RAM config, to allocate some memory to PSRAM

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

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

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

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

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

**Important:** Don’t use ROM code in `xTaskCreateStatic` task The ROM code itself is linked in components/esp32/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.
## 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”

<table>
<thead>
<tr>
<th>Component</th>
<th>Internal RAM Required</th>
<th>With PSRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wi-Fi</td>
<td>50kB+</td>
<td>50kB+</td>
</tr>
<tr>
<td>Bluetooth</td>
<td>140kB (50kB if only BLE needed)</td>
<td>95kB (50kB if only BLE needed)</td>
</tr>
<tr>
<td>Flash Card</td>
<td>12kB+</td>
<td>12kB+</td>
</tr>
<tr>
<td>I2S</td>
<td>Configurable, 8kB for reference</td>
<td>Configurable, 8kB for reference</td>
</tr>
<tr>
<td>RingBuffer</td>
<td>Configurable, 30kB for reference</td>
<td>0kB, all moved into PSRAM</td>
</tr>
</tbody>
</table>
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

```c
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 = {
    .ssid = CONFIG_WIFI_SSID,
    .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.
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.

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

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;
- **Setup Path to ESP-ADF** to make the framework aware of the audio specific API;
- **Start a Project** that will provide a sample audio application for the 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
- Functional Description
  - Allocation of ESP32 Pins to Test Points
    - JTAG Test Point
    - UART Test Point
- MicroSD Card
- GPIO Allocation Summary
- Notes on Power Distribution
  - Power Supply over USB and from Battery
  - Independent Audio and Digital Power Supply
- Selecting of the Audio Output
- Related Documents
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](image)

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.

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.

Fig. 7: ESP32 LyraT-Mini V1.2 Board Layout
USB Power Port  Provides the power supply for the board.

Standby/Charging LEDs  The Standby green LED indicates that power has been applied to the 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**

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

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>Pin Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RXD0 RX</td>
</tr>
<tr>
<td>2</td>
<td>TXD0 TX</td>
</tr>
<tr>
<td>3</td>
<td>GND GND</td>
</tr>
<tr>
<td>4</td>
<td>n/a 3.3 V</td>
</tr>
</tbody>
</table>

MicroSD Card

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

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>MicroSD Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MTDI / GPIO12</td>
</tr>
<tr>
<td>2</td>
<td>MTCK / GPIO13</td>
</tr>
<tr>
<td>3</td>
<td>MTD0 / GPIO15</td>
</tr>
<tr>
<td>4</td>
<td>MTMS / GPIO14</td>
</tr>
<tr>
<td>5</td>
<td>GPIO2 DATA0</td>
</tr>
<tr>
<td>6</td>
<td>GPIO4 –</td>
</tr>
<tr>
<td>7</td>
<td>GPIO34 CD</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Pin Name</th>
<th>ES8311</th>
<th>ES7243</th>
<th>Keys</th>
<th>MicroSD</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>EN</td>
<td></td>
<td></td>
<td>EN_KEY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S_VP</td>
<td></td>
<td></td>
<td>I2S_DATA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S_VN</td>
<td></td>
<td></td>
<td>REC, MODE, PLAY, SET, VOL-, VOL+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>IO34</td>
<td></td>
<td></td>
<td></td>
<td>CD</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>IO35</td>
<td></td>
<td></td>
<td>I2S0_ASDOUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>IO32</td>
<td></td>
<td></td>
<td>I2S1_SCLK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>IO33</td>
<td></td>
<td></td>
<td>I2S1_LRCK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>IO25</td>
<td></td>
<td></td>
<td>I2S0_LRCK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>IO26</td>
<td></td>
<td></td>
<td>I2S0_DSDIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>IO27</td>
<td></td>
<td></td>
<td></td>
<td>Blue_LED</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>IO14</td>
<td></td>
<td></td>
<td></td>
<td>CLK</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>IO12</td>
<td></td>
<td></td>
<td></td>
<td>NC (DATA2)</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>IO13</td>
<td></td>
<td></td>
<td></td>
<td>NC (DATA3)</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>SD2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>SD3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>CMD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>CLK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>SD0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>SD1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Pin</th>
<th>Pin Name</th>
<th>ES8311</th>
<th>ES7243</th>
<th>Keys</th>
<th>MicroSD</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>IO15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>IO2</td>
<td>CO2_KEY</td>
<td></td>
<td></td>
<td>CMD</td>
<td>DATA0</td>
</tr>
<tr>
<td>25</td>
<td>IO0</td>
<td>I2S0_MCLK</td>
<td>I2S1_MCLK</td>
<td></td>
<td>IO0_KEY</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>IO4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NC (DATA1)</td>
</tr>
<tr>
<td>27</td>
<td>NC (IO16)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>NC (IO17)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>IO5</td>
<td>I2S0_SCLK</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>IO18</td>
<td></td>
<td>I2C_SDA</td>
<td>I2C_SDA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>IO19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PA_CTRL 2</td>
</tr>
<tr>
<td>33</td>
<td>IO21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PA_CTRL 3</td>
</tr>
<tr>
<td>34</td>
<td>RXD0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RXD0 4</td>
</tr>
<tr>
<td>35</td>
<td>TXD0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TXD0 4</td>
</tr>
<tr>
<td>36</td>
<td>IO22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Green_LED</td>
</tr>
<tr>
<td>37</td>
<td>IO23</td>
<td>I2C_SCK</td>
<td>I2C_SCL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

3.4. Development Boards
Power System:

USB<>UART:

Fig. 8: ESP32-LyraT-Mini V1.2 - Dedicated USB Power Supply Socket
3.4. Development Boards

Fig. 9: ESP32-LyraT-Mini V1.2 - Power Supply from a Battery

Fig. 10: ESP32-LyraT-Mini V1.2 - Digital Power Supply
Two digital IO signals are provided to implement selection between the speaker and the headphones:

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

The application running on ESP32 may then enable 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;
- **Setup Path to ESP-ADF** to make the framework aware of the audio specific API;
- **Start a Project** that will provide a sample audio application for the 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**
- **Functional Description**
  - Hardware Setup Options
    - Enable MicroSD Card in 1-wire Mode
    - Enable MicroSD Card in 4-wire Mode
    - Enable JTAG
    - Using Automatic Upload
  - Allocation of ESP32 Pins
  - Pinout of Extension Headers
    - UART Header / JP2
    - I2S Header / JP4
    - I2C Header / JP5
    - JTAG Header / JP7
  - Notes of Power Distribution
    - Power Supply Separation
    - Three Dedicated LDOs
    - Separate Power Feed for the PAs
  - Selecting of the Audio Output
- **Other Versions of LyraT**
- **Related Documents**

Overview

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

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

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

**ESP32-WROVER-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.

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

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

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

In this mode:
- **JTAG** functionality is not available
- **Vol**- touch button is available for use with the API

Enable MicroSD Card in 4-wire Mode

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

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

Enable JTAG

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

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

**Using Automatic Upload**

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

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

**Allocation of ESP32 Pins**

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

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

- (TP) - touch pad
- (PB) - push button
Pinout of Extension Headers

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

**UART Header / JP2**

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

**I2S Header / JP4**

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

**I2C Header / JP5**

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

**JTAG Header / JP7**

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>JTAG Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MTDI / GPIO12</td>
</tr>
<tr>
<td>2</td>
<td>MTCK / GPIO13</td>
</tr>
<tr>
<td>3</td>
<td>MTDI / GPIO12</td>
</tr>
<tr>
<td>4</td>
<td>MTMS / GPIO14</td>
</tr>
</tbody>
</table>
Notes of Power Distribution

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

Power Supply Separation

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

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.

MicroSD Card and Audio Codec

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

SDIO Power Supply:
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](image)

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 21: ESP32-LyraT V4.2 Board Layout
**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**

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

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

In this mode:

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

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

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

In this mode:

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

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

In this mode:

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

Allocation of ESP32 Pins

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

Red / Green LEDs

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>LED Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 GPIO19</td>
<td>Red LED</td>
</tr>
<tr>
<td>2 GPIO22</td>
<td>Green LED</td>
</tr>
</tbody>
</table>

Touch Pads

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

1. **Vol-** function is not available if JTAG is used. It is also not available for the MicroSD Card configured to operate in 4-wire mode.
MicroSD Card / J5

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>MicroSD Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MTDI/GPIO12</td>
<td>DATA2</td>
</tr>
<tr>
<td>2 MTCK/GPIO13</td>
<td>CD/DATA3</td>
</tr>
<tr>
<td>3 MTD0/GPIO15</td>
<td>CMD</td>
</tr>
<tr>
<td>4 MTMS/GPIO14</td>
<td>CLK</td>
</tr>
<tr>
<td>5 GPIO2</td>
<td>DATA0</td>
</tr>
<tr>
<td>6 GPIO4</td>
<td>DATA1</td>
</tr>
<tr>
<td>7 GPIO21</td>
<td>CD</td>
</tr>
</tbody>
</table>

UART Header / JP2

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

EN and IO0 Headers / JP23 and J24

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

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

I2S Header / JP4

<table>
<thead>
<tr>
<th>I2C Header Pin</th>
<th>ESP32 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MCLK</td>
<td>GPIO0</td>
</tr>
<tr>
<td>2 SCLK</td>
<td>GPIO5</td>
</tr>
<tr>
<td>1 LRCK</td>
<td>GPIO25</td>
</tr>
<tr>
<td>2 DSDIN</td>
<td>GPIO26</td>
</tr>
<tr>
<td>3 ASDOUT</td>
<td>GPIO35</td>
</tr>
<tr>
<td>3 GND</td>
<td>GND</td>
</tr>
</tbody>
</table>
I2C Header / JP5

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

JTAG Header / JP7

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

Function DIP Switch / JP8

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

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

Start Application Development

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

Initial Setup

Prepare the board for loading of the first sample application:

1. Install jumpers on **IO0** and **EN** headers to enable automatic application upload. If there are no jumpers then upload may be triggered using **Boot / RST** buttons.

2. Connect 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;
- **Setup Path to ESP-ADF** to make the framework aware of the audio specific API;
- **Start a Project** that will provide a sample audio application for the 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)
- ESP32-WROVER 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

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

![ESP32-LyraT block diagram](image)

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**

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

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

In this mode:

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

---

3.4. Development Boards  295
Enable MicroSD Card in 4-wire Mode

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

In this mode:

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

Enable JTAG

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

In this mode:

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

Allocation of ESP32 Pins

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

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

Touch Pads

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

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

MicroSD Card / J5

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

UART Header / JP2

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

I2S Header / JP4

<table>
<thead>
<tr>
<th>I2C Header Pin</th>
<th>ESP32 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCLK</td>
<td>GPIO0</td>
</tr>
<tr>
<td>SCLK</td>
<td>GPIO5</td>
</tr>
<tr>
<td>LRCK</td>
<td>GPIO25</td>
</tr>
<tr>
<td>DSDIN</td>
<td>GPIO26</td>
</tr>
<tr>
<td>ASDOUT</td>
<td>GPIO35</td>
</tr>
<tr>
<td>GND</td>
<td>GND</td>
</tr>
</tbody>
</table>
I2C Header / JP5

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

JTAG Header / JP7

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

Function DIP Switch / JP8

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

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

Start Application Development

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

Initial Setup

Prepare the board for loading of the first sample application:

1. Connect 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;
- **Setup Path to ESP-ADF** to make the framework aware of the audio specific API;
- **Start a Project** that will provide a sample audio application for the 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)
- ESP32-WROVER 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*. 

---

3.4. Development Boards 299
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.
3.4. Development Boards

Fig. 25: ESP32-LyraTD-MSC Block Diagram

Legend
PB = Push Button  
PA = Power Amplifier

ESP32-LyraTD-MSC V2.2
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.
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;
- Setup Path to ESP-ADF to make the framework aware of the audio specific API;
- Start a Project that will provide a sample audio application for the 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.

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 documentation.
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 a 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.
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.
<table>
<thead>
<tr>
<th>Key Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP32-DU1906</td>
<td>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.</td>
</tr>
<tr>
<td>DIP Connector for SPI LCD</td>
<td>ESP32-Korvo-DU1906 has a 2.54 mm pitch connector to connect SPI LCD.</td>
</tr>
<tr>
<td>Audio ADC (Audio Analog-to-Digital Converter)</td>
<td>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.</td>
</tr>
<tr>
<td>Line-in/out Connector (Earphone Jacks)</td>
<td>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.</td>
</tr>
<tr>
<td>Speaker Connector</td>
<td>Output sockets to connect two 4-ohm speakers to provide stereo sound via digital Audio PA.</td>
</tr>
<tr>
<td>Audio DAC (Audio Digital-to-Analog Converter)</td>
<td>ES7148 Stereo DAC is able to convert digital signals into analog audio outputs.</td>
</tr>
<tr>
<td>Audio PA (Digital Audio Power Amplifier)</td>
<td>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.</td>
</tr>
<tr>
<td>Battery Connector</td>
<td>Connect a battery.</td>
</tr>
<tr>
<td>Battery Charger</td>
<td>AP5056, a constant-current/constant-voltage linear power management chip, can be used for charging management to a single lithium-ion battery.</td>
</tr>
<tr>
<td>PWR Slide Switch</td>
<td>Power switch for the board, turn on/off the power supply.</td>
</tr>
<tr>
<td>USB to UART</td>
<td>CP2102N supports USB-to-UART conversion for easy download and debugging of software.</td>
</tr>
<tr>
<td>DBG USB (Debugging USB)</td>
<td>USB communication between PC and ESP32-DU1906 module.</td>
</tr>
<tr>
<td>PWR USB (Power supply USB)</td>
<td>Provide power supply for the whole system. It is recommended to be connected to an at least 5 V / 2 A power adapter for sufficient current supply.</td>
</tr>
<tr>
<td>Charging LEDs</td>
<td>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 green (indicating the charging is completed). If there is no battery connected, the BAT_CHRG and BAT_STBY LEDs will be red and green respectively by default.</td>
</tr>
</tbody>
</table>
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;
- Setup Path to ESP-ADF to make the framework aware of the audio specific API;
- Start a Project that will provide a sample audio application for the 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
The key components of the board are described in a clockwise direction.

Fig. 32: ESP32-S3-Korvo-2 V3.0 (click to enlarge)
### Key Component

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ESP32-S3-WROOM-1 Module</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Left Microphone</strong></td>
<td>Onboard microphone connected to ADC.</td>
</tr>
<tr>
<td><strong>Audio ADC Chip</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Audio Codec Chip</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Audio PA Chip</strong></td>
<td>NS4150 is an EMI, 3 W mono Class D audio power amplifier, amplifying audio signals from audio codec chips to drive speakers.</td>
</tr>
<tr>
<td><strong>Right Microphone</strong></td>
<td>Onboard microphone connected to ADC.</td>
</tr>
<tr>
<td><strong>Speaker Output Port</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>USB-to-UART Bridge Chip</strong></td>
<td>A single chip USB-UART bridge CP2102N provides up to 3 Mbps transfer rates for software download and debugging.</td>
</tr>
<tr>
<td><strong>USB-to-UART Port</strong></td>
<td>Functions as the communication interface between a PC and the ESP32-S3-WROOM-1 module.</td>
</tr>
<tr>
<td><strong>USB Power Port</strong></td>
<td>Provides power to the board. It is recommended to use at least 5V/2A power adapter to ensure a stable power supply.</td>
</tr>
<tr>
<td><strong>Battery Socket</strong></td>
<td>Two pins socket to connect a single cell Li-ion battery.</td>
</tr>
<tr>
<td><strong>Power Switch</strong></td>
<td>Power on/off knob: toggling it down powers the board on; toggling it up powers the board off.</td>
</tr>
<tr>
<td><strong>Battery Charger</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Function Press Keys</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Boot/Reset Press Keys</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>MicroSD Slot</strong></td>
<td>The development board supports a microSD card in 1-bit mode, and can store or play audio files in the microSD card.</td>
</tr>
<tr>
<td><strong>LCD Connector</strong></td>
<td>A FPC connector with 0.5 mm pitch to connect to the LCD extension board.</td>
</tr>
<tr>
<td><strong>System LEDs</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Camera Connector</strong></td>
<td>An external camera module that can be connected to the development board with the connector to transmit images.</td>
</tr>
</tbody>
</table>
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
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. 35: ESP32-S3-Korvo-2 V3.0 - Power Supply from a Battery

Fig. 36: ESP32-S3-Korvo-2 V3.0 - Power Supply Options

Fig. 37: ESP32-S3-Korvo-2 V3.0 - Digital Power Supply
**Power for Audio:**

![Power for Audio Diagram](image)

*Notes: 1.Vout=1.20*(1+R1/R2)=3.296Vz
R1=52.3K,R2=30.1K are recommended for better performance.*

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

<table>
<thead>
<tr>
<th>Pin†</th>
<th>Pin Name</th>
<th>ES8311</th>
<th>ES7210</th>
<th>Camera</th>
<th>LCD</th>
<th>Keys</th>
<th>MicroSD</th>
<th>IO Expander</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>EN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EN_KEY</td>
</tr>
<tr>
<td>4</td>
<td>IO4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DATA0</td>
</tr>
<tr>
<td>5</td>
<td>IO5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REC, MUTE, PLAY, SET, VOL-, VOL+</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>IO6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BAT_MEAS_ADC</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>IO7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CMD</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>IO15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CLK</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>IO16</td>
<td>12S0_MCLK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I2C_SDA</td>
<td>I2C_SDA</td>
</tr>
<tr>
<td>10</td>
<td>IO17</td>
<td>12C_SDA</td>
<td>12C_SDA</td>
<td>SIOD</td>
<td></td>
<td></td>
<td></td>
<td>TP_12C_SDA</td>
<td>I2C_SDA</td>
</tr>
<tr>
<td>11</td>
<td>IO18</td>
<td>12C_CLK</td>
<td>12C_CLK</td>
<td>SIOC</td>
<td></td>
<td></td>
<td></td>
<td>TP_12C_CLK</td>
<td>I2C_CLK</td>
</tr>
<tr>
<td>12</td>
<td>IO8</td>
<td>12S0_DSDIN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>IO19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>IO20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>IO3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D5</td>
</tr>
<tr>
<td>16</td>
<td>IO46</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NC</td>
</tr>
<tr>
<td>17</td>
<td>IO9</td>
<td>12S0_SCL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ISCLK</td>
</tr>
<tr>
<td>18</td>
<td>IO10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SDOUT</td>
</tr>
</tbody>
</table>

Continued on next page
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Pin</th>
<th>Pin Name</th>
<th>ES8311</th>
<th>ES7210</th>
<th>Camera</th>
<th>LCD</th>
<th>Keys</th>
<th>MicroSD</th>
<th>IO Expander</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>IO11</td>
<td>PCLK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>IO12</td>
<td>D6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>IO13</td>
<td>D2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>IO14</td>
<td>D4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>IO21</td>
<td>VSYNC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>IO47</td>
<td>D3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>IO48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PA_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>IO45</td>
<td>I2S0_LRC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>IO0</td>
<td>LCD_SPI_SDA_BOOT_KEY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>IO35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>IO36</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>IO37</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>IO38</td>
<td>HREF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>IO39</td>
<td>D9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>IO40</td>
<td>XCLK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>IO41</td>
<td>D8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>IO42</td>
<td>D7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>RXD0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ESP0_UART0_RX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>TXD0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ESP0_UART0_TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>IO2</td>
<td>LCD_SPI_DC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>IO1</td>
<td>LCD_SPI_CLK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>EPAD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The GPIOs allocated to the IO expander are further expanded to multiple GPIOs.

