ESP-IoT-Solution User Guide

Release master
Espressif Systems
Mar 18, 2026

Table of contents

Table of contents i

1 Get Started 3
- 1.1 ESP-IoT-Solution Version Description . . . 3
- 1.2 Development Environment Setup . . . 3
- 1.3 Hardware Preparation . . . 4
- 1.4 How to Obtain and Use Components . . . 4
- 1.5 How to Use Example Programs . . . 4

2 Basic Component 7
- 2.1 Boards Component . . . 7
  - 2.1.1 i2c_bus . . . 7
  - 2.1.2 spi_bus . . . 14
- 2.2 I2S LCD Driver . . . 18
  - 2.2.1 API Reference . . . 18
- 2.3 Boards Component . . . 21
  - 2.3.1 Instructions . . . 22
  - 2.3.2 The Switch and Configuration of a Development Board . . . 23
  - 2.3.3 Supported Development Boards . . . 24
  - 2.3.4 Add a New Development Board . . . 25
  - 2.3.5 Component Dependencies . . . 25
  - 2.3.6 Adapted IDF Versions . . . 25
  - 2.3.7 Supported Chips . . . 25
- 2.4 CMake Utilities . . . 25
  - 2.4.1 How to Use . . . 26
  - 2.4.2 Detailed Reference . . . 26

3 Bluetooth 27
- 3.1 BLE Connection Management . . . 27
  - 3.1.1 Application Example . . . 27
  - 3.1.2 Examples . . . 27
  - 3.1.3 API Reference . . . 27
- 3.2 BLE L2CAP Connection Oriented Channels (CoC) . . . 61
  - 3.2.1 Application Example . . . 61
  - 3.2.2 Examples . . . 62
  - 3.2.3 API Reference . . . 62
- 3.3 BLE Services . . . 62
  - 3.3.1 Alert Notification Service . . . 62
  - 3.3.2 Battery Service . . . 65
  - 3.3.3 Body Composition Service . . . 78
  - 3.3.4 Current Time Service . . . 82
  - 3.3.5 Device Information Service . . . 85
  - 3.3.6 Heart Rate Service . . . 89
  - 3.3.7 Health Thermometer Service . . . 91
  - 3.3.8 Object Transfer Service . . . 95
  - 3.3.9 TX Power Service . . . 106
  - 3.3.10 User Data Service . . . 107

3.3.11 Weight Scale Service . . . 110
3.4 BLE Profiles . . . 113
3.4.1 Alert Notification Profile . . . 114
3.4.2 Bluetooth Low Energy MIDI Profile . . . 118
3.4.3 Heart Rate Profile . . . 123
3.4.4 Health Thermometer Profile . . . 125
3.4.5 Object Transfer Profile . . . 129
3.5 BLE HCI Components . . . 146
3.5.1 How to Use BLE HCI . . . 146
3.5.2 API Reference . . . 147
3.6 BTHome Component . . . 153
3.6.1 BTHome Usage . . . 153
3.6.2 Example Code . . . 153
3.6.3 API Reference . . . 154

4 Display **163**
4.1 LCD Screen . . . 163
4.1.1 LCD Introduction . . . 163
4.1.2 LCD Terms Table . . . 174
4.1.3 LCD Development Guide . . . 174
4.1.4 SPI LCD Introduction . . . 179
4.1.5 RGB LCD Introduction . . . 186
4.1.6 MIPI DSI LCD Detailed Guide . . . 200
4.1.7 Detailed Explanation of LCD Screen Tearing . . . 215
4.1.8 LCD Application Solution . . . 217
4.1.9 GUI Optimization Solutions . . . 226
4.2 Digital Tube . . . 228
4.2.1 CH450 Driver . . . 229
4.2.2 HT16C21 Driver . . . 230
4.2.3 IS31FL3XXX Driver . . . 231
4.3 LED Indicator . . . 231
4.3.1 Supported Indicator Light Types . . . 232
4.3.2 Defining Blinking Type . . . 232
4.3.3 Predefined Blinking Priorities . . . 236
4.3.4 Control Indicator Blinks . . . 236
4.3.5 Custom light blink . . . 237
4.3.6 Adjustment of Gamma . . . 237
4.3.7 Drive Level Setting . . . 238
4.4 LCD Tools . . . 241
4.4.1 ESP LVGL ADAPTER . . . 241
4.4.2 ESP LV DECODER . . . 256
4.4.3 ESP LV FS . . . 258
4.4.4 ESP MMAP ASSETS . . . 259

5 USB Host & Device **267**
5.1 USB Host & Device . . . 267
5.1.1 **ESP USB Peripheral Introduction** . . . 267
5.1.2 USB-OTG Peripheral Introduction . . . 270
5.1.3 USB-Serial-JTAG Peripheral Introduction . . . 273
5.1.4 USB PHY/Transceiver Introduction . . . 275
5.1.5 USB VID and PID . . . 278
5.1.6 USB Host Solution . . . 278
5.1.7 USB Device Solution . . . 281
5.1.8 Self-Powered USB Device Solutions . . . 284
5.1.9 Prevent Windows from incrementing COM numbers based on USB device serial number . 285
5.1.10 USB Signal Quality Test . . . 286
5.1.11 TinyUSB Application Guide . . . 291
5.1.12 Developing with Native tinyusb . . . 296

5.1.13 **USB Type-C Hardware Design Guide** . . . 300
5.2 USB Host Drivers . . . 305
5.2.1 USB Stream Component . . . 305
5.2.2 ESP MSC OTA . . . 316
5.2.3 USB Host CDC . . . 321
5.2.4 USB RNDIS Host Driver . . . 335
5.2.5 USB PPP . . . 339
5.2.6 USB ECM Host Driver . . . 341
5.3 USB Device Drivers . . . 346
5.3.1 USB Device UVC . . . 346
5.3.2 USB Device UAC . . . 349
5.3.3 ESP TinyUF2 . . . 351

6 Audio **357**
6.1 PWM Audio . . . 357
6.1.1 Features . . . 357
6.1.2 Structure . . . 357
6.1.3 PWM Frequency . . . 358
6.1.4 Application Example . . . 358
6.1.5 API Reference . . . 358
6.2 DAC Audio . . . 362
6.2.1 API Reference . . . 362
6.3 ADC Microphone . . . 364
6.3.1 Features . . . 364
6.3.2 Reference Circuit . . . 364
6.3.3 External Configuration for ADC Continuous . . . 365
6.3.4 Using with ADC Oneshot Mode . . . 365
6.3.5 Example Code . . . 365
6.3.6 API Reference . . . 366

7 Audio/Video Codec **369**
7.1 AVI Player . . . 369
7.1.1 API Reference . . . 369

8 AI **375**
8.1 OpenAI . . . 375
8.1.1 FAQ . . . 375
8.1.2 API Reference . . . 376
8.2 Model Context Protocol (MCP) . . . 390
8.2.1 Examples . . . 390
8.2.2 API Reference . . . 390
8.3 Touch Digit Recognition . . . 397
8.3.1 Touch Principle and Data Acquisition . . . 397
8.3.2 Model Training and Deployment . . . 399
8.3.3 On-device Inference . . . 404

9 Input Device **405**
9.1 **Button** . . . 405
9.1.1 Button event . . . 406
9.1.2 Configuration . . . 407
9.1.3 Demonstration . . . 408
9.1.4 API Reference . . . 412
9.2 **Keyboard Scanning** . . . 417
9.2.1 **Component Events** . . . 417
9.2.2 **Application Example** . . . 418
9.2.3 **API Reference** . . . 419
9.3 Knob . . . 424
9.3.1 Applicable Scenarios . . . 424
9.3.2 Hardware Design . . . 424

9.3.3 Knob Event . . . 424
9.3.4 Configuration items . . . 426
9.3.5 Application Examples . . . 426
9.3.6 Low Power Support . . . 426
9.3.7 Enable and Disable . . . 426
9.3.8 API Reference . . . 428
9.4 Touch Panel . . . 430
9.4.1 Supported Touch Controllers . . . 431
9.4.2 Architecture . . . 431
9.4.3 Touch Panel Features . . . 431
9.4.4 Application Example . . . 431
9.4.5 Using ESP-IoT-Solution Touch Drivers . . . 433
9.4.6 Touch Rotation and Mirroring . . . 434
9.4.7 Related Examples . . . 434
9.4.8 Component Links . . . 434
9.4.9 API Reference . . . 434

10 IR **435**
10.1 IR learn . . . 435
10.1.1 Application Examples . . . 435
10.1.2 API Reference . . . 437

11 Low-Power solution **441**
11.1 ESP32-P4 Battery Backup Solution . . . 441
11.1.1 Battery Connection and Circuit Protection . . . 441
11.1.2 Power Switching . . . 443
11.1.3 Brown-out Detector & BOD filter . . . 444
11.1.4 Battery Charging Circuit . . . 445
11.1.5 Peripheral Support in VBAT Power Supply Scenario . . . 446

12 Sensors **447**
12.1 Sensor Hub . . . 447
12.1.1 Instructions . . . 447
12.1.2 Examples . . . 449
12.1.3 API Reference . . . 449
12.2 Humidity and Temperature Sensor . . . 458
12.2.1 Adapted Products . . . 458
12.2.2 API Reference . . . 459
12.3 Inertial Measurement Unit (IMU) . . . 459
12.3.1 Adapted Products . . . 459
12.3.2 API Reference . . . 459
12.4 Ambient Light Sensor . . . 459
12.4.1 Adapted Products . . . 459
12.4.2 API Reference . . . 460
12.5 Pressure Sensor . . . 460
12.5.1 Adapted Products . . . 460
12.6 Gesture Sensor . . . 460
12.6.1 Adapted Products . . . 460
12.7 **NTC_Driver** . . . 460
12.7.1 Demonstration . . . 461
12.7.2 API reference . . . 462
12.8 **Power Monitor** . . . 463
12.8.1 Adapted Products . . . 464
12.8.2 API Reference . . . 464
12.9 **Power Measure** . . . 465
12.9.1 Features . . . 466
12.9.2 BL0937 Calibration Process and Method . . . 466
12.9.3 How It Works . . . 466
12.9.4 Code Snippets . . . 467

12.9.5 Troubleshooting . . . 469
12.9.6 Adapted Products . . . 470
12.9.7 API Reference . . . 470
12.10 Battery Fuel Gauge . . . 474
12.10.1 Adapted Products . . . 474
12.11 ADC Sensor . . . 474
12.11.1 Adapted Products . . . 474
12.12 Gas Sensor . . . 474
12.12.1 Adapted Products . . . 474

13 Touch Sensor **475**
13.1 Touch Proximity Sensor . . . 475
13.1.1 Principle of Operation . . . 475
13.1.2 Hardware Reference for Testing . . . 476
13.1.3 Configuration Reference . . . 476
13.1.4 Parameter Adjustment Reference . . . 477
13.1.5 Examples . . . 482
13.1.6 API Reference . . . 482
13.2 Touch Button Sensor . . . 484
13.2.1 Overview . . . 485
13.2.2 Configuration Parameters . . . 485
13.2.3 API Usage Examples . . . 486
13.2.4 Examples . . . 487
13.2.5 API Reference . . . 487
13.3 Touch Button Sensor . . . 489
13.3.1 Overview . . . 490
13.3.2 Configuration Parameters . . . 490
13.3.3 API Usage Examples . . . 491
13.3.4 Examples . . . 493
13.3.5 API Reference . . . 493

14 Storage **497**
14.1 Storage Media . . . 497
14.1.1 SPI Flash . . . 498
14.1.2 SD Card . . . 498
14.1.3 eMMC . . . 498
14.1.4 EEPROM . . . 499
14.2 File System . . . 499
14.2.1 NVS Library . . . 500
14.2.2 FAT File System . . . 500
14.2.3 SPIFFS File System . . . 501
14.2.4 LittleFS File System . . . 501
14.2.5 Virtual File System (VFS) . . . 502
14.2.6 Storage Security . . . 502
14.2.7 File System Design Recommendations . . . 502

15 Motor **503**
15.1 BLDC Motor . . . 503
15.1.1 Overview of Bldc Motor Control . . . 503
15.1.2 Sensorless Square Wave Motor Control Based on ADC Sampling . . . 507
15.1.3 Sensorless Square Wave Motor Control Based on Comparator Detection . . . 511
15.1.4 ESP Sensorless BLDC Control Components . . . 513
15.2 Servo . . . 524
15.2.1 Instructions . . . 524
15.2.2 Application Example . . . 524
15.2.3 API Reference . . . 525
15.3 ESP SimpleFOC . . . 527
15.3.1 Using SimpleFOC in Your Project . . . 527
15.3.2 API Reference . . . 528

16 Solutions **529**
16.1 4G Solutions . . . 529
16.1.1 IOT ETH Ethernet . . . 529

17 Security & Encryption **533**
17.1 Flash Encryption . . . 533
17.1.1 Flash Encryption and Secure Boot . . . 533
17.1.2 Encryption Scope . . . 533
17.1.3 Flash Encryption Feature Support List . . . 534
17.1.4 Development Mode and Release Mode . . . 534
17.1.5 Encryption Process . . . 535
17.1.6 Application Access to Encrypted Partitions . . . 535
17.1.7 Reflashing Flash . . . 535
17.1.8 Disabling Encryption . . . 535
17.1.9 Example Code . . . 536
17.1.10 Best Practices . . . 536
17.1.11 Frequently Asked Questions (FAQ) . . . 536
17.2 Secure Boot . . . 536
17.2.1 Secure Boot and Flash Encryption . . . 536
17.2.2 Secure Boot Chain of Trust . . . 537
17.2.3 Secure Boot Feature Support List . . . 537
17.2.4 Signature and Key Management . . . 538
17.2.5 Signature Block Format . . . 538
17.2.6 Enabling Hardware Secure Boot . . . 538
17.2.7 Key Revocation . . . 539
17.2.8 Restrictions After Enabling . . . 539
17.2.9 Software Signature Verification . . . 540
17.2.10 Example Code . . . 540
17.2.11 Best Practices . . . 541
17.2.12 Frequently Asked Questions (FAQ) . . . 541
17.3 Mass Production . . . 541
17.3.1 **Flash Download Tool** . . . 541
17.3.2 **Espressif Production Test Tool** . . . 542
17.3.3 **Module Fixture Design Specifications** . . . 543
17.3.4 **Matter QR Code Generation Tool** . . . 544
17.3.5 **EspRFTestTool Package** . . . 545

18 Electrical&Lighting Solution **547**
18.1 Lightbulb Driver . . . 547
18.1.1 Dimming scheme . . . 547
18.1.2 Fade principle . . . 550
18.1.3 Low-power implementation process . . . 550
18.1.4 Color calibration scheme . . . 550
18.1.5 Power limiting scheme . . . 551
18.1.6 API Reference . . . 552

19 Other Resources **569**
19.1 GPIO Expander . . . 569
19.1.1 Adapted Products . . . 569
19.2 ADC Range Extension Solution . . . 569
19.2.1 Patch Use Guide . . . 570
19.2.2 API Guide . . . 570
19.2.3 Comparison of ADC Range Extension Effects . . . 571
19.3 **Zero_Detection** . . . 572
19.3.1 Zero Detection event . . . 572
19.3.2 Configuration . . . 572
19.3.3 Demonstration . . . 572
19.3.4 API Reference . . . 573
19.4 ADC Two-Point Calibration Scheme . . . 578

19.4.1 Principle of Two-Point Calibration for ESP32 . . . 578
19.4.2 Principle of Two-Point Calibration (Method 2) for ESP32-S2 . . . 579
19.4.3 API Reference . . . 580

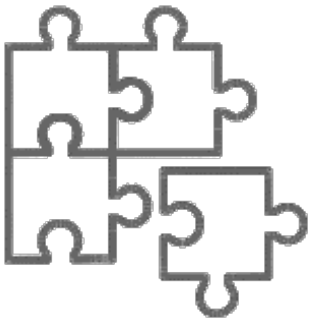
20 Contributions Guide **583**
20.1 How to Contribute . . . 583
20.2 Before Contributing . . . 583
20.3 Pull Request Process . . . 583
20.4 Legal Part . . . 583
20.5 Related Documents . . . 584
20.5.1 esp-iot-solution 编码规范 . . . 584

Index **591**

Index **591**

This is the documentation for ESP-IoT-Solution Development Framework.

ESP-IoT-Solution contains device drivers and code frameworks for the development of IoT system, which works as extra components of ESP-IDF and much easier to start.

Get Started	Display	USB Host&Device
GUI	Input	Sensors
		
Audio	Security&Encryption	Contribute

Chapter 1

Get Started

This document briefly introduces how to obtain and use components in ESP-IoT-Solution, how to compile and run examples, helping beginners get started quickly.

1.1 ESP-IoT-Solution Version Description

ESP-IoT-Solution has adopted component-based management since `release/v2.0`, with each component and example independently iterating. Please check the component's `idf_component.yml` file for the dependent ESP-IDF version. Release branches only maintain historical major versions, while the master branch continuously integrates new features. It is recommended that new projects obtain required components through the component registry.

Different version descriptions are as follows. For details and component lists, please see README_CN.md:

Table 1: ESP-IoT-Solution Version Support

ESP-IoT-Solution	Dependent ESP-IDF	Main Changes	Support Status
master	>= v5.3	New chip support	New feature development branch
release/v2.0	<= v5.3, >= v4.4	Component manager support	Historical version maintenance
release/v1.1	v4.0.1	IDF version update, code migration	Backup, maintenance stopped
release/v1.0	v3.2.2	Historical version	Backup, maintenance stopped

Note: The recommended ESP-IDF version varies for different chips. For details, please refer to Compatibility Between ESP-IDF Releases and Revisions of Espressif SoCs.

1.2 Development Environment Setup

ESP-IDF is the IoT development framework provided by Espressif for ESP series chips:

- ESP-IDF includes a series of libraries and header files, providing core components required for building software projects based on ESP SoC
- ESP-IDF also provides the most commonly used tools and functions during development and mass production, such as: build, flash, debug and measurement.

Note: Please refer to: ESP-IDF Programming Guide to complete the ESP-IDF development environment setup.

1.3 Hardware Preparation

You can choose any ESP series development board, or refer to the supported development boards in esp-bsp for a quick start. For specifications of each series chip, please see ESP Product Selector.

ESP series SoCs support the following features:

- Wi-Fi (2.4 GHz/5 GHz dual-band)
- Bluetooth 5.x (BLE/Mesh)
- High-performance multi-core processor, maximum frequency up to 400 MHz
- Ultra-low power coprocessor and deep sleep mode
- **Rich peripheral interfaces:**
  - General interfaces: GPIO, UART, I2C, I2S, SPI, SDIO, USB OTG, etc.
  - Dedicated interfaces: LCD, camera, Ethernet, CAN, Touch, LED PWM, temperature sensor, etc.
- **Large capacity memory:**
  - Internal RAM up to 768 KB
  - Support for external PSRAM expansion
  - Support for external Flash storage
- **Enhanced security features:**
  - Hardware encryption engine
  - Secure boot
  - Flash encryption
  - Digital signature

ESP series SoCs use advanced process technology, providing industry-leading RF performance, low power consumption characteristics and stable reliability, suitable for IoT, industrial control, smart home, wearable devices and other application scenarios.

Note: For specific specifications and functions of each series chip, please refer to ESP Product Selector.

1.4 How to Obtain and Use Components

It is recommended to obtain ESP-IoT-Solution components through ESP Component Registry.

Taking the button component as an example, the steps to add dependencies are as follows:

1. Execute in the project root directory:

```
idf.py add-dependency "espressif/button"
```

2. Reference header files in code and call APIs, for example:

```
#include "iot_button.h"
// For specific API usage, please refer to component documentation
```

For more information on component manager usage, please refer to ESP Registry Docs.

1.5 How to Use Example Programs

ESP-IoT-Solution provides rich example programs to help users get started quickly. Taking the `button_power_save` example:

1. Ensure that the ESP-IDF development environment setup has been completed and environment variables have been successfully configured
2. Download the ESP-IoT-Solution code repository:

```
git clone https://github.com/espressif/esp-iot-solution.git
```

3. Enter the example directory or copy it to your working directory:

```
cd examples/get-started/button_power_save
```

Note: If you copy the example to another directory, due to file path changes, please delete all `override_path` configurations in `main/idf_component.yml`.

4. Select target chip (such as ESP32, required when using for the first time or switching chips):

```
idf.py set-target esp32
```

5. Configure project (optional):

```
idf.py menuconfig
```

6. Compile and flash to development board:

```
idf.py build
idf.py -p <PORT> flash
```

7. Monitor output through serial port:

```
idf.py -p <PORT> monitor
```

For more examples, please see the `examples/` directory. For specific usage methods, please refer to the README files under each example.

Chapter 2

Basic Component

2.1 Boards Component

2.1.1 i2c_bus

How to Use i2c_bus

1. Create a bus: create a bus object using `i2c_bus_create()`. During the process, you need to specify the I2C port number and the bus configuration option `i2c_config_t`, which includes SDA and SCL pin numbers, pull-up and pull-down modes, as these are determined when the system is designed and normally will not be changed at runtime. The bus configuration option also includes the default clock frequency of the bus, which is used when the device does not specify a frequency.
2. Create a device: use `i2c_bus_device_create()` to create a device on the bus object created in the first step. During the process, you need to specify bus handle, the I2C address of the device, and the clock frequency when the device is running. The frequency will be dynamically changed during I2C transmission based on device configuration options. The device clock frequency can be configured as 0, indicating the current bus frequency is used by default.
3. Data reading: use `i2c_bus_read_byte()` or `i2c_bus_read_bytes()` to read `Byte` data; use `i2c_bus_read_bit()` or `i2c_bus_read_bits()` to read `bit` data. During the process, you only need to pass in the device handle, the device register address, a buffer to hold the read data, the read length, etc. The register address can be configured as `NULL_I2C_MEM_ADDR` for devices without internal registers.
4. Data writing: use `i2c_bus_write_byte()` or `i2c_bus_write_bytes()` to write `Byte` data; use `i2c_bus_write_bit()` or `i2c_bus_write_bits()` to write `bit` data. During the process, you only need to pass in the device handle, the device register address, the data location to be written, the write length, etc. The register address can be configured as `NULL_I2C_MEM_ADDR` for devices without internal registers.
5. Delete device and bus: if all i2c_bus communication has been completed, you can free your system resources by deleting devices and bus objects. Use `i2c_bus_device_delete()` to delete created devices respectively, then use `i2c_bus_delete()` to delete bus resources. If the bus is deleted with the device not being deleted yet, this operation will not take effect.