Table 3: IO Expander GPIO Allocation

<table>
<thead>
<tr>
<th>IO Expander Pin</th>
<th>Pin Name</th>
<th>LCD</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>P0</td>
<td></td>
<td>PA_CTRL</td>
</tr>
<tr>
<td>5</td>
<td>P1</td>
<td>LCD_CTRL</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>P2</td>
<td>LCD_RST</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>P3</td>
<td>LCD_CS</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>P4</td>
<td>TP_INT</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>P5</td>
<td></td>
<td>PERI_PWR_ON</td>
</tr>
<tr>
<td>11</td>
<td>P6</td>
<td></td>
<td>LED1</td>
</tr>
<tr>
<td>12</td>
<td>P7</td>
<td></td>
<td>LED2</td>
</tr>
</tbody>
</table>

1 Pin - ESP32-S3-WROOM-1 module pin number, GND and power supply pins are not listed.
Connector

Camera Connector

<table>
<thead>
<tr>
<th>No.</th>
<th>Camera Signal</th>
<th>ESP32-S3 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIOD</td>
<td>GPIO17</td>
</tr>
<tr>
<td>2</td>
<td>SIOC</td>
<td>GPIO18</td>
</tr>
<tr>
<td>3</td>
<td>D5</td>
<td>GPIO3</td>
</tr>
<tr>
<td>4</td>
<td>PCLK</td>
<td>GPIO11</td>
</tr>
<tr>
<td>5</td>
<td>D6</td>
<td>GPIO12</td>
</tr>
<tr>
<td>6</td>
<td>D2</td>
<td>GPIO13</td>
</tr>
<tr>
<td>7</td>
<td>D4</td>
<td>GPIO14</td>
</tr>
<tr>
<td>8</td>
<td>VSYNC</td>
<td>GPIO21</td>
</tr>
<tr>
<td>9</td>
<td>D3</td>
<td>GPIO47</td>
</tr>
<tr>
<td>10</td>
<td>HREF</td>
<td>GPIO38</td>
</tr>
<tr>
<td>11</td>
<td>D9</td>
<td>GPIO39</td>
</tr>
<tr>
<td>12</td>
<td>XCLK</td>
<td>GPIO40</td>
</tr>
<tr>
<td>13</td>
<td>D8</td>
<td>GPIO41</td>
</tr>
<tr>
<td>14</td>
<td>D7</td>
<td>GPIO42</td>
</tr>
</tbody>
</table>

LCD Connector

<table>
<thead>
<tr>
<th>No.</th>
<th>LCD Signal</th>
<th>ESP32-S3 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TP_I2C_SDA</td>
<td>GPIO17</td>
</tr>
<tr>
<td>2</td>
<td>TP_I2C_CLK</td>
<td>GPIO18</td>
</tr>
<tr>
<td>3</td>
<td>LCD_SPI_SDA</td>
<td>GPIO0</td>
</tr>
<tr>
<td>4</td>
<td>LCD_SPI_DC</td>
<td>GPIO2</td>
</tr>
<tr>
<td>5</td>
<td>LCD_SPI_CLK</td>
<td>GPIO1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No.</th>
<th>LCD Signal</th>
<th>IO Expander</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ESP_LCD_CTRL</td>
<td>P1</td>
</tr>
<tr>
<td>2</td>
<td>ESP_LCD_RST</td>
<td>P2</td>
</tr>
<tr>
<td>3</td>
<td>ESP_LCD_CS</td>
<td>P3</td>
</tr>
<tr>
<td>4</td>
<td>ESP_TP_INT</td>
<td>P4</td>
</tr>
</tbody>
</table>

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.
3.4. Development Boards

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

<table>
<thead>
<tr>
<th>No.</th>
<th>Codec Pin</th>
<th>ESP32-S3 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MCLK</td>
<td>GPIO16</td>
</tr>
<tr>
<td>2</td>
<td>SCLK</td>
<td>GPIO9</td>
</tr>
<tr>
<td>3</td>
<td>LRCK</td>
<td>GPIO45</td>
</tr>
<tr>
<td>4</td>
<td>DSDIN</td>
<td>GPIO8</td>
</tr>
<tr>
<td>5</td>
<td>ASDOUT</td>
<td>–</td>
</tr>
<tr>
<td>6</td>
<td>GND</td>
<td>–</td>
</tr>
</tbody>
</table>

ADC Test Point/J16

<table>
<thead>
<tr>
<th>No.</th>
<th>ADC Pin</th>
<th>ESP32-S3 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MCLK</td>
<td>GPIO16</td>
</tr>
<tr>
<td>2</td>
<td>SCLK</td>
<td>GPIO9</td>
</tr>
<tr>
<td>3</td>
<td>LRCK</td>
<td>GPIO45</td>
</tr>
<tr>
<td>4</td>
<td>SDOUT</td>
<td>GPIO10</td>
</tr>
<tr>
<td>5</td>
<td>INT</td>
<td>–</td>
</tr>
<tr>
<td>6</td>
<td>GND</td>
<td>–</td>
</tr>
</tbody>
</table>

UART Test Point/J17

<table>
<thead>
<tr>
<th>No.</th>
<th>UART Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.3V</td>
</tr>
<tr>
<td>2</td>
<td>TXD</td>
</tr>
<tr>
<td>3</td>
<td>RXD</td>
</tr>
<tr>
<td>4</td>
<td>IO0</td>
</tr>
<tr>
<td>5</td>
<td>EN</td>
</tr>
<tr>
<td>6</td>
<td>GND</td>
</tr>
</tbody>
</table>
I2C Test Point/J18

<table>
<thead>
<tr>
<th>No.</th>
<th>I2C Pin</th>
<th>ESP32-S3 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.3V</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>CLK</td>
<td>GPIO18</td>
</tr>
<tr>
<td>3</td>
<td>SDA</td>
<td>GPIO17</td>
</tr>
<tr>
<td>4</td>
<td>GND</td>
<td>–</td>
</tr>
</tbody>
</table>

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

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.

<table>
<thead>
<tr>
<th>Key Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCD Display</td>
<td>A 2.4&quot; 320x240 SPI LCD display module; the display driver/controller is Ilitek ILI934.</td>
</tr>
<tr>
<td>Home Key</td>
<td>(Reserved) Returns to homepage or previous page.</td>
</tr>
<tr>
<td>Signal Connector</td>
<td>Connects the power, ground, and signal wires between the extension board and the main board with a FPC cable.</td>
</tr>
<tr>
<td>LCD Connector</td>
<td>Connects LCD display to the driver circuit of this board.</td>
</tr>
<tr>
<td>TP Connector</td>
<td>Connects LCD display to the touch circuit of this board.</td>
</tr>
</tbody>
</table>

---

3.4. Development Boards 325
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.
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.
<table>
<thead>
<tr>
<th>Key Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP32-C3-WROOM-02 Module</td>
<td>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.</td>
</tr>
<tr>
<td>Speaker PA Chip</td>
<td>NS4150 is an EMI, 3 W mono Class D audio power amplifier, amplifying audio signals from ESP32-C3 PDM_TX to drive speakers.</td>
</tr>
<tr>
<td>Speaker Output Port</td>
<td>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.</td>
</tr>
<tr>
<td>Function Press Keys</td>
<td>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.</td>
</tr>
<tr>
<td>Boot/Reset Press Keys</td>
<td>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.</td>
</tr>
<tr>
<td>USB-to-UART Port</td>
<td>Functions as the communication interface between the PC and the ESP32-C3-WROOM-02 module.</td>
</tr>
<tr>
<td>USB-to-UART Bridge Chip</td>
<td>The single-chip USB-UART bridge CP2102N provides up to 3 Mbps transfer rates for software download and debugging.</td>
</tr>
<tr>
<td>USB Power Port</td>
<td>Provides power to the board. It is recommended to use at least 5 V/2 A power adapters to ensure a stable power supply.</td>
</tr>
<tr>
<td>System Power Switch</td>
<td>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.</td>
</tr>
<tr>
<td>LED Strip Power Selection Switch</td>
<td>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.</td>
</tr>
<tr>
<td>12 V DC Power Port</td>
<td>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.</td>
</tr>
<tr>
<td>12 V to 5 V Buck Chip</td>
<td>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.</td>
</tr>
<tr>
<td>Addressable LED Strip Port</td>
<td>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.</td>
</tr>
<tr>
<td>RGB LED Strip Port</td>
<td>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.</td>
</tr>
<tr>
<td>System Power LED</td>
<td>It turns red when System Power Switch is toggled to ON.</td>
</tr>
<tr>
<td>IR Receiver</td>
<td>IRM-H638T/TR2 is a miniature SMD type infrared remote control system receiver. The demodulated output signal can directly be decoded by ESP32-C3.</td>
</tr>
<tr>
<td>IR Transmitter</td>
<td>IR67-21C/TR8 is an infrared emitting diode. It is spectrally matched with silicon photodiode and phototransistor.</td>
</tr>
<tr>
<td>Microphone</td>
<td>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.</td>
</tr>
<tr>
<td>System LED</td>
<td>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.</td>
</tr>
</tbody>
</table>
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:
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:

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

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Qty</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP32-C3-Lyra</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>USB 2.0 cables (Standard-A to Micro-B)</td>
<td>2</td>
<td>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.</td>
</tr>
<tr>
<td>Computer running Windows, Linux, or macOS</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Speaker</td>
<td>1</td>
<td>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.</td>
</tr>
</tbody>
</table>
Optional Hardware

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Qty</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 V DC adapter</td>
<td>1</td>
<td>The maximum operating current of the adapter is 2 A. It provides power supply for 12 V LED strips.</td>
</tr>
<tr>
<td>5 V or 12 V addressable LED strip/ring</td>
<td>1</td>
<td>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).</td>
</tr>
<tr>
<td>5 V or 12 V RGB LED strip</td>
<td>1</td>
<td>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).</td>
</tr>
</tbody>
</table>

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
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.

12 V to 5 V Buck Power

System 3.3 V Power

Connector

RGB LED Strip Connector (JP1)

<table>
<thead>
<tr>
<th>No.</th>
<th>Signal Name</th>
<th>ESP32-C3 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VCC_12V_5V</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>LED_G</td>
<td>GPIO6</td>
</tr>
<tr>
<td>3</td>
<td>LED_R</td>
<td>GPIO5</td>
</tr>
<tr>
<td>4</td>
<td>LED_B</td>
<td>GPIO4</td>
</tr>
</tbody>
</table>
Fig. 52: 12 V to 5 V Buck Power Circuit

Fig. 53: System 3.3 V Power Circuit
Addressable LED Strip Connector (JP2)

<table>
<thead>
<tr>
<th>No.</th>
<th>Signal Name</th>
<th>ESP32-C3 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VCC_12V_5V</td>
<td>~</td>
</tr>
<tr>
<td>2</td>
<td>DIN</td>
<td>GPIO7</td>
</tr>
<tr>
<td>3</td>
<td>DIN</td>
<td>GPIO7</td>
</tr>
<tr>
<td>4</td>
<td>GND</td>
<td>~</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>No.</th>
<th>Signal Name</th>
<th>ESP32-C3 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VCC_3V3</td>
<td>~</td>
</tr>
<tr>
<td>2</td>
<td>ESP_EN</td>
<td>EN</td>
</tr>
<tr>
<td>3</td>
<td>ESP_BOOT</td>
<td>GPIO9</td>
</tr>
<tr>
<td>4</td>
<td>ESP_UART_RXD</td>
<td>U0RXD</td>
</tr>
<tr>
<td>5</td>
<td>ESP_UART_TXD</td>
<td>U0TXD</td>
</tr>
<tr>
<td>6</td>
<td>GND</td>
<td>~</td>
</tr>
</tbody>
</table>

I2C Header (JP8)

<table>
<thead>
<tr>
<th>No.</th>
<th>Signal Name</th>
<th>ESP32-C3 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VCC_3V3</td>
<td>~</td>
</tr>
<tr>
<td>2</td>
<td>I2C_CLK</td>
<td>GPIO8</td>
</tr>
<tr>
<td>3</td>
<td>I2C_DATA</td>
<td>GPIO9</td>
</tr>
<tr>
<td>4</td>
<td>GND</td>
<td>~</td>
</tr>
</tbody>
</table>

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.

3.4. Development Boards
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.

3.4.12 ESP32-C3-Lyra

3.5 Audio Samples

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

3.5.1 Formats

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

Long Samples

The audio track duration in this section is 3 minutes and 7 seconds.

Two Channel Audio

<table>
<thead>
<tr>
<th>No</th>
<th>Format</th>
<th>Audio File</th>
<th>Size [kB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>aac</td>
<td>ff-16b-2c-44100hz.aac</td>
<td>2,995</td>
</tr>
<tr>
<td>2</td>
<td>ac3</td>
<td>ff-16b-2c-44100hz.ac3</td>
<td>2,994</td>
</tr>
<tr>
<td>3</td>
<td>aiff</td>
<td>ff-16b-2c-44100hz.aiff</td>
<td>33,002</td>
</tr>
<tr>
<td>4</td>
<td>flac</td>
<td>ff-16b-2c-44100hz.flac</td>
<td>22,406</td>
</tr>
<tr>
<td>5</td>
<td>m4a</td>
<td>ff-16b-2c-44100hz.m4a</td>
<td>3,028</td>
</tr>
<tr>
<td>6</td>
<td>mp3</td>
<td>ff-16b-2c-44100hz.mp3</td>
<td>2,994</td>
</tr>
<tr>
<td>7</td>
<td>mp4</td>
<td>ff-16b-2c-44100hz.mp4</td>
<td>3,079</td>
</tr>
<tr>
<td>8</td>
<td>ogg</td>
<td>ff-16b-2c-44100hz.ogg</td>
<td>2,612</td>
</tr>
<tr>
<td>9</td>
<td>opus</td>
<td>ff-16b-2c-44100hz.opus</td>
<td>2,598</td>
</tr>
<tr>
<td>10</td>
<td>ts</td>
<td>ff-16b-2c-44100hz.ts</td>
<td>5,510</td>
</tr>
<tr>
<td>11</td>
<td>wav</td>
<td>ff-16b-2c-44100hz.wav</td>
<td>32,229</td>
</tr>
<tr>
<td>12</td>
<td>wma</td>
<td>ff-16b-2c-44100hz.wma</td>
<td>3,227</td>
</tr>
</tbody>
</table>

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

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

Playlist containing all above files: ff-16b-1c-playlist.m3u

Short Samples

If you need shorter audio files for testing, this section provides 16 seconds audio tracks.

Two Channel Audio

<table>
<thead>
<tr>
<th>No</th>
<th>Format</th>
<th>Audio File</th>
<th>Size [kB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>aac</td>
<td>gs-16b-2c-44100hz.aac</td>
<td>241</td>
</tr>
<tr>
<td>2</td>
<td>ac3</td>
<td>gs-16b-2c-44100hz.ac3</td>
<td>380</td>
</tr>
<tr>
<td>3</td>
<td>aiff</td>
<td>gs-16b-2c-44100hz.aiff</td>
<td>2,792</td>
</tr>
<tr>
<td>4</td>
<td>flac</td>
<td>gs-16b-2c-44100hz.flac</td>
<td>1,336</td>
</tr>
<tr>
<td>5</td>
<td>m4a</td>
<td>gs-16b-2c-44100hz.m4a</td>
<td>258</td>
</tr>
<tr>
<td>6</td>
<td>mp3</td>
<td>gs-16b-2c-44100hz.mp3</td>
<td>254</td>
</tr>
<tr>
<td>7</td>
<td>mp4</td>
<td>gs-16b-2c-44100hz.mp4</td>
<td>259</td>
</tr>
<tr>
<td>8</td>
<td>ogg</td>
<td>gs-16b-2c-44100hz.ogg</td>
<td>229</td>
</tr>
<tr>
<td>9</td>
<td>opus</td>
<td>gs-16b-2c-44100hz.opus</td>
<td>219</td>
</tr>
<tr>
<td>10</td>
<td>ts</td>
<td>gs-16b-2c-44100hz.ts</td>
<td>286</td>
</tr>
<tr>
<td>11</td>
<td>wav</td>
<td>gs-16b-2c-44100hz.wav</td>
<td>2,792</td>
</tr>
<tr>
<td>12</td>
<td>wma</td>
<td>gs-16b-2c-44100hz.wma</td>
<td>276</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>No</th>
<th>Format</th>
<th>Audio File</th>
<th>Size [kB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>amr</td>
<td>gs-16b-1c-8000hz.amr</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>aac</td>
<td>gs-16b-1c-44100hz.aac</td>
<td>137</td>
</tr>
<tr>
<td>3</td>
<td>ac3</td>
<td>gs-16b-1c-44100hz.ac3</td>
<td>190</td>
</tr>
<tr>
<td>4</td>
<td>aiff</td>
<td>gs-16b-1c-44100hz.aiff</td>
<td>1,397</td>
</tr>
<tr>
<td>5</td>
<td>flac</td>
<td>gs-16b-1c-44100hz.flac</td>
<td>645</td>
</tr>
<tr>
<td>6</td>
<td>m4a</td>
<td>gs-16b-1c-44100hz.m4a</td>
<td>258</td>
</tr>
<tr>
<td>7</td>
<td>mp3</td>
<td>gs-16b-1c-44100hz.mp3</td>
<td>127</td>
</tr>
<tr>
<td>8</td>
<td>ogg</td>
<td>gs-16b-1c-44100hz.ogg</td>
<td>144</td>
</tr>
<tr>
<td>9</td>
<td>opus</td>
<td>gs-16b-1c-44100hz.opus</td>
<td>132</td>
</tr>
<tr>
<td>10</td>
<td>wav</td>
<td>gs-16b-1c-44100hz.wav</td>
<td>1,497</td>
</tr>
<tr>
<td>11</td>
<td>wma</td>
<td>gs-16b-1c-44100hz.wma</td>
<td>276</td>
</tr>
</tbody>
</table>

Playlist containing all above files: gs-16b-1c-playlist.m3u

### 3.5.2 Sample Rates

The files in this section have been prepared by converting a single audio file into different sampling rates defined in MPEG Layer III specification. Both mono and stereo versions of files are provided. The bit depth of files is 16 bits.

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

Playlist containing all above files: ff-16b-mp3-playlist.m3u

Original music files: “Furious Freak” and “Galway”, Kevin MacLeod (incompetech.com), Licensed under Creative Commons: By Attribution 3.0, http://creativecommons.org/licenses/by/3.0/
CHAPTER 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.
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.
- 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.

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.
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 un-provisioned 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.
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
A2DP, 347
A2DP sink, 347
A2DP source, 347
AAC, 347
aac_decoder_cfg_t (C++ class), 115
aac_decoder_cfg_t::out_rb_size (C++ member), 115
aac_decoder_cfg_t::plus_enable (C++ member), 115
aac_decoder_cfg_t::stack_in_ext (C++ member), 115
aac_decoder_cfg_t::task_core (C++ member), 115
aac_decoder_cfg_t::task_prio (C++ member), 115
aac_decoder_cfg_t::task_stack (C++ member), 115
aac_decoder_init (C++ function), 115
AAC_DECODER_RINGBUFFER_SIZE (C macro), 116
AAC_DECODER_TASK_CORE (C macro), 116
AAC_DECODER_TASK_Prio (C macro), 116
AAC_DECODER_TASK_STACK_SIZE (C macro), 116
acoustic, 347
acoustic calibrator, 347
acoustic echo cancellation, 347
ADC, 348
ADC_BUTTON_STACK_SIZE (C macro), 215
ADC_BUTTON_TASK_CORE_ID (C macro), 215
ADC_BUTTON_TASK_PRIORITY (C macro), 215
ADC_DEFAULT.ARR (C macro), 215
Advanced Audio Distribution Profile, 347
AEC, 347
AEL_MSG_CMD_ABORT (C++ enumerator), 38
AEL_MSG_CMD_DONE (C++ enumerator), 38
AEL_MSG_CMD_FAIL (C++ enumerator), 38
AEL_MSG_CMD_OK (C++ enumerator), 38
AEL_MSG_CMD_TIMEOUT (C++ enumerator), 38
AEL_MSG_CMD_DESTROY (C++ enumerator), 38
AEL_MSG_CMD_FINISH (C++ enumerator), 38
AEL_MSG_CMD_NONE (C++ enumerator), 38
AEL_MSG_CMD_PAUSE (C++ enumerator), 38
AEL_MSG_CMD_REPORT_CODEC_FMT (C++ enumerator), 38
AEL_MSG_CMD_REPORT_MUSIC_INFO (C++ enumerator), 38
AEL_MSG_CMD_REPORT_POSITION (C++ enumerator), 38
AEL_MSG_CMD_REPORT_STATUS (C++ enumerator), 38
AEL_MSG_CMD_RESUME (C++ enumerator), 38
AEL_MSG_CMD_STOP (C++ enumerator), 38
AEL_PROCESS_FAIL (C++ enumerator), 38
AEL_STATE_ERROR (C++ enumerator), 38
AEL_STATE_FINISHED (C++ enumerator), 38
AEL_STATE_INIT (C++ enumerator), 38
AEL_STATE_INITIALIZING (C++ enumerator), 38
AEL_STATE_NONE (C++ enumerator), 38
AEL_STATE_PAUSED (C++ enumerator), 38
AEL_STATE_RUNNING (C++ enumerator), 38
AEL_STATE_STOPPED (C++ enumerator), 38
AEL_STATUS_ERROR_CLOSE (C++ enumerator), 39
AEL_STATUS_ERROR_INPUT (C++ enumerator), 39
AEL_STATUS_ERROR_PROCESS (C++ enumerator), 39
AEL_STATUS_ERROR_TIMEOUT (C++ enumerator), 39
AEL_STATUS_ERROR_UNKNOWN (C++ enumerator), 39
AEL_STATUS_INPUT_BUFFERING (C++ enumerator), 39
AEL_STATUS_INPUT_DONE (C++ enumerator), 39
AEL_STATUS_MOUNTED (C++ enumerator), 39
AEL_STATUS_NONE (C++ enumerator), 39
AEL_STATUS_OUTPUT_BUFFERING (C++ enumerator), 39
AEL_STATUS_OUTPUT_DONE (C++ enumerator), 39
AEL_STATUS_STATE_FINISHED (C++ enumerator), 39
AEL_STATUS_STATE_PAUSED (C++ enumerator), 39

359
AEL_STATUS_STATE_RUNNING (C++ enumerator), 39
AEL_STATUS_STATE_STOPPED (C++ enumerator), 39
AEL_STATUS_UNMOUNTED (C++ enumerator), 39
AFL, 347
AG, 348
AGC, 348
AirKiss, 347
airkiss_config_create (C++ function), 155
AIRKISS_CONFIG_INFO_DEFAULT (C macro), 156
airkiss_config_info_t (C++ class), 156
airkiss_config_info_t::aes_key (C++ member), 156
airkiss_config_info_t::lan_pack (C++ member), 156
airkiss_config_info_t::ssdp_notify_enable (C++ member), 156
airkiss_lan_pack_param_t (C++ class), 155
airkiss_lan_pack_param_t::appid (C++ member), 156
airkiss_lan_pack_param_t::deviceid (C++ member), 156
ALC, 348
algo_stream_init (C++ function), 71
algo_stream_set_delay (C++ function), 71
algorithm_mono_fix (C++ function), 71
ALGORITHM_STREAM_CFG_DEFAULT (C macro), 73
algorithm_stream_cfg_t (C++ class), 72
algorithm_stream_cfg_t::aec_low_cost (C++ member), 72
algorithm_stream_cfg_t::agc_gain (C++ member), 72
algorithm_stream_cfg_t::algo_mask (C++ member), 72
algorithm_stream_cfg_t::debug_input (C++ member), 72
algorithm_stream_cfg_t::input_type (C++ member), 72
algorithm_stream_cfg_t::mic_ch (C++ member), 72
algorithm_stream_cfg_t::out_rb_size (C++ member), 72
algorithm_stream_cfg_t::rec_linear_factor (C++ member), 72
algorithm_stream_cfg_t::ref_linear_factor (C++ member), 72
algorithm_stream_cfg_t::sample_rate (C++ member), 72
algorithm_stream_cfg_t::stack_in_ext (C++ member), 72
algorithm_stream_cfg_t::swap_ch (C++ member), 72
algorithm_stream_cfg_t::task_core (C++ member), 72
algorithm_stream_cfg_t::task_prio (C++ member), 72
algorithm_stream_cfg_t::task_stack (C++ member), 72
ALGORITHM_STREAM_DEFAULT_AGC_GAIN_DB (C macro), 73
ALGORITHM_STREAM_DEFAULT_MASK (C macro), 73
ALGORITHM_STREAM_DEFAULT_MIC_CHANNELS (C macro), 73
ALGORITHM_STREAM_DEFAULT_SAMPLE_BIT (C macro), 73
ALGORITHM_STREAM_DEFAULT_SAMPLE_RATE_HZ (C macro), 73
ALGORITHM_STREAM_INPUT_TYPE1 (C++ enumerator), 73
ALGORITHM_STREAM_INPUT_TYPE2 (C++ enumerator), 73
algorithm_stream_input_type_t (C++ enum), 73
algorithm_stream_mask_t (C++ enum), 73
ALGORITHM_STREAM_PINNED_TO_CORE (C macro), 73
ALGORITHM_STREAM_RINGBUFFER_SIZE (C macro), 73
ALGORITHM_STREAM_TASK_PERIOD (C macro), 73
ALGORITHM_STREAM_TASK_STACK_SIZE (C macro), 73
AMR, 347
amr_decoder_init (C++ function), 116
amr_decoder_cfg_t (C++ class), 117
amr_decoder_cfg_t::out_rb_size (C++ member), 117
amr_decoder_cfg_t::stack_in_ext (C++ member), 117
amr_decoder_cfg_t::task_core (C++ member), 117
amr_decoder_cfg_t::task_prio (C++ member), 117
AMR_DECODER_RINGBUFFER_SIZE (C macro), 117
AMR_DECODER_TASK_CORE (C macro), 117
AMR_DECODER_TASK_PRIO (C macro), 117
AMR_DECODER_TASK_STACK_SIZE (C macro), 117
AMR-NB, 347
AMR-WB, 348
AMRNB_ENC_BITRATE_MR102 (C++ enumerator), 119
AMRNB_ENC_BITRATE_MR122 (C++ enumerator), 119
AMRNB_ENC_BITRATE_MR475 (C++ enumerator), 119
AMRNB_ENC_BITRATE_MR515 (C++ enumerator), 119
AMRNB_ENC_BITRATE_MR59 (C++ enumerator), 119
AMRNB_ENC_BITRATE_MR67 (C++ enumerator), 119
AMRNB_ENC_BITRATE_MR74 (C++ enumerator), 119
AMRNB_ENC_BITRATE_MR795 (C++ enumerator), 119
AMRNB_ENC_BITRATE_MRD TX (C++ enumerator), 119
AMRNB_ENC_BITRATE_N_MODES (C++ enumerator), 119
AMRNB_ENC_BITRATE_UNKN OW (C++ enumerator), 119
amrnb_encoder_bitrate_t (C++ enum), 119
amrnb_encoder_cfg_t (C++ class), 118
amrnb_encoder_cfg_t::bitrate_mode (C++ member), 118
amrnb_encoder_cfg_t::contain_amrnb_header (C++ member), 118
amrnb_encoder_cfg_t::out_rb_size (C++ member), 118
amrnb_encoder_cfg_t::stack_in_ext (C++ member), 118
amrnb_encoder_cfg_t::task_core (C++ member), 118
amrnb_encoder_cfg_t::task_prio (C++ member), 118
amrnb_encoder_cfg_t::task_stack (C++ member), 118
amrnb_encoder_init (C++ function), 117
AMRNB_ENCODER_RINGBUFFER_SIZE (C macro), 118
amrnb_encoder_set_bitrate (C++ function), 117
AMRNB_ENCODER_TASK_CORE (C macro), 118
AMRNB_ENCODER_TASK_PARIO (C macro), 118
AMRNB_ENCODER_TASK_STACK (C macro), 118
AMRWB_ENC_BITRATE_MD1265 (C++ enumerator), 120
AMRWB_ENC_BITRATE_MD1425 (C++ enumerator), 121
AMRWB_ENC_BITRATE_MD1585 (C++ enumerator), 121
AMRWB_ENC_BITRATE_MD1825 (C++ enumerator), 121
AMRWB_ENC_BITRATE_MD1985 (C++ enumerator), 121
AMRWB_ENC_BITRATE_MD2305 (C++ enumerator), 121
AMRWB_ENC_BITRATE_MD2385 (C++ enumerator), 121
AMRWB_ENC_BITRATE_MD66 (C++ enumerator), 120
AMRWB_ENC_BITRATE_MD885 (C++ enumerator), 120
AMRWB_ENC_BITRATE_MDNONE (C++ enumerator), 120
AMRWB_ENC_BITRATE_N_MODES (C++ enumerator), 121
amrwb_encoder_bitrate_t (C++ enum), 120
amrwb_encoder_cfg_t (C++ class), 120
amrwb_encoder_cfg_t::bitrate_mode (C++ member), 120
amrwb_encoder_cfg_t::contain_amrwb_header (C++ member), 120
amrwb_encoder_cfg_t::out_rb_size (C++ member), 120
amrwb_encoder_cfg_t::stack_in_ext (C++ member), 120
amrwb_encoder_cfg_t::task_core (C++ member), 120
amrwb_encoder_cfg_t::task_prio (C++ member), 120
amrwb_encoder_cfg_t::task_stack (C++ member), 120
amrwb_encoder_init (C++ function), 119
AMRWB_ENCODER_RINGBUFFER_SIZE (C macro), 120
amrwb_encoder_set_bitrate (C++ function), 119
AMRWB_ENCODER_TASK_CORE (C macro), 120
AMRWB_ENCODER_TASK_PARIO (C macro), 120
AMRWB_ENCODER_TASK_STACK (C macro), 120
analog-to-digital converter, 348
audio codec, 348
audio forge, 348
audio front end, 348
audio gate, 348
audio passthru, 348
audio pipeline, 348
Audio Video Remote Control Profile, 348
AUDIO_CODEC_TYPE_DECODER (C++ enumerator), 55
AUDIO_CODEC_TYPE_ENCODER (C++ enumerator), 55
AUDIO_CODEC_TYPE_NONE (C++ enumerator), 55
audio_element_abort_input_ringbuf (C++ function), 24
audio_element_abort_output_ringbuf (C++ function), 24
function), 24
audio_element_cfg_t (C++ class), 36
audio_element_cfg_t::buffer_len (C++ member), 36
audio_element_cfg_t::close (C++ member), 36
audio_element_cfg_t::data (C++ member), 37
audio_element_cfg_t::destroy (C++ member), 36
audio_element_cfg_t::multi_in_rb_num (C++ member), 37
audio_element_cfg_t::multi_out_rb_num (C++ member), 37
audio_element_cfg_t::open (C++ member), 36
audio_element_cfg_t::out_rb_size (C++ member), 37
audio_element_cfg_t::process (C++ member), 36
audio_element_cfg_t::read (C++ member), 36
audio_element_cfg_t::seek (C++ member), 36
audio_element_cfg_t::stack_in_ext (C++ member), 37
audio_element_cfg_t::tag (C++ member), 37
audio_element_cfg_t::task_core (C++ member), 37
audio_element_cfg_t::task_prio (C++ member), 37
audio_element_cfg_t::task_stack (C++ member), 36
audio_element_cfg_t::write (C++ member), 36
audio_element_change_cmd (C++ function), 26
audio_element_deinit (C++ function), 18
audio_element_err_t (C++ enum), 38
audio_element_finish_state (C++ function), 26
audio_element_get_event_queue (C++ function), 28
audio_element_get_input_ringbuf (C++ function), 23
audio_element_get_input_ringbuf (C++ function), 31
audio_element_get_multi_input_ringbuf (C++ function), 31
audio_element_get_multi_output_ringbuf (C++ function), 23
audio_element_get_output_ringbuf (C++ function), 23
audio_element_get_output_ringbuf_size (C++ function), 29
audio_element_get_read_cb (C++ function), 28
audio_element_get_state (C++ function), 24
audio_element_get_tag (C++ function), 19
audio_element_get_uri (C++ function), 20
audio_element_get_write_cb (C++ function), 28
audio_element_getdata (C++ function), 19
audio_element_getinfo (C++ function), 19
audio_element_handle_t (C++ type), 37
AUDIO_ELEMENT_INFO_DEFAULT (C macro), 37
audio_element_info_t (C++ class), 35
audio_element_info_t::bits (C++ member), 36
audio_element_info_t::bps (C++ member), 36
audio_element_info_t::byte_pos (C++ member), 36
audio_element_info_t::channels (C++ member), 36
audio_element_info_t::codec_fmt (C++ member), 36
audio_element_info_t::duration (C++ member), 36