Example:

```
i2c_config_t conf = {
    .mode = I2C_MODE_MASTER,
    .sda_io_num = I2C_MASTER_SDA_IO,
    .sda_pullup_en = GPIO_PULLUP_ENABLE,
    .scl_io_num = I2C_MASTER_SCL_IO,
    .scl_pullup_en = GPIO_PULLUP_ENABLE,
    .master.clk_speed = 100000,
}; // i2c_bus configurations

uint8_t data_rd[2] = {0};
uint8_t data_wr[2] = {0x01, 0x21};

i2c_bus_handle_t i2c0_bus = i2c_bus_create(I2C_NUM_0, &conf); // create i2c_bus
i2c_bus_device_handle_t i2c_device1 = i2c_bus_device_create(i2c0_bus, 0x28,␣
↪400000); // create device1, address: 0x28 , clk_speed: 400000
i2c_bus_device_handle_t i2c_device2 = i2c_bus_device_create(i2c0_bus, 0x32, 0); //␣
↪create device2, address: 0x32 , clk_speed: no-specified

i2c_bus_read_bytes(i2c_device1, NULL_I2C_MEM_ADDR, 2, data_rd); // read bytes from␣
↪device1 with no register address
i2c_bus_write_bytes(i2c_device2, 0x10, 2, data_wr); // write bytes to device2␣
↪register 0x10

i2c_bus_device_delete(&i2c_device1); //delete device1
i2c_bus_device_delete(&i2c_device2); //delete device2
i2c_bus_delete(&i2c0_bus);  //delete i2c_bus
```

Note: For some special application scenarios:

1. When the address of a register is 16-bit, you can use *i2c_bus_read_reg16()* or *i2c_bus_write_reg16()* to read or write its data;
2. For devices that need to skip the address phase or need to add a command phase, you can operate using *i2c_bus_cmd_begin()* combined with I2C command link.
3. For scenarios where I2C ports are insufficient or software I2C debugging is required, you can enable `Enable software I2C support` in `menuconfig` under `(Top) → Component config → Bus Options → I2C Bus Options`. Then, pass a `port` of type `i2c_sw_port_t` (e.g., `I2C_NUM_SW_0`) to the `port` parameter in the *i2c_bus_create()* function.

Adapted IDF Versions

- ESP-IDF v4.0 and later versions.

API Reference

Header File

- components/i2c_bus/include/i2c_bus.h

Functions

i2c_bus_handle_t **i2c_bus_create**(i2c_port_t port, const *i2c_config_t* *conf)

Create an I2C bus instance then return a handle if created successfully. Each I2C bus works in a singleton mode, which means for an i2c port only one group parameter works. When i2c_bus_create is called more than one time for the same i2c port, following parameter will override the previous one.

Parameters

- **port** –I2C port number. Please note that enabling I2C_BUS_SUPPORT_SOFTWARE in menuconfig allows you to use ports in i2c_sw_port_t to enable software I2C.

- **conf** –Pointer to I2C bus configuration

Returns i2c_bus_handle_t Return the I2C bus handle if created successfully, return NULL if failed.

esp_err_t **i2c_bus_delete**(*i2c_bus_handle_t* *p_bus_handle)

Delete and release the I2C bus resource.

Parameters **p_bus_handle** –Point to the I2C bus handle, if delete succeed handle will set to NULL.

Returns
- ESP_OK Success
- ESP_FAIL Fail

i2c_master_bus_handle_t **i2c_bus_get_internal_bus_handle**(*i2c_bus_handle_t* bus_handle)

Get internal idf bus_handle from i2c_bus_handle.

Parameters **bus_handle** –I2C bus handle

Returns i2c_master_bus_handle_t Return the idf bus_handle if obtained successfully, return NULL if failed.

uint8_t **i2c_bus_scan**(*i2c_bus_handle_t* bus_handle, uint8_t *buf, uint8_t num)

Scan i2c devices attached on i2c bus.

Parameters
- **bus_handle** –I2C bus handle
- **buf** –Pointer to a buffer to save devices' address, if NULL no address will be saved.
- **num** –Maximum number of addresses to save, invalid if buf set to NULL, higher addresses will be discarded if num less-than the total number found on the I2C bus.

Returns uint8_t Total number of devices found on the I2C bus

uint32_t **i2c_bus_get_current_clk_speed**(*i2c_bus_handle_t* bus_handle)

Get current active clock speed.

Parameters **bus_handle** –I2C bus handle

Returns uint32_t current clock speed

uint8_t **i2c_bus_get_created_device_num**(*i2c_bus_handle_t* bus_handle)

Get created device number of the bus.

Parameters **bus_handle** –I2C bus handle

Returns uint8_t created device number of the bus

i2c_bus_device_handle_t **i2c_bus_device_create**(*i2c_bus_handle_t* bus_handle, uint8_t dev_addr, uint32_t clk_speed)

Create an I2C device on specific bus. Dynamic configuration must be enable to achieve multiple devices with different configs on a single bus. menuconfig:Bus Options->I2C Bus Options->enable dynamic configuration.

Parameters
- **bus_handle** –Point to the I2C bus handle
- **dev_addr** –i2c device address
- **clk_speed** –device specified clock frequency the i2c_bus will switch to during each transfer. 0 if use current bus speed.

Returns i2c_bus_device_handle_t return a device handle if created successfully, return NULL if failed.

esp_err_t **i2c_bus_device_delete**(*i2c_bus_device_handle_t* *p_dev_handle)

Delete and release the I2C device resource, i2c_bus_device_delete should be used in pairs with i2c_bus_device_create.

Parameters **p_dev_handle** –Point to the I2C device handle, if delete succeed handle will set to NULL.

Returns
- ESP_OK Success
- ESP_FAIL Fail

uint8_t **i2c_bus_device_get_address**(*i2c_bus_device_handle_t* dev_handle)

Get device's I2C address.

Parameters **dev_handle** –I2C device handle

Returns uint8_t I2C address, return NULL_I2C_DEV_ADDR if dev_handle is invalid.

esp_err_t **i2c_bus_read_byte**(*i2c_bus_device_handle_t* dev_handle, uint8_t mem_address, uint8_t *data)

Read single byte from i2c device with 8-bit internal register/memory address.

Parameters

- **dev_handle** –I2C device handle
- **mem_address** –The internal reg/mem address to read from, set to NULL_I2C_MEM_ADDR if no internal address.
- **data** –Pointer to a buffer to save the data that was read

Returns esp_err_t

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Sending command error, slave doesn't ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

esp_err_t **i2c_bus_read_bytes**(*i2c_bus_device_handle_t* dev_handle, uint8_t mem_address, size_t data_len, uint8_t *data)

Read multiple bytes from i2c device with 8-bit internal register/memory address. If internal reg/mem address is 16-bit, please refer i2c_bus_read_reg16.

Parameters

- **dev_handle** –I2C device handle
- **mem_address** –The internal reg/mem address to read from, set to NULL_I2C_MEM_ADDR if no internal address.
- **data_len** –Number of bytes to read
- **data** –Pointer to a buffer to save the data that was read

Returns esp_err_t

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Sending command error, slave doesn't ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

esp_err_t **i2c_bus_read_bit**(*i2c_bus_device_handle_t* dev_handle, uint8_t mem_address, uint8_t bit_num, uint8_t *data)

Read single bit of a byte from i2c device with 8-bit internal register/memory address.

Parameters

- **dev_handle** –I2C device handle
- **mem_address** –The internal reg/mem address to read from, set to NULL_I2C_MEM_ADDR if no internal address.
- **bit_num** –The bit number 0 - 7 to read
- **data** –Pointer to a buffer to save the data that was read. *data == 0 -> bit = 0, *data !=0 -> bit = 1.

Returns esp_err_t

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Sending command error, slave doesn't ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

esp_err_t **i2c_bus_read_bits**(*i2c_bus_device_handle_t* dev_handle, uint8_t mem_address, uint8_t bit_start, uint8_t length, uint8_t *data)

Read multiple bits of a byte from i2c device with 8-bit internal register/memory address.

Parameters

- **dev_handle** –I2C device handle
- **mem_address** –The internal reg/mem address to read from, set to NULL_I2C_MEM_ADDR if no internal address.
- **bit_start** –The bit to start from, 0 - 7, MSB at 0
- **length** –The number of bits to read, 1 - 8
- **data** –Pointer to a buffer to save the data that was read

Returns esp_err_t

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Sending command error, slave doesn't ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

esp_err_t **i2c_bus_write_byte**(*i2c_bus_device_handle_t* dev_handle, uint8_t mem_address, uint8_t data)

Write single byte to i2c device with 8-bit internal register/memory address.

Parameters

- **dev_handle** –I2C device handle
- **mem_address** –The internal reg/mem address to write to, set to NULL_I2C_MEM_ADDR if no internal address.
- **data** –The byte to write.

Returns esp_err_t

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Sending command error, slave doesn't ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

esp_err_t **i2c_bus_write_bytes**(*i2c_bus_device_handle_t* dev_handle, uint8_t mem_address, size_t data_len, const uint8_t *data)

Write multiple byte to i2c device with 8-bit internal register/memory address If internal reg/mem address is 16-bit, please refer i2c_bus_write_reg16.

Parameters

- **dev_handle** –I2C device handle
- **mem_address** –The internal reg/mem address to write to, set to NULL_I2C_MEM_ADDR if no internal address.
- **data_len** –Number of bytes to write
- **data** –Pointer to the bytes to write.

Returns esp_err_t

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Sending command error, slave doesn't ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

esp_err_t **i2c_bus_write_bit**(*i2c_bus_device_handle_t* dev_handle, uint8_t mem_address, uint8_t bit_num, uint8_t data)

Write single bit of a byte to an i2c device with 8-bit internal register/memory address.

Parameters

- **dev_handle** –I2C device handle
- **mem_address** –The internal reg/mem address to write to, set to NULL_I2C_MEM_ADDR if no internal address.
- **bit_num** –The bit number 0 - 7 to write
- **data** –The bit to write, data == 0 means set bit = 0, data !=0 means set bit = 1.

Returns esp_err_t

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

- ESP_FAIL Sending command error, slave doesn't ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

esp_err_t **i2c_bus_write_bits**(*i2c_bus_device_handle_t* dev_handle, uint8_t mem_address, uint8_t bit_start, uint8_t length, uint8_t data)

Write multiple bits of a byte to an i2c device with 8-bit internal register/memory address.

Parameters
- **dev_handle** –I2C device handle
- **mem_address** –The internal reg/mem address to write to, set to NULL_I2C_MEM_ADDR if no internal address.
- **bit_start** –The bit to start from, 0 - 7, MSB at 0
- **length** –The number of bits to write, 1 - 8
- **data** –The bits to write.

Returns esp_err_t
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Sending command error, slave doesn't ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

esp_err_t **i2c_bus_cmd_begin**(*i2c_bus_device_handle_t* dev_handle, *i2c_cmd_handle_t* cmd)

I2C master send queued commands create by `i2c_cmd_link_create`. This function will trigger sending all queued commands. The task will be blocked until all the commands have been sent out. If I2C_BUS_DYNAMIC_CONFIG enable, i2c_bus will dynamically check configs and re-install i2c driver before each transfer, hence multiple devices with different configs on a single bus can be supported.

Note: Only call this function when `i2c_bus_read/write_xx` do not meet the requirements

Parameters
- **dev_handle** –I2C device handle
- **cmd** –I2C command handler

Returns esp_err_t
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Sending command error, slave doesn't ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

esp_err_t **i2c_bus_write_reg16**(*i2c_bus_device_handle_t* dev_handle, uint16_t mem_address, size_t data_len, const uint8_t *data)

Write date to an i2c device with 16-bit internal reg/mem address.

Parameters
- **dev_handle** –I2C device handle
- **mem_address** –The internal 16-bit reg/mem address to write to, set to NULL_I2C_MEM_ADDR if no internal address.
- **data_len** –Number of bytes to write
- **data** –Pointer to the bytes to write.

Returns esp_err_t
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Sending command error, slave doesn't ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

esp_err_t **i2c_bus_read_reg16**(*i2c_bus_device_handle_t* dev_handle, uint16_t mem_address, size_t data_len, uint8_t *data)

Read date from i2c device with 16-bit internal reg/mem address.

Parameters

- **dev_handle** –I2C device handle
- **mem_address** –The internal 16-bit reg/mem address to read from, set to NULL_I2C_MEM_ADDR if no internal address.
- **data_len** –Number of bytes to read
- **data** –Pointer to a buffer to save the data that was read

Returns esp_err_t

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Sending command error, slave doesn’ t ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

Structures

struct **i2c_config_t**

I2C initialization parameters.

Public Members

i2c_mode_t **mode**

I2C mode

int **sda_io_num**

GPIO number for I2C sda signal

int **scl_io_num**

GPIO number for I2C scl signal

bool **sda_pullup_en**

Internal GPIO pull mode for I2C sda signal

bool **scl_pullup_en**

Internal GPIO pull mode for I2C scl signal

uint32_t **clk_speed**

I2C clock frequency for master mode, (no higher than 1MHz for now)

struct *i2c_config_t*::[anonymous] **master**

I2C master config

uint32_t **clk_flags**

Bitwise of `I2C_SCLK_SRC_FLAG_**FOR_DFS**` for clk source choice

Macros

NULL_I2C_MEM_ADDR

set mem_address to NULL_I2C_MEM_ADDR if i2c device has no internal address during read/write

NULL_I2C_MEM_16BIT_ADDR

set 16bit mem_address to NULL_I2C_MEM_16BIT_ADDR if i2c device has no internal address during read/write

NULL_I2C_DEV_ADDR

invalid i2c device address

gpio_pad_select_gpio

portTICK_RATE_MS

Type Definitions

typedef void ***i2c_bus_handle_t**

i2c bus handle

typedef void ***i2c_bus_device_handle_t**

i2c device handle

typedef void ***i2c_cmd_handle_t**

I2C command handle

2.1.2 spi_bus

How to Use spi_bus

1. Create a bus: use *spi_bus_create()* to create a bus object. During the process, you need to specify the SPI port number (can choose between `SPI2_HOST` and `SPI3_HOST`) and the bus configuration option `spi_config_t`, which includes the pin numbers of `MOSI`, `MISO` and `SCLK`, as these are determined when the system is designed and normally will not be changed at runtime. The bus configuration option also includes `max_transfer_sz` to configure the maximum data size during a transmission. When `max_transfer_sz` is configured to 0, it means the maximum size will be the default value 4096.
2. Create a device: use *spi_bus_device_create()* to create a device on the bus object created in the first step. During the process, you need to specify the bus handle, the `CS` pin number of the device, device operation mode, the clock frequency when the device is running. The device mode and frequency will be dynamically changed during SPI transmissions based on device configuration options.
3. Data transmission: use *spi_bus_transfer_byte()*, *spi_bus_transfer_bytes()*, *spi_bus_transfer_reg16()* or *spi_bus_transfer_reg32()* to transfer data directly. Data send and receive can be operated at the same time since SPI communication is a full-duplex communication. During the process, you only need to pass in the device handle, data to be transmitted, a buffer to hold the read data, transmission length, etc.
4. Delete device and bus: if all spi_bus communication has been completed, you can free your system resources by deleting devices and bus objects. Use *spi_bus_device_delete()* to delete created devices respectively, then use *spi_bus_delete()* to delete bus resources. If the bus is deleted with the device not being deleted yet, this operation will not take effect.

Example:

```
spi_bus_handle_t bus_handle = NULL;
spi_bus_device_handle_t device_handle = NULL;
uint8_t data8_in = 0;
```

(continues on next page)

(continued from previous page)

```
uint8_t data8_out = 0xff;
uint16_t data16_in = 0;
uint32_t data32_in = 0;

spi_config_t bus_conf = {
    .miso_io_num = 19,
    .mosi_io_num = 23,
    .sclk_io_num = 18,
}; // spi_bus configurations

spi_device_config_t device_conf = {
    .cs_io_num = 19,
    .mode = 0,
    .clock_speed_hz = 20 * 1000 * 1000,
}; // spi_device configurations

bus_handle = spi_bus_create(SPI2_HOST, &bus_conf); // create spi bus
device_handle = spi_bus_device_create(bus_handle, &device_conf); // create spi␣
↪device

spi_bus_transfer_bytes(device_handle, &data8_out, &data8_in, 1); // transfer 1␣
↪byte with spi device
spi_bus_transfer_bytes(device_handle, NULL, &data8_in, 1); // only read 1 byte␣
↪with spi device
spi_bus_transfer_bytes(device_handle, &data8_out, NULL, 1); // only write 1 byte␣
↪with spi device
spi_bus_transfer_reg16(device_handle, 0x1020, &data16_in); // transfer 16-bit␣
↪value with the device
spi_bus_transfer_reg32(device_handle, 0x10203040, &data32_in); // transfer 32-bit␣
↪value with the device

spi_bus_device_delete(&device_handle);
spi_bus_delete(&bus_handle);
```

Note: For some special application scenarios, you can operate using `spi_bus_transmit_begin()` combined with spi_transaction_t directly.

Adapted IDF Versions

- ESP-IDF v4.0 and later versions.

API Reference

Header File

- components/spi_bus/include/spi_bus.h

Functions

spi_bus_handle_t **spi_bus_create** (spi_host_device_t host_id, const *spi_config_t* *bus_conf)

Create and initialize a spi bus and return the spi bus handle.

Parameters

- **host_id** –SPI peripheral that controls this bus, SPI2_HOST or SPI3_HOST
- **bus_conf** –spi bus configurations details in *spi_config_t*

Returns spi_bus_handle_t handle for spi bus operation, NULL if failed.

esp_err_t **spi_bus_delete** (*spi_bus_handle_t* *p_bus_handle)

Deinitialize and delete the spi bus.

Parameters **p_bus_handle** –pointer to spi bus handle, if delete succeed handle will set to NULL.

Returns esp_err_t

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_FAIL Fail
- ESP_OK Success

spi_bus_device_handle_t **spi_bus_device_create** (*spi_bus_handle_t* bus_handle, const *spi_device_config_t* *device_conf)

Create and add a device on the spi bus.

Parameters

- **bus_handle** –handle for spi bus operation.
- **device_conf** –spi device configurations details in *spi_device_config_t*

Returns spi_bus_device_handle_t handle for device operation, NULL if failed.

esp_err_t **spi_bus_device_delete** (*spi_bus_device_handle_t* *p_dev_handle)

Deinitialize and remove the device from spi bus.

Parameters **p_dev_handle** –pointer to device handle, if delete succeed handle will set to NULL.

Returns esp_err_t

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_FAIL Fail
- ESP_OK Success

esp_err_t **spi_bus_transfer_byte** (*spi_bus_device_handle_t* dev_handle, uint8_t data_out, uint8_t *data_in)

Transfer one byte with the device.

Parameters

- **dev_handle** –handle for device operation.
- **data_out** –data will send to device.
- **data_in** –pointer to receive buffer, set NULL to skip receive phase.

Returns esp_err_t

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_TIMEOUT if bus is busy
- ESP_OK on success

esp_err_t **spi_bus_transfer_bytes** (*spi_bus_device_handle_t* dev_handle, const uint8_t *data_out, uint8_t *data_in, uint32_t data_len)

Transfer multi-bytes with the device.

Parameters

- **dev_handle** –handle for device operation.
- **data_out** –pointer to sent buffer, set NULL to skip sent phase.
- **data_in** –pointer to receive buffer, set NULL to skip receive phase.
- **data_len** –number of bytes will transfer.

Returns esp_err_t

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_TIMEOUT if bus is busy
- ESP_OK on success

esp_err_t **spi_bus_transmit_begin** (*spi_bus_device_handle_t* dev_handle, spi_transaction_t *p_trans)

Send a polling transaction, wait for it to complete, and return the result.

Note: Only call this function when `spi_bus_transfer_xx` do not meet the requirements

Parameters
- **dev_handle** –handle for device operation.
- **p_trans** –Description of transaction to execute

Returns esp_err_t
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_TIMEOUT if bus is busy
- ESP_OK on success

esp_err_t **spi_bus_transfer_reg16** (*spi_bus_device_handle_t* dev_handle, uint16_t data_out, uint16_t *data_in)

Transfer one 16-bit value with the device. using msb by default. For example 0x1234, 0x12 will send first then 0x34.

Parameters
- **dev_handle** –handle for device operation.
- **data_out** –data will send to device.
- **data_in** –pointer to receive buffer, set NULL to skip receive phase.

Returns esp_err_t
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_TIMEOUT if bus is busy
- ESP_OK on success

esp_err_t **spi_bus_transfer_reg32** (*spi_bus_device_handle_t* dev_handle, uint32_t data_out, uint32_t *data_in)

Transfer one 32-bit value with the device. using msb by default. For example 0x12345678, 0x12 will send first, 0x78 will send in the end.

Parameters
- **dev_handle** –handle for device operation.
- **data_out** –data will send to device.
- **data_in** –pointer to receive buffer, set NULL to skip receive phase.

Returns esp_err_t
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_TIMEOUT if bus is busy
- ESP_OK on success

Structures

struct **spi_config_t**

spi bus initialization parameters.

Public Members

gpio_num_t **miso_io_num**

GPIO pin for Master In Slave Out (=spi_q) signal, or -1 if not used.

gpio_num_t **mosi_io_num**

GPIO pin for Master Out Slave In (=spi_d) signal, or -1 if not used.

gpio_num_t **sclk_io_num**

GPIO pin for Spi CLocK signal, or -1 if not used

int **max_transfer_sz**

<Maximum length of bytes available to send, if < 4096, 4096 will be set

struct **spi_device_config_t**

spi device initialization parameters.

Public Members

gpio_num_t **cs_io_num**

GPIO pin to select this device (CS), or -1 if not used

uint8_t **mode**

modes (0,1,2,3) that correspond to the four possible clocking configurations

int **clock_speed_hz**

spi clock speed, divisors of 80MHz, in Hz. See `SPI_MASTER_FREQ_*

Macros

NULL_SPI_CS_PIN

set cs_io_num to NULL_SPI_CS_PIN if spi device has no CP pin

Type Definitions

typedef void ***spi_bus_handle_t**

spi bus handle

typedef void ***spi_bus_device_handle_t**

spi device handle

The communication bus component (Bus) is a set of application-layer code built on top of the ESP-IDF peripheral driver code, including `i2c_bus`, `spi_bus` and etc. It is mainly used for bus communication between ESP chips and external devices. From the point of application development, this component has the following features:

1. Simplified peripheral initialization processes
2. Thread-safe device operations
3. Simple and flexible RW operations

This component abstracts the following concepts:

1. Bus: the resource and configuration option shared between devices during communication
2. Device: device specific resource and configuration option during communication

Each physical peripheral bus can mount one or more devices if the electrical condition allows, with the SPI bus addressing devices based on CS pins and the I2C bus addressing devices based on their addresses, thus achieving software independence between different devices on the same bus.