audio_element_info_t::reserve_data (C++ member), 36
audio_element_info_t::sample_rates (C++ member), 36
audio_element_info_t::total_bytes (C++ member), 36
audio_element_info_t::uri (C++ member), 36
audio_element_init (C++ function), 18
audio_element_input (C++ function), 27
audio_element_is_stopping (C++ function), 32
audio_element_msg_cmd_t (C++ enum), 38
audio_element_msg_remove_listener (C++ function), 23
audio_element_msg_set_listener (C++ function), 22
audio_element_multi_input (C++ function), 29
audio_element_multi_output (C++ function), 30
audio_element_output (C++ function), 27
audio_element_pause (C++ function), 22
audio_element_process_deinit (C++ function), 31
audio_element_process_init (C++ function), 31
audio_element_report_codec_fmt (C++ function), 25
audio_element_report_info (C++ function), 25
audio_element_report_pos (C++ function), 25
audio_element_report_status (C++ function), 24
audio_element_reserve_data_t (C++ class), 35
audio_element_reserve_data_t::user_data_0 (C++ member), 35
audio_element_reserve_data_t::user_data_1 (C++ member), 35
audio_element_reserve_data_t::user_data_2 (C++ member), 35
audio_element_reserve_data_t::user_data_3 (C++ member), 35
audio_element_reserve_data_t::user_data_4 (C++ member), 35
audio_element_reset_input_ringbuf (C++ function), 26
audio_element_reset_output_ringbuf (C++ function), 27
audio_element_reset_state (C++ function), 29
audio_element_resume (C++ function), 22
audio_element_run (C++ function), 20
audio_element_seek (C++ function), 32
audio_element_set_bps (C++ function), 33
audio_element_set_byte_pos (C++ function), 32
audio_element_set_codec_fmt (C++ function), 33
audio_element_set_duration (C++ function), 22
audio_element_set_event_callback (C++ function), 23
audio_element_set_input_ringbuf (C++ function), 23
audio_element_set_input_timeout (C++ function), 25
audio_element_set_multi_input_ringbuf (C++ function), 30
audio_element_set_multi_output_ringbuf (C++ function), 30
audio_element_set_music_info (C++ function), 33
audio_element_set_output_ringbuf (C++ function), 23
audio_element_set_output_ringbuf_size (C++ function), 29
audio_element_set_output_timeout (C++ function), 26
audio_element_set_read_cb (C++ function), 27
audio_element_set_reserve_user0 (C++ function), 34
audio_element_set_reserve_user1 (C++ function), 34
audio_element_set_reserve_user2 (C++ function), 34
audio_element_set_reserve_user3 (C++ function), 35
audio_element_set_reserve_user4 (C++ function), 35
audio_element_set_ringbuf_done (C++ function), 28
audio_element_set_tag (C++ function), 19
audio_element_set_total_bytes (C++ function), 33
audio_element_set_uri (C++ function), 20
audio_element_set_write_cb (C++ function), 28
audio_element_setdata (C++ function), 18
audio_element_setinfo (C++ function), 19
audio_element_state_t (C++ enum), 38
audio_element_status_t (C++ enum), 38
audio_element_stop (C++ function), 21
audio_element_terminate (C++ function), 20
audio_element_terminate_with_ticks (C++ function), 20
AUDIO_ELEMENT_TYPE_ELEMENT (C++ enumerator), 54
AUDIO_ELEMENT_TYPE_PERIPH (C++ enumerator), 54
AUDIO_ELEMENT_TYPE_PLAYER (C++ enumerator), 54
AUDIO_ELEMENT_TYPE_SERVICE (C++ enumerator), 54
audio_element_type_t (C++ enum), 54
AUDIO_ELEMENT_TYPE_UNKNOW (C++ enumerator), 54
audio_element_update_byte_pos (C++ function), 32
audio_element_update_total_bytes (C++ function), 32
audio_element_wait_for_buffer (C++ function), 24
audio_element_wait_for_stop (C++ function), 21
audio_element_wait_for_stop_ms (C++ function), 21
audio_err_t (C++ enum), 56
audio_event_iface_cfg_t (C++ class), 53
audio_event_iface_cfg_t::context (C++ member), 53
audio_event_iface_cfg_t::external_queue_size (C++ member), 53
audio_event_iface_cfg_t::internal_queue_size (C++ member), 53
audio_event_iface_cfg_t::type (C++ member), 54
audio_event_iface_cfg_t::on_cmd (C++ member), 53
audio_event_iface_cmd (C++ function), 51
audio_event_iface_cmd_from_isr (C++ function), 51
AUDIOEVENT_DEFAULT_CFG (C macro), 54
audio_event_iface_destroy (C++ function), 50
audio_event_iface_discard (C++ function), 51

Index 363
audio_event_iface_get_msg_queue_handle (C++ function), 52
audio_event_iface_get_queue_handle (C++ function), 52
audio_event_iface_handle_t (C++ type), 54
audio_event_iface_init (C++ function), 49
audio_event_iface_listen (C++ function), 52
audio_event_iface_msg_t (C++ class), 53
audio_event_iface_msg_t::cmd (C++ member), 53
audio_event_iface_msg_t::data (C++ member), 53
audio_event_iface_msg_t::data_len (C++ member), 53
audio_event_iface_msg_t::need_free_data (C++ member), 53
audio_event_iface_msg_t::source (C++ member), 53
audio_event_iface_msg_t::source_type (C++ member), 53
audio_event_iface_read (C++ function), 52
audio_event_iface_remove_listener (C++ function), 50
audio_event_iface_sendout (C++ function), 51
audio_event_iface_set_cmd_waiting_timeout (C++ function), 50
audio_event_iface_set_listener (C++ function), 50
audio_event_iface_set_msg_listener (C++ function), 52
audio_event_iface_waiting_cmd_msg (C++ function), 50
audio_hal (C++ class), 226
audio_hal::audio_codec_config_iface (C++ member), 227
audio_hal::audio_codec_ctrl (C++ member), 227
audio_hal::audio_codec_deinititalize (C++ member), 227
audio_hal::audio_codec_disable (C++ member), 227
audio_hal::audio_codec_enable (C++ member), 227
audio_hal::audio_codec_enable_volume (C++ member), 227
audio_hal::audio_codec_get_volume (C++ member), 227
audio_hal::audio_codec_initialize (C++ member), 227
audio_hal::audio_codec_set_mute (C++ member), 227
audio_hal::audio_codec_set_volume (C++ member), 227
audio_hal::audio_hal_lock (C++ member), 227
audio_hal::handle (C++ member), 227
AUDIO_HAL_08K_SAMPLES (C++ enumerator), 228
AUDIO_HAL_11K_SAMPLES (C++ enumerator), 229
AUDIO_HAL_16K_SAMPLES (C++ enumerator), 229
AUDIO_HAL_22K_SAMPLES (C++ enumerator), 229
AUDIO_HAL_24K_SAMPLES (C++ enumerator), 229
AUDIO_HAL_32K_SAMPLES (C++ enumerator), 229
AUDIO_HAL_ADC_INPUT_ALL (C++ enumerator), 228
AUDIO_HAL_ADC_INPUT_DIFFERENCE (C++ enumerator), 228
AUDIO_HAL_ADC_INPUT_LINE1 (C++ enumerator), 228
audio_hal_adc_input_t (C++ enum), 228
AUDIO_HAL_BIT_LENGTH_16BITS (C++ enumerator), 229
AUDIO_HAL_BIT_LENGTH_24BITS (C++ enumerator), 229
AUDIO_HAL_BIT_LENGTH_32BITS (C++ enumerator), 229
audio_hal_codec_config_t (C++ class), 226
audio_hal_codec_config_t::adc_input (C++ member), 226
audio_hal_codec_config_t::dac_output (C++ member), 226
audio_hal_codec_config_t::i2s_iface (C++ member), 226
audio_hal_codec_i2s_iface_t (C++ class), 226
audio_hal_codec_i2s_iface_t::bits (C++ member), 226
audio_hal_codec_i2s_iface_t::fmt (C++ member), 226
audio_hal_codec_i2s_iface_t::mode (C++ member), 226
audio_hal_codec_i2s_iface_t::samples (C++ member), 226
audio_hal_codec_iface_config (C++ function), 225
AUDIO_HAL_CODEC_MODE_BOTH (C++ enumerator), 227
AUDIO_HAL_CODEC_MODE_Decode (C++ enumerator), 227
AUDIO_HAL_CODEC_MODE_encode (C++ enumerator), 227
AUDIO_HAL_CODEC_MODE_LINE_IN (C++ enumerator), 228
audio_hal_codec_mode_t (C++ enum), 227
audio_hal::ctrl_code (C++ function), 224
AUDIO_HAL_CTRL_START (C++ enumerator), 228
audio_hal::ctrl_stop (C++ function), 228
audio_hal_ctrl_t (C++ enum), 228
AUDIO_HAL_DAC_OUTPUT_ALL (C++ enumerator), 228
AUDIO_HAL_DAC_OUTPUT_LINE1 (C++ enumerator), 228
AUDIO_HAL_DAC_OUTPUT_LINE2 (C++ enumerator), 228
audio_hal_dac_output_t (C++ function), 48
audio_hal_handle_t (C++ type), 227
audio_hal_get_volume (C++ function), 225
audio_hal_init (C++ function), 224
audio_hal_iface_samples_t::data_len (C++ function), 42
audio_hal_iface_samples_t::duty_resolution (C++ function), 42
audio_hal_iface_samples_t::gpio_num_left (C++ function), 42
audio_hal_iface_samples_t::gpio_num_right (C++ function), 42
audio_hal_iface_samples_t::timer_num (C++ function), 42
audio_hal ידי t (C++ function), 225
audio_hal_dac_output_t (C++ function), 224
audio_hal_enable_pa (C++ function), 226
audio_hallfw_func_t (C++ type), 227
audio_hal_get_volume (C++ function), 225
audio_hal_lkt (C++ function), 227
audio_pipeline_register (C++ function), 47
audio_pipeline_register (C++ function), 45
audio_pipeline_relink (C++ function), 47
audio_pipeline_relink_more (C++ function), 48
audio_pipeline_remove_listener (C++ function), 44
audio_pipeline_remove_listener (C++ function), 44
audio_pipeline_reset_elements (C++ function), 47
audio_pipeline_reset_items_state (C++ function), 46
audio_pipeline_reset_kept_state (C++ function), 47
audio_pipeline_reset_ringbuffer (C++ function), 46
audio_pipeline_resume (C++ function), 42
audio_pipeline_run (C++ function), 41
audio_pipeline_set_listener (C++ function), 44
audio_pipeline_unlink (C++ function), 43
audio_pipeline_unregister (C++ function), 40
audio_pipeline_unregister_more (C++ function), 45
audio_pipeline_wait_for_stop (C++ function), 42
audio_pipeline_wait_for_stop_with_ticks (C++ function), 42
audio_pipeline_config (C++ class), 82
audio_pipeline_config_t::data_len (C++ member), 83
audio_pipeline_config_t::duty_resolution (C++ member), 83
audio_pipeline_config_t::gpio_num_left (C++ member), 83
audio_pipeline_config_t::gpio_num_right (C++ member), 83
audio_pipeline_config_t::ledc_channel_left (C++ member), 83
audio_pipeline_config_t::ledc_channel_right (C++ member), 83
audio_pipeline_config_t::ledc_timer_sel (C++ member), 83
audio_pipeline_config_t::tg_num (C++ member), 83
audio_pipeline_config_t::timer_num (C++ member), 83
audio_pipeline_config_t::service_connect (C++ member), 184
audio_pipeline_cfg (C++ member), 49
audio_pipeline_cfg::rb_size (C++ member), 49
audio_pipeline_cfg::t (C++ type), 48
audio_pipeline_change_state (C++ function), 46
audio_pipeline_check_items_state (C++ function), 46
audio_pipeline_deinit (C++ function), 40
audio_pipeline_get_el_by_tag (C++ function), 43
audio_pipeline_get_el_by_tag (C++ function), 43
audio_pipeline_get_el_by_tag (C++ function), 43
audio_pipeline_get_el_by_tag (C++ function), 43
audio_pipeline_get_el_by_tag (C++ function), 43
audio_pipeline_get_el_once (C++ function), 44
audio_pipeline_get_event_iface (C++ function), 44
audio_pipeline_handle_t (C++ type), 49
audio_pipeline_init (C++ function), 40
audio_pipeline_link (C++ function), 43
audio_pipeline_link_insert (C++ function), 44
audio_pipeline_link_more (C++ function), 45
audio_pipeline_link_more (C++ function), 45
audio_pipeline_link_more (C++ function), 45
audio_pipeline_listen_more (C++ function), 46
audio_pipeline_pause (C++ function), 42
audio_pipeline_register (C++ function), 40
audio_pipeline_register_more (C++ function), 45
audio_pipeline_relink (C++ function), 47
audio_pipeline_relink_more (C++ function), 48
audio_pipeline_remove_listener (C++ function), 44
audio_pipeline_reset_elements (C++ function), 47
audio_pipeline_reset_items_state (C++ function), 46
audio_pipeline_reset_kept_state (C++ function), 47
audio_pipeline_reset_ringbuffer (C++ function), 46
audio_pipeline_resume (C++ function), 42
audio_pipeline_run (C++ function), 41
audio_pipeline_set_listener (C++ function), 44
audio_pipeline_unlink (C++ function), 43
audio_pipeline_unregister (C++ function), 40
audio_pipeline_unregister_more (C++ function), 45
audio_pipeline_wait_for_stop (C++ function), 42
audio_pipeline_wait_for_stop_with_ticks (C++ function), 42
audio_pipeline_config (C++ class), 82
audio_pipeline_config_t::data_len (C++ member), 83
audio_pipeline_config_t::duty_resolution (C++ member), 83
audio_pipeline_config_t::gpio_num_left (C++ member), 83
audio_pipeline_config_t::gpio_num_right (C++ member), 83
audio_pipeline_config_t::ledc_channel_left (C++ member), 83
audio_pipeline_config_t::ledc_channel_right (C++ member), 83
audio_pipeline_config_t::ledc_timer_sel (C++ member), 83
audio_pipeline_config_t::tg_num (C++ member), 83
audio_pipeline_config_t::timer_num (C++ member), 83
audio_pipeline_config_t::service_connect (C++ member), 184
Index 365
audio_service_set_data (C++ member), 184
audio_service_stop (C++ member), 184
audio_service_start (C++ member), 184
audio_service_set_callback (C++ function), 184
audio_service_get_data (C++ member), 184
audio_service_disconnect (C++ member), 184
audio_service_destroy (C++ member), 184
audio_service_create (C++ function), 184
audio_service_connect (C++ function), 184
audio_service_config_t::service_destroy (C++ member), 184
audio_service_config_t::service_disconnect (C++ member), 184
audio_service_config_t::service_stop (C++ member), 184
audio_service_config_t::service_start (C++ member), 184
audio_service_config_t::task_core (C++ member), 184
audio_service_config_t::task_func (C++ member), 184
audio_service_config_t::task_prio (C++ member), 184
audio_service_config_t::task_stack (C++ member), 184
audio_service_config_t::user_data (C++ member), 184
audio_service_connect (C++ function), 182
audio_service_create (C++ function), 181
audio_service_destroy (C++ function), 181
audio_service_disconnect (C++ function), 182
audio_service_get_data (C++ function), 183
audio_service_handle_t (C++ type), 184
audio_service_set_callback (C++ function), 182
audio_service_set_data (C++ function), 183
audio_service_start (C++ function), 181
audio_service_stop (C++ function), 181
AUDIO_STATUS_ERROR (C++ enumerator), 57
AUDIO_STATUS_FINISHED (C++ enumerator), 57