2.2 I2S LCD Driver

2.2.1 API Reference

Header File

- components/bus/include/i2s_lcd_driver.h

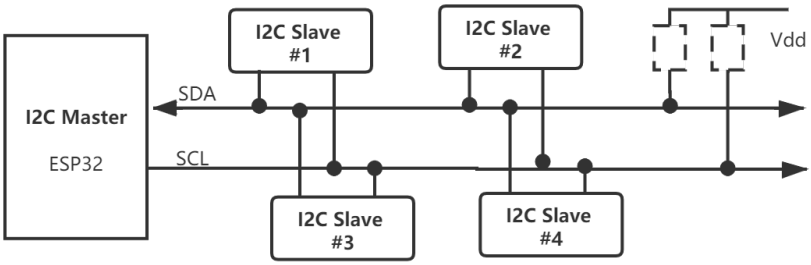


Fig. 1: i2c_bus Connection Diagram

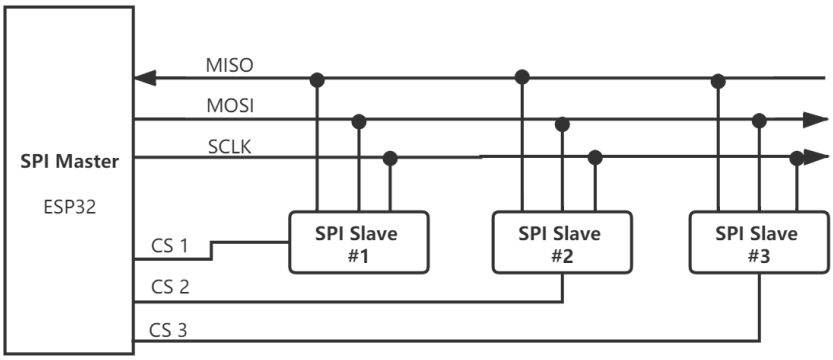


Fig. 2: spi_bus Connection Diagram

Functions

i2s_lcd_handle_t **i2s_lcd_driver_init** (const *i2s_lcd_config_t* *config)

Initialize i2s lcd driver.

Parameters **config** –configuration of i2s

Returns A handle to the created i2s lcd driver, or NULL in case of error.

esp_err_t **i2s_lcd_driver_deinit** (*i2s_lcd_handle_t* handle)

Deinit i2s lcd driver.

Parameters **handle** –i2s lcd driver handle to deinitilize

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG handle is invalid

esp_err_t **i2s_lcd_write_data** (*i2s_lcd_handle_t* handle, uint16_t data)

Write a data to LCD.

Parameters

- **handle** –i2s lcd driver handle
- **data** –Data to write

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG handle is invalid

esp_err_t **i2s_lcd_write_cmd** (*i2s_lcd_handle_t* handle, uint16_t cmd)

Write a command to LCD.

Parameters

- **handle** –Handle of i2s lcd driver
- **cmd** –command to write

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG handle is invalid

esp_err_t **i2s_lcd_write_command**(*i2s_lcd_handle_t* handle, const uint8_t *cmd, uint32_t length)

Write a command to LCD.

Parameters

- **handle** –Handle of i2s lcd driver
- **cmd** –command to write
- **length** –length of command

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG handle is invalid

esp_err_t **i2s_lcd_write**(*i2s_lcd_handle_t* handle, const uint8_t *data, uint32_t length)

Write block data to LCD.

Parameters

- **handle** –Handle of i2s lcd driver
- **data** –Pointer of data
- **length** –length of data

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG handle is invalid

esp_err_t **i2s_lcd_acquire**(*i2s_lcd_handle_t* handle)

acquire a lock

Parameters **handle** –Handle of i2s lcd driver

Returns Always return ESP_OK

esp_err_t **i2s_lcd_release**(*i2s_lcd_handle_t* handle)

release a lock

Parameters **handle** –Handle of i2s lcd driver

Returns Always return ESP_OK

Structures

struct **i2s_lcd_config_t**

Configuration of i2s lcd mode.

Handle of i2s lcd driver

Public Members

int8_t **data_width**

Parallel data width, 16bit or 8bit available

int8_t **pin_data_num**[16]

Parallel data output IO

int8_t **pin_num_cs**

CS io num

int8_t **pin_num_wr**

Write clk io

int8_t **pin_num_rs**

RS io num

int **clk_freq**

I2s clock frequency

i2s_port_t **i2s_port**

I2S port number

bool **swap_data**

Swap the 2 bytes of RGB565 color

uint32_t **buffer_size**

DMA buffer size

Macros

LCD_CMD_LEV

LCD_DATA_LEV

Type Definitions

typedef void ***i2s_lcd_handle_t**

2.3 Boards Component

This document mainly introduces the use of a board support component (Boards). As a common component of examples, this component can provide unified pin macro definitions and hardware-independent initialization operations to applications. Applications developed based on this component are compatible with different development boards at the same time with the following features:

1. Provides unified macro definitions for pins
2. Provides default peripheral configuration parameters
3. Provides unified board-level initialization interfaces
4. Provides hardware control interfaces for development boards

The following figure shows the structure of the Boards component:

- The Boards component contains the following:
  - `board_common.h`, contains the function declaration of the common API;
  - `board_common.c`, contains the function implementation of the common API (weak function);
  - `Kconfig.projbuild`, contains common configuration items;
- The subfolders named after the development board name includes the following:
  - `iot_board.h` provides the gpio definition of the development board, and the board' s unique custom API function declaration

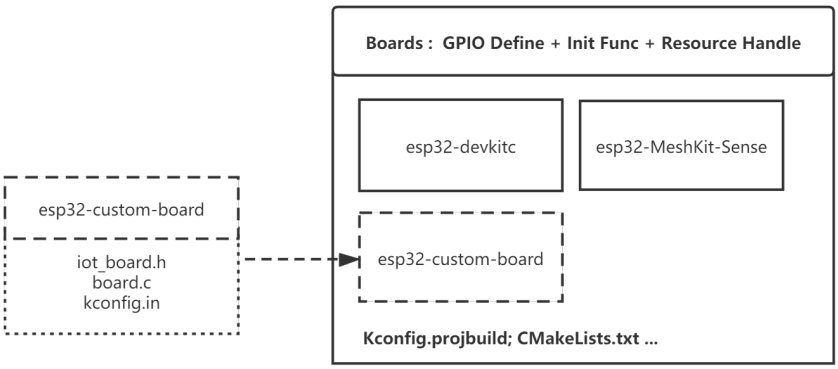


Fig. 3: Boards Component Diagram

- `board.c` provides user implementation of common API (Covering default weak function), custom API function implementation
- `kconfig.in` provides custom configuration items unique to the development board.

Note: The Boards component is provided in `examples/common_components/boards`.

2.3.1 Instructions

1. Initialize development board: use `iot_board_init` in `app_main` to initialize the development board. you can also do some configurations regarding this process using *The Switch and Configuration of a Development Board* in `menuconfig`;
2. Get the handle of a peripheral: use `iot_board_get_handle` and `board_res_id_t` to get peripheral resources. `NULL` will be returned if this peripheral is not initialized;
3. Operate on peripherals with handles directly.

Example:

```
void app_main(void)
{
    /*initialize board with default parameters,
    you can use menuconfig to choose a target board*/
    esp_err_t err = iot_board_init();
    if (err != ESP_OK) {
        goto error;
    }

    /*get the i2c0 bus handle with a board_res_id,
    BOARD_I2C0_ID is declared in board_res_id_t in each iot_board.h*/
    bus_handle_t i2c0_bus_handle = (bus_handle_t)iot_board_get_handle(BOARD_I2C0_
↪ID);
    if (i2c0_bus_handle == NULL) {
        goto error;
    }

    /*
    * use initialized peripheral with handles directly,
    * no configurations required anymore.
    */
}
```

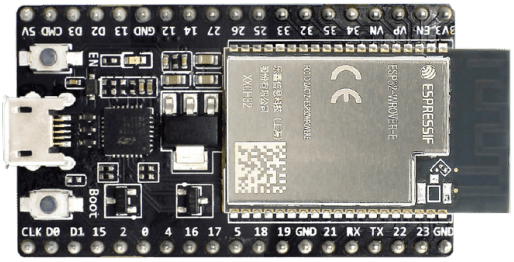
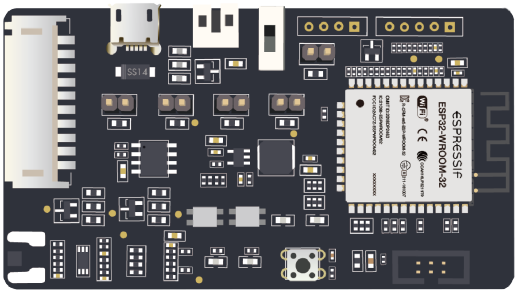
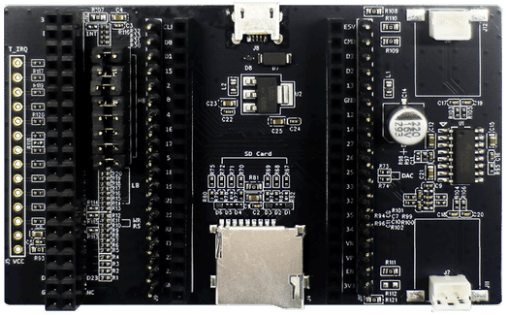
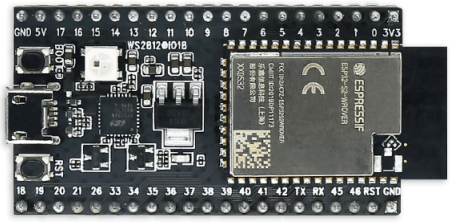
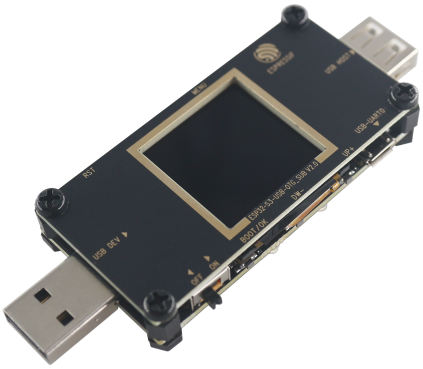
2.3.2 The Switch and Configuration of a Development Board

For applications developed basing on `Boards`, the following steps can be used to switch and configure boards:

1. Select the target development board: select a development board in `menuconfig->Board Options->Choose Target Board`;
2. Configure the development board parameters: `Board Common Options` contains common configurations, such as if enable i2c0 during the initialization of the development board; `XXX Board Options` contains the development board-specific configurations, such as switching the power supply status of the development board, etc.
3. Use `idf.py build flash monitor` to recompile and download the code.

Note: The default target of this build system is `ESP32`, please set the target before building via `idf.py set-target esp32s2` if you need to use `ESP32-S2`.

2.3.3 Supported Development Boards

ESP32 Development Boards	
	
esp32-devkitc	esp32-meshkit-sense
	
esp32-lcdkit	
ESP32-S2 Development Boards	
	
esp32s2-saola	
ESP32-S3 Development Boards	
	

2.3.4 Add a New Development Board

A new development board can be added to quickly adapt to applications developed basing on the `Boards` component.

The main process is as follows:

1. Prepare the necessary `iot_board.h` based on existing example;
2. Add board specific functions or cover the common weak function in `board_xxx.c` according to the requirements;
3. Add configuration options specific to this board in `kconfig.in` according to your needs;
4. Add the information of this board to `Kconfig.projbuild` for users;
5. Add the directory of this board to `CMakeLists.txt` so that it can be indexed by the build system. Please also update `component.mk` if you need to support the old `make` system.

Note: An easy way is to directly copy files of the existing development boards in `Boards` and make simple modifications to add your new board.

2.3.5 Component Dependencies

- Common dependencies: bus, button, led_strip

2.3.6 Adapted IDF Versions

- ESP-IDF v4.4 and later versions.

2.3.7 Supported Chips

- ESP32
- ESP32-S2
- ESP32-S3

2.4 CMake Utilities

`cmake_utilities` is a collection of CMake utilities that are commonly used in the components of `esp-iot-solution`.

Supported Features:

- `project_include.cmake`: add additional features like `DIAGNOSTICS_COLOR` to the project. The file will be automatically parsed, for details, please refer project-include-cmake.
- `package_manager.cmake`: provides functions to manager components' versions, etc.
- `gcc.cmake`: manager the GCC compiler options like `LTO` through menuconfig.
- `relinker.cmake` provides a way to move IRAM functions to flash to save RAM space.
- `gen_compressed_ota.cmake`: add new command `idf.py gen_compressed_ota` to generate `xz` compressed OTA binary. please refer xz.
- `gen_single_bin.cmake`: add new command `idf.py gen_single_bin` to generate single combined bin file (combine app, bootloader, partition table, etc).

2.4.1 How to Use

1. Add dependency of this component in your component or project' s `idf_component.yml`.

```yaml
dependencies:
  espressif/cmake_utilities: "0.*"
```

2. Include the CMake Features/Script in your component' s or project' s `CMakeLists.txt` after `idf_component_register`.

```
// Note: should remove .cmake postfix when using include(), otherwise the␣
↪requested file will not found
// Note: should place this line after `idf_component_register` function
// only include the one you needed.
include(package_manager)
```

3. Then you can use the corresponding features provided by the CMake script.

2.4.2 Detailed Reference

1. relinker
2. gen_compressed_ota
3. GCC Optimization

Chapter 3

Bluetooth

3.1 BLE Connection Management

Supported Socs	ESP32	ESP32-C2	ESP32-C3	ESP32-S3

The BLE connection management provides a simplified API interface for accessing the commonly used BLE functionality. It supports common scenarios like peripheral, central among others.

3.1.1 Application Example

```
esp_ble_conn_config_t config = {
    .device_name = CONFIG_EXAMPLE_BLE_ADV_NAME,
    .broadcast_data = CONFIG_EXAMPLE_BLE_SUB_ADV
};

ESP_ERROR_CHECK(esp_ble_conn_init(&config));
ESP_ERROR_CHECK(esp_ble_conn_start());
```

3.1.2 Examples

1. BLE periodic advertiser example: bluetooth/ble_conn_mgr/ble_periodic_adv.
2. BLE periodic sync example: bluetooth/ble_conn_mgr/ble_periodic_sync.
3. BLE serial port profile example: bluetooth/ble_conn_mgr/ble_spp.
4. BLE L2CAP CoC Central example: bluetooth/ble_l2cap_coc/l2cap_coc_central.
5. BLE L2CAP CoC Peripheral example: bluetooth/ble_l2cap_coc/l2cap_coc_peripheral.

3.1.3 API Reference

Header File

- components/bluetooth/ble_conn_mgr/include/esp_ble_conn_mgr.h

Functions

esp_err_t **esp_ble_conn_l2cap_coc_mem_init** (void)

Initialize L2CAP CoC SDU memory pool.

Note: This API is thread-safe, but should be called from task context. Repeated calls are safe and return ESP_OK.

Returns
- ESP_OK on success
- ESP_ERR_TIMEOUT if lock acquisition times out
- ESP_ERR_NOT_SUPPORTED if L2CAP CoC is disabled
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_l2cap_coc_mem_release** (void)

Release L2CAP CoC SDU memory pool.

Note: This API is thread-safe, but should be called from task context. It must only be called when there are no active L2CAP CoC servers or channels.

Returns
- ESP_OK on success
- ESP_ERR_TIMEOUT if lock acquisition times out
- ESP_ERR_INVALID_STATE if active L2CAP CoC contexts exist
- ESP_ERR_NOT_SUPPORTED if L2CAP CoC is disabled
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_l2cap_coc_accept** (uint16_t conn_handle, uint16_t peer_sdu_size, *esp_ble_conn_l2cap_coc_chan_t* chan)

Provide receive buffer for L2CAP CoC channel (accept or next SDU)

Parameters
- **conn_handle** –**[in]** Connection handle
- **peer_sdu_size** –**[in]** Peer SDU size from accept event
- **chan** –**[in]** L2CAP CoC channel

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_NO_MEM if memory pool not initialized or allocation failed
- ESP_ERR_NOT_SUPPORTED if L2CAP CoC is disabled
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_init** (*esp_ble_conn_config_t* *config)

Initialization *BLE* connection management based on the configuration This function must be the first function to call, This call MUST have a corresponding call to esp_ble_conn_deinit when the operation is complete.

Parameters **config** –**[in]** The configurations, see `esp_ble_conn_config_t`.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

esp_err_t **esp_ble_conn_deinit** (void)

Deinitialization *BLE* connection management This function must be the last function to call, It is the opposite of the esp_ble_conn_init function.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong deinitialization
- ESP_FAIL on error

esp_err_t **esp_ble_conn_start** (void)

Starts *BLE* session.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong start
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_stop** (void)

Stop *BLE* session.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong stop
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_set_mtu** (uint16_t mtu)

Legacy API to set preferred MTU.

This API is kept for backward compatibility and is a no-op in multi-connection implementation. Use esp_ble_conn_mtu_update(conn_handle, mtu) instead.

Parameters **mtu** –**[in]** The maximum transmission unit value to update

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong update
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_connect** (void)

This api is typically used to connect actively.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if BLE connection manager is not initialized
- ESP_ERR_TIMEOUT if connect event is not received within wait timeout
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_disconnect** (void)

This api is typically used to disconnect actively.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if BLE connection manager is not initialized
- ESP_ERR_NOT_FOUND if no active connection exists
- ESP_ERR_TIMEOUT if disconnect event is not received within wait timeout
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_connect_to_addr** (const uint8_t peer_addr[6], uint8_t peer_addr_type)

Initiate BLE connection to a peer (Central role).

Note: Only one outgoing connection attempt is allowed at a time. Returns ESP_ERR_INVALID_STATE if a connection attempt is already in progress.

Parameters
- **peer_addr** –**[in]** Peer BLE address (6 bytes)
- **peer_addr_type** –**[in]** Peer address type: esp_ble_conn_addr_type_t

Returns
- ESP_OK on success (connection initiated; result via ESP_BLE_CONN_EVENT_CONNECTED/DISCONNECTED)
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if max connections reached, peer already connected/pending, or another attempt in progress
- ESP_ERR_NOT_SUPPORTED if Central role is not enabled
- ESP_FAIL if connection initiation fails

esp_err_t **esp_ble_conn_disconnect_by_handle**(uint16_t conn_handle)

Disconnect a BLE connection by handle.

Parameters **conn_handle** –**[in]** Connection handle to terminate (valid range 0x0000–0x0EFF)

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if conn_handle is BLE_CONN_HANDLE_INVALID or out of range
- ESP_FAIL if disconnect fails

esp_err_t **esp_ble_conn_get_conn_handle**(uint16_t *out_conn_handle)

Get current (default) connection handle (read-only).

In multi-connection mode this returns the first/primary connection handle, or BLE_CONN_HANDLE_INVALID if none.

Parameters **out_conn_handle** –**[out]** Pointer to store the handle. Set to BLE_CONN_HANDLE_INVALID when not connected.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if out_conn_handle is NULL

esp_err_t **esp_ble_conn_get_conn_handle_by_addr**(const uint8_t peer_addr[6], uint8_t peer_addr_type, uint16_t *out_conn_handle)

Get connection handle by peer address.

Use this in multi-connection scenarios to resolve a peer address to its connection handle (valid range 0x0000–0x0EFF).

Parameters
- **peer_addr** –**[in]** Peer BLE address (6 bytes)
- **peer_addr_type** –**[in]** Peer address type: esp_ble_conn_addr_type_t
- **out_conn_handle** –**[out]** Pointer to store the connection handle for that peer

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if peer_addr or out_conn_handle is NULL
- ESP_ERR_NOT_FOUND if no connection exists for the given peer address

esp_err_t **esp_ble_conn_get_mtu**(uint16_t *out_mtu)

Get current negotiated GATT MTU (read-only).

Parameters **out_mtu** –**[out]** Pointer to store current MTU. Defaults to 23 if unknown.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if out_mtu is NULL

esp_err_t **esp_ble_conn_get_mtu_by_handle**(uint16_t conn_handle, uint16_t *out_mtu)

Get negotiated GATT MTU by connection handle.

Parameters
- **conn_handle** –**[in]** Connection handle (valid range 0x0000–0x0EFF; BLE_CONN_HANDLE_INVALID is invalid)
- **out_mtu** –**[out]** Pointer to store MTU

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters (e.g. conn_handle out of range)
- ESP_ERR_NOT_FOUND if connection does not exist

esp_err_t **esp_ble_conn_get_peer_addr**(uint8_t out_addr[6])

Get current peer address (read-only).

Parameters **out_addr** –**[out]** Pointer to store peer address (6 bytes).

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if out_addr is NULL
- ESP_ERR_NOT_FOUND if not connected

esp_err_t **esp_ble_conn_get_peer_addr_by_handle**(uint16_t conn_handle, uint8_t out_peer_addr[6], uint8_t *out_peer_addr_type)

Get peer address by connection handle.

Parameters
- **conn_handle** –**[in]** Connection handle
- **out_peer_addr** –**[out]** Pointer to store peer address (6 bytes)
- **out_peer_addr_type** –**[out]** Pointer to store peer address type (can be NULL)

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_NOT_FOUND if connection does not exist

esp_err_t **esp_ble_conn_find_conn_handle_by_peer_addr**(const uint8_t peer_addr[6], uint8_t peer_addr_type, uint16_t *out_conn_handle)

Find connection handle by peer address.

Parameters
- **peer_addr** –**[in]** Peer address (6 bytes)
- **peer_addr_type** –**[in]** Peer address type
- **out_conn_handle** –**[out]** Pointer to store matched connection handle

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_NOT_FOUND if no matched connection exists

esp_err_t **esp_ble_conn_find_peer_addr_by_conn_handle**(uint16_t conn_handle, uint8_t out_peer_addr[6], uint8_t *out_peer_addr_type)

Find peer address by connection handle (deprecated).

Deprecated:

Use esp_ble_conn_get_peer_addr_by_handle() instead.

Parameters
- **conn_handle** –**[in]** Connection handle
- **out_peer_addr** –**[out]** Output peer address (6 bytes)
- **out_peer_addr_type** –**[out]** Output peer address type (can be NULL)

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_NOT_FOUND if the connection handle does not exist

esp_err_t **esp_ble_conn_get_disconnect_reason**(uint16_t *out_reason)

Get last disconnect reason (read-only).