AUDIO_STATUS_PAUSED (C++ enumerator), 57
AUDIO_STATUS_RUNNING (C++ enumerator), 56
AUDIO_STATUS_STOPPED (C++ enumerator), 56
AUDIO_STATUS_UNKNOWN (C++ enumerator), 56
AUDIO_STREAM_NONE (C++ enumerator), 54
AUDIO_STREAM_READER (C++ enumerator), 55
audio_stream_type_t (C++ enum), 54
AUDIO_STREAM_WRITER (C++ enumerator), 55
audio_termination_type_t (C++ enum), 57
audio_volume_get (C++ type), 56
audio_volume_set (C++ type), 56
automatic_gain_control, 348
automatic_level_control, 348
automatic_speech_recognition, 348
aux_cable, 349
AVRCP, 348
aw2013_led_bar_task (C++ function), 167
B
bandwidth, 349
Bass Frequency, 349

B

bathtub, 349
Bluetooth, 349
BLUE_LED_MAX_NUM (C macro), 219
BLUETOOTH_A2DP_LINK (C++ enumerator), 145
BLUETOOTH_A2DP_SOURCE (C++ enumerator), 145
BLUETOOTH_ADDR_LEN (C macro), 144
BLUETOOTH_ADDR_LEN (C++ enum), 145
bluetooth_addr_t (C++ type), 145
bluetooth_service_cfg_t (C++ class), 144
bluetooth_service_cfg_t::device_name (C++ member), 144
bluetooth_service_cfg_t::mode (C++ member), 144
bluetooth_service_cfg_t::remote_name (C++ member), 144
bluetooth_service_cfg_t::user_callback (C++ member), 144
bluetooth_service_create_periph (C++ function), 141
bluetooth_service_create_stream (C++ function), 141
bluetooth_service_destroy (C++ function), 143
bluetooth_service_mode_t (C++ enum), 145
bluetooth_service_start (C++ function), 141
bluetooth_service_user_cb_t (C++ class), 144
bluetooth_service_user_cb_t::user_a2d_cb (C++ member), 144
bluetooth_service_user_cb_t::user_a2d_sbc (C++ member), 144
bluetooth_service_user_cb_t::user_a2d_sbc_sbc (C++ member), 144
bluetooth_service_user_cb_t::user_avrc_ct_cb (C++ member), 144
BluFi, 349
blufi_config_create (C++ function), 154
blufi_set_customized_data (C++ function), 154
blufi_set_customized_data (C++ function), 154
blufi_set_sta_connected_flag (C++ function), 154
C
Cavity, 349
command word, 349
console_cmd_callback_t (C++ type), 209
CONSOLE_DEFAULT_BUFFER_SIZE (C macro), 209
CONSOLE_DEFAULT_PROMPT_STRING (C macro), 209
CONSOLE_DEFAULT_TASK_PRIO (C macro), 209
CONSOLE_DEFAULT_TASK_STACK (C macro), 209
coredump_need_upload (C++ function), 176
coredump_upload (C++ function), 176
coredump_upload_service_config_t (C++ class), 176
coredump_upload_service_config_t::cb_ctx (C++ member), 176
coredump_upload_service_config_t::do_post (C++ member), 177
coredump_upload_service_config_t::evt_cb (C++ member), 176
coredump_upload_service_config_t::task_config (C++ member), 176
coredump_upload_service_config_t::task_post (C++ member), 176
coredump_upload_service_config_t::task_pre (C++ member), 176
coredump_upload_service_create (C++ function), 176
COREDUMP_UPLOAD_SERVICE_DEFAULT_CONFIG (C macro), 177
ctrl_func (C++ type), 37
cutoff frequency, 349
D
DAC, 349
decoder, 349
decoder_opus_init (C++ function), 125
default_AAC_DECODER_CONFIG (C macro), 116
default_AMR_DECODER_CONFIG (C macro), 117
default_AMRNB_DECODER_CONFIG (C macro), 118
default_AMRWB_DECODER_CONFIG (C macro), 120
default_AUDIO_ELEMENT_CONFIG (C macro), 37
default_AUDIO_EVENT_IFACE_SIZE (C macro), 54
default_AUDIO_PIPELINE_CONFIG (C macro), 49
default_DOWNMIX_CONFIG (C macro), 132
default_ELEMENT_BUFFER_LENGTH (C macro), 37
default_ELEMENT_RINGBUF_SIZE (C macro), 37
default_ELEMENT_STACK_SIZE (C macro), 37
default_ELEMENT_TASK_CORE (C macro), 37
default_ELEMENT_TASK_PRIO (C macro), 37
default_EQUALIZER_CONFIG (C macro), 134
default_ESP_AUDIO_CONFIG (C macro), 69
default_ESP_PERIPH_SET_CONFIG (C macro), 199
default_ESP_PERIPH_STACK_SIZE (C macro), 199
default_ESP_PERIPH_TASK_CORE (C macro), 199
default_ESP_PERIPH_TASK_PRIO (C macro), 199
default_FLAC_DECODER_CONFIG (C macro), 122
default_MP3_DECODER_CONFIG (C macro), 123
default_OGG_DECODER_CONFIG (C macro), 124
default_OPS_DECODER_CONFIG (C macro), 126
default_PIPELINE_RINGBUF_SIZE (C macro), 49
default_REC_ENGINE_CONFIG (C macro), 190
default_RESAMPLE_FILTER_CONFIG (C macro), 137
default_SONIC_CONFIG (C macro), 140
default WAV_DECODER_CONFIG (C macro), 127
default WAV_ENCODER_CONFIG (C macro), 128
digital living network alliance, 349
digital media renderer, 349
digital signal processor, 349
digital-to-analog converter, 349
dispatcher, 349
display_destroy (C++ function), 165
display_pattern_battery_charging (C++ enumerator), 167
display_pattern_battery_full (C++ enumerator), 167
display_pattern_battery_low (C++ enumerator), 167
Index 367
dram_list_reset (C++ function), 101

dram_list_save (C++ function), 100

dram_list_show (C++ function), 102

DSP, 349

DuerOS, 350

dueros_service_create (C++ function), 164
dueros_service_state_get (C++ function), 164
dueros_voice_cancel (C++ function), 165
dueros_voice_upload (C++ function), 164

E

echo, 350

echo reference signal, 350

ECM, 350

el_i0_func (C++ type), 37
electret condenser microphone, 350

element, 350

ELEMENT_SUB_TYPE_OFFSET (C macro), 54

EMBED_FLASH_STREAM_BUF_SIZE (C macro), 92

EMBED_FLASH_STREAM_CFG_DEFAULT (C macro), 92

embed_flash_stream_cfg_t (C++ class), 91

embed_flash_stream_cfg_t::buf_sz (C++ member), 92

embed_flash_stream_cfg_t::external_stack (C++ member), 92

embed_flash_stream_cfg_t::out_rb_size (C++ member), 92

embed_flash_stream_cfg_t::task_core (C++ member), 92

embed_flash_stream_cfg_t::task_prio (C++ member), 92

embed_flash_stream_cfg_t::task_stack (C++ member), 92

EMBED_FLASH_STREAM_EXT_STACK (C macro), 92

EMBED_FLASH_STREAM_INIT (C++ function), 91

EMBED_FLASH_STREAM_RINGBUFFER_SIZE (C macro), 92

embed_flash_stream_set_context (C++ function), 91

EMBED_FLASH_STREAM_TASK_CORE (C macro), 92

EMBED_FLASH_STREAM_TASK_PRI0 (C macro), 92

EMBED_FLASH_STREAM_TASK_STACK (C macro), 92

embed_item_info (C++ class), 92

embed_item_info::address (C++ member), 92

embed_item_info::size (C++ member), 92

encoder, 350

equalizer, 350

equalizerCfg (C++ class), 134

equalizerCfg::channel (C++ member), 134

equalizerCfg::out_rb_size (C++ member), 134
ES8388_ADCPOWER (C macro), 234
ES8388_ADDR (C macro), 234
ES8388_ANAVOLMANAG (C macro), 234
ES8388_CHIPLOPOW1 (C macro), 234
ES8388_CHIPLOPOW2 (C macro), 234
ES8388_CHIPPOWER (C macro), 234
es8388_config_adc_input (C++ function), 232
es8388_config_dac_output (C++ function), 232
es8388_config_fmt (C++ function), 230
es8388_config_i2s (C++ function), 233
ES8388_CTRL_CON1 (C macro), 234
ES8388_CTRL_CON2 (C macro), 234
es8388_ctrl_state (C++ function), 233
ES8388_DACCONTROL10 (C macro), 235
ES8388_DACCONTROL11 (C macro), 235
ES8388_DACCONTROL12 (C macro), 235
ES8388_DACCONTROL13 (C macro), 235
ES8388_DACCONTROL14 (C macro), 235
ES8388_DACCONTROL15 (C macro), 235
ES8388_DACCONTROL16 (C macro), 235
ES8388_DACCONTROL17 (C macro), 235
ES8388_DACCONTROL18 (C macro), 235
ES8388_DACCONTROL19 (C macro), 235
ES8388_DACCONTROL20 (C macro), 234
ES8388_DACCONTROL21 (C macro), 235
ES8388_DACCONTROL22 (C macro), 235
ES8388_DACCONTROL23 (C macro), 235
ES8388_DACCONTROL24 (C macro), 235
ES8388_DACCONTROL25 (C macro), 235
ES8388_DACCONTROL26 (C macro), 235
ES8388_DACCONTROL27 (C macro), 235
ES8388_DACCONTROL28 (C macro), 235
ES8388_DACCONTROL29 (C macro), 235
ES8388_DACCONTROL3 (C macro), 234
ES8388_DACCONTROL30 (C macro), 235
ES8388_DACCONTROL31 (C macro), 234
ES8388_DACCONTROL32 (C macro), 234
ES8388_DACPOWER (C macro), 234
es8388_deinit (C++ function), 230
es8388_get_voice_mute (C++ function), 232
es8388_get_voice_volume (C++ function), 231
es8388_i2s_config_clock (C++ function), 230
es8388_init (C++ function), 230
ES8388_MASTERMODE (C macro), 234
es8388_pa_power (C++ function), 233
es8388_read_all (C++ function), 233
es8388_set_bits_per_sample (C++ function), 231
es8388_set_control (C++ function), 232
es8388_set_voice_mute (C++ function), 232
es8388_set_voice_volume (C++ function), 231
es8388_start (C++ function), 231
es8388_stop (C++ function), 231
es8388_write_reg (C++ function), 233
ESP VoIP, 350
ESP_A2DP_SAMPLE_RATE (C macro), 144
es8388_callback_set (C++ function), 65
es8388_cfg_t (C class), 67
es8388_cfg_t::cb_ctx (C++ member), 67
es8388_cfg_t::cb_func (C++ member), 67
es8388_cfg_t::component_select (C++ member), 67
es8388_cfg_t::evt_que (C++ member), 67
es8388_cfg_t::in_stream_buf_size (C++ member), 67
es8388_cfg_t::out_stream_buf_size (C++ member), 67
es8388_cfg_t::prefer_type (C++ member), 67
es8388_cfg_t::resample_rate (C++ member), 67
es8388_cfg_t::task_prio (C++ member), 67
es8388_cfg_t::task_stack (C++ member), 67
es8388_cfg_t::vol_get (C++ member), 67
es8388_cfg_t::vol_handle (C++ member), 67
es8388_cfg_t::vol_set (C++ member), 67
es8388_codec_lib_add (C++ function), 59
es8388_codec_lib_query (C++ function), 59
ESP_AUDIO_COMPONENT_SELECT_ALC (C macro), 69
ESP_AUDIO_COMPONENT_SELECT_DEFAULT (C macro), 69
ESP_AUDIO_COMPONENT_SELECT_EQUALIZER (C macro), 69
es8388_create (C++ function), 58
es8388_destroy (C++ function), 58
es8388_duration_get (C++ function), 65
es8388_eq_gain_get (C++ function), 62
es8388_eq_gain_set (C++ function), 62
es8388_event_callback (C++ type), 56
es8388_event_queue_set (C++ function), 66
es8388_handle_t (C++ type), 69
es8388_info_get (C++ function), 65
es8388_info_set (C++ function), 65
es8388_info_t (C class), 68
es8388_info_t::audio_speed (C++ member), 68
es8388_info_t::codec_el (C++ member), 68
esp_audio_info_t::codec_info (C++ member), 68
esp_audio_info_t::filter_el (C++ member), 68
esp_audio_info_t::in_el (C++ member), 68
esp_audio_info_t::in_stream_total_size (C++ member), 68
esp_audio_info_t::out_el (C++ member), 68
esp_audio_info_t::st (C++ member), 68
esp_audio_info_t::time_pos (C++ member), 68
esp_audio_input_stream_add (C++ function), 58
esp_audio_media_type_set (C++ function), 65
esp_audio_music_info_get (C++ function), 65
esp_audio_music_info_t (C++ class), 68
esp_audio_music_info_t::bits (C++ member), 69
esp_audio_music_info_t::bps (C++ member), 69
esp_audio_music_info_t::channels (C++ member), 69
esp_audio_music_info_t::codec_fmt (C++ member), 69
esp_audio_music_info_t::sample_rates (C++ member), 69
esp_audio_output_stream_add (C++ function), 58
esp_audio_pause (C++ function), 61
esp_audio_play (C++ function), 59
ESP_AUDIO_PLAY_SPEED_0_50 (C++ enumerator), 59
ESP_AUDIO_PLAY_SPEED_0_75 (C++ enumerator), 59
ESP_AUDIO_PLAY_SPEED_1_00 (C++ enumerator), 59
ESP_AUDIO_PLAY_SPEED_1_25 (C++ enumerator), 59
ESP_AUDIO_PLAY_SPEED_1_50 (C++ enumerator), 59
ESP_AUDIO_PLAY_SPEED_1_75 (C++ enumerator), 59
ESP_AUDIO_PLAY_SPEED_2_00 (C++ enumerator), 59
ESP_AUDIO_PLAY_SPEED_MAX (C++ enumerator), 70
esp_audio_play_speed_t (C++ enum), 69
ESP_AUDIO_PLAY_SPEED_UNKNOW (C++ enumerator), 69
esp_audio_play_timeout_set (C++ function), 66
esp_audio_pos_get (C++ function), 64
ESP_AUDIO_PREFER_MEM (C++ enumerator), 57
ESP_AUDIO_PREFER_SPEED (C++ enumerator), 57

esp_audio_prefer_t (C++ enum), 57
esp_audio_prefer_type_get (C++ function), 66
esp_audio_resume (C++ function), 62
esp_audio_seek (C++ function), 65
esp_audio_setup (C++ function), 65
esp_audio_setup_t (C++ class), 67
esp_audio_setup_t::set_channel (C++ member), 68
esp_audio_setup_t::set_codec (C++ member), 68
esp_audio_setup_t::set_in_stream (C++ member), 68
esp_audio_setup_t::set_out_stream (C++ member), 68
esp_audio_setup_t::set_pos (C++ member), 68
esp_audio_setup_t::set_sample_rate (C++ member), 68
esp_audio_setup_t::set_time (C++ member), 68
esp_audio_setup_t::set_type (C++ member), 68
esp_audio_setup_t::set_uri (C++ member), 68
esp_audio_speed_get (C++ uri), 68
esp_audio_speed_idx_to_float (C++ function), 63
esp_audio_speed_set (C++ function), 63
esp_audio_state_get (C++ function), 64
esp_audio_state_t (C++ class), 55
esp_audio_state_t::err_msg (C++ member), 55
esp_audio_state_t::media_src (C++ member), 55
esp_audio_state_t::status (C++ member), 55
esp_audio_status_t (C++ member), 56
esp_audio_stop (C++ function), 61
esp_audio_sync_play (C++ function), 60
esp_audio_time_get (C++ function), 64
esp_audio_vol_get (C++ function), 64
esp_audio_vol_set (C++ function), 63
ESP_ERR_AUDIO_ALREADY_EXISTS (C++ enumerator), 56
ESP_ERR_AUDIO_BASE (C macro), 55
ESP_ERR_AUDIO_CLOSE (C++ enumerator), 56
ESP_ERR_AUDIO_FAIL (C++ enumerator), 56
ESP_ERR_AUDIO_HAL_FAIL (C++ enumerator), 56
ESP_ERR_AUDIO_INPUT (C++ enumerator), 56
ESP_ERR_AUDIO_INVALID_PARAMETER (C++ enumerator), 56
ESP_ERR_AUDIO_INVALID_PATH (C++ enumerator), 56
ESP_ERR_AUDIO_INVALID_URI (C++ enumerator), 56
ESP_ERR_AUDIO_LINK_FAIL (C++ enumerator), 56
ESP_ERR_AUDIO_MEMORY_LACK (C++ enumerator), 56
ESP_ERR_AUDIO_NO_CODEC (C++ enumerator), 56
ESP_ERR_AUDIO_NO_ERROR (C++ enumerator), 56
ESP_ERR_AUDIO_NO_INPUT_STREAM (C++ enumerator), 56
ESP_ERR_AUDIO_NO_OUTPUT_STREAM (C++ enumerator), 56
ESP_ERR_AUDIO_NOT_READY (C++ enumerator), 56
ESP_ERR_AUDIO_NOT_SUPPORT (C++ enumerator), 56
ESP_ERR_AUDIO_OUTPUT (C++ enumerator), 56
ESP_ERR_AUDIO_OUT_OF_RANGE (C++ enumerator), 56
ESP_ERR_AUDIO_OUTPUT (C++ enumerator), 56
ESP_ERR_AUDIO_PROCESS (C++ enumerator), 56
ESP_ERR_AUDIO_STOP_BY_USER (C++ enumerator), 56
ESP_ERR_AUDIO_TIMEOUT (C++ enumerator), 56
ESP_ERR_AUDIO_UNKNOWN (C++ enumerator), 56
ESP_ERR_FS_OTA_BASE (C macro), 164
ESP_ERR_FS_OTA_IN_PROGRESS (C macro), 164
esp_fs_ota (C++ function), 161
esp_fs_ota_begin (C++ function), 162
esp_fs_ota_config_t::buffer_size (C++ member), 163
esp_fs_ota_config_t::path (C++ member), 164