Parameters **out_reason** –**[out]** Pointer to store last disconnect reason.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if out_reason is NULL
- ESP_ERR_NOT_FOUND if no disconnect reason is available

esp_err_t **esp_ble_conn_get_disconnect_reason_by_handle**(uint16_t conn_handle, uint16_t *out_reason)

Get disconnect reason by connection handle.

Parameters

- **conn_handle** –**[in]** Connection handle
- **out_reason** –**[out]** Pointer to store disconnect reason

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_NOT_FOUND if reason is unavailable for the connection

esp_err_t **esp_ble_conn_update_params**(uint16_t conn_handle, const *esp_ble_conn_params_t* *params)

Request connection parameter update.

Parameters

- **conn_handle** –**[in]** Connection handle (use esp_ble_conn_get_conn_handle to query)
- **params** –**[in]** Desired connection parameters (min_ce_len and max_ce_len are forwarded to NimBLE).

Returns

- ESP_OK on success (request accepted)
- ESP_ERR_INVALID_ARG on wrong parameters
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_get_phy**(uint16_t conn_handle, uint8_t *out_tx_phy, uint8_t *out_rx_phy)

Get current LE PHY for a connection.

Parameters

- **conn_handle** –**[in]** Connection handle (valid range 0x0000–0x0EFF).
- **out_tx_phy** –**[out]** Pointer to store TX PHY (esp_ble_conn_phy_t: 1=1M, 2=2M, 3=Coded). Can be NULL.
- **out_rx_phy** –**[out]** Pointer to store RX PHY (esp_ble_conn_phy_t). Can be NULL.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if conn_handle is invalid or both out_tx_phy and out_rx_phy are NULL
- ESP_ERR_NOT_FOUND if connection does not exist
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_set_preferred_phy**(uint16_t conn_handle, uint8_t tx_phys_mask, uint8_t rx_phys_mask, uint16_t phy_opts)

Set preferred LE PHY for a connection.

The controller may not apply the change if the peer does not support the requested PHY.

Parameters

- **conn_handle** –**[in]** Connection handle (valid range 0x0000–0x0EFF).
- **tx_phys_mask** –**[in]** Preferred TX PHY mask: ESP_BLE_CONN_PHY_MASK_1M, _2M, _CODED, or _ALL (or combination).
- **rx_phys_mask** –**[in]** Preferred RX PHY mask (same as tx_phys_mask).
- **phy_opts** –**[in]** Coded PHY option when using CODED: ESP_BLE_CONN_PHY_CODED_OPT_ANY, _S2, or _S8; otherwise 0.

Returns

- ESP_OK on success (request accepted)
- ESP_ERR_INVALID_ARG if conn_handle is invalid or out of range
- ESP_ERR_NOT_FOUND if connection does not exist
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_set_data_len** (uint16_t conn_handle, uint16_t tx_octets, uint16_t tx_time)

Set preferred data length for a connection (LE Data Length Extension).

Sends LE Set Data Length command to the controller. The controller negotiates with the peer asynchronously. Per BLE spec: tx_octets 27–251 bytes, tx_time 328–17040 microseconds.

Parameters
- **conn_handle** –**[in]** Connection handle (valid range 0x0000–0x0EFF).
- **tx_octets** –**[in]** Preferred TX payload size in bytes (27–251).
- **tx_time** –**[in]** Preferred TX time in microseconds (328–17040).

Returns
- ESP_OK on success (command accepted)
- ESP_ERR_INVALID_ARG if conn_handle or parameters are out of range
- ESP_ERR_NOT_FOUND if connection does not exist
- ESP_ERR_NOT_SUPPORTED if Data Length Extension is not supported
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_adv_params_set** (const *esp_ble_conn_adv_params_t* *params)

Configure advertising parameters.

Parameters **params** –**[in]** Advertising parameters (NULL to use defaults)

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if BLE connection manager is not initialized

esp_err_t **esp_ble_conn_adv_data_set** (const uint8_t *data, uint16_t len)

Set advertising data (legacy or extended).

Parameters
- **data** –**[in]** Raw advertising data (format: length-byte + type + value per AD structure). Can be NULL to clear.
- **len** –**[in]** Data length in bytes. Max: `ESP_BLE_CONN_ADV_DATA_MAX_LEN` (31) for legacy; larger when extended advertising is enabled (platform-dependent).

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if BLE connection manager is not initialized
- ESP_ERR_INVALID_ARG if len exceeds maximum or data is NULL while len > 0
- ESP_ERR_NO_MEM on allocation failure

esp_err_t **esp_ble_conn_scan_rsp_data_set** (const uint8_t *data, uint16_t len)

Set scan response data (legacy or extended).

Parameters
- **data** –**[in]** Raw scan response data (format: length-byte + type + value per AD structure). Can be NULL to clear.
- **len** –**[in]** Data length in bytes. Max: `ESP_BLE_CONN_ADV_DATA_MAX_LEN` (31) for legacy; larger when extended advertising is enabled (platform-dependent).

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if BLE connection manager is not initialized
- ESP_ERR_INVALID_ARG if len exceeds maximum or data is NULL while len > 0
- ESP_ERR_NO_MEM on allocation failure

esp_err_t **esp_ble_conn_periodic_adv_data_set** (const uint8_t *data, uint16_t len)

Set periodic advertising data (extended advertising only).

Parameters

- **data** –**[in]** Raw periodic advertising data. Can be NULL to clear.
- **len** –**[in]** Data length in bytes (platform-dependent max when CONFIG_BLE_CONN_MGR_PERIODIC_ADV is set).

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if BLE connection manager is not initialized
- ESP_ERR_INVALID_ARG if len exceeds maximum or data is NULL while len > 0
- ESP_ERR_NO_MEM on allocation failure
- ESP_ERR_NOT_SUPPORTED if periodic advertising is not enabled

esp_err_t **esp_ble_conn_adv_start** (void)

Start advertising (peripheral role).

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if not initialized or not in peripheral role
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_adv_stop** (void)

Stop advertising (peripheral role).

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if not initialized
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_scan_params_set** (const *esp_ble_conn_scan_params_t* *params)

Configure scan parameters.

Parameters **params** –**[in]** Scan parameters (NULL to use defaults)

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if BLE connection manager is not initialized

esp_err_t **esp_ble_conn_scan_start** (void)

Start scanning (central role).

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if not initialized or not in central role
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_scan_stop** (void)

Stop ongoing BLE scan (central role).

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if scan is not active
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_mtu_update** (uint16_t conn_handle, uint16_t mtu)

Initiate MTU exchange on a connection.

Sets preferred MTU and initiates exchange (central) or sets preferred for future (peripheral).

Parameters
- **conn_handle** –**[in]** Connection handle
- **mtu** –**[in]** Desired MTU (BLE_ATT_MTU_DFLT to BLE_ATT_MTU_MAX, typically 512)

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_set_local_passkey**(uint32_t passkey)

Set local passkey for pairing (DISP mode).

When the device displays a passkey during pairing (action DISP), this key is used and auto-injected. Set to 0 to disable (application handles via esp_ble_conn_passkey_reply).

Parameters **passkey** –**[in]** 6-digit passkey (0-999999); 0 to clear

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if BLE connection manager is not initialized

esp_err_t **esp_ble_conn_security_initiate**(uint16_t conn_handle)

Initiate BLE security procedure.

Call this to start pairing. When ESP_BLE_CONN_EVENT_PASSKEY_ACTION occurs, use esp_ble_conn_passkey_reply() or esp_ble_conn_numcmp_reply() to respond.

Parameters **conn_handle** –**[in]** Connection handle

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if conn_handle is BLE_CONN_HANDLE_INVALID or out of range
- ESP_ERR_NOT_SUPPORTED if stack SMP is disabled
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_passkey_reply**(uint16_t conn_handle, uint32_t passkey)

Reply to passkey action.

Call when ESP_BLE_CONN_EVENT_PASSKEY_ACTION has action INPUT. Pass the passkey entered by the user.

Parameters

- **conn_handle** –**[in]** Connection handle
- **passkey** –**[in]** 6-digit passkey (0-999999)

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_NOT_SUPPORTED if NimBLE SMP is disabled
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_numcmp_reply**(uint16_t conn_handle, bool accept)

Reply to numeric comparison (NUMCMP).

Call when ESP_BLE_CONN_EVENT_PASSKEY_ACTION has action NUMCMP.

Parameters

- **conn_handle** –**[in]** Connection handle
- **accept** –**[in]** true to accept, false to reject

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_NOT_SUPPORTED if NimBLE SMP is disabled
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_l2cap_coc_create_server**(uint16_t psm, uint16_t mtu, *esp_ble_conn_l2cap_coc_event_cb_t* cb, void *cb_arg)

Create an L2CAP CoC server.

Note: The callback context is persistent for this PSM and is reused across incoming connections. It is not auto-freed until the L2CAP CoC memory pool is released.

Parameters

- **psm** –**[in]** Protocol/Service Multiplexer to listen on (must be in range 0x0001-0x00FF for LE)
- **mtu** –**[in]** Maximum SDU size to use for the server (must be in range 23-512 bytes)
- **cb** –**[in]** L2CAP CoC event callback
- **cb_arg** –**[in]** Callback argument

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters (NULL callback, PSM out of range 0x0001-0x00FF, or MTU out of range)
- ESP_ERR_NOT_SUPPORTED if L2CAP CoC is disabled
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_l2cap_coc_connect**(uint16_t conn_handle, uint16_t psm, uint16_t mtu, uint16_t sdu_size, *esp_ble_conn_l2cap_coc_event_cb_t* cb, void *cb_arg)

Initiate an L2CAP CoC connection.

Note: The callback context is auto-freed if the connection attempt fails (ESP_BLE_CONN_L2CAP_COC_EVENT_CONNECTED with non-zero status) and after ESP_BLE_CONN_L2CAP_COC_EVENT_DISCONNECTED.

Parameters

- **conn_handle** –**[in]** Connection handle (valid range 0x0000–0x0EFF; BLE_CONN_HANDLE_INVALID is invalid)
- **psm** –**[in]** Protocol/Service Multiplexer to connect to (must be in range 0x0001-0x00FF for LE)
- **mtu** –**[in]** Maximum SDU size to use for the connection (must be in range 23-512 bytes)
- **sdu_size** –**[in]** Receive buffer size for the first SDU (must be in range 23-65535 bytes). This size is used to allocate the initial RX buffer for the channel. Applications typically set it to the expected peer CoC MTU.
- **cb** –**[in]** L2CAP CoC event callback
- **cb_arg** –**[in]** Callback argument

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters (NULL callback, conn_handle/PSM/sdu_size out of range, or MTU out of range)
- ESP_ERR_NOT_SUPPORTED if L2CAP CoC is disabled
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_l2cap_coc_send**(*esp_ble_conn_l2cap_coc_chan_t* chan, const *esp_ble_conn_l2cap_coc_sdu_t* *sdu)

Send an SDU over an L2CAP CoC channel.

Note: Flow control: If the channel is stalled (buffer full), this function returns ESP_ERR_NOT_FINISHED. The caller should wait for ESP_BLE_CONN_L2CAP_COC_EVENT_TX_UNSTALLED event before retrying the send operation.

Note: Memory pool: If mbuf allocation fails (ESP_ERR_NO_MEM), it indicates the memory pool is exhausted. The caller should wait and retry later, or check if there are pending operations that need to complete.

Parameters

- **chan** –**[in]** L2CAP CoC channel

- **sdu** –**[in]** SDU to send (data buffer is not owned by the stack, can be freed immediately after this function returns)

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters (NULL channel, NULL sdu, invalid data pointer, or zero length)
- ESP_ERR_INVALID_STATE if memory pool not initialized or channel not connected
- ESP_ERR_NO_MEM if mbuf allocation failed (memory pool exhausted)
- ESP_ERR_NOT_FINISHED if channel is stalled or send would block (wait for TX_UNSTALLED event)
- ESP_ERR_NOT_SUPPORTED if L2CAP CoC is disabled
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_l2cap_coc_recv_ready**(*esp_ble_conn_l2cap_coc_chan_t* chan, uint16_t sdu_size)

Provide a receive buffer for an L2CAP CoC channel.

Parameters

- **chan** –**[in]** L2CAP CoC channel
- **sdu_size** –**[in]** Receive buffer size for next SDU

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_NO_MEM if memory pool not initialized or allocation failed
- ESP_ERR_NOT_SUPPORTED if L2CAP CoC is disabled
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_l2cap_coc_disconnect**(*esp_ble_conn_l2cap_coc_chan_t* chan)

Disconnect an L2CAP CoC channel.

Parameters **chan** –**[in]** L2CAP CoC channel

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_NOT_SUPPORTED if L2CAP CoC is disabled
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_l2cap_coc_get_chan_info**(*esp_ble_conn_l2cap_coc_chan_t* chan, *esp_ble_conn_l2cap_coc_chan_info_t* *chan_info)

Get L2CAP CoC channel information.

Parameters

- **chan** –**[in]** L2CAP CoC channel
- **chan_info** –**[out]** Channel info output

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_NOT_SUPPORTED if L2CAP CoC is disabled
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_register_scan_callback**(*esp_ble_conn_scan_cb_t* cb, void *cb_arg)

Register scan callback for BLE central role.

Note: When the callback returns true, a connection is initiated to that peer. Only one outgoing connection attempt is allowed at a time. If a connection attempt is already in progress, further scan results that would trigger connect are skipped until the current attempt completes (success or fail).

Parameters

- **cb** –**[in]** Scan callback (can be NULL to disable)
- **cb_arg** –**[in]** Callback argument

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters

esp_err_t **esp_ble_conn_parse_adv_data**(const uint8_t *adv_data, uint8_t adv_len, uint8_t ad_type, const uint8_t **out_data, uint8_t *out_len)

Parse AD type data from raw advertising data.

Parameters
- **adv_data** –**[in]** Raw advertising data
- **adv_len** –**[in]** Raw advertising data length
- **ad_type** –**[in]** Advertising data type
- **out_data** –**[out]** Output data pointer (points into adv_data)
- **out_len** –**[out]** Output data length

Returns
- ESP_OK if AD type is found
- ESP_ERR_NOT_FOUND if AD type is not present
- ESP_ERR_INVALID_ARG on invalid parameters

esp_err_t **esp_ble_conn_notify**(const *esp_ble_conn_data_t* *inbuff)

This api is typically used to notify actively.

Parameters **inbuff** –**[in]** The pointer to store notify data.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on wrong notify
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_notify_by_handle**(uint16_t conn_handle, const *esp_ble_conn_data_t* *inbuff)

Send notification/indication to a specific connection.

Note: The given `conn_handle` is the effective target; `inbuff->write_conn_id` is ignored. Use this API for multi-connection when sending to a specific link.

Parameters
- **conn_handle** –**[in]** Connection handle (valid range 0x0000–0x0EFF; BLE_CONN_HANDLE_INVALID is invalid).
- **inbuff** –**[in]** Notify payload (characteristic UUID and data).

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters (e.g. conn_handle out of range)
- ESP_ERR_NO_MEM if system resources are insufficient
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_read**(*esp_ble_conn_data_t* *outbuf)

This api is typically used to read actively.

Parameters **outbuf** –**[in]** The pointer to store read data.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on wrong read
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_read_by_handle**(uint16_t conn_handle, *esp_ble_conn_data_t* *outbuf)

Read characteristic on a specific connection.

Parameters

- **conn_handle** –**[in]** Connection handle (valid range 0x0000–0x0EFF; BLE_CONN_HANDLE_INVALID is invalid).
- **outbuf** –**[inout]** Read request/response buffer (UUID and optional write_conn_id; response in outbuf).

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters (e.g. conn_handle out of range)
- ESP_ERR_TIMEOUT if read completion is not received within wait timeout
- ESP_FAIL or stack-specific error code on read failure

esp_err_t **esp_ble_conn_write**(const *esp_ble_conn_data_t* *inbuff)

This api is typically used to write actively.

Parameters **inbuff** –**[in]** The pointer to store write data.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on wrong write
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_write_by_handle**(uint16_t conn_handle, const *esp_ble_conn_data_t* *inbuff)

Write characteristic on a specific connection.

Parameters

- **conn_handle** –**[in]** Connection handle (valid range 0x0000–0x0EFF; BLE_CONN_HANDLE_INVALID is invalid).
- **inbuff** –**[in]** Write payload (characteristic UUID and data).

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters (e.g. conn_handle out of range)
- ESP_ERR_NO_MEM if system resources are insufficient
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_subscribe**(*esp_ble_conn_desc_t* desc, const *esp_ble_conn_data_t* *inbuff)

This api is typically used to subscribe actively.

Parameters

- **desc** –**[in]** The declarations of descriptors
- **inbuff** –**[in]** The pointer to store subscribe data.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on wrong subscribe
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_subscribe_by_handle**(uint16_t conn_handle, *esp_ble_conn_desc_t* desc, const *esp_ble_conn_data_t* *inbuff)

Subscribe to notifications/indications on a specific connection.

Parameters

- **conn_handle** –**[in]** Connection handle (valid range 0x0000–0x0EFF; BLE_CONN_HANDLE_INVALID is invalid).
- **desc** –**[in]** Descriptor type (e.g. ESP_BLE_CONN_DESC_CIENT_CONFIG for CCCD).
- **inbuff** –**[in]** Subscribe payload (characteristic UUID and enable flags).

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters (e.g. conn_handle out of range)
- ESP_ERR_NO_MEM if system resources are insufficient
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_add_svc**(const *esp_ble_conn_svc_t* *svc)

This api is typically used to add service actively.

Parameters **svc** –**[in]** The pointer to store service.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on wrong add service
- ESP_FAIL on other error

esp_err_t **esp_ble_conn_remove_svc**(const *esp_ble_conn_svc_t* *svc)

This api is typically used to remove service actively.

Parameters **svc** –**[in]** The pointer to store service.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on wrong remove service
- ESP_FAIL on other error

Unions

union **esp_ble_conn_uuid_t**

#include <esp_ble_conn_mgr.h> Universal UUID, to be used for any-UUID static allocation.

Public Members

uint16_t **uuid16**

16 bit UUID of the service

uint32_t **uuid32**

32 bit UUID of the service

uint8_t **uuid128**[BLE_UUID128_VAL_LEN]

128 bit UUID of the service

Structures

struct **esp_ble_conn_l2cap_coc_chan_info_t**

L2CAP CoC channel information.

Public Members

uint16_t **scid**

Source Channel ID

uint16_t **dcid**

Destination Channel ID

uint16_t **our_l2cap_mtu**

Local L2CAP MTU

uint16_t **peer_l2cap_mtu**

Peer L2CAP MTU

uint16_t **psm**

Protocol/Service Multiplexer

uint16_t **our_coc_mtu**

Local CoC MTU

uint16_t **peer_coc_mtu**

Peer CoC MTU

struct **esp_ble_conn_l2cap_coc_sdu_t**

L2CAP CoC SDU buffer.

Public Members

uint16_t **len**

SDU length in bytes

uint8_t ***data**

SDU data buffer

struct **esp_ble_conn_scan_result_t**

BLE scan result data.

Public Members

uint8_t **addr**[6]

Peer address

uint8_t **addr_type**

Peer address type

int8_t **rssi**

RSSI in dBm

uint8_t **adv_data_len**

Advertising data length

uint8_t **adv_data**[ESP_BLE_CONN_ADV_DATA_MAX_LEN]

Advertising data

struct **esp_ble_conn_l2cap_coc_event**

L2CAP CoC event payload.

Public Members

esp_ble_conn_l2cap_coc_event_type_t **type**

Event type

int **status**

Connection status (0 on success)

TX status

Reconfigure status

uint16_t **conn_handle**

Connection handle

esp_ble_conn_l2cap_coc_chan_t **chan**

CoC channel

struct *esp_ble_conn_l2cap_coc_event*::[anonymous]::[anonymous] **connect**

Connected event data

struct *esp_ble_conn_l2cap_coc_event*::[anonymous]::[anonymous] **disconnect**

Disconnected event data

uint16_t **peer_sdu_size**

Peer SDU size

struct *esp_ble_conn_l2cap_coc_event*::[anonymous]::[anonymous] **accept**

Accept event data

esp_ble_conn_l2cap_coc_sdu_t **sdu**

SDU payload. Note: The sdu.data buffer is managed internally and will be freed immediately after the callback returns. The callback must not save the pointer for later use. If asynchronous processing is needed, the callback should copy the data.

struct *esp_ble_conn_l2cap_coc_event*::[anonymous]::[anonymous] **receive**

Receive event data

struct *esp_ble_conn_l2cap_coc_event*::[anonymous]::[anonymous] **tx_unstalled**

TX unstalled event data

struct *esp_ble_conn_l2cap_coc_event*::[anonymous]::[anonymous] **reconfigured**

Reconfigure event data

struct **esp_ble_conn_character_t**

This structure maps handler required by UUID which are used to BLE characteristics.

Public Members

const char ***name**

Name of the handler

uint8_t **type**

Type of the UUID

uint16_t **flag**

Flag for visiting right

esp_ble_conn_uuid_t **uuid**

Universal UUID, to be used for any-UUID static allocation

esp_ble_conn_cb_t **uuid_fn**

BLE UUID callback

struct **esp_ble_conn_svc_t**

This structure maps handler required by UUID which are used to BLE services.

Public Members

uint8_t **type**

Type of the UUID

uint16_t **nu_lookup_count**

Number of entries in the Name-UUID lookup table

esp_ble_conn_uuid_t **uuid**

Universal UUID, to be used for any-UUID static allocation

esp_ble_conn_character_t ***nu_lookup**

Pointer to the Name-UUID lookup table

struct **esp_ble_conn_config_t**

This structure maps handler required which are used to configure.

Public Members

uint8_t **device_name**[MAX_BLE_DEVNAME_LEN]

Local name for peripheral role (name broadcast when advertising)

uint8_t **remote_name**[MAX_BLE_DEVNAME_LEN]

Remote name for central role (target device name to connect to when scan_cb is NULL)

uint8_t **broadcast_data**[BROADCAST_PARAM_LEN]

BLE device manufacturer being announce

uint16_t **extended_adv_len**

Extended advertising data length

uint16_t **periodic_adv_len**

Periodic advertising data length

uint16_t **extended_adv_rsp_len**

Extended advertising responses data length

const char ***extended_adv_data**

Extended advertising data

const char ***periodic_adv_data**

Periodic advertising data

const char ***extended_adv_rsp_data**

Extended advertising responses data

uint16_t **include_service_uuid**

If include service UUID in advertising

esp_ble_conn_uuid_type_t **adv_uuid_type**

UUID type: BLE_CONN_UUID_TYPE_16 or BLE_CONN_UUID_TYPE_128.

uint16_t **adv_uuid16**

16-bit UUID (when adv_uuid_type is BLE_CONN_UUID_TYPE_16)

uint8_t **adv_uuid128**[BLE_UUID128_VAL_LEN]

128-bit UUID (when adv_uuid_type is BLE_CONN_UUID_TYPE_128)

union *esp_ble_conn_config_t*::[anonymous] **[anonymous]**

Service UUID for advertising

struct **esp_ble_conn_data_t**

This structure maps handler required by UUID which are used to data.