esp_fs_ota_finish (C++ function), 163
esp_fs_ota_get_image_len_read (C++ function), 163
esp_fs_ota_get_img_desc (C++ function), 163
esp_fs_ota_handle_t (C++ type), 164
esp_fs_ota_perform (C++ function), 162
esp_periph_config_t (C++ class), 198
esp_periph_config_t::extern_stack (C++ member), 198
esp_periph_config_t::task_core (C++ member), 198
esp_periph_config_t::task_prio (C++ member), 198
esp_periph_config_t::task_stack (C++ member), 198
esp_periph_create (C++ function), 194
esp_periph_destroy (C++ function), 197
esp_periph_event (C++ class), 198
esp_periph_event::cb (C++ member), 198
esp_periph_event::iface (C++ member), 198
esp_periph_event::user_ctx (C++ member), 198
esp_periph_event_handle_t (C++ type), 199
esp_periph_event_t (C++ type), 199
esp_periph_func (C++ type), 199
esp_periph_get_data (C++ function), 196
esp_periph_get_id (C++ function), 197
esp_periph_get_state (C++ function), 196
esp_periph_handle_t (C++ type), 199
esp_periph_id_t (C++ enum), 199
esp_periph_init (C++ function), 197
esp_periph_register_on_events (C++ function), 198
esp_periph_remove_from_set (C++ function), 193
esp_periph_run (C++ function), 197
esp_periph_run_func (C++ type), 199
esp_periph_send_cmd (C++ function), 195
esp_periph_send_cmd_from_isr (C++ function), 195
esp_periph_send_event (C++ function), 195
esp_periph_set_change_waiting_time (C++ function), 193
esp_periph_set_data (C++ function), 196
esp_periph_set_destroy (C++ function), 191
esp_periph_set_function (C++ function), 194
esp_periph_set_get_by_id (C++ function), 191
esp_periph_set_get_event_iface (C++ function), 192
esp_periph_set_get_queue (C++ function), 192
esp_periph_set_handle_t (C++ function), 199
esp_periph_set_id (C++ function), 197
esp_periph_set_init (C++ function), 191
esp_periph_set_list_destroy (C++ function), 193
esp_periph_set_list_init (C++ function), 192
esp_periph_set_list_run (C++ function), 193
esp_periph_set_register_callback (C++ function), 192
esp_periph_set_stop_all (C++ function), 191
esp_periph_start (C++ function), 194
esp_periph_start_timer (C++ function), 195
esp_periph_state_t (C++ enum), 200
esp_periph_stop (C++ function), 194
esp_periph_stop_timer (C++ function), 196
touch_pad_sel_t (C++ enum), 210
esp_wifi_set_listen_interval (C++ function), 201
esp_wifi_setting_create (C++ function), 151
esp_wifi_setting_destroy (C++ function), 151
esp_wifi_setting_get_data (C++ function), 152
esp_wifi_setting_handle_t (C++ type), 153
esp_wifi_setting_info_notify (C++ function), 152
esp_wifi_setting_register_function (C++ function), 151
esp_wifi_setting_register_notify_handler (C++ function), 151
esp_wifi_setting_set_data (C++ function), 152
esp_wifi_setting_start (C++ function), 152
esp_wifi_setting_stop (C++ function), 152
esp_wifi_setting_teardown (C++ function), 153
event_cb_func (C++ type), 37

F
fast Fourier transform, 350
FatFs, 350
FatFs stream, 350
FATFS_STREAM_BUF_SIZE (C macro), 75
FATFS_STREAM_CFG_DEFAULT (C macro), 75
fatfs_stream_cfg_t (C++ class), 74
fatfs_stream_cfg_t::buf_sz (C++ member), 74
fatfs_stream_cfg_t::ext_stack (C++ member), 74
fatfs_stream_cfg_t::out_rb_size (C++ member), 74
fatfs_stream_cfg_t::task_core (C++ member), 74
fatfs_stream_cfg_t::task_prio (C++ member), 74
fatfs_stream_cfg_t::task_stack (C++ member), 74
fatfs_stream_cfg_t::type (C++ member), 74
fatfs_stream_cfg_t::write_header (C++ member), 74
fatfs_stream_init (C++ function), 74
FATFS_STREAM_RINGBUFFER_SIZE (C macro), 75
FATFS_STREAM_TASK_CORE (C macro), 75
FATFS_STREAM_TASK_PRIO (C macro), 75
FATFS_STREAM_TASK_STACK (C macro), 75
FB, 350
FFT, 350
FLAC, 350
flac_decoderCfg_t (C++ class), 121
flac_decoderCfg_t::out_rb_size (C++ member), 122
flac_decoderCfg_t::stack_in_ext (C++ member), 122
flac_decoderCfg_t::task_core (C++ member), 122
flac_decoderCfg_t::task_prio (C++ member), 122
flac_decoderCfg_t::task_stack (C++ member), 122
flac_decoder_init (C++ function), 121
FLAC_DECODER_RINGBUFFER_SIZE (C macro), 122
FLAC_DECODER_TASK_CORE (C macro), 122
FLAC_DECODER_TASK_PRIO (C macro), 122
FLAC_DECODER_TASK_STACK_SIZE (C macro), 122
flash_list_choose (C++ function), 105
flash_list_create (C++ function), 105
flash_list_current (C++ function), 104
flash_list_destroy (C++ function), 106
flash_list_exist (C++ function), 105
flash_list_get_url_id (C++ function), 105
flash_list_get_url_num (C++ function), 105
flash_list_next (C++ function), 104
flash_list_prev (C++ function), 104
flash_list_reset (C++ function), 105
flash_list_save (C++ function), 104
flexible pipeline, 350
FPS, 350
frames per second, 350
frequency response, 350
full band, 350
G
get_input_key_service_state (C++ function), 146
H
HAL, 351
Hands-Free, 350
Hands-Free Audio Gateway, 351
Hands-Free Profile, 351
Hands-Free Unit, 351
hardware abstraction layer, 351
headset, 351
HF, 351
HFP, 351
HFP-AG, 351
Hi-Fi speaker, 351
High Frequency, 351
high-fidelity microphone, 351
HLS, 351
HTTP Live Streaming, 351
HTTP stream, 351
HTTP_STREAM_CFG_DEFAULT (C macro), 78
http_stream_cfg_t (C++ class), 78
http_stream_cfg_t::auto_connect_next_track (C++ member), 77
http_stream_cfg_t::cert_pem (C++ member), 77
http_stream_cfg_t::crt_bundle_attach (C++ member), 77
http_stream_cfg_t::enable_playlist_parser (C++ member), 77
http_stream_cfg_t::out_rb_size (C++ member), 78
http_stream_cfg_t::task_core (C++ member), 77
http_stream_cfg_t::task_prio (C++ member), 77
http_stream_cfg_t::task_stack (C++ member), 77
Index 373
http_stream_cfg_t::event_handle (C++ member), 77
http_stream_cfg_t::multi_out_num (C++ member), 77
http_stream_cfg_t::out_rb_size (C++ member), 77
http_stream_cfg_t::request_range_size (C++ member), 78
http_stream_cfg_t::request_size (C++ member), 78
http_stream_cfg_t::stack_in_ext (C++ member), 77
http_stream_cfg_t::task_core (C++ member), 77
http_stream_cfg_t::task_prio (C++ member), 77
http_stream_cfg_t::task_stack (C++ member), 77
http_stream_cfg_t::type (C++ member), 77
http_stream_cfg_t::user_data (C++ member), 77
http_stream_event_handle_t (C++ type), 78
http_stream_event_id_t (C++ enum), 78
http_stream_event_msg_t (C++ class), 76
http_stream_event_msg_t::buffer (C++ member), 77
http_stream_event_msg_t::buffer_len (C++ member), 77
http_stream_event_msg_t::el (C++ member), 77
http_stream_event_msg_t::event_id (C++ member), 77
http_stream_event_msg_t::http_client (C++ member), 77
http_stream_event_msg_t::user_data (C++ member), 77
http_stream_fetch_again (C++ function), 76
HTTP_STREAM_FINISH_PLAYLIST (C++ enumerator), 78
HTTP_STREAM_FINISH_REQUEST (C++ enumerator), 78
HTTP_STREAM_FINISH_TRACK (C++ enumerator), 78
http_stream_init (C++ function), 75
http_stream_next_track (C++ function), 75
HTTP_STREAM_ON_REQUEST (C++ enumerator), 78
HTTP_STREAM_ON_RESPONSE (C++ enumerator), 78
HTTP_STREAM_POST_REQUEST (C++ enumerator), 78
HTTP_STREAM_PRE_REQUEST (C++ enumerator), 78
HTTP_STREAM_RESOLVE_ALL_TRACKS (C++ enumerator), 78
http_stream_restart (C++ function), 76
HTTP_STREAM_RINGBUFFER_SIZE (C macro), 78
http_stream_set_server_cert (C macro), 78
HTTP_STREAM_TASK_CORE (C macro), 78
HTTP_STREAM_TASK_PRIO (C macro), 78
HTTP_STREAM_TASK_STACK (C macro), 78
I2S stream, 351
i2s_alc_volume_get (C++ function), 80
i2s_alc_volume_set (C++ function), 79
I2S_STREAM_BUF_SIZE (C macro), 81
I2S_STREAM_CFG_DEFAULT (C macro), 81
i2s_stream_cfg_t (C++ class), 80
i2s_stream_cfg_t::buffer_len (C++ member), 81
i2s_stream_cfg_t::i2s_config (C++ member), 81
i2s_stream_cfg_t::i2s_port (C++ member), 80
i2s_stream_cfg_t::multi_out_num (C++ member), 81
i2s_stream_cfg_t::need_expand (C++ member), 81
i2s_stream_cfg_t::out_rb_size (C++ member), 81
i2s_stream_cfg_t::stack_in_ext (C++ member), 81
i2s_stream_cfg_t::task_core (C++ member), 81
i2s_stream_cfg_t::task_prio (C++ member), 81
i2s_stream_cfg_t::task_stack (C++ member), 81
i2s_stream_cfg_t::type (C++ member), 80
i2s_stream_cfg_t::uninstall_drv (C++ member), 81
i2s_stream_cfg_t::use_alc (C++ member), 80
i2s_stream_cfg_t::volume (C++ member), 80
i2s_stream_init (C++ function), 79
I2S_STREAM_INTERNAL_DAC_CFG_DEFAULT (C macro), 81
I2S_STREAM_RINGBUFFER_SIZE (C macro), 81
i2s_stream_set_clk (C++ function), 79
i2s_stream_sync_delay (C++ function), 80
I2S_STREAM_TASK_CORE (C macro), 81
I2S_STREAM_TASK_PRIO (C macro), 81
I2S_STREAM_TASK_STACK (C macro), 81
I2S_STREAM_TX_PDM_CFG_DEFAULT (C macro), 81
374 Index
INPUT_KEY_SERVICE_ACTION_CLICK (C++ enumerator), 147
INPUT_KEY_SERVICE_ACTION_CLICK_RELEASE (C++ enumerator), 147
input_key_service_action_id_t (C++ enum), 147
INPUT_KEY_SERVICE_ACTION_PRESS (C++ enumerator), 147
INPUT_KEY_SERVICE_ACTION_PRESS_RELEASE (C++ enumerator), 147
INPUT_KEY_SERVICE_ACTION_UNKNOWN (C++ enumerator), 147
input_key_service_add_key (C++ function), 146
input_key_service_cfg_t (C++ class), 146
input_key_service_cfg_t::based_cfg (C++ member), 147
input_key_service_cfg_t::handle (C++ member), 147
input_key_service_create (C++ function), 146
INPUT_KEY_SERVICE_DEFAULT_CONFIG (C macro), 147
input_key_service_info_t (C++ class), 146
input_key_service_info_t::act_id (C++ member), 146
input_key_service_info_t::type (C++ member), 146
input_key_service_info_t::user_id (C++ member), 146
INPUT_KEY_SERVICE_TASK_ON_CORE (C macro), 147
INPUT_KEY_SERVICE_TASK_PRIORITY (C macro), 147
INPUT_KEY_SERVICE_TASK_STACK_SIZE (C macro), 147
INPUT_KEY_USER_ID_BATTERY_CHARGING (C++ enumerator), 148
INPUT_KEY_USER_ID_CAPTURE (C++ enumerator), 148
INPUT_KEY_USER_ID_COLOR (C++ enumerator), 148
INPUT_KEY_USER_ID_MAX (C++ enumerator), 148
INPUT_KEY_USER_ID_MODE (C++ enumerator), 148
INPUT_KEY_USER_ID_MSG (C++ enumerator), 148
INPUT_KEY_USER_ID_MUTE (C++ enumerator), 148
INPUT_KEY_USER_ID_PLAY (C++ enumerator), 148
INPUT_KEY_USER_ID_REC (C++ enumerator), 148
input_key_user_id_t (C++ enum), 147
INPUT_KEY_USER_ID_UNKNOWN (C++ enumerator), 148
INPUT_KEY_USER_ID_WAKEUP (C++ enumerator), 148
insertion loss, 351
Internet of Things, 351
Internet radio, 351
IoT, 351
IS31FL3216_CH_NUM (C macro), 219
IS31FL3216_STATE_BY_AUDIO (C++ enumerator), 219
IS31FL3216_STATE_FLASH (C++ enumerator), 219
IS31FL3216_STATE_OFF (C++ enumerator), 219
IS31FL3216_STATE_ON (C++ enumerator), 219
IS31FL3216_STATE_SHIFT (C++ enumerator), 219
IS31FL3216_STATE_UNKNOWN (C++ enumerator), 219
JPEG, 351
JPG, 351
L
led_bar_aw2013_deinit (C++ function), 168
led_bar_aw2013_init (C++ function), 167
led_bar_aw2013_pattern (C++ function), 167
led_bar_aw2013_set_blink_time (C++ function), 167
led_bar_is31x_deinit (C++ function), 168
led_bar_is31x_init (C++ function), 168
led_bar_is31x_pattern (C++ function), 168
led_bar_is31x_set_blink_time (C++ function), 168
led_bar_is31x_set_blink_time (C++ function), 168
led_bar_is31x_set_blink_time (C++ function), 168
led_bar_ws2812_deinit (C++ function), 169
led_bar_ws2812_init (C++ function), 170
led_bar_ws2812_handle_t (C++ type), 170
led_bar_ws2812_pattern (C++ function), 169
led_bar_ws2812_pattern (C++ function), 169
led_bar_ws2812_pattern (C++ function), 169
led_indicator_deinit (C++ function), 170
led_indicator_handle_t (C++ type), 171
led_indicator_init (C++ function), 170
led_indicator_pattern (C++ function), 170
Light and Versatile Graphics Library, 351
low-pass filter, 352
LVGL, 352
M
M3UB, 352
M4A, 352
mass production, 352
maximum output power, 352
MCLK, 352
media_source_type_t (C++ enum), 57
MEDIA_SRC_TYPE_MUSIC_A2DP (C++ enumerator), 57
MEDIA_SRC_TYPE_MUSIC_BASE (C++ enumerator), 57
MEDIA_SRC_TYPE_MUSIC_DLNA (C++ enumerator), 57
MEDIA_SRC_TYPE_MUSIC_FLASH (C++ enumerator), 57
MEDIA_SRC_TYPE_MUSIC_HTTP (C++ enumerator), 57
MEDIA_SRC_TYPE_MUSIC_MAX (C++ enumerator), 57
MEDIA_SRC_TYPE_MUSIC_RAW (C++ enumerator), 57
MEDIA_SRC_TYPE_MUSIC_SD (C++ enumerator), 57
MEDIA_SRC_TYPE_MUSIC_NULL (C++ enumerator), 57
MEDIA_SRC_TYPE_reserve_BASE (C++ enumerator), 57
MEDIA_SRC_TYPE_RESERVE_MAX (C++ enumerator), 57
MEDIA_SRC_TYPE_TONE_BASE (C++ enumerator), 57
MEDIA_SRC_TYPE_TONE_FLASH (C++ enumerator), 57
MEDIA_SRC_TYPE_TONE_HTTP (C++ enumerator), 57
MEDIA_SRC_TYPE_TONE_MAX (C++ enumerator), 57
MEDIA_SRC_TYPE_TONE_SD (C++ enumerator), 57
mem_assert (C macro), 54
MEDS mic, 352
MFCC, 352
mic, 352
micro-electro-mechanical systems microphone, 352
microphone, 352
microphone gain, 352
microphone hole, 352
microSD card, 352
MP3, 352
mp3_decoder_cfg_t (C++ class), 123
mp3_decoder_cfg_t::out_rb_size (C++ member), 123
mp3_decoder_cfg_t::stack_in_ext (C++ member), 123
mp3_decoder_cfg_t::task_core (C++ member), 123
mp3_decoder_cfg_t::task_prio (C++ member), 123
mp3_decoder_cfg_t::task_stack (C++ member), 123
mp3_decoder_init (C++ function), 123
MP3_DECODER_RINGBUFFER_SIZE (C macro), 123
MP3_DECODER_TASK_CORE (C macro), 123
MP3_DECODER_TASK_PRIO (C macro), 123
MP3_DECODER_TASK_STACK_SIZE (C macro), 123
MP4, 352
multi-room, 352
Multi-Room Music, 352
MultiNet, 352
N
narrowband, 352
NB, 353
NimBLE, 353
noise criteria curve, 353
noise floor, 353
noise rating curve, 353
noise suppression, 353
non-volatile storage, 353
NS, 353
NVS, 353
O
OGG, 353
ogg_decoder_cfg_t (C++ class), 124
ogg_decoder_cfg_t::out_rb_size (C++ member), 124
ogg_decoder_cfg_t::stack_in_ext (C++ member), 124
ogg_decoder_cfg_t::task_core (C++ member), 124
ogg_decoder_cfg_t::task_prio (C++ member), 124
ogg_decoder_cfg_t::task_stack (C++ member), 124