Public Members

uint8_t **type**

Type of the UUID

uint16_t **write_conn_id**

Connection handle ID; use BLE_CONN_HANDLE_INVALID to use default link

esp_ble_conn_uuid_t **uuid**

Universal UUID, to be used for any-UUID static allocation

uint8_t ***data**

Data buffer

uint16_t **data_len**

Data size

struct **esp_ble_conn_periodic_sync_t**

This structure represents a periodic advertising sync established during discovery procedure.

Public Members

uint8_t **status**

Periodic sync status

uint16_t **sync_handle**

Periodic sync handle

uint8_t **sid**

Advertising Set ID

uint8_t **adv_addr**[6]

Advertiser address

uint8_t **adv_phy**

Advertising PHY

uint16_t **per_adv_ival**

Periodic advertising interval

uint8_t **adv_clk_accuracy**

Advertiser clock accuracy

struct **esp_ble_conn_periodic_report_t**

This structure represents a periodic advertising report received on established sync.

Public Members

uint16_t **sync_handle**

Periodic sync handle

int8_t **tx_power**

Advertiser transmit power in dBm (127 if unavailable)

int8_t **rssi**

Received signal strength indication in dBm (127 if unavailable)

uint8_t **data_status**

Advertising data status

uint8_t **data_length**

Advertising Data length

const uint8_t ***data**

Advertising data

struct **esp_ble_conn_periodic_sync_lost_t**

This structure represents a periodic advertising sync lost of established sync.

Public Members

uint16_t **sync_handle**

Periodic sync handle

int **reason**

Reason for sync lost

struct **esp_ble_conn_cccd_update_t**

This structure represents CCCD (Client Characteristic Configuration Descriptor) update event.

Public Members

uint16_t **conn_handle**

Connection handle

uint16_t **char_handle**

Characteristic handle

uint8_t **uuid_type**

UUID type

esp_ble_conn_uuid_t **uuid**

Characteristic UUID

bool **notify_enable**

Notification enabled (bit 0)

bool **indicate_enable**

Indication enabled (bit 1)

struct **esp_ble_conn_params_t**

BLE connection parameters.

Public Members

uint16_t **itvl_min**

Connection event interval minimum. This is in units of 1.25ms. (0x0006 = 7.5ms). Valid range: 0x0006–0x0C80

uint16_t **itvl_max**

Connection event interval maximum. This is in units of 1.25ms. (0x0006 = 7.5ms). Valid range: 0x0006–0x0C80

uint16_t **latency**

Peripheral latency. This is in units of connection events to skip. Valid range: 0x0000–0x01F3

uint16_t **supervision_timeout**

Supervision timeout. This is in units of 10ms. (e.g. 400 = 4s). Valid range: 0x000A–0x0C80. Must be larger than the connection interval (itvl_max * 1.25ms)

uint16_t **min_ce_len**

Minimum connection event length. Valid range: 0x0000–0x0C80

uint16_t **max_ce_len**

Maximum connection event length. Valid range: 0x0000–0x0C80

struct **esp_ble_conn_adv_params_t**

BLE advertising parameters. Aligned with LE Set Extended Advertising Parameters (HCI). Interval units: 0.625ms. e.g. 0x100 = 100ms. Legacy advertising uses itvl_min, itvl_max, conn_mode, disc_mode; extended fields are applied when CONFIG_BLE_CONN_MGR_EXTENDED_ADV is enabled.

Public Members

uint8_t **adv_handle**

Advertising_Handle: instance index; 0 = use default (e.g. 1)

uint16_t **adv_event_properties**

Advertising_Event_Properties: BIT0=connectable, BIT1=scannable, BIT2=directed, BIT3=high_duty_directed, BIT4=legacy_pdu, BIT5=anonymous, BIT6=include_tx_power, BIT7=scan_req_notif

uint16_t **itvl_min**

Primary_Advertising_Interval_Min (0.625ms). Valid 0x0020–0x4000, 0 = default

uint16_t **itvl_max**

Primary_Advertising_Interval_Max (0.625ms). Valid 0x0020–0x4000, 0 = default

uint8_t **conn_mode**

Legacy: esp_ble_conn_adv_conn_mode_t

uint8_t **disc_mode**

Legacy: esp_ble_conn_adv_disc_mode_t

uint8_t **channel_map**

Primary_Advertising_Channel_Map: BIT0=ch37, BIT1=ch38, BIT2=ch39. 0 = default (all)

uint8_t **own_addr_type**

Own address type: esp_ble_conn_addr_type_t

uint8_t **peer_addr_type**

Peer address type for directed adv: esp_ble_conn_addr_type_t; 0 = not directed

uint8_t **peer_addr**[6]

Peer_Address for directed advertising

uint8_t **filter_policy**

Advertising filter: esp_ble_conn_scan_filt_policy_t

int8_t **tx_power**

Advertising_TX_Power in dBm; 127 = use stack default

uint8_t **primary_phy**

Primary advertising PHY: esp_ble_conn_phy_t

uint8_t **secondary_adv_max_skip**

Secondary_Advertising_Max_Skip (0 = send every primary)

uint8_t **secondary_phy**

Secondary advertising PHY: esp_ble_conn_phy_t

uint8_t **sid**

Advertising_SID (0–15)

uint8_t **scan_req_notif**

Scan_Request_Notification_Enable: 0 = disabled, non-zero = enabled

uint8_t **primary_phy_options**

Primary_Advertising_PHY_Options, 0 = default

uint8_t **secondary_phy_options**

Secondary_Advertising_PHY_Options, 0 = default

uint8_t **ext_adv_cap**

Deprecated: use adv_event_properties; applied when adv_event_properties is 0

struct **esp_ble_conn_scan_params_t**

BLE scan parameters. Aligned with LE Set Scan Parameters / Extended Scan (HCI). Interval/window units: 0.625ms. e.g. 0x12 = 11.25ms.

Public Members

uint8_t **own_addr_type**

Own address type: esp_ble_conn_addr_type_t

uint8_t **filter_policy**

Scanning filter: esp_ble_conn_scan_filt_policy_t

uint16_t **itvl**

Scan_Interval (0.625ms). 0 = use stack default

uint16_t **window**

Scan_Window (0.625ms). 0 = use stack default. Must be <= itvl

bool **passive**

Scan_Type: true = passive scan (no scan requests), false = active

bool **filter_duplicates**

true = filter duplicate advertisements

bool **limited**

true = limited discovery procedure

uint8_t **scan_phys**

Scanning PHYs (bitmask). 0 = standard scan (1M). Reserved and currently ignored by NimBLE backend

struct **esp_ble_conn_event_data_t**

Unified event data for BLE connection manager events. Use the union member corresponding to the event type (esp_ble_conn_event_t).

Public Members

uint16_t **conn_handle**

Connection handle

uint8_t **peer_addr**[6]

Peer address (big-endian)

Peer address

uint8_t **peer_addr_type**

Peer address type: esp_ble_conn_addr_type_t

Peer address type

struct *esp_ble_conn_event_data_t*::[anonymous]::[anonymous] **connected**

ESP_BLE_CONN_EVENT_CONNECTED

uint16_t **reason**

Disconnect reason (stack/HCI error code)

struct *esp_ble_conn_event_data_t*::[anonymous]::[anonymous] **disconnected**

ESP_BLE_CONN_EVENT_DISCONNECTED

esp_ble_conn_data_t **data_receive**

ESP_BLE_CONN_EVENT_DATA_RECEIVE

esp_ble_conn_periodic_report_t **periodic_report**

ESP_BLE_CONN_EVENT_PERIODIC_REPORT

esp_ble_conn_periodic_sync_lost_t **periodic_sync_lost**

ESP_BLE_CONN_EVENT_PERIODIC_SYNC_LOST

esp_ble_conn_periodic_sync_t **periodic_sync**

ESP_BLE_CONN_EVENT_PERIODIC_SYNC

esp_ble_conn_cccd_update_t **cccd_update**

ESP_BLE_CONN_EVENT_CCCD_UPDATE

uint16_t **channel_id**

GATT channel ID (0 for ATT)

uint16_t **mtu**

Negotiated MTU value

struct *esp_ble_conn_event_data_t*::[anonymous]::[anonymous] **mtu_update**

ESP_BLE_CONN_EVENT_MTU

int **status**

Update status (0 on success, non-zero on failure)

0 on success, non-zero on failure

esp_ble_conn_params_t **params**

Updated connection parameters

struct *esp_ble_conn_event_data_t*::[anonymous]::[anonymous] **conn_param_update**

ESP_BLE_CONN_EVENT_CONN_PARAM_UPDATE

esp_ble_conn_scan_result_t **scan_result**

ESP_BLE_CONN_EVENT_SCAN_RESULT

bool **encrypted**

Link is encrypted

bool **authenticated**

Link is authenticated

struct *esp_ble_conn_event_data_t*::[anonymous]::[anonymous] **enc_change**

ESP_BLE_CONN_EVENT_ENC_CHANGE

esp_ble_conn_sm_action_t **action**

Passkey action type, see esp_ble_conn_sm_action_t

uint32_t **passkey**

Passkey to display (DISP), or for user to enter (INPUT), or to compare (NUMCMP)

struct *esp_ble_conn_event_data_t*::[anonymous]::[anonymous] **passkey_action**

ESP_BLE_CONN_EVENT_PASSKEY_ACTION

Macros

BLE_CONN_MGR_ADDR_STR

BLE address format string for printf-style logging.

BLE_CONN_MGR_ADDR_HEX(addr)

BLE address formatter arguments for printf-style logging.

ESP_BLE_CONN_ADV_TYPE_FLAGS

Common advertising data types

ESP_BLE_CONN_ADV_TYPE_NAME_SHORT

ESP_BLE_CONN_ADV_TYPE_NAME_COMPLETE

ESP_BLE_CONN_ADV_TYPE_TX_POWER

ESP_BLE_CONN_ADV_TYPE_UUID16_INCOMP

ESP_BLE_CONN_ADV_TYPE_UUID16_COMPLETE

ESP_BLE_CONN_ADV_TYPE_UUID32_INCOMP

ESP_BLE_CONN_ADV_TYPE_UUID32_COMPLETE

ESP_BLE_CONN_ADV_TYPE_UUID128_INCOMP

ESP_BLE_CONN_ADV_TYPE_UUID128_COMPLETE

ESP_BLE_CONN_ADV_TYPE_SERVICE_DATA16

ESP_BLE_CONN_ADV_TYPE_SERVICE_DATA32

ESP_BLE_CONN_ADV_TYPE_SERVICE_DATA128

ESP_BLE_CONN_ADV_TYPE_APPEARANCE

ESP_BLE_CONN_ADV_TYPE_MANUFACTURER

ESP_BLE_CONN_ADV_DATA_MAX_LEN

BLE_CONN_HANDLE_INVALID

Invalid connection handle (no connection). Use to check "not connected" or invalid handle.

BLE_CONN_HANDLE_MAX

Maximum valid connection handle. Valid range is 0x0000 –BLE_CONN_HANDLE_MAX (0x0EFF).

ESP_IOT_ATT_SUCCESS

Success code and error codes

ESP_IOT_ATT_INVALID_HANDLE

ESP_IOT_ATT_READ_NOT_PERMIT

ESP_IOT_ATT_WRITE_NOT_PERMIT

ESP_IOT_ATT_INVALID_PDU

ESP_IOT_ATT_INSUF_AUTHENTICATION

ESP_IOT_ATT_REQ_NOT_SUPPORTED

ESP_IOT_ATT_INVALID_OFFSET

ESP_IOT_ATT_INSUF_AUTHORIZATION

ESP_IOT_ATT_PREPARE_Q_FULL

ESP_IOT_ATT_NOT_FOUND

ESP_IOT_ATT_NOT_LONG

ESP_IOT_ATT_INSUF_KEY_SIZE

ESP_IOT_ATT_INVALID_ATTR_LEN

ESP_IOT_ATT_ERR_UNLIKELY

ESP_IOT_ATT_INSUF_ENCRYPTION

ESP_IOT_ATT_UNSUPPORT_GRP_TYPE

ESP_IOT_ATT_INSUF_RESOURCE

ESP_IOT_ATT_DATABASE_OUT_OF_SYNC

ESP_IOT_ATT_VALUE_NOT_ALLOWED

ESP_IOT_ATT_NO_RESOURCES

Application error codes

ESP_IOT_ATT_INTERNAL_ERROR

ESP_IOT_ATT_WRONG_STATE

ESP_IOT_ATT_DB_FULL

ESP_IOT_ATT_BUSY

ESP_IOT_ATT_ERROR

ESP_IOT_ATT_CMD_STARTED

ESP_IOT_ATT_ILLEGAL_PARAMETER

ESP_IOT_ATT_PENDING

ESP_IOT_ATT_AUTH_FAIL

ESP_IOT_ATT_MORE

ESP_IOT_ATT_INVALID_CFG

ESP_IOT_ATT_SERVICE_STARTED

ESP_IOT_ATT_ENCRYPED_MITM

ESP_IOT_ATT_ENCRYPED_NO_MITM

ESP_IOT_ATT_NOT_ENCRYPTED

ESP_IOT_ATT_CONGESTED

ESP_IOT_ATT_DUP_REG

ESP_IOT_ATT_ALREADY_OPEN

ESP_IOT_ATT_CANCEL

ESP_IOT_ATT_STACK_RSP

Common profile and service error codes

ESP_IOT_ATT_APP_RSP

ESP_IOT_ATT_UNKNOWN_ERROR

ESP_IOT_ATT_CCC_CFG_ERR

ESP_IOT_ATT_PRC_IN_PROGRESS

ESP_IOT_ATT_OUT_OF_RANGE

MAX_BLE_DEVNAME_LEN

BLE device name cannot be larger than this value 31 bytes (max scan response size) - 1 byte (length) - 1 byte (type) = 29 bytes BLE device name length

BLE_UUID128_VAL_LEN

128 bit UUID length

BROADCAST_PARAM_LEN

BLE device broadcast param length

BLE_CONN_GATT_CHR_BROADCAST

Characteristic broadcast properties

BLE_CONN_GATT_CHR_READ

Characteristic read properties

BLE_CONN_GATT_CHR_WRITE_NO_RSP

Characteristic write properties

BLE_CONN_GATT_CHR_WRITE

Characteristic write properties

BLE_CONN_GATT_CHR_NOTIFY

Characteristic notify properties

BLE_CONN_GATT_CHR_INDICATE

Characteristic indicate properties

BLE_CONN_GATT_CHR_READ_ENC

Characteristic read requires encryption

BLE_CONN_GATT_CHR_READ_AUTHEN

Characteristic read requires authentication

BLE_CONN_GATT_CHR_READ_AUTHOR

Characteristic read requires authorization

BLE_CONN_GATT_CHR_WRITE_ENC

Characteristic write requires encryption

BLE_CONN_GATT_CHR_WRITE_AUTHEN

Characteristic write requires authentication

BLE_CONN_GATT_CHR_WRITE_AUTHOR

Characteristic write requires authorization

ESP_BLE_CONN_PHY_MASK_1M

LE PHY preference mask (for set preferred PHY). Values match the underlying stack. Prefer 1M PHY

ESP_BLE_CONN_PHY_MASK_2M

Prefer 2M PHY

ESP_BLE_CONN_PHY_MASK_CODED

Prefer Coded PHY

ESP_BLE_CONN_PHY_MASK_ALL

All PHYs (1M | 2M | Coded)

ESP_BLE_CONN_PHY_CODED_OPT_ANY

Coded PHY option (phy_opts when using ESP_BLE_CONN_PHY_MASK_CODED). Coded: any

ESP_BLE_CONN_PHY_CODED_OPT_S2

Coded: S=2

ESP_BLE_CONN_PHY_CODED_OPT_S8

Coded: S=8

BLE_UUID_CMP(type, src, dst)

BLE_UUID_TYPE(type)

MIN(a, b)

Type Definitions

typedef void ***esp_ble_conn_l2cap_coc_chan_t**

L2CAP CoC channel handle (opaque pointer)

> **Note:** This is an opaque handle managed by the stack. Applications must not dereference, cast, or persist it beyond the lifetime of the associated connection. Use it only with the L2CAP CoC APIs provided here.

typedef bool (***esp_ble_conn_scan_cb_t**)(const *esp_ble_conn_scan_result_t* *result, void *arg)

typedef struct *esp_ble_conn_l2cap_coc_event* **esp_ble_conn_l2cap_coc_event_t**

L2CAP CoC event payload.

typedef int (***esp_ble_conn_l2cap_coc_event_cb_t**)(*esp_ble_conn_l2cap_coc_event_t* *event, void *arg)

L2CAP CoC event callback.

Note: For ESP_BLE_CONN_L2CAP_COC_EVENT_DATA_RECEIVED events, the sdu.data buffer is managed internally and will be freed immediately after this callback returns. The callback must not save the pointer for later use. If asynchronous processing is needed, the callback should copy the data before returning.

Param event **[in]** L2CAP CoC event payload
Param arg **[in]** User context
Return
- 0 to indicate the event was handled successfully
- Non-zero to indicate an application error (stack behavior may vary by event type)

typedef esp_err_t (***esp_ble_conn_cb_t**)(const uint8_t *inbuf, uint16_t inlen, uint8_t **outbuf, uint16_t *outlen, void *priv_data, uint8_t *att_status)

This is type of function that will handle the registered characteristic.

Param inbuf **[in]** The pointer to store data: read operation if NULL or write operation if not NULL
Param inlen **[in]** The store data length
Param outbuf **[out]** Variable to store data, it' ll free by connection management component
Param outlen **[out]** Variable to store data length
Param priv_data **[in]** Private data context
Param att_status **[in]** The attribute return status
Return
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter
- ESP_FAIL on error

Enumerations

enum **esp_ble_conn_sm_action_t**

Passkey action types for Security Manager pairing.

Values:

enumerator **ESP_BLE_CONN_SM_ACT_NONE**

No action / unknown / other

enumerator **ESP_BLE_CONN_SM_ACT_OOB**

Out-of-band (OOB) data required

enumerator **ESP_BLE_CONN_SM_ACT_INPUT**

User must enter passkey; call esp_ble_conn_passkey_reply()

enumerator **ESP_BLE_CONN_SM_ACT_DISP**

Display passkey to user, application should display the passkey

enumerator **ESP_BLE_CONN_SM_ACT_NUMCMP**

Numeric comparison; call esp_ble_conn_numcmp_reply()

enum **esp_ble_conn_l2cap_coc_event_type_t**

L2CAP CoC event types.

Values:

enumerator **ESP_BLE_CONN_L2CAP_COC_EVENT_CONNECTED**

CoC channel connected

enumerator **ESP_BLE_CONN_L2CAP_COC_EVENT_DISCONNECTED**

CoC channel disconnected

enumerator **ESP_BLE_CONN_L2CAP_COC_EVENT_ACCEPT**

CoC connection accepted

enumerator **ESP_BLE_CONN_L2CAP_COC_EVENT_DATA_RECEIVED**

SDU received

enumerator **ESP_BLE_CONN_L2CAP_COC_EVENT_TX_UNSTALLED**

TX unstalled

enumerator **ESP_BLE_CONN_L2CAP_COC_EVENT_RECONFIG_COMPLETED**

Local reconfigure completed

enumerator **ESP_BLE_CONN_L2CAP_COC_EVENT_PEER_RECONFIGURED**

Peer reconfigured

enum **esp_ble_conn_adv_conn_mode_t**

Legacy advertising connection mode (conn_mode in *esp_ble_conn_adv_params_t*). Values match the underlying BLE stack.

Values:

enumerator **ESP_BLE_CONN_ADV_CONN_MODE_NON**

Non-connectable

enumerator **ESP_BLE_CONN_ADV_CONN_MODE_DIR**

Connectable directed

enumerator **ESP_BLE_CONN_ADV_CONN_MODE_UND**

Connectable undirected

enum **esp_ble_conn_adv_disc_mode_t**

Legacy advertising discoverable mode (disc_mode in *esp_ble_conn_adv_params_t*)

Values:

enumerator **ESP_BLE_CONN_ADV_DISC_MODE_NON**

Non-discoverable

enumerator **ESP_BLE_CONN_ADV_DISC_MODE_LTD**

Limited discoverable

enumerator **ESP_BLE_CONN_ADV_DISC_MODE_GEN**

General discoverable

enum **esp_ble_conn_addr_type_t**

BLE address type (for own_addr_type, peer_addr_type in adv/scan/event data)

Values:

enumerator **ESP_BLE_CONN_ADDR_PUBLIC**

Public device address

enumerator **ESP_BLE_CONN_ADDR_RANDOM**

Random device address

enum **esp_ble_conn_phy_t**

LE PHY (primary_phy, secondary_phy in adv params; scan_phys bitmask)

Values:

enumerator **ESP_BLE_CONN_PHY_1M**

1M PHY

enumerator **ESP_BLE_CONN_PHY_2M**

2M PHY

enumerator **ESP_BLE_CONN_PHY_CODED**

Coded PHY

enum **esp_ble_conn_scan_filt_policy_t**

Scan filter policy (filter_policy in scan params)

Values:

enumerator **ESP_BLE_CONN_SCAN_FILT_NO_WL**

Accept any advertiser

enumerator **ESP_BLE_CONN_SCAN_FILT_USE_WL**

Use whitelist only

enum **esp_ble_conn_event_t**

Type of event.

Values:

enumerator **ESP_BLE_CONN_EVENT_UNKNOWN**

Unknown event

enumerator **ESP_BLE_CONN_EVENT_STARTED**

When BLE connection management start, the event comes

enumerator **ESP_BLE_CONN_EVENT_STOPPED**

When BLE connection management stop, the event comes

enumerator **ESP_BLE_CONN_EVENT_CONNECTED**

When a new connection was established, the event comes

enumerator **ESP_BLE_CONN_EVENT_DISCONNECTED**

When a connection was terminated, the event comes

enumerator **ESP_BLE_CONN_EVENT_DATA_RECEIVE**

When receive a notification or indication data, the event comes

enumerator **ESP_BLE_CONN_EVENT_DISC_COMPLETE**

When the ble discover service complete, the event comes

enumerator **ESP_BLE_CONN_EVENT_PERIODIC_REPORT**

When the periodic adv report, the event comes

enumerator **ESP_BLE_CONN_EVENT_PERIODIC_SYNC_LOST**

When the periodic sync lost, the event comes

enumerator **ESP_BLE_CONN_EVENT_PERIODIC_SYNC**

When the periodic sync, the event comes

enumerator **ESP_BLE_CONN_EVENT_CCCD_UPDATE**

When CCCD (Client Characteristic Configuration Descriptor) is written, the event comes

enumerator **ESP_BLE_CONN_EVENT_MTU**

When GATT MTU is negotiated/updated, the event comes

enumerator **ESP_BLE_CONN_EVENT_CONN_PARAM_UPDATE**

When connection parameters are updated, the event comes

enumerator **ESP_BLE_CONN_EVENT_SCAN_RESULT**

When a BLE advertisement is received during scan (central role), the event comes

enumerator **ESP_BLE_CONN_EVENT_ENC_CHANGE**

When link encryption state changes (pairing/encryption complete or failed), the event comes

enumerator **ESP_BLE_CONN_EVENT_PASSKEY_ACTION**

When passkey input/display/confirm is needed during pairing, the event comes

enum **esp_ble_conn_desc_t**

Type of descriptors.

Values:

enumerator **ESP_BLE_CONN_DESC_UNKNOWN**

Unknown descriptors

enumerator **ESP_BLE_CONN_DESC_EXTENDED**

Characteristic Extended Properties

enumerator **ESP_BLE_CONN_DESC_USER**

Characteristic User Description

enumerator **ESP_BLE_CONN_DESC_CIENT_CONFIG**

Client Characteristic Configuration

enumerator **ESP_BLE_CONN_DESC_SERVER_CONFIG**

Server Characteristic Configuration

enumerator **ESP_BLE_CONN_DESC_PRE_FORMAT**

Characteristic Presentation Format

enumerator **ESP_BLE_CONN_DESC_AGG_FORMAT**

Characteristic Aggregate Format

enumerator **ESP_BLE_CONN_DESC_VALID_RANGE**

Valid Range

enumerator **ESP_BLE_CONN_DESC_EXTREPORT**

External Report Reference

enumerator **ESP_BLE_CONN_DESC_REPORT**

Report Reference

enumerator **ESP_BLE_CONN_DESC_DIGITAL**

Number of Digitals

enumerator **ESP_BLE_CONN_DESC_VALUE_TRIGGER**

Value Trigger Setting

enumerator **ESP_BLE_CONN_DESC_ENV_SENS_CONFIG**

Environmental Sensing Configuration

enumerator **ESP_BLE_CONN_DESC_ENV_SENS_MEASURE**

Environmental Sensing Measurement

enumerator **ESP_BLE_CONN_DESC_ENV_TRIGGER**

Environmental Sensing Trigger Setting

enumerator **ESP_BLE_CONN_DESC_TIME_TRIGGER**

Time Trigger Setting

enumerator **ESP_BLE_CONN_DESC_COMPLETE_BLOCK**

Complete BR-EDR Transport Block Data

enum **esp_ble_conn_uuid_type_t**

Type of UUID.

Values:

enumerator **BLE_CONN_UUID_TYPE_16**

16-bit UUID (BT SIG assigned)

enumerator **BLE_CONN_UUID_TYPE_32**

32-bit UUID (BT SIG assigned)

enumerator **BLE_CONN_UUID_TYPE_128**

128-bit UUID

3.2 BLE L2CAP Connection Oriented Channels (CoC)

Supported Socs	ESP32	ESP32-C2	ESP32-C3	ESP32-S3

L2CAP Connection Oriented Channels (CoC) is a BLE feature that enables reliable, stream-oriented data transfer between BLE devices. Unlike GATT, which is based on attribute protocol, L2CAP CoC provides a connection-oriented channel model similar to TCP, allowing for larger data transfers with flow control.

Key features of L2CAP CoC:

- **Connection-oriented**: Establishes dedicated channels between devices
- **Flow control**: Prevents data loss by controlling transmission rate
- **Larger MTU**: Supports MTU sizes up to 512 bytes (configurable)
- **Bidirectional**: Full-duplex communication on established channels
- **Multiple channels**: Support for multiple concurrent CoC channels

L2CAP CoC is particularly useful for applications that require:

- Large file transfers
- Audio/video streaming
- Bulk data synchronization
- Any application needing reliable, stream-oriented communication

3.2.1 Application Example

The PSM (Protocol/Service Multiplexer) value can be configured via Kconfig. In the example projects, it can be set in `Example Configuration → L2CAP CoC PSM` menu. Valid range: 0x0001-0x00FF (1-255).

- **0x0001-0x007F**: Fixed Bluetooth SIG-defined services
- **0x0080-0x00FF**: Dynamic custom services

Both central and peripheral devices must use the same PSM value to establish a connection.

```
// PSM value is configured via Kconfig (CONFIG_EXAMPLE_L2CAP_COC_PSM)
// Default value: 0x00EF (239)
#define L2CAP_COC_PSM CONFIG_EXAMPLE_L2CAP_COC_PSM
#define L2CAP_COC_MTU CONFIG_BLE_CONN_MGR_L2CAP_COC_MTU  // Maximum␣
↪Transmission Unit

// Initialize L2CAP CoC memory pool
```

(continues on next page)

(continued from previous page)

```
ESP_ERROR_CHECK(esp_ble_conn_l2cap_coc_mem_init());

// Create L2CAP CoC server
// app_ble_conn_l2cap_coc_event_handler is your callback function
esp_err_t rc = esp_ble_conn_l2cap_coc_create_server(
    L2CAP_COC_PSM, L2CAP_COC_MTU,
    app_ble_conn_l2cap_coc_event_handler, NULL);
```

3.2.2 Examples

1. BLE L2CAP CoC Central example: bluetooth/ble_l2cap_coc/l2cap_coc_central.
2. BLE L2CAP CoC Peripheral example: bluetooth/ble_l2cap_coc/l2cap_coc_peripheral.

3.2.3 API Reference

See *BLE Connection Management* for the L2CAP CoC API reference.

3.3 BLE Services

3.3.1 Alert Notification Service

The Alert Notification service exposes alert information in a device. This information includes the following:

1. Type of alert occurring in a device
2. Additional text information such as caller ID or sender ID
3. Count of new alerts
4. Count of unread alert items.

Examples

bluetooth/ble_services/ble_ans.

API Reference

Header File

- components/bluetooth/ble_services/ans/include/esp_ans.h

Functions

esp_err_t **esp_ble_ans_get_new_alert** (uint8_t cat_id, uint8_t *cat_val)

Read the value of or check supported new alert category.

Attention 1. When cat_id is 0xFF, read the value of supported new alert category.
Attention 2. When cat_id isn't 0xFF, check supported new alert category is enable or disable.

Parameters

- **cat_id** –**[in]** The ID of the category to read or check
- **cat_val** –**[out]** The value of read or check supported new alert category

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong category of the new alert

esp_err_t **esp_ble_ans_set_new_alert**(uint8_t cat_id, const char *cat_info)

Send a new alert notification to the given category with the given info string.

Parameters
- **cat_id** –The ID of the category to send the notification to
- **cat_info** –The info string to send with the notification

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong category of the new alert

esp_err_t **esp_ble_ans_get_unread_alert**(uint8_t cat_id, uint8_t *cat_val)

Read the value of or check supported unread alert status category.

Attention 1. When cat_id is 0xFF, read the value of supported unread alert status category.
Attention 2. When cat_id isn't 0xFF, check supported unread alert status category is enable or disable.

Parameters
- **cat_id** –**[in]** The ID of the category to read or check
- **cat_val** –**[out]** The value of read or check supported unread alert status category

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong category of the unread alert

esp_err_t **esp_ble_ans_set_unread_alert**(uint8_t cat_id)

Send an unread alert to the given category then notifies the client if the given category is valid and enabled.

Parameters **cat_id** –The ID of the category which should be incremented and notified

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong category of the unread alert

esp_err_t **esp_ble_ans_init**(void)

Initialization GATT Alert Notification Service.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

Macros

BLE_ANS_UUID16

BLE_ANS_CHR_UUID16_SUP_NEW_ALERT_CAT

BLE_ANS_CHR_UUID16_NEW_ALERT

BLE_ANS_CHR_UUID16_SUP_UNR_ALERT_CAT

BLE_ANS_CHR_UUID16_UNR_ALERT_STAT

BLE_ANS_CHR_UUID16_ALERT_NOT_CTRL_PT

BLE_ANS_CAT_BM_NONE

BLE_ANS_CAT_BM_SIMPLE_ALERT

BLE_ANS_CAT_BM_EMAIL

BLE_ANS_CAT_BM_NEWS

BLE_ANS_CAT_BM_CALL

BLE_ANS_CAT_BM_MISSED_CALL

BLE_ANS_CAT_BM_SMS

BLE_ANS_CAT_BM_VOICE_MAIL

BLE_ANS_CAT_BM_SCHEDULE

BLE_ANS_CAT_ID_SIMPLE_ALERT

BLE_ANS_CAT_ID_EMAIL

BLE_ANS_CAT_ID_NEWS

BLE_ANS_CAT_ID_CALL

BLE_ANS_CAT_ID_MISSED_CALL

BLE_ANS_CAT_ID_SMS

BLE_ANS_CAT_ID_VOICE_MAIL

BLE_ANS_CAT_ID_SCHEDULE

BLE_ANS_CAT_NUM

BLE_ANS_CMD_EN_NEW_ALERT_CAT

BLE_ANS_CMD_EN_UNR_ALERT_CAT

BLE_ANS_CMD_DIS_NEW_ALERT_CAT

BLE_ANS_CMD_DIS_UNR_ALERT_CAT

BLE_ANS_CMD_NOT_NEW_ALERT_IMMEDIATE

BLE_ANS_CMD_NOT_UNR_ALERT_IMMEDIATE

BLE_ANS_INFO_STR_MAX_LEN

BLE_ANS_NEW_ALERT_MAX_LEN

3.3.2 Battery Service

The battery service exposes the Battery Level and other information for a battery in the context of the battery' s electrical connection to a device.

Examples

bluetooth/ble_services/ble_bas.

API Reference

Header File

- components/bluetooth/ble_services/bas/include/esp_bas.h

Functions

esp_err_t **esp_ble_bas_get_battery_level**(uint8_t *level)

Get the current battery level of the device.

Parameters **level** –**[in]** The pointer to store the current battery level

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery level

esp_err_t **esp_ble_bas_set_battery_level**(uint8_t *level)

Set the current battery level of the device.

Parameters **level** –**[in]** The pointer to store the current battery level

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery level

esp_err_t **esp_ble_bas_get_level_status**(*esp_ble_bas_level_status_t* *status)

Get the summary information related to the battery status of the device.

Parameters **status** –**[in]** The pointer to store the the summary information related to the battery status

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery status

esp_err_t **esp_ble_bas_set_level_status**(*esp_ble_bas_level_status_t* *status)

Set the summary information related to the battery status of the device.

Parameters **status** –**[in]** The pointer to store the the summary information related to the battery status

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery status

esp_err_t **esp_ble_bas_get_estimated_date**(uint32_t *estimated_date)

Get the current estimated date when the battery is likely to require service or replacement.

Parameters **estimated_date** –**[in]** The pointer to store the current estimated date

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery estimated date

esp_err_t **esp_ble_bas_set_estimated_date**(uint32_t *estimated_date)

Set the current estimated date when the battery is likely to require service or replacement.

Parameters **estimated_date** –**[in]** The pointer to store the current estimated date

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery estimated date

esp_err_t **esp_ble_bas_get_critical_status**(*esp_ble_bas_critical_status_t* *status)

Get the status bits related to potential battery malfunction.

Parameters **status** –**[in]** The pointer to store the status bits related to potential battery malfunction

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery malfunction

esp_err_t **esp_ble_bas_set_critical_status**(*esp_ble_bas_critical_status_t* *status)

Set the status bits related to potential battery malfunction.

Parameters **status** –**[in]** The pointer to store the status bits related to potential battery malfunction

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery malfunction

esp_err_t **esp_ble_bas_get_energy_status**(*esp_ble_bas_energy_status_t* *status)

Get the current energy status details of the battery.

Parameters **status** –**[in]** The pointer to store the status bits related to potential battery energy

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery energy

esp_err_t **esp_ble_bas_set_energy_status**(*esp_ble_bas_energy_status_t* *status)

Set the current energy status details of the battery.

Parameters **status** –**[in]** The pointer to store the status bits related to potential battery energy

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery energy

esp_err_t **esp_ble_bas_get_time_status**(*esp_ble_bas_time_status_t* *status)

Get the current estimates on times for discharging and charging.

Parameters **status** –**[in]** The pointer to store the current estimates on times

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery estimates

esp_err_t **esp_ble_bas_set_time_status** (*esp_ble_bas_time_status_t* *status)

Set the current estimates on times for discharging and charging.

Parameters **status** –**[in]** The pointer to store the current estimates on times

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery estimates

esp_err_t **esp_ble_bas_get_health_status** (*esp_ble_bas_health_status_t* *status)

Get the aspects of battery health.

Parameters **status** –**[in]** The pointer to store the aspects of battery health

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery health

esp_err_t **esp_ble_bas_set_health_status** (*esp_ble_bas_health_status_t* *status)

Set the aspects of battery health.

Parameters **status** –**[in]** The pointer to store the aspects of battery health

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery health

esp_err_t **esp_ble_bas_get_health_info** (*esp_ble_bas_health_info_t* *info)

Get the static aspects of battery health.

Parameters **info** –**[in]** The pointer to store the static aspects of battery health

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery health information

esp_err_t **esp_ble_bas_set_health_info** (*esp_ble_bas_health_info_t* *info)

Set the static aspects of battery health.

Parameters **info** –**[in]** The pointer to store the static aspects of battery health

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery health information

esp_err_t **esp_ble_bas_get_battery_info** (*esp_ble_bas_battery_info_t* *info)

Get the physical characteristics of the battery.

Parameters **info** –**[in]** The pointer to store the physical characteristics of the battery

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery information

esp_err_t **esp_ble_bas_set_battery_info** (*esp_ble_bas_battery_info_t* *info)

Set the physical characteristics of the battery.

Parameters **info** –**[in]** The pointer to store the physical characteristics of the battery

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery information

esp_err_t **esp_ble_bas_init** (void)

Initialization Battery Service.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

Structures

struct **uint24_t**

This structure represents a 24bits data type.

Public Members

uint32_t **u24**

A 24bits data

struct **esp_ble_bas_level_status_t**

Battery Level Status Characteristic.

Public Members

uint8_t **en_identifier**

Identifier Present

uint8_t **en_battery_level**

Battery Level Present

uint8_t **en_additional_status**

Additional Status Present

uint8_t **flags_reserved**

Reserve Feature Used

struct *esp_ble_bas_level_status_t*::[anonymous] **flags**

Flags of Battery Level Status

uint16_t **battery**

Battery Present

uint16_t **wired_external_power_source**

Wired External Power Source Connected:

uint16_t **wireless_external_power_source**

Wireless External Power Source Connected:

uint16_t **battery_charge_state**

Battery Charge State:

uint16_t **battery_charge_level**

Battery Charge Level:

uint16_t **battery_charge_type**

Battery Charging Type

uint16_t **charging_fault_reason**

Charging Fault Reason

uint16_t **power_state_reserved**

Reserve Feature Used

struct *esp_ble_bas_level_status_t*::[anonymous] **power_state**

Power State of Battery Level Status

uint16_t **identifier**

Used as an identifier for a service instance.

uint8_t **battery_level**

Refer to the Battery Level

uint8_t **service_required**

Service Required

uint8_t **battery_fault**

Battery Fault:

uint8_t **reserved**

Reserve Feature Used

struct *esp_ble_bas_level_status_t*::[anonymous] **additional_status**

Contains additional status information such as whether or not service is required

struct **esp_ble_bas_critical_status_t**

Battery Critical Status Characteristic.

Public Members

uint8_t **critical_power_state**

Critical Power State

uint8_t **immediate_service**

Immediate Service Required

uint8_t **reserved**

Reserve Feature Used

struct *esp_ble_bas_critical_status_t*::[anonymous] **status**

Battery Critical Status

struct **esp_ble_bas_energy_status_t**

Battery Energy Status Characteristic.

Public Members

uint8_t **en_external_source_power**

External Source Power Present

uint8_t **en_voltage**

Present Voltage Present

uint8_t **en_available_energy**

Available Energy Present

uint8_t **en_available_battery_capacity**

Available Battery Capacity Present

uint8_t **en_charge_rate**

Charge Rate Present

uint8_t **en_available_energy_last_charge**

Available Energy at Last Charge Present

uint8_t **reserved**

Reserve Feature Used

struct *esp_ble_bas_energy_status_t*::[anonymous] **flags**

Flags of Battery Energy Status

uint16_t **external_source_power**

The total power being consumed from an external power source

uint16_t **voltage**

The present terminal voltage of the battery in volts.

uint16_t **available_energy**

The available energy of the battery in kilowatt-hours in its current charge state

uint16_t **available_battery_capacity**

The capacity of the battery in kilowatt-hours at full charge in its current condition

uint16_t **charge_rate**

The energy flowing into the battery in watts

uint16_t **available_energy_last_charge**

The available energy of the battery in kilowatt-hours in its last charge state

struct **esp_ble_bas_time_status_t**

Battery Time Status Characteristic.

Public Members

uint8_t **en_discharged_standby**

Time until Discharged on Standby Present

uint8_t **en_recharged**

Time until Recharged Present

uint8_t **reserved**

Reserve Feature Used

struct *esp_ble_bas_time_status_t*::[anonymous] **flags**

Flags of Battery Time Status

uint24_t **discharged**

Estimated time in minutes until discharged

uint24_t **discharged_standby**

Estimated time in minutes until discharged assuming for the remaining time the device is in standby.

uint24_t **recharged**

Estimated time in minutes until recharged

struct **esp_ble_bas_health_status_t**

Battery Health Status Characteristic.

Public Members

uint8_t **en_battery_health_summary**

Battery Health Summary Present

uint8_t **en_cycle_count**

Cycle Count Present

uint8_t **en_current_temperature**

Current Temperature Present

uint8_t **en_deep_discharge_count**

Deep Discharge Count Present

uint8_t **reserved**

Reserve Feature Used

struct *esp_ble_bas_health_status_t*::[anonymous] **flags**

Flags of Battery Health Status

uint8_t **battery_health_summary**

Represents aggregation of the overall health of the battery

uint16_t **cycle_count**

Represents the count value of charge cycles

int8_t **current_temperature**

Represents the current temperature of the battery

uint16_t **deep_discharge_count**

Represents the number of times the battery was completely discharged

struct **esp_ble_bas_health_info_t**

Battery Health Information Characteristic.

Public Members

uint8_t **en_cycle_count_designed_lifetime**

Cycle Count Designed Lifetime Present

uint8_t **min_max_designed_operating_temperature**

Min and Max Designed Operating Temperature Present

uint8_t **reserved**

Reserve Feature Used

struct *esp_ble_bas_health_info_t*::[anonymous] **flags**

Flags of Battery Health Information

uint16_t **cycle_count_designed_lifetime**

Represents the designed number of charge cycles supported by the device

int8_t **min_designed_operating_temperature**

Represents the minimum designed operating temperature of the battery

int8_t **max_designed_operating_temperature**

Represents the maximum designed operating temperature of the battery

struct **esp_ble_bas_battery_info_t**

Battery Information Characteristic.

Public Members

uint16_t **en_manufacture_date**

Battery Manufacture Date Present

uint16_t **en_expiration_date**

Battery Expiration Date Present

uint16_t **en_designed_capacity**

Battery Designed Capacity Present

uint16_t **en_low_energy**

Battery Low Energy Present

uint16_t **en_critical_energy**

Battery Critical Energy Present

uint16_t **en_chemistry**

Battery Chemistry Present

uint16_t **en_nominalvoltage**

Nominal Voltage Present

uint16_t **en_aggregation_group**

Battery Aggregation Group Present

uint16_t **flags_reserved**

Reserve Feature Used

struct *esp_ble_bas_battery_info_t*::[anonymous] **flags**

Flags of Battery Information

uint8_t **replace_able**

Battery Replaceable

uint8_t **recharge_able**

Battery Rechargeable

uint8_t **reserved**

Reserve Feature Used

struct *esp_ble_bas_battery_info_t*::[anonymous] **features**

Features of Battery Information

uint24_t **manufacture_date**

Battery date of manufacture specified as days

uint24_t **expiration_date**

Battery expiration date specified as days

uint16_t **designed_capacity**

The capacity of the battery in kilowatt-hours at full charge in original (new) condition.

uint16_t **low_energy**

The battery energy value in kilowatt-hours when the battery is low

uint16_t **critical_energy**

The battery energy value in kilowatt-hours when the battery is critical.

uint8_t **chemistry**

The value of battery chemistry

uint16_t **nominalvoltage**

Nominal voltage of the battery in units of volts

uint8_t **aggregation_group**

Indicates the Battery Aggregation Group to which this instance of the battery service is associated

struct **esp_ble_bas_data_t**

Battery Service.