ogg_decoder_init (C++ function), 124
OGG_DECODER_RINGBUFFER_SIZE (C macro), 124
OGG_DECODER_TASK_CORE (C macro), 124
OGG_DECODER_TASK_PRIO (C macro), 124
OGG_DECODER_TASK_STACK_SIZE (C macro), 124
on_event_iface_func (C++ type), 54
OPUS, 353
opus_decoder_cfg_t (C++ class), 125
opus_decoder_cfg_t::out_rb_size (C++ member), 125
opus_decoder_cfg_t::stack_in_ext (C++ member), 125
opus_decoder_cfg_t::task_core (C++ member), 125
opus_decoder_cfg_t::task_prio (C++ member), 125
opus_decoder_cfg_t::task_stack (C++ member), 125
OPUS_DECODER_RINGBUFFER_SIZE (C macro), 126
OPUS_DECODER_TASK_CORE (C macro), 126
OPUS_DECODER_TASK_PRIO (C macro), 126
OPUS_DECODER_TASK_STACK_SIZE (C macro), 126
ota_app_get_default_proc (C++ function), 160
ota_data_get_default_proc (C++ function), 160
ota_data_image_stream_read (C++ function), 160
ota_data_partition_erase_mark (C++ function), 161
ota_data_partition_write (C++ function), 160
ota_get_version_number (C++ function), 161
ota_node_attr_t (C++ class), 158
ota_node_attr_t::cert_pem (C++ member), 158
ota_node_attr_t::label (C++ member), 158
ota_node_attr_t::type (C++ member), 158
ota_node_attr_t::uri (C++ member), 158
ota_result_t (C++ class), 159
ota_result_t::id (C++ member), 159
ota_result_t::result (C++ member), 159
OTA_SERV_ERR_REASON_ERROR_MAGIC_WORD (C++ enumerator), 159
OTA_SERV_ERR_REASON_ERROR_PROJECT_NAME (C++ enumerator), 159
OTA_SERV_ERR_REASON_ERROR_VERSION (C++ enumerator), 159
OTA_SERV_ERR_REASON_FILE_NOT_FOUND (C++ enumerator), 159
OTA_SERV_ERR_REASON_GET_NEW_APP_DESC_FAIL (C++ enumerator), 160
OTA_SERV_ERR_REASON_NO_HIGHER_VERSION (C++ enumerator), 159
OTA_SERV_ERR_REASON_NULL_POINTER (C++ enumerator), 159
OTA_SERV_ERR_REASON_PARTITION_NOT_FOUND (C++ enumerator), 159
OTA_SERV_ERR_REASON_PARTITION_RD_FAIL (C++ enumerator), 160
OTA_SERV_ERR_REASON_PARTITION_WT_FAIL (C++ enumerator), 160
OTA_SERV_ERR_REASON_STREAM_INIT_FAIL (C++ enumerator), 160
OTA_SERV_ERR_REASON_STREAM_RD_FAIL (C++ enumerator), 160
OTA_SERV_ERR_REASON_SUCCESS (C++ enumerator), 159
OTA_SERV_ERR_REASON_UNKNOWN (C++ enumerator), 159
OTA_SERV_ERR_REASON_URL_PARSE_FAIL (C++ enumerator), 159
OTA_SERV_EVENT_TYPE_FINISH (C++ enumerator), 159
OTA_SERV_EVENT_TYPE_RESULT (C++ enumerator), 159
ota_service_config_t (C++ class), 158
ota_service_config_t::cb_ctx (C++ member), 158
ota_service_config_t::evt_cb (C++ member), 158
ota_service_config_t::task_core (C++ member), 158
ota_service_config_t::task_prio (C++ member), 158
ota_service_config_t::task_stack (C++ member), 158
ota_service_create (C++ function), 157
OTA_SERVICE_DEFAULT_CONFIG (C macro), 159
OTA_SERVICE_ERR_REASON_BASE (C macro), 159
ota_service_err_reason_t (C++ enum), 159
ota_service_event_type_t (C++ enum), 159
ota_service_set_upgrade_param (C++ function), 157
ota_upgrade_ops_t (C++ class), 158
ota_upgrade_ops_t::break_after_fail (C++ member), 159
ota_upgrade_ops_t::execute_upgrade (C++ member), 158
ota_upgrade_ops_t::finished_check (C++ member), 158
ota_upgrade_ops_t::need_upgrade (C++ member), 158
ota_upgrade_ops_t::node (C++ member), 158
ota_upgrade_ops_t::prepare (C++ member), 158
ota_upgrade_ops_t::reboot_flag (C++ member), 159
partition_list_choose (C++ function), 108
partition_list_create (C++ function), 106
partition_list_current (C++ function), 107
partition_list_destroy (C++ function), 109
partition_list_exist (C++ function), 107
partition_list_get_url_id (C++ function), 108
partition_list_get_url_num (C++ function), 108
partition_list_next (C++ function), 107
partition_list_prev (C++ function), 107
partition_list_reset (C++ function), 107
partition_list_save (C++ function), 106
partition_list_show (C++ function), 108
periph_adc_button_cfg_t (C++ class), 215
periph_adc_button_cfg_t::arr (C++ member), 215
periph_adc_button_cfg_t::arr_size (C++ member), 215
periph_adc_button_cfg_t::task_cfg (C++ member), 215
PERIPH_ADC_BUTTON_DEFAULT_CONFIG (C++ macro), 215
periph_adc_button_event_id_t (C++ enum), 216
PERIPH_ADC_BUTTON_IDLE (C++ enumerator), 216
periph_adc_button_init (C++ function), 215
PERIPH_ADC_BUTTON_LONG_PRESSED (C++ enumerator), 216
PERIPH_ADC_BUTTON_LONG_RELEASE (C++ enumerator), 216
PERIPH_ADC_BUTTON_PRESSED (C++ enumerator), 216
PERIPH_ADC_BUTTON_RELEASE (C++ enumerator), 216
periph_bluetooth_cancel_discover (C++ function), 143
periph_bluetooth_connect (C++ function), 143
periph_bluetooth_discover (C++ function), 143
periph_bluetooth_fast_forward (C++ function), 142
periph_bluetooth_get_a2dp_sample_rate (C++ function), 143
periph_bluetooth_next (C++ function), 142
periph_bluetooth_pause (C++ function), 141
periph_bluetooth_play (C++ function), 141
periph_bluetooth_prev (C++ function), 142
periph_bluetooth_rewind (C++ function), 142
periph_bluetooth_stop (C++ function), 141
periph_button_cfg_t::gpio_mask (C++ member), 211
periph_button_cfg_t::long_press_time_ms (C++ member), 211
periph_button_event_id_t (C++ enum), 212
periph_button_init (C++ function), 211
PERIPH_BUTTON_LONG_PRESSED (C++ enumerator), 212
PERIPH_BUTTON_LONG_RELEASE (C++ enumerator), 212
PERIPH_BUTTON_PRESSED (C++ enumerator), 212
PERIPH_BUTTON_RELEASE (C++ enumerator), 212
PERIPH_BUTTON_UNCHANGE (C++ enumerator), 212
periph_console_cfg_t::duty (C++ member), 208
periph_console_cfg_t::buffer_size (C++ member), 208
periph_console_cfg_t::command_num (C++ member), 208
periph_console_cfg_t::commands (C++ member), 208
periph_console_cfg_t::prompt_string (C++ member), 208
periph_console_cfg_t::task_prio (C++ member), 208
periph_console_cfg_t::task_stack (C++ member), 208
periph_console_cmd_t::func (C++ member), 208
periph_console_cmd_t::help (C++ member), 208
periph_console_cmd_t::id (C++ member), 208
periph_console_init (C++ function), 208
PERIPH_ID_ADC (C++ enumerator), 199
PERIPH_ID_ADC_BTN (C++ enumerator), 199
PERIPH_ID_AUXIN (C++ enumerator), 199
PERIPH_ID_AW2013 (C++ enumerator), 200
PERIPH_ID_BLUETOOTH (C++ enumerator), 199
PERIPH_ID_BUTTON (C++ enumerator), 199
PERIPH_ID_CONSOLE (C++ enumerator), 199
PERIPH_ID_FLASH (C++ enumerator), 199
PERIPH_ID_GPION_ISR (C++ enumerator), 199
PERIPH_ID_IS31FL3216 (C++ enumerator), 199
PERIPH_ID_LCD (C++ enumerator), 200
PERIPH_ID_LED (C++ enumerator), 199
PERIPH_ID_SDCARD (C++ enumerator), 199
PERIPH_ID_SPIFFS (C++ enumerator), 199
PERIPH_ID_TOUCH (C++ enumerator), 199
PERIPH_ID_WIFI (C++ enumerator), 199
PERIPH_ID_WS2812 (C++ enumerator), 199
PERIPH_IS31_SHIFT_MODE_ACC (C++ enumerator), 219
PERIPH_IS31_SHIFT_MODE_SINGLE (C++ enumerator), 219
PERIPH_IS31_SHIFT_MODE_UNKNOWN (C++ enumerator), 219
periph_is31f13216_cfg_t::duty (C++ member), 209
periph_is31f13216_cfg_t::is31f13216_pattern (C++ member), 219
periph_is31f13216_cfg_t::state (C++ member), 219
periph_is31f13216_init (C++ function), 216
periph_is31f13216_set_act_time (C++ function), 218
periph_is31f13216_set_blink_pattern (C++ function), 217
periph_is31f13216_set_duty (C++ function), 217
periph_is31f13216_set_duty_step (C++ function), 217
periph_is31f13216_set_interval (C++ function), 217
periph_is31f13216_set_light_on_num (C++ member), 219
periph_is31f13216_set_light_off_num (C++ member), 219
periph_is31f13216_set_light_on_num (C++ member), 219
periph_is31f13216_set_light_off_num (C++ member), 219
periph_is31fl3216_set_light_on_num (C++ member), 219
periph_is31fl3216_set_light_off_num (C++ member), 219
periph_is31fl3216_set_light_on_num (C++ member), 219
periph_is31fl3216_set_light_off_num (C++ member), 219
periph_is31fl3216_set_light_on_num (C++ member), 219
periph_is31fl3216_set_light_off_num (C++ member), 219
periph_is31fl3216_set_light_on_num (C++ member), 219
periph_is31fl3216_set_light_off_num (C++ member), 219
periph_is31fl3216_set_light_on_num (C++ member), 219
periph_is31fl3216_set_light_off_num (C++ member), 219
periph_is31fl3216_set_light_on_num (C++ member), 219
periph_is31fl3216_set_light_off_num (C++ member), 219
periph_is31fl3216_set_light_on_num (C++ member), 219
periph_is31fl3216_set_light_off_num (C++ member), 219
PERIPH_STATE_INIT (C++ enumerator), 200
PERIPH_STATE_NULL (C++ enumerator), 200
PERIPH_STATE_PAUSE (C++ enumerator), 200
PERIPH_STATE_RUNNING (C++ enumerator), 200
PERIPH_STATE_STATUS_MAX (C++ enumerator), 200
PERIPH_STATE_STOPPING (C++ enumerator), 200
periph_touch_init (C++ function), 199
periph_touch_cfg_t (C++ class), 210
periph_touch_cfg_t::long_tap_time_ms (C++ member), 210
periph_touch_cfg_t::touch_mask (C++ member), 210
periph_touch_cfg_t::touch_mask (C++ member), 210
periph_touch_event_id_t (C++ enum), 210
periph_touch_init (C++ function), 209
PERIPH_TOUCH_LONG_RELEASE (C++ enumerator), 210
PERIPH_TOUCH_LONG_TAP (C++ enumerator), 210
PERIPH_TOUCH_RELEASE (C++ enumerator), 210
PERIPH_TOUCH_TAP (C++ enumerator), 210
PERIPH_TOUCH_UNCHANGE (C++ enumerator), 210
periph_wifi_cfg_t (C++ class), 202
periph_wifi_cfg_t::disable_auto_reconnect (C++ member), 203
periph_wifi_cfg_t::password (C++ member), 203
periph_wifi_cfg_t::reconnect_timeout_ms (C++ member), 203
periph_wifi_cfg_t::ssid (C++ member), 203
periph_wifi_cfg_t::wpa2_e_cfg (C++ member), 203
PERIPH_WIFI_CONFIG_DONE (C++ enumerator), 203
PERIPH_WIFI_CONFIG_ERROR (C++ enumerator), 203
periph_wifi_config_mode_t (C++ enum), 203
periph_wifi_config_start (C++ function), 201
periph_wifi_config_wait_done (C++ function), 201
PERIPH_WIFI_CONNECTED (C++ enumerator), 203
PERIPH_WIFI_CONNECTING (C++ enumerator), 203
PERIPH_WIFI_DISCONNECTED (C++ enumerator), 203
PERIPH_WIFI_ERROR (C++ enumerator), 203
periph_wifi_init (C++ function), 200
periph_wifi_is_connected (C++ function), 201
PERIPH_WIFI_SETTING (C++ enumerator), 203
periph_wifi_state_t (C++ enum), 203
PERIPH_WIFI_UNCHANGE (C++ enumerator), 203
periph_wifi_wait_for_connected (C++ function), 200
periph_wifi_wait_for_connected (C++ function), 200
periph_wpa2_enterprise_cfg_t (C++ class), 202
periph_wpa2_enterprise_cfg_t::ca_pem_end (C++ member), 202
periph_wpa2_enterprise_cfg_t::ca_pem_start (C++ member), 202
periph_wpa2_enterprise_cfg_t::diaseble_wpa2_e (C++ member), 202
periph_wpa2_enterprise_cfg_t::eap_id (C++ member), 202
periph_wpa2_enterprise_cfg_t::eap_method (C++ member), 202
periph_wpa2_enterprise_cfg_t::eap_password (C++ member), 202
periph_wpa2_enterprise_cfg_t::eap_username (C++ member), 202
periph_wpa2_enterprise_cfg_t::wpa2_e_cert_end (C++ member), 202
periph_wpa2_enterprise_cfg_t::wpa2_e_cert_start (C++ member), 202
periph_wpa2_enterprise_cfg_t::wpa2_e_key_end (C++ member), 202
periph_wpa2_enterprise_cfg_t::wpa2_e_key_start (C++ member), 202
pixel, 353
playback, 353
playlist_add (C++ function), 109
playlist_checkout_by_id (C++ function), 109
playlist_choose (C++ function), 112
playlist_create (C++ function), 109
playlist_destroy (C++ function), 113
PLAYLIST_DRAM (C++ enumerator), 114
playlist_exist (C++ function), 113
PLAYLIST_FLASH (C++ enumerator), 114
playlist_get_current_list_id (C++ function), 110
playlist_get_current_list_type (C++ function), 110
playlist_get_current_list_url (C++ function), 110
playlist_get_current_list_url_id (C++ function), 111
playlist_get_current_list_url_num (C++ function), 111
playlist_get_list_num (C++ function), 110
playlist_handle_t (C++ type), 114
playlist_next (C++ function), 111
playlist_operation_t (C++ class), 113
playlist_operation_t::choose (C++ member), 113
playlist_operation_t::current (C++ member), 114
playlist_operation_t::destroy (C++ mem-
playlist_operation_t::exist (C++ member), 114
playlist_operation_t::get_url_id (C++ member), 114
playlist_operation_t::get_url_num (C++ member), 114
playlist_operation_t::next (C++ member), 113
playlist_operation_t::prev (C++ member), 113
playlist_operation_t::remove_by_id (C++ member), 114
playlist_operation_t::remove_by_url (C++ member), 114
playlist_operation_t::reset (C++ member), 113
playlist_operation_t::save (C++ member), 113
playlist_operation_t::show (C++ member), 114
playlist_operation_t::type (C++ member), 114
playlist_operator_handle_t (C++ type), 114
playlist_operator_t (C++ class), 114
playlist_operator_t::get_operation (C++ member), 114
playlist_operator_t::playlist (C++ member), 114
PLAYLIST_PARTITION (C++ enumerator), 115
playlist_prev (C++ function), 111
playlist_remove_by_url (C++ function), 112
playlist_remove_by_url_id (C++ function), 112
playlist_reset (C++ function), 112
playlist_save (C++ function), 111
PLAYLIST_SDCARD (C++ enumerator), 114
playlist_show (C++ function), 112
playlist_type_t (C++ enum), 114
PLAYLIST_UNKNOWN (C++ enumerator), 114
process_func (C++ type), 37
programmable_gain_amplifier, 353
protractor, 353
pulse-code_modulation, 353
PWM_CONFIG_RINGBUFFER_SIZE (C macro), 84
PWM_STREAM_BUF_SIZE (C macro), 84
PWM_STREAM_CFG_DEFAULT (C macro), 84
pwm_stream_cfg_t (C++ class), 83
pwm_stream_cfg_t::buffer_len (C++ member), 83
pwm_stream_cfg_t::ext_stack (C++ member), 83
pwm_stream_cfg_t::out_rb_size (C++ member), 83
pwm_stream_cfg_t::pwm_config (C++ member), 83
pwm_stream_cfg_t::task_core (C++ member), 83
pwm_stream_cfg_t::task_prio (C++ member), 83
pwm_stream_cfg_t::task_stack (C++ member), 83
pwm_stream_cfg_t::type (C++ member), 83
PWM_STREAM_GPIO_NUM_LEFT (C macro), 84
PWM_STREAM_GPIO_NUM_RIGHT (C macro), 84
pwm_stream_init (C++ function), 82
PWM_STREAM_RINGBUFFER_SIZE (C macro), 84
pwm_stream_set_clk (C++ function), 82