Public Members

uint8_t **battery_level**

The charge level of a battery

esp_ble_bas_level_status_t **level_status**

The power state of a battery

uint24_t **estimated_service_date**

The estimated date when replacement or servicing is required.

esp_ble_bas_critical_status_t **critical_status**

The device will possibly not function as expected due to low energy or service required

esp_ble_bas_energy_status_t **energy_status**

Details about the energy status of the battery

esp_ble_bas_time_status_t **time_status**

Time estimates for discharging and charging

esp_ble_bas_health_status_t **health_status**

Several aspects of battery health

esp_ble_bas_health_info_t **health_info**

The health of a battery

esp_ble_bas_battery_info_t **battery_info**

The physical characteristics of a battery in the context of the battery' s connection in a device

Macros

BLE_BAS_UUID16

BLE_BAS_CHR_UUID16_LEVEL

BLE_BAS_CHR_UUID16_LEVEL_STATUS

BLE_BAS_CHR_UUID16_ESTIMATED_SERVICE_DATE

BLE_BAS_CHR_UUID16_CRITICAL_STATUS

BLE_BAS_CHR_UUID16_ENERGY_STATUS

BLE_BAS_CHR_UUID16_TIME_STATUS

BLE_BAS_CHR_UUID16_HEALTH_STATUS

BLE_BAS_CHR_UUID16_HEALTH_INFO

BLE_BAS_CHR_UUID16_BATTERY_INFO

BAS_CHR_LEVEL_STATUS_FLAGS_BM_NONE

BAS_CHR_LEVEL_STATUS_FLAGS_BM_IDENTIFY

BAS_CHR_LEVEL_STATUS_FLAGS_BM_BATTERY_LEVEL

BAS_CHR_LEVEL_STATUS_FLAGS_BM_ADDITIONAL_STATUS

BAS_CHR_LEVEL_STATUS_FLAGS_BM_RFU

BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_NOT

BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_SET

BAS_CHR_LEVEL_STATUS_POWER_STATE_WIRED_EXTERNAL_POWER_SOURCE_NOTCONNECT

BAS_CHR_LEVEL_STATUS_POWER_STATE_WIRED_EXTERNAL_POWER_SOURCE_CONNECTED

BAS_CHR_LEVEL_STATUS_POWER_STATE_WIRED_EXTERNAL_POWER_SOURCE_UNKNOWN

BAS_CHR_LEVEL_STATUS_POWER_STATE_WIRED_EXTERNAL_POWER_SOURCE_RFU

BAS_CHR_LEVEL_STATUS_POWER_STATE_WIRELESS_EXTERNAL_POWER_SOURCE_NOTCONNECT

BAS_CHR_LEVEL_STATUS_POWER_STATE_WIRELESS_EXTERNAL_POWER_SOURCE_CONNECTED

BAS_CHR_LEVEL_STATUS_POWER_STATE_WIRELESS_EXTERNAL_POWER_SOURCE_UNKNOWN

BAS_CHR_LEVEL_STATUS_POWER_STATE_WIRELESS_EXTERNAL_POWER_SOURCE_RFU

BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_CHARGE_STATE_UNKNOWN

BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_CHARGE_STATE_CHARGING

BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_CHARGE_STATE_DISCHARGING_ACTIVE

BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_CHARGE_STATE_DISCHARGING_INACTIVE

BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_CHARGE_LEVEL_UNKNOWN

BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_CHARGE_LEVEL_GOOD

BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_CHARGE_LEVEL_LOW

BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_CHARGE_LEVEL_CRITICAL

BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_TYPE_UNKNOWN

BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_TYPE_CURRENT

BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_TYPE_VOLTAGE

BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_TYPE_TRICKLE

BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_TYPE_FLOAT

BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_TYPE_RFU

BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_FAULT_NONE

BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_FAULT_BATTERY

BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_FAULT_EXTERNAL_POWER_SOURCE

BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_FAULT_OTHER

BAS_CHR_LEVEL_STATUS_ASSITIONAL_STATUS_SERVICE_FALSE

BAS_CHR_LEVEL_STATUS_ASSITIONAL_STATUS_SERVICE_TRUE

BAS_CHR_LEVEL_STATUS_ASSITIONAL_STATUS_SERVICE_UNKNOWN

BAS_CHR_LEVEL_STATUS_ASSITIONAL_STATUS_SERVICE_RFU

BAS_CHR_LEVEL_STATUS_ASSITIONAL_STATUS_BATTERY_FAULT_UNKNOWN

BAS_CHR_LEVEL_STATUS_ASSITIONAL_STATUS_BATTERY_FAULT_TRUE

BAS_CHR_CRITICAL_STATUS_FLAGS_BM_NONE

BAS_CHR_CRITICAL_STATUS_FLAGS_BM_CRITICAL_POWER_STATE

BAS_CHR_CRITICAL_STATUS_FLAGS_BM_IMMEDIATE_SERVICE

BAS_CHR_ENERGY_STATUS_FLAGS_BM_NONE

BAS_CHR_ENERGY_STATUS_FLAGS_BM_EXTERNAL_SOURCE_POWER

BAS_CHR_ENERGY_STATUS_FLAGS_BM_VOLTAGE

BAS_CHR_ENERGY_STATUS_FLAGS_BM_AVAILALBE_ENERGY

BAS_CHR_ENERGY_STATUS_FLAGS_BM_AVAILALBE_BATTERY_CAPACITY

BAS_CHR_ENERGY_STATUS_FLAGS_BM_CHARGE_RATE

BAS_CHR_ENERGY_STATUS_FLAGS_BM_AVAILALBE_ENERGY_LAST_CHARGE

BAS_CHR_ENERGY_STATUS_FLAGS_BM_RFU

BAS_CHR_TIME_STATUS_FLAGS_BM_NONE

BAS_CHR_TIME_STATUS_FLAGS_BM_DISCHARGED_STANDBY

BAS_CHR_TIME_STATUS_FLAGS_BM_RECHARGE

BAS_CHR_TIME_STATUS_FLAGS_BM_RFU

BAS_CHR_HEALTH_STATUS_FLAGS_BM_NONE

BAS_CHR_HEALTH_STATUS_FLAGS_BM_BATTERY_HEALTH_SUMMARY

BAS_CHR_HEALTH_STATUS_FLAGS_BM_RCYCLE_COUNT

BAS_CHR_HEALTH_STATUS_FLAGS_BM_CURRENT_TEMPERATURE

BAS_CHR_HEALTH_STATUS_FLAGS_BM_DEEP_DISCHARGE_COUNT

BAS_CHR_HEALTH_STATUS_FLAGS_BM_RFU

BAS_CHR_HEALTH_INFO_FLAGS_BM_NONE

BAS_CHR_HEALTH_INFO_FLAGS_BM_CYCLE_COUNT_DESIGNED_LIFETIME

BAS_CHR_HEALTH_INFO_FLAGS_BM_DESIGNED_OPERATING_TEMPERATURE

BAS_CHR_HEALTH_INFO_FLAGS_BM_RFU

BAS_CHR_BATTERY_INFO_FLAGS_BM_NONE

BAS_CHR_BATTERY_INFO_FLAGS_BM_MANUFACTURE_DATE

BAS_CHR_BATTERY_INFO_FLAGS_BM_EXPIRATION_DATE

BAS_CHR_BATTERY_INFO_FLAGS_BM_DESIGNED_CAPACITY

BAS_CHR_BATTERY_INFO_FLAGS_BM_LOW_ENERGY

BAS_CHR_BATTERY_INFO_FLAGS_BM_CRITICAL_ENERGY

BAS_CHR_BATTERY_INFO_FLAGS_BM_CHEMISTRY

BAS_CHR_BATTERY_INFO_FLAGS_BM_NOMINAL_VOLTAGE

BAS_CHR_BATTERY_INFO_FLAGS_BM_AGGREGATION_GROUP

BAS_CHR_BATTERY_INFO_FLAGS_BM_RFU

BAS_CHR_BATTERY_INFO_FEATURE_BM_NONE

BAS_CHR_BATTERY_INFO_FEATURE_BM_REPLACE

BAS_CHR_BATTERY_INFO_FEATURE_BM_RECHARGE

BAS_CHR_BATTERY_INFO_FEATURE_BM_RFU

3.3.3 Body Composition Service

The Body Composition Service (BCS) exposes data related to body composition from a body composition analyzer (Server) intended for consumer healthcare as well as sports/fitness applications.

Examples

bluetooth/ble_services/ble_bcs.

API Reference

Header File

- components/bluetooth/ble_services/bcs/include/esp_bcs.h

Functions

esp_err_t **esp_ble_bcs_set_measurement** (*esp_bcs_val_t* *in_val, bool need_send)

Set the Body Composition value.

Parameters

- **in_val** –**[in]** The pointer to store the Body Composition Information.
- **need_send** –**[in]** If set to true, the Body Composition Info will send to remote client.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

esp_err_t **esp_ble_bcs_init** (void)

Initialization Body Composion Service.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

Structures

struct **esp_bcs_val_t**

Body Composition Measurement Characteristic.

Public Members

uint32_t **bcs_flag**

Body Composition flag field

uint16_t **year**

Year as defined by the Gregorian calendar, Valid range 1582 to 9999

uint8_t **month**

Month of the year as defined by the Gregorian calendar, Valid range 1 (January) to 12 (December)

uint8_t **day**

Day of the month as defined by the Gregorian calendar, Valid range 1 to 31

uint8_t **hours**

Number of hours past midnight, Valid range 0 to 23

uint8_t **minutes**

Number of minutes since the start of the hour. Valid range 0 to 59

uint8_t **seconds**

Number of seconds since the start of the minute. Valid range 0 to 59

struct *esp_bcs_val_t*::[anonymous] **timestamp**

The date and time

uint8_t **user_id**

User ID field

uint16_t **basal_metabolism**

Basal Metabolism field

uint16_t **muscle_percentage**

Muscle Percentage field

uint16_t **muscle_mass**

Muscle Mass field

uint16_t **fat_free_mass**

Fat Free Mass field

uint16_t **soft_lean_mass**

Soft Lean Mass field

uint16_t **body_water_mass**

Body Water Mass field

uint8_t **impedance**

Impedance field

uint16_t **weight**

Weight field

uint16_t **height**

Height field

Macros

BLE_BCS_MAX_VAL_LEN

BLE_BCS_UUID16

BLE_BCS_CHR_UUID16_FEATURE

BLE_BCS_CHR_UUID16_MEASUREMENT

BLE_BCS_FEAT_TIME_STAMP

BLE_BCS_FEAT_MULTI_USER

BLE_BCS_FEAT_BASAL_METABOLISM

BLE_BCS_FEAT_MUSCLE_PERCENTAGE

BLE_BCS_FEAT_MUSCLE_MASS

BLE_BCS_FEAT_FAT_FREE_MASS

BLE_BCS_FEAT_SOFT_LEAN_MASS

BLE_BCS_FEAT_BODY_WATER_MASS

BLE_BCS_FEAT_IMPEDENCE

BLE_BCS_FEAT_WEIGHT

BLE_BCS_FEAT_HEIGHT

BLE_BCS_FEAT_MASS_MEASUREMENT_RESOLUTION

BLE_BCS_FEAT_HEIGHT_RESOLUTION

BLE_BCS_FLAG_MEASUREMENT_UNITS

BLE_BCS_FLAG_TIME_STAMP

BLE_BCS_FLAG_MULTI_USER

BLE_BCS_FLAG_BASAL_METABOLISM

BLE_BCS_FLAG_MUSCLE_PERCENTAGE

BLE_BCS_FLAG_MUSCLE_MASS

BLE_BCS_FLAG_FAT_FREE_MASS

BLE_BCS_FLAG_SOFT_LEAN_MASS

BLE_BCS_FLAG_BODY_WATER_MASS

BLE_BCS_FLAG_IMPEDENCE

BLE_BCS_FLAG_WEIGHT

BLE_BCS_FLAG_HEIGHT

BLE_BCS_FLAG_MULTIPLE_PACKET

3.3.4 Current Time Service

The Current Time service defines how a Bluetooth device can expose date and time information to other Bluetooth devices.

Examples

bluetooth/ble_services/ble_cts.

API Reference

Header File

- components/bluetooth/ble_services/cts/include/esp_cts.h

Functions

esp_err_t **esp_ble_cts_get_current_time** (*esp_ble_cts_cur_time_t* *out_val)

Read the value of Current Time characteristic.

Parameters **out_val** –**[in]** The pointer to store the Current Time Increment.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_cts_set_current_time** (*esp_ble_cts_cur_time_t* *in_val, bool need_send)

Set the Current Time characteristic value.

Parameters
- **in_val** –**[in]** The pointer to store the Current Time.
- **need_send** –**[in]** send the current time to remote client.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

esp_err_t **esp_ble_cts_get_local_time** (*esp_ble_cts_local_time_t* *out_val)

Read the value of Local Time characteristic.

Parameters **out_val** –**[in]** The pointer to store the Local Time Increment.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_cts_set_local_time** (*esp_ble_cts_local_time_t* *in_val)

Set the Local Time characteristic value.

Parameters **in_val** –**[in]** The pointer to store the Local Time.

Returns
- ESP_OK on successful

esp_err_t **esp_ble_cts_get_reference_time** (*esp_ble_cts_ref_time_t* *out_val)

Read the value of Reference Time characteristic.

Parameters **out_val** –**[in]** The pointer to store the Reference Time Increment.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_cts_set_reference_time** (*esp_ble_cts_ref_time_t* *in_val)

Set the Reference Time characteristic value.

Parameters **in_val** –**[in]** The pointer to store the Reference Time.

Returns

- ESP_OK on successful

esp_err_t **esp_ble_cts_init** (void)

Initialization Current Time Service.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

Structures

struct **esp_ble_cts_cur_time_t**

Current Time.

Public Members

uint16_t **year**

Year as defined by the Gregorian calendar, Valid range 1582 to 9999

uint8_t **month**

Month of the year as defined by the Gregorian calendar, Valid range 1 (January) to 12 (December)

uint8_t **day**

Day of the month as defined by the Gregorian calendar, Valid range 1 to 31

uint8_t **hours**

Number of hours past midnight, Valid range 0 to 23

uint8_t **minutes**

Number of minutes since the start of the hour. Valid range 0 to 59

uint8_t **seconds**

Number of seconds since the start of the minute. Valid range 0 to 59

struct *esp_ble_cts_cur_time_t*::[anonymous] **timestamp**

The date and time

uint8_t **day_of_week**

Day_of_week. valid range : 1(Monday) - 7(Sunday), 0 means day of week not known

uint8_t **fractions_256**

Fractions_256. the number of 1 / 256 fractions of second, valid range : 0 - 255

uint8_t **adjust_reason**

This field represents reason(s) for adjusting time

struct **esp_ble_cts_local_time_t**

Local Time Information.

Public Members

int8_t **timezone**

Time zone

uint8_t **dst_offset**

dst_offset. allowed values : 0, 2, 4, 8, 255 Dst offset

struct **esp_ble_cts_ref_time_t**

reference time information

Public Members

uint8_t **time_source**

time_source. valid range : 0 - 253 255 means not known Time source

uint8_t **time_accuracy**

Time accuracy

uint8_t **days_since_update**

days_since_update. valid range : 0 - 254 A value of 255 is used when the time span is greater than or equal to 255 days Days since update

uint8_t **hours_since_update**

hours_since_update. valid range : 0 - 23 A value of 255 is used when the time span is greater than or equal to 255 days Hours since update

Macros

BLE_CTS_UUID16

BLE_CTS_CHR_UUID16_CURRENT_TIME

BLE_CTS_CHR_UUID16_LOCAL_TIME

BLE_CTS_CHR_UUID16_REFERENCE_TIME

BLE_CTS_MANUAL_TIME_UPDATE_MASK

BLE_CTS_EXTERNAL_REFERENCE_TIME_UPDATE_MASK

BLE_CTS_CHANGE_OF_TIME_ZONE_MASK

BLE_CTS_CHANGE_OF_DST_MASK

3.3.5 Device Information Service

The device information service exposes manufacturer and/or vendor information about a device.

Examples

bluetooth/ble_services/ble_dis.

API Reference

Header File

- components/bluetooth/ble_services/dis/include/esp_dis.h

Functions

esp_err_t **esp_ble_dis_get_model_number**(const char **value)

Get the model number that is assigned by the device vendor.

Parameters **value** –**[out]** The pointer to store the model number.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong model number

esp_err_t **esp_ble_dis_set_model_number**(const char *value)

Set the model number of the device.

Parameters **value** –**[in]** The pointer to store the model number.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong model number

esp_err_t **esp_ble_dis_get_serial_number**(const char **value)

Get the serial number for a particular instance of the device.

Parameters **value** –**[out]** The pointer to store the serial number.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong serial number

esp_err_t **esp_ble_dis_set_serial_number**(const char *value)

Set the serial number of the device.

Parameters **value** –**[in]** The pointer to store the serial number.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong serial number

esp_err_t **esp_ble_dis_get_firmware_revision**(const char **value)

Get the firmware revision for the firmware within the device.

Parameters **value** –**[out]** The pointer to store the firmware revision.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong firmware revision

esp_err_t **esp_ble_dis_set_firmware_revision**(const char *value)

Set the firmware revision of the device.

Parameters **value** –**[in]** The pointer to store the firmware revision.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong firmware revision

esp_err_t **esp_ble_dis_get_hardware_revision**(const char **value)

Get the hardware revision for the hardware within the device.

Parameters **value** –**[out]** The pointer to store the hardware revision.
Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong hardware revision

esp_err_t **esp_ble_dis_set_hardware_revision**(const char *value)

Set the hardware revision of the device.

Parameters **value** –**[in]** The pointer to store the hardware revision.
Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong hardware revision

esp_err_t **esp_ble_dis_get_software_revision**(const char **value)

Get the software revision for the software within the device.

Parameters **value** –**[out]** The pointer to store the software revision.
Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong software revision

esp_err_t **esp_ble_dis_set_software_revision**(const char *value)

Set the software revision of the device.

Parameters **value** –**[in]** The pointer to store the software revision.
Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong software revision

esp_err_t **esp_ble_dis_get_manufacturer_name**(const char **value)

Get the name of the manufacturer of the device.

Parameters **value** –**[out]** The pointer to store the name of the manufacturer.
Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong name of the manufacturer

esp_err_t **esp_ble_dis_set_manufacturer_name**(const char *value)

Set the name of the manufacturer of the device.

Parameters **value** –**[in]** The pointer to store the name of the manufacturer.
Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong name of the manufacturer

esp_err_t **esp_ble_dis_get_system_id**(const char **value)

Get the System Id of the device.

Parameters **value** –**[out]** The pointer to store the System Id.
Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong System Id

esp_err_t **esp_ble_dis_set_system_id**(const char *value)

Set the System Id of the device.

Parameters **value** –**[in]** The pointer to store the System Id.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong System Id

esp_err_t **esp_ble_dis_get_pnp_id**(*esp_ble_dis_pnp_t* *pnp_id)

Get the PnP Id of the device.

Parameters **pnp_id** –**[in]** The pointer to store the PnP Id.
Returns :
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong PnP Id

esp_err_t **esp_ble_dis_set_pnp_id**(*esp_ble_dis_pnp_t* *pnp_id)

Set the PnP Id of the device.

Parameters **pnp_id** –**[in]** The pointer to store the PnP Id.
Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong PnP Id

esp_err_t **esp_ble_dis_init**(void)

Initialization GATT Device Information Service.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

Structures

struct **esp_ble_dis_pnp**

The structure represent a set of values that are used to create a device ID value. These values are used to identify all devices of a given type/model/version using numbers.

Public Members

uint8_t **src**

The vendor ID source

uint16_t **vid**

The product vendor from the namespace in the vendor ID source

uint16_t **pid**

Manufacturer managed identifier for this product

uint16_t **ver**

Manufacturer managed version for this product

struct **esp_ble_dis_data**

Structure holding data for the main characteristics

Public Members

char **model_number**[CONFIG_BLE_DIS_STR_MAX]

Model number. Represent the model number that is assigned by the device vendor.

char **serial_number**[CONFIG_BLE_DIS_STR_MAX]

Serial number. Represent the serial number for a particular instance of the device.

char **firmware_revision**[CONFIG_BLE_DIS_STR_MAX]

Firmware revision. Represent the firmware revision for the firmware within the device.

char **hardware_revision**[CONFIG_BLE_DIS_STR_MAX]

Hardware revision. Represent the hardware revision for the hardware within the device.

char **software_revision**[CONFIG_BLE_DIS_STR_MAX]

Software revision. Represent the software revision for the software within the device.

char **manufacturer_name**[CONFIG_BLE_DIS_STR_MAX]

Manufacturer name. Represent the name of the manufacturer of the device.

char **system_id**[CONFIG_BLE_DIS_STR_MAX]

System ID. Represent the System Id of the device.

esp_ble_dis_pnp_t **pnp_id**

PnP ID. Represent the PnP Id of the device.

Macros

BLE_DIS_UUID16

BLE_DIS_CHR_UUID16_SYSTEM_ID

BLE_DIS_CHR_UUID16_MODEL_NUMBER

BLE_DIS_CHR_UUID16_SERIAL_NUMBER

BLE_DIS_CHR_UUID16_FIRMWARE_REVISION

BLE_DIS_CHR_UUID16_HARDWARE_REVISION

BLE_DIS_CHR_UUID16_SOFTWARE_REVISION

BLE_DIS_CHR_UUID16_MANUFACTURER_NAME

BLE_DIS_CHR_UUID16_REG_CERT

BLE_DIS_CHR_UUID16_PNP_ID

Type Definitions

typedef struct *esp_ble_dis_pnp* **esp_ble_dis_pnp_t**

The structure represent a set of values that are used to create a device ID value. These values are used to identify all devices of a given type/model/version using numbers.

typedef struct *esp_ble_dis_data* **esp_ble_dis_data_t**

Structure holding data for the main characteristics

3.3.6 Heart Rate Service

The heart rate service exposes heart rate and other data related to a heart rate sensor intended for fitness applications.

Examples

bluetooth/ble_services/ble_hrs.

API Reference

Header File

- components/bluetooth/ble_services/hrs/include/esp_hrs.h

Functions

esp_err_t **esp_ble_hrs_get_location**(uint8_t *location)

Get the sensor location value of the device.

Parameters **location** –**[in]** The pointer to store the sensor location value

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong location

esp_err_t **esp_ble_hrs_set_location**(uint8_t location)

Set the sensor location value of the device.

Parameters **location** –**[in]** The sensor location value

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong location

esp_err_t **esp_ble_hrs_get_hrm**(*esp_ble_hrs_hrm_t* *hrm)

Get the value of the heart rate measurement of the device.

Parameters **hrm** –**[in]** The pointer to store the value of the heart rate measurement

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong heart rate measurement

esp_err_t **esp_ble_hrs_set_hrm**(*esp_ble_hrs_hrm_t* *hrm)

Set the value of the heart rate measurement of the device.

Parameters **hrm** –**[in]** The pointer to store the value of the heart rate measurement

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong heart rate measurement

esp_err_t **esp_ble_hrs_init** (void)

Initialization Heart Rate Service.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

Structures

struct **esp_ble_hrs_hrm_t**

Heart Rate Measurement Characteristic.