PWM_STREAM_TASK_CORE (C macro), 84
PWM_STREAM_TASK_PRIO (C macro), 84
PWM_STREAM_TASK_STACK (C macro), 84
R
raw_stream, 353
RAW_STREAM_CFG_DEFAULT (C macro), 85
raw_stream_cfg_t (C++ class), 85
raw_stream_cfg_t::out_rb_size (C++ member), 85
raw_stream_cfg_t::type (C++ member), 85
raw_stream_init (C++ function), 84
raw_stream_read (C++ function), 84
RAW_STREAM_RINGBUFFER_SIZE (C macro), 85
raw_stream_write (C++ function), 85
RB_ABORT (C macro), 223
rb_abort (C++ function), 221
rb_bytes_available (C++ function), 222
rb_bytes_filled (C++ function), 222
rb_create (C++ function), 221
rb_destroy (C++ function), 221
RB_DONE (C macro), 223
rb_done_write (C++ function), 223
RB_FAIL (C macro), 223
rb_get_size (C++ function), 222
RB_OK (C macro), 223
rb_read (C++ function), 222
rb_reset (C++ function), 221
rb_reset_is_done_write (C++ function), 222
RB_TIMEOUT (C macro), 223
rb_unblock_reader (C++ function), 223
rb_write (C++ function), 222
rec_callback (C++ type), 190
rec_close (C++ type), 190
rec_config_t (C++ class), 189
rec_config_t::close (C++ member), 189
rec_config_t::enable_wwe (C++ member), 189
rec_config_t::evt_cb (C++ member), 189
rec_config_t::extension (C++ member), 189
rec_config_t::fetch (C++ member), 189
rec_config_t::one_frame_duration_ms (C++ member), 189
rec_config_t::open (C++ member), 189
rec_config_t::sensitivity (C++ member), 189
rec_config_t::support_encoding (C++ member), 189
rec_config_t::task_core (C++ member), 189
rec_config_t::vad_off_delay_ms (C++ member), 189
rec_config_t::wakep_time_ms (C++ member), 189
rec_engine_create (C++ function), 186
rec_engine_data_read (C++ function), 186
rec_engine_destroy (C++ function), 187
rec_engine_detect_suspen (C++ function), 187
rec_engine_enc_data_read (C++ function), 188
rec_engine_enable (C++ function), 188
rec_engine_mute_enable (C++ function), 188
rec_engine_trigger_start (C++ function), 187
rec_engine_vad_enable (C++ function), 187
rec_event_type_t
REC_EVENT_VAD_STOP
REC_EVENT_VAD_START
rec_fetch (C++ type), 190
REC_ONE_BLOCK_SIZE
rec_get_wakeup_stat (C++ function), 188
rec_get_wakeup_stat (C++ function), 188
rec_open (C++ type), 190
REC_VOICE_SUSPEND_OFF (C++ enumerator), 190
REC_VOICE_SUSPEND_ON (C++ enumerator), 190
rec_voice_suspend_t
SD card, 354
SD_MODE_1_LINE (C++ enumerator), 205
SD_MODE_4_LINE (C++ enumerator), 205
SD_MODE_8_LINE (C++ enumerator), 205
SD_MODE_MAX (C++ enumerator), 205
SD_MODE_SPI (C++ enumerator), 205
sdcard_list_choose (C++ function), 98
sdcard_list_create (C++ function), 97
sdcard_list_current (C++ function), 98
sdcard_list_destroy (C++ function), 99
sdcard_list_exist (C++ function), 98
sdcard_list_get_url_id (C++ function), 99
sdcard_list_get_url_num (C++ function), 99
sdcard_list_next (C++ function), 97
sdcard_list_prev (C++ function), 97
sdcard_list_reset (C++ function), 98
sdcard_list_save (C++ function), 99
sdcard_list_show (C++ function), 97

S

SBC, 354

resample filter, 353
resonant frequency, 354
reverberation, 354
RGB, 354
ring buffer, 354
ringbuf_handle_t (C++ type), 224
RSP_FILTER_BUFFER_BYTE (C macro), 137
rsp_filter_cfg_t::class, 136
rsp_filter_cfg_t::complexity (C++ member), 136
rsp_filter_cfg_t::dest_bits (C++ member), 136
rsp_filter_cfg_t::dest_ch (C++ member), 136
rsp_filter_cfg_t::dest_rate (C++ member), 136
rsp_filter_cfg_t::down_ch_idx (C++ member), 137
rsp_filter_cfg_t::max_indata_bytes (C++ member), 136
rsp_filter_cfg_t::mode (C++ member), 136
rsp_filter_cfg_t::out_len_bytes (C++ member), 136
rsp_filter_cfg_t::out_rb_size (C++ member), 137
rsp_filter_cfg_t::prefer_flag (C++ member), 137
rsp_filter_cfg_t::sina bits (C++ member), 136
rsp_filter_cfg_t::src_ch (C++ member), 136
rsp_filter_cfg_t::src_rate (C++ member), 136
rsp_filter_cfg_t::stack_in_ext (C++ member), 137
rsp_filter_cfg_t::task_core (C++ member), 137
rsp_filter_cfg_t::task_prio (C++ member), 137
rsp_filter_cfg_t::task_stack (C++ member), 137
rsp_filter_cfg_t::type (C++ member), 136
rsp_filter_change_src_info (C++ function), 135
rsp_filter_init (C++ function), 135
RSP_FILTER_RINGBUFFER_SIZE (C macro), 137
rsp_filter_set_src_info (C++ function), 135
RSP_FILTER_TASK_CORE (C macro), 137
RSP_FILTER_TASK_PRIO (C macro), 137
RSP_FILTER_TASK_STACK (C macro), 137
sdcard_scan (C++ function), 96
sdcard_scan_cb_t (C++ type), 96
SDCARD_STATUS_CARD_DETECT_CHANGE (C++ enumerator), 205
SDCARD_STATUS_MOUNT_ERROR (C++ enumerator), 205
SDCARD_STATUS_MOUNTED (C++ enumerator), 205
SDCARD_STATUS_UNKNOWN (C++ enumerator), 205
SDCARD_STATUS_UNMOUNT_ERROR (C++ enumerator), 205
SDCARD_STATUS_UNMOUNTED (C++ enumerator), 205
service_callback (C++ type), 184
service_ctrl (C++ type), 184
service_event_t (C++ class), 183
service_event_t::data (C++ member), 183
service_event_t::len (C++ member), 183
service_event_t::source (C++ member), 183
service_event_t::type (C++ member), 183
SERVICE_STATE_CONNECTED (C++ enumerator), 184
SERVICE_STATE_CONNECTING (C++ enumerator), 184
SERVICE_STATE_IDLE (C++ enumerator), 184
SERVICE_STATE_RUNNING (C++ enumerator), 184
SERVICE_STATE_STOPPED (C++ enumerator), 184
service_state_t (C++ enum), 184
SESSION_STATE_UNKNOWN (C++ enumerator), 184
Session Initiation Protocol, 354
signal-to-echo ratio, 354
signal-to-noise ratio, 354
SIP, 354
smart_config_create (C++ function), 153
SMART_CONFIG_INFO_DEFAULT (C macro), 154
smart_config_info_t (C++ class), 154
smart_config_info_t::type (C++ member), 154
SmartConfig, 354
SNR, 354
sonic, 354
sonic_cfg_t (C++ class), 139
sonic_cfg_t::out_rb_size (C++ member), 139
sonic_cfg_t::sonic_info (C++ member), 139
sonic_cfg_t::stack_in_ext (C++ member), 139
sonic_cfg_t::task_core (C++ member), 139
sonic_cfg_t::task_prio (C++ member), 139
sonic_cfg_t::task_stack (C++ member), 139
sonic_info_t (C++ class), 139
sonic_info_t::channel (C++ member), 139
sonic_info_t::pitch (C++ member), 139
sonic_info_t::resample_linear_interpolate (C++ member), 139
sonic_info_t::samplerate (C++ member), 139
sonic_info_t::speed (C++ member), 139
sonic_init (C++ function), 138
SONIC_RINGBUFFER_SIZE (C macro), 140
sonic_set_info (C++ function), 138
sonic_set_pitch_and_speed_info (C++ function), 138
SONIC_SET_VALUE_FOR_INITIALIZATION (C macro), 140
SONIC_TASK_CORE (C macro), 140
SONIC_TASK_Prio (C macro), 140
SONIC_TASK_STACK (C macro), 140
sound card, 354
sound level meter, 354
sound pickup hole, 354
sound pickup tube, 354
sound transmission loss, 354
source_info_init (C++ function), 131
speech, 354
speech recognition, 354
SPIFFS, 355
SPIFFS stream, 355
SPIFFS_STATUS_MOUNT_ERROR (C++ enumerator), 207
SPIFFS_STATUS_MOUNTED (C++ enumerator), 207
SPIFFS_STATUS_UNKNOWN (C++ enumerator), 207
SPIFFS_STATUS_UNMOUNT_ERROR (C++ enumerator), 207
SPIFFS_STATUS_UNMOUNTED (C++ enumerator), 207
SPIFFS_STREAM_BUF_SIZE (C macro), 87
SPIFFS_STREAM_CFG_DEFAULT (C macro), 87
spiffs_stream_cfg_t (C++ class), 86
spiffs_stream_cfg_t::buf_sz (C++ member), 86
spiffs_stream_cfg_t::out_rb_size (C++ member), 86
spiffs_stream_cfg_t::task_core (C++ member), 86
spiffs_stream_cfg_t::task_prio (C++ member), 86
spiffs_stream_init (C++ function), 86
SPIFFS_STREAM_RINGBUFFER_SIZE (C macro), 87
SPIFFS_STREAM_TASK_CORE (C macro), 87
SPIFFS_STREAM_TASK_Prio (C macro), 87
SPIFFS_STREAM_TASK_STACK (C macro), 87
SR, 354
STL, 355
stream_func (C++ type), 37
subband codec, 355
super wide band, 355
SWB, 355

T

Tape measure, 355
TCP_SERVER_DEFAULT_RESPONSE_LENGTH (C macro), 88
TCP_STREAM_BUF_SIZE (C macro), 88
TCP_STREAM_CFG_DEFAULT (C macro), 89
tcp_stream_cfg_t::event_ctx (C++ class), 88
tcp_stream_cfg_t::event_handler (C++ member), 88
tcp_stream_cfg_t::ext_stack (C++ member), 88
tcp_stream_cfg_t::host (C++ member), 88
tcp_stream_cfg_t::port (C++ member), 88
tcp_stream_cfg_t::task_core (C++ member), 88
tcp_stream_cfg_t::task_prio (C++ member), 88
tcp_stream_cfg_t::task_stack (C++ member), 88
tcp_stream_cfg_t::timeout_ms (C++ member), 88
tcp_stream_cfg_t::type (C++ member), 88
TCP_STREAM_DEFAULT_PORT (C macro), 88
tcp_stream_event_handle_cb (C++ type), 89
tcp_stream_event_msg (C++ class), 87
tcp_stream_event_msg::data (C++ member), 88
tcp_stream_event_msg::data_len (C++ member), 88
tcp_stream_event_msg::sock_fd (C++ member), 88
tcp_stream_event_msg::source (C++ member), 88
tcp_stream_event_msg_t (C++ type), 89
tcp_stream_init (C++ function), 87
TCP_STREAM_STATE_CONNECTED (C++ enumerator), 89
TCP_STREAM_STATE_NONE (C++ enumerator), 89
tcp_stream_status_t (C++ enum), 89
TCP_STREAM_TASK_CORE (C macro), 88
TCP_STREAM_TASK_PRIO (C macro), 88
TCP_STREAM_TASK_STACK (C macro), 88
TERMINATION_TYPE_DONE (C++ enumerator), 57
TERMINATION_TYPE_MAX (C++ enumerator), 57
TERMINATION_TYPE_NOW (C++ enumerator), 57
text-to-speech, 355
THD, 355
timer_callback (C++ type), 199
tolerance, 355
tone, 355
TONE_STREAM_BUF_SIZE (C macro), 90
TONE_STREAM_CFG_DEFAULT (C macro), 90
tone_stream_cfg_t (C++ class), 90
tone_stream_cfg_t::buf_sz (C++ member), 90
tone_stream_cfg_t::event_handler (C++ member), 90
tone_stream_cfg_t::label (C++ member), 90
tone_stream_cfg_t::out_rb_size (C++ member), 90
tone_stream_cfg_t::task_core (C++ member), 90
tone_stream_cfg_t::task_prio (C++ member), 90
tone_stream_cfg_t::task_stack (C++ member), 90
tone_stream_init (C++ function), 89
TONE_STREAM_RINGBUFFER_SIZE (C macro), 90
TONE_STREAM_TASK_CORE (C macro), 90
TONE_STREAM_TASK_PRIO (C macro), 90
TONE_STREAM_TASK_STACK (C macro), 90
TONE_STREAM_USE_DELEGATE (C macro), 90
total harmonic distortion, 355
TOUCH_PAD_SEL0 (C++ enumerator), 210
TOUCH_PAD_SEL1 (C++ enumerator), 210
TOUCH_PAD_SEL2 (C++ enumerator), 210
TOUCH_PAD_SEL3 (C++ enumerator), 210
TOUCH_PAD_SEL4 (C++ enumerator), 210
TOUCH_PAD_SEL5 (C++ enumerator), 210
TOUCH_PAD_SEL6 (C++ enumerator), 210
TOUCH_PAD_SEL7 (C++ enumerator), 210
TOUCH_PAD_SEL8 (C++ enumerator), 210
TOUCH_PAD_SEL9 (C++ enumerator), 210
TTS, 355
TTS_STREAM_BUF_SIZE (C macro), 94
TTS_STREAM_CFG_DEFAULT (C macro), 94
tts_stream_cfg_t (C++ class), 94
tts_stream_cfg_t::buf_sz (C++ member), 94
tts_stream_cfg_t::ext_stack (C++ member), 94
tts_stream_cfg_t::out_rb_size (C++ member), 94
tts_stream_init (C++ function), 94
tts_stream_cfg_t::task_core (C++ member), 94
tts_stream_cfg_t::task_prio (C++ member), 94
tts_stream_cfg_t::task_stack (C++ member), 94

384
tts_stream_cfg_t::type (C++ member), 94
tts_stream_get_speed (C++ function), 93
tts_stream_init (C++ function), 93
TTS_STREAM_RINGBUFFER_SIZE (C macro), 94
tts_stream_set_speed (C++ function), 93
tts_stream_set_strings (C++ function), 93
TTS_STREAM_TASK_CORE (C macro), 94
TTS_STREAM_TASK_Prio (C macro), 94
TTS_STREAM_TASK_STACK (C macro), 94
TTS_VOICE_SPEED_0 (C++ enumerator), 95
TTS_VOICE_SPEED_1 (C++ enumerator), 95
TTS_VOICE_SPEED_2 (C++ enumerator), 95
TTS_VOICE_SPEED_3 (C++ enumerator), 95
TTS_VOICE_SPEED_4 (C++ enumerator), 95
TTS_VOICE_SPEED_5 (C++ enumerator), 95
TTS_VOICE_SPEED_MAX (C++ enumerator), 95
tts_voice_speed_t (C++ enumerator), 95

V
VAD, 355
voice activity detection, 355
VoIP, 355
vol_monitor_create (C++ function), 173
vol_monitor_destroy (C++ function), 173
VOL_MONITOR_EVENT_BAT_FULL (C++ enumerator), 175
VOL_MONITOR_EVENT_BAT_LOW (C++ enumerator), 175
vol_monitor_event_cb (C++ type), 175
VOL_MONITOR_EVENT_FREQ_REPORT (C++ enumerator), 175
vol_monitor_event_t (C++ enum), 175
vol_monitor_handle_t (C++ type), 175
vol_monitor_param_t (C++ class), 174
vol_monitor_param_t::deinit (C++ member), 174
vol_monitor_param_t::init (C++ member), 174
vol_monitor_param_t::read_freq (C++ member), 174
vol_monitor_param_t::report_freq (C++ member), 174
vol_monitor_param_t::user_data (C++ member), 174
vol_monitor_param_t::vol_full_threshold (C++ member), 174
vol_monitor_param_t::vol_get (C++ member), 174
vol_monitor_param_t::vol_low_threshold (C++ member), 175
vol_monitor_set_event_cb (C++ function), 173
vol_monitor_set_report_freq (C++ function), 174
vol_monitor_start_freq_report (C++ function), 173
vol_monitor_stop_freq_report (C++ function), 174

W
wake word, 355
wake word engine, 355
wake-up, 356
WakeNet, 355
wav_decoder_cfg_t (C++ class), 126
wav_decoder_cfg_t::out_rb_size (C++ member), 127
wav_decoder_cfg_t::stack_in_ext (C++ member), 127
wav_decoder_cfg_t::task_core (C++ member), 127
wav_decoder_cfg_t::task_prio (C++ member), 127
wav_decoder_cfg_t::task_stack (C++ member), 127
wav_decoder_init (C++ function), 126
WAV_DECODER_RINGBUFFER_SIZE (C macro), 127
WAV_DECODER_TASK_CORE (C macro), 127
WAV_DECODER_TASK_Prio (C macro), 127
WAV_DECODER_TASK_STACK (C macro), 127
wav_encoder_cfg_t (C++ class), 128
wav_encoder_cfg_t::out_rb_size (C++ member), 128
wav_encoder_cfg_t::stack_in_ext (C++ member), 128
wav_encoder_cfg_t::task_core (C++ member), 128
wav_encoder_cfg_t::task_prio (C++ member), 128
wav_encoder_cfg_t::task_stack (C++ member), 128
wav_encoder_init (C++ function), 127
WAV_ENCODER_RINGBUFFER_SIZE (C macro), 128
WAV_ENCODER_TASK_CORE (C macro), 128
WAV_ENCODER_TASK_Prio (C macro), 128
WAV_ENCODER_TASK_STACK (C macro), 128

WB, 356
wideband, 356
WIFI_CONFIG_AIRKISS (C++ enumerator), 203
WIFI_CONFIG_BLUEFI (C++ enumerator), 203
WIFI_CONFIG_ESPTOUCH (C++ enumerator), 203
WIFI_CONFIG_ESPTOUCH_AIRKISS (C++ enumerator), 203
WIFI_CONFIG_WEB (C++ enumerator), 203
WIFI_CONFIG_WPS (C++ enumerator), 203
WIFI_SERV_EVENT_CONNECTED (C++ enumerator), 150