Public Members

uint8_t **format**

Heart rate value format flag

uint8_t **detected**

Sensor contact detected flag

uint8_t **supported**

Sensor contact cupported flag

uint8_t **energy**

Energy expended present flag

uint8_t **interval**

RR-Interval present flag

uint8_t **reserved**

Reserved for future use flag

struct *esp_ble_hrs_hrm_t*::[anonymous] **flags**

Flags of heart rate measurement

uint8_t **u8**

8 bit resolution

uint16_t **u16**

16 bit resolution

union *esp_ble_hrs_hrm_t*::[anonymous] **heartrate**

Heart rate measurement value

uint16_t **energy_val**

Expended energy value

uint16_t **interval_buf**[BLE_HRS_CHR_MERSUREMENT_RR_INTERVAL_MAX_NUM]

The RR-Interval value represents the time between two R-Wave detections

Macros

BLE_HRS_UUID16

BLE_HRS_CHR_UUID16_MEASUREMENT

BLE_HRS_CHR_UUID16_BODY_SENSOR_LOC

BLE_HRS_CHR_UUID16_HEART_RATE_CNTL_POINT

BLE_HRS_CHR_MERSUREMENT_RR_INTERVAL_MAX_NUM

BLE_HRS_CHR_BODY_SENSOR_LOC_OTHER

BLE_HRS_CHR_BODY_SENSOR_LOC_CHEST

BLE_HRS_CHR_BODY_SENSOR_LOC_WRIST

BLE_HRS_CHR_BODY_SENSOR_LOC_FINGER

BLE_HRS_CHR_BODY_SENSOR_LOC_HAND

BLE_HRS_CHR_BODY_SENSOR_LOC_EAR_LOBE

BLE_HRS_CHR_BODY_SENSOR_LOC_FOOT

BLE_HRS_CHR_BODY_SENSOR_LOC_RFU

BLE_HRS_CHR_BODY_SENSOR_LOC_MAX

3.3.7 Health Thermometer Service

The health thermometer service exposes temperature and other data related to a thermometer used for healthcare applications.

Examples

bluetooth/ble_services/ble_hts.

API Reference

Header File

- components/bluetooth/ble_services/hts/include/esp_hts.h

Functions

esp_err_t **esp_ble_hts_get_temp_type** (uint8_t *temp_type)

Get the current temperature type value of the device.

Parameters **temp_type** –**[in]** The pointer to store the current temperature type value

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong temperature type

esp_err_t **esp_ble_hts_set_temp_type** (uint8_t temp_type)

Set the current temperature type value of the device.

Parameters **temp_type** –**[in]** The current temperature type value

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong temperature type

esp_err_t **esp_ble_hts_get_measurement_temp** (*esp_ble_hts_temp_t* *temp_val)

Get the value of the temperature measurement of the device.

Parameters **temp_val** –**[in]** The pointer to store the value of the temperature measurement

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong temperature measurement

esp_err_t **esp_ble_hts_set_measurement_temp** (*esp_ble_hts_temp_t* *temp_val)

Set the value of the temperature measurement of the device.

Parameters **temp_val** –**[in]** The pointer to store the value of the temperature measurement

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong temperature measurement

esp_err_t **esp_ble_hts_get_intermediate_temp** (*esp_ble_hts_temp_t* *temp_val)

Get the value of the intermediate temperature of the device.

Parameters **temp_val** –**[in]** The pointer to store the value of the intermediate temperature

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong intermediate temperature

esp_err_t **esp_ble_hts_set_intermediate_temp** (*esp_ble_hts_temp_t* *temp_val)

Set the value of the intermediate temperature of the device.

Parameters **temp_val** –**[in]** The pointer to store the value of the intermediate temperature

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong intermediate temperature

esp_err_t **esp_ble_hts_get_measurement_interval** (uint16_t *interval_val)

Get the measurement interval value of the device.

Parameters **interval_val** –**[in]** The pointer to store the measurement interval value

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong measurement interval

esp_err_t **esp_ble_hts_set_measurement_interval** (uint16_t interval_val)

Set the measurement interval value of the device.

Parameters **interval_val** –**[in]** The measurement interval value

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong measurement interval

esp_err_t **esp_ble_hts_init** (void)

Initialization Health Thermometer Service.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

Structures

struct **esp_ble_hts_temp_t**

Temperature Measurement and Intermediate Temperature Characteristic.

Public Members

uint8_t **temperature_unit**

Temperature units flag

uint8_t **time_stamp**

Time stamp flag

uint8_t **temperature_type**

Temperature type flag

uint8_t **reserved**

Reserved for future use

struct *esp_ble_hts_temp_t*::[anonymous] **flags**

Flags of temperature

uint32_t **celsius**

Celsius unit

uint32_t **fahrenheit**

Fahrenheit unit

union *esp_ble_hts_temp_t*::[anonymous] **temperature**

Temperature value

uint16_t **year**

Year as defined by the Gregorian calendar, Valid range 1582 to 9999

uint8_t **month**

Month of the year as defined by the Gregorian calendar, Valid range 1 (January) to 12 (December)

uint8_t **day**

Day of the month as defined by the Gregorian calendar, Valid range 1 to 31

uint8_t **hours**

Number of hours past midnight, Valid range 0 to 23

uint8_t **minutes**

Number of minutes since the start of the hour. Valid range 0 to 59

uint8_t **seconds**

Number of seconds since the start of the minute. Valid range 0 to 59

struct *esp_ble_hts_temp_t*::[anonymous] **timestamp**

The date and time

uint8_t **location**

The location of a temperature measurement

Macros

BLE_HTS_UUID16

BLE_HTS_CHR_UUID16_TEMPERATURE_MEASUREMENT

BLE_HTS_CHR_UUID16_TEMPERATURE_TYPE

BLE_HTS_CHR_UUID16_INTERMEDIATE_TEMPERATURE

BLE_HTS_CHR_UUID16_MEASUREMENT_INTERVAL

BLE_HTS_CHR_TEMPERATURE_UNITS_CELSIUS

BLE_HTS_CHR_TEMPERATURE_UNITS_FAHRENHEIT

BLE_HTS_CHR_TEMPERATURE_FLAGS_NOT

BLE_HTS_CHR_TEMPERATURE_FLAGS_SET

BLE_HTS_CHR_TEMPERATURE_TYPE_RFU

BLE_HTS_CHR_TEMPERATURE_TYPE_ARMPIT

BLE_HTS_CHR_TEMPERATURE_TYPE_BODY

BLE_HTS_CHR_TEMPERATURE_TYPE_EAR

BLE_HTS_CHR_TEMPERATURE_TYPE_FINGER

BLE_HTS_CHR_TEMPERATURE_TYPE_GAST_TRACT

BLE_HTS_CHR_TEMPERATURE_TYPE_MOUTH

BLE_HTS_CHR_TEMPERATURE_TYPE_RECTUM

BLE_HTS_CHR_TEMPERATURE_TYPE_TOE

BLE_HTS_CHR_TEMPERATURE_TYPE_TYMPANUM

BLE_HTS_CHR_TEMPERATURE_TYPE_MAX

3.3.8 Object Transfer Service

The Object Transfer Service provides management and control features supporting bulk data transfers which occur via a separate L2CAP connection-oriented channel.

Examples

bluetooth/ble_services/ble_ots.

API Reference

Header File

- components/bluetooth/ble_services/ots/include/esp_ots.h

Functions

esp_err_t **esp_ble_ots_get_feature**(*esp_ble_ots_feature_t* *out_val)

Read the value of OTS Feature Characteristic.

Parameters **out_val** –**[out]** The pointer to store the OTS Feature Value.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_ots_set_feature**(*esp_ble_ots_feature_t* *in_val)

Set the value of OTS Feature Characteristic.

Parameters **in_val** –**[in]** The pointer to store OTS Feature value.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization

esp_err_t **esp_ble_ots_get_name**(uint8_t *out_buf, size_t buf_len, size_t *out_len)

Read the value of Object Name Characteristic.

Parameters
- **out_buf** –**[out]** The pointer to store the Object Name.
- **buf_len** –**[in]** Maximum length of the output buffer.
- **out_len** –**[out]** The pointer to store the actual length of the Object Name (can be NULL if not needed).

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_ots_set_name** (const uint8_t *in_val, size_t in_len)

Set the Object Name Characteristic value.

Parameters
- **in_val** –**[in]** The pointer to the Object Name data.
- **in_len** –**[in]** The length of the Object Name data.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization

esp_err_t **esp_ble_ots_get_type** (uint16_t *out_val)

Read the value of Object Type Characteristic value.

Parameters **out_val** –**[out]** The pointer to store the Object Type.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_ots_set_type** (uint16_t *in_val)

Set the Object Type Characteristic value.

Parameters **in_val** –**[in]** The pointer to store Object Type.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization

esp_err_t **esp_ble_ots_get_size** (*esp_ble_ots_size_t* *out_val)

Read the value of Object Size Characteristic value.

Parameters **out_val** –**[out]** The pointer to store the Object Size.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_ots_set_size** (*esp_ble_ots_size_t* *in_val)

Set the Object Size Characteristic value.

Parameters **in_val** –**[in]** The pointer to store Object Size.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization

esp_err_t **esp_ble_ots_get_first_create_time** (*esp_ble_ots_utc_t* *out_val)

Read the value of Object First Create Characteristic value.

Parameters **out_val** –**[out]** The pointer to store the Object First Create Time.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_ots_set_first_create_time** (*esp_ble_ots_utc_t* *in_val)

Set the Object First Create Characteristic value.

Parameters **in_val** –**[in]** The pointer to store the Object First Create Time.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization

esp_err_t **esp_ble_ots_get_last_modify_time** (*esp_ble_ots_utc_t* *out_val)

Read the value of Object Last Modify Characteristic value.

Parameters **out_val** –**[out]** The pointer to store the Object Last Modify Time.

Returns
- ESP_OK on successful

- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_ots_set_last_modify_time** (*esp_ble_ots_utc_t* *in_val)

Set the Object Last Modify Characteristic value.

Parameters **in_val** –**[in]** The pointer to store the Object Last Modify Time.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization

esp_err_t **esp_ble_ots_get_id** (*esp_ble_ots_id_t* *out_val)

Read the value of Object ID value.

Parameters **out_val** –**[out]** The pointer to store the Object ID.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_ots_set_id** (*esp_ble_ots_id_t* *in_val)

Set the Object ID value.

Parameters **in_val** –**[in]** The pointer to store the Object ID.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization

esp_err_t **esp_ble_ots_get_prop** (*esp_ble_ots_prop_t* *out_val)

Read the value of Object Properties value.

Parameters **out_val** –**[out]** The pointer to store the Object Properties.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_ots_set_prop** (*esp_ble_ots_prop_t* *in_val)

Set the Object Properties value.

Parameters **in_val** –**[in]** The pointer to store the Object Properties.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization

esp_err_t **esp_ble_ots_get_oacp** (*esp_ble_ots_oacp_t* *out_val)

Read the value of Object Action Control Point value.

Parameters **out_val** –**[out]** The pointer to store the Object Action Control Point.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_ots_set_oacp** (*esp_ble_ots_oacp_t* *in_val, bool need_send)

Set the Object Action Control Point value.

Parameters

- **in_val** –**[in]** The pointer to store the Object Action Control Point value
- **need_send** –**[in]** If set to true, the OACP Value will send to remote client.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization

esp_err_t **esp_ble_ots_get_olcp** (*esp_ble_ots_olcp_t* *out_val)

Read the value of Object List Control Point value.

Parameters **out_val** –**[out]** The pointer to store the Object List Control Point.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_ots_set_olcp** (*esp_ble_ots_olcp_t* *in_val, bool need_send)

Set the Object List Control Point value.

Parameters
- **in_val** –**[in]** The pointer to store the Object List Control Point value
- **need_send** –**[in]** If set to true, the OLCP Value will send to remote client.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization

esp_err_t **esp_ble_ots_get_filter** (*esp_ble_ots_filter_t* *out_val)

Read the value of Object Filter value.

Parameters **out_val** –**[out]** The pointer to store the Object Filter.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_ots_set_filter** (*esp_ble_ots_filter_t* *in_val)

Set the value of Object Filter value.

Parameters **in_val** –**[in]** The pointer to store the Object Filter.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization

esp_err_t **esp_ble_ots_get_change** (*esp_ble_ots_change_t* *out_val)

Read the value of Object Change value.

Parameters **out_val** –**[out]** The pointer to store the Object Change.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_ots_set_change** (*esp_ble_ots_change_t* *in_val, bool need_send)

Set the Object Change value.

Parameters
- **in_val** –**[in]** The pointer to store the Object Change.
- **need_send** –**[in]** If set to true, the Object Change value will send to remote client.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization

esp_err_t **esp_ble_ots_init** (void)

Initialization Object Transfer Service.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_ERR_NO_MEM if failed to create mutex
- ESP_FAIL on error

esp_err_t **esp_ble_ots_deinit** (void)

Deinitialization Object Transfer Service.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong deinitialization
- ESP_FAIL on error

Structures

struct **esp_ble_ots_feature_t**

OTS Feature Characteristic.

Public Members

uint32_t **create_op**

Create Op Code Supported. 0: False, 1: True

uint32_t **delete_op**

Delete Op Code Supported. 0: False, 1: True

uint32_t **calculate_op**

Calculate Checksum Op Code Supported. 0: False, 1: True

uint32_t **execute_op**

Execute Op Code Supported. 0: False, 1: True

uint32_t **read_op**

Read Op Code Supported. 0: False, 1: True

uint32_t **write_op**

Write Op Code Supported. 0: False, 1: True

uint32_t **appending_op**

Appending Op Code Supported. 0: False, 1: True

uint32_t **truncation_op**

Truncation Op Code Supported. 0: False, 1: True

uint32_t **patching_op**

Patching Op Code Supported. 0: False, 1: True

uint32_t **abort_op**

Abort Op Code Supported. 0: False, 1: True

struct *esp_ble_ots_feature_t*::[anonymous] **oacp**

The structure of the OACP Features field

uint32_t **goto_op**

Go to Op Code Supported. 0: False, 1: True

uint32_t **order_op**

Order Op Code Supported. 0: False, 1: True

uint32_t **req_num_op**

Request Number of Objects Op Code Supported. 0: False, 1: True

uint32_t **clear_mark_op**

Clear Marking Op Code Supported. 0: False, 1: True

struct *esp_ble_ots_feature_t*::[anonymous] **olcp**

The structure of the OLCP Features field

struct **esp_ble_ots_size_t**

Object Size Characteristic.

Public Members

uint32_t **current_size**

Current Object Size

uint32_t **allocated_size**

Allocated Object Size

struct **esp_ble_ots_utc_t**

UTC.

Public Members

uint16_t **year**

Year as defined by the Gregorian calendar, Valid range 1582 to 9999

uint8_t **month**

Month of the year as defined by the Gregorian calendar, Valid range 1 (January) to 12 (December)

uint8_t **day**

Day of the month as defined by the Gregorian calendar, Valid range 1 to 31

uint8_t **hours**

Number of hours past midnight, Valid range 0 to 23

uint8_t **minutes**

Number of minutes since the start of the hour. Valid range 0 to 59

uint8_t **seconds**

Number of seconds since the start of the minute. Valid range 0 to 59

struct **esp_ble_ots_id_t**

Object ID.

Public Members

uint8_t **id**[6]

Allocated ID. Valid range 0x000000000100 to 0xffffffffffff

struct **esp_ble_ots_prop_t**

Object Properties Characteristic.

Public Members

uint32_t **delete_prop**

Deletion of this object is permitted. 0: False, 1: True

uint32_t **execute_prop**

Execute of this object is permitted. 0: False, 1: True

uint32_t **read_prop**

Reading of this object is permitted. 0: False, 1: True

uint32_t **write_prop**

Writing of this object is permitted. 0: False, 1: True

uint32_t **append_prop**

Appending of this object is permitted. 0: False, 1: True

uint32_t **truncate_prop**

Truncation of this object is permitted. 0: False, 1: True

uint32_t **patch_prop**

Patching of this object is permitted. 0: False, 1: True

uint32_t **mark_prop**

This object is a marked object. 0: False, 1: True

struct **esp_ble_ots_oacp_t**

Object Action Control Point (OACP)

Public Members

uint8_t **op_code**

OACP op code

uint8_t **parameter**[20]

OACP parameter

struct **esp_ble_ots_olcp_t**

Object List Control Point (OLCP)

Public Members

uint8_t **op_code**

OLCP op code

uint8_t **parameter**[6]

OLCP parameter

struct **esp_ble_ots_olcp_rsp_t**

Object List Control Point (OLCP) Response Value.

Public Members

uint8_t **req_code**

OLCP op code

uint8_t **rsp_code**

OLCP rsp op code

uint8_t **rsp_parameter**[6]

OLCP rsp parameter

struct **esp_ble_ots_filter_t**

Object List Filter Characteristic.

Public Members

uint8_t **filter**

Object filter type

uint8_t **filter_parameter**[BLE_OTS_ATT_VAL_LEN]

Object filter parameter

struct **esp_ble_ots_change_t**

Object Change Characteristic.

Public Members

uint8_t **flag**

Flag fields

uint8_t **object_id**[6]

Object ID

Macros

BLE_OTS_ATT_VAL_LEN

Attribute value length

BLE_OTS_UUID16

BLE_OTS_CHR_UUID16_OTS_FEATURE

BLE_OTS_CHR_UUID16_OBJECT_NAME

BLE_OTS_CHR_UUID16_OBJECT_TYPE

BLE_OTS_CHR_UUID16_OBJECT_SIZE

BLE_OTS_CHR_UUID16_OBJECT_FIRST_CREATED

BLE_OTS_CHR_UUID16_OBJECT_LAST_MODIFIED

BLE_OTS_CHR_UUID16_OBJECT_ID

BLE_OTS_CHR_UUID16_OBJECT_PROP

BLE_OTS_CHR_UUID16_OBJECT_ACTION_CONTROL_POINT

BLE_OTS_CHR_UUID16_OBJECT_LIST_CONTROL_POINT

BLE_OTS_CHR_UUID16_OBJECT_LIST_FILTER

BLE_OTS_CHR_UUID16_OBJECT_CHANGED

BLE_OTS_OACP_CREATE

BLE_OTS_OACP_DELETE

BLE_OTS_OACP_CALCULATE_CHECKSUM

BLE_OTS_OACP_EXECUTE

BLE_OTS_OACP_READ

BLE_OTS_OACP_WRITE

BLE_OTS_OACP_ABORT

BLE_OTS_OACP_RESPONSE

BLE_OTS_OACP_RSP_SUCCESS

BLE_OTS_OACP_RSP_NOT_SUPPORT

BLE_OTS_OACP_RSP_INVALID_PARAMETER

BLE_OTS_OACP_RSP_INSUFFICIENT_RESOURCES

BLE_OTS_OACP_RSP_INVALID_OBJECT

BLE_OTS_OACP_RSP_CHANNEL_UNAVAILABLE

BLE_OTS_OACP_RSP_UNSUPPORT_TYPE

BLE_OTS_OACP_RSP_NOT_PERMIT

BLE_OTS_OACP_RSP_OBJECT_LOCK

BLE_OTS_OACP_RSP_OPERATION_FAIL

BLE_OTS_OLCP_FIRST

BLE_OTS_OLCP_LAST

BLE_OTS_OLCP_PREVIOUS

BLE_OTS_OLCP_NEXT

BLE_OTS_OLCP_GO_TO

BLE_OTS_OLCP_ORDER

BLE_OTS_OLCP_REQ_NUM_OF_OBJ

BLE_OTS_OLCP_CLEAR_MARK

BLE_OTS_OLCP_RESPONSE

BLE_OTS_OLCP_SORT_BY_NAME_ASCENDING

BLE_OTS_OLCP_SORT_BY_TYPE_ASCENDING

BLE_OTS_OLCP_SORT_BY_SIZE_ASCENDING

BLE_OTS_OLCP_SORT_BY_FIRST_CREATE_TIME_ASCENDING

BLE_OTS_OLCP_SORT_BY_LAST_MODIFY_TIME_ASCENDING

BLE_OTS_OLCP_SORT_BY_NAME_DESCENDING

BLE_OTS_OLCP_SORT_BY_TYPE_DESCENDING

BLE_OTS_OLCP_SORT_BY_SIZE_DESCENDING

BLE_OTS_OLCP_SORT_BY_FIRST_CREATE_TIME_DESCENDING

BLE_OTS_OLCP_SORT_BY_LAST_MODIFY_TIME_DESCENDING

BLE_OTS_OLCP_RSP_SUCCESS

BLE_OTS_OLCP_RSP_NOT_SUPPORT

BLE_OTS_OLCP_RSP_INVALID_PARAMETER

BLE_OTS_OLCP_RSP_OPERATION_FAIL

BLE_OTS_OLCP_RSP_OUT_OF_BOUNDS

BLE_OTS_OLCP_RSP_TOO_MANY_OBJECTS

BLE_OTS_OLCP_RSP_NO_OBJECT

BLE_OTS_OLCP_RSP_OBJECT_ID_NOT_FOUND

BLE_OTS_FILTER_NAME_START

BLE_OTS_FILTER_NAME_END

BLE_OTS_FILTER_NAME_CONTAIN

BLE_OTS_FILTER_NAME_EXACTLY

BLE_OTS_FILTER_OBJ_TYPE

BLE_OTS_FILTER_CREATE_TIME

BLE_OTS_FILTER_MODIFY_TIME

BLE_OTS_FILTER_CURRENT_SIZE

BLE_OTS_FILTER_ALLOC_SIZE

BLE_OTS_FILTER_MARK_OBJECT

BLE_OTS_OBJ_CHG_SRC

BLE_OTS_OBJ_CHG_CONTENT

BLE_OTS_OBJ_CHG_METADATA

BLE_OTS_OBJ_CHG_CREATION

BLE_OTS_OBJ_CHG_DELETION

3.3.9 TX Power Service

The Tx power service expose the current transmit power level of a device when in a connection.

Examples

bluetooth/ble_services/ble_tps.

API Reference

Header File

- components/bluetooth/ble_services/tps/include/esp_tps.h

Functions

int8_t **esp_ble_tps_get_tx_power_level** (void)

Get the TX Power Level of the device.

Returns : The TX Power Level of the device

esp_err_t **esp_ble_tps_set_tx_power_level** (int8_t tx_power_level)

Set the TX Power Level of the device.

Parameters **tx_power_level** –**[in]** The TX Power Level of the device

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong TX Power Level

esp_err_t **esp_ble_tps_init** (void)

Initialization TX Power Service.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

Macros

BLE_TPS_UUID16

BLE_TPS_CHR_UUID16_TX_POWER_LEVEL

3.3.10 User Data Service

The User Data Service (UDS) exposes user-related data in a sports and fitness, home, or healthcare environment. This allows remote access and update of user data by a client as well as the synchronization of user data between a server and a client.

Examples

bluetooth/ble_services/ble_uds.

API Reference

Header File

- components/bluetooth/ble_services/uds/include/esp_uds.h

Functions

esp_err_t **esp_ble_uds_get_db_incre** (*esp_ble_uds_db_t* *out_val)

Read the value of Database Change Increment characteristic.

Parameters **out_val** –**[in]** The pointer to store the Database Change Increment.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_uds_set_db_incre** (*esp_ble_uds_db_t* *in_val, bool need_send)

Set the Database Change Increment characteristic value.

Parameters
- **in_val** –**[in]** The pointer to store the Database Change Increment Information.
- **need_send** –**[in]** If set to true, the User Data Info will send to remote client.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

esp_err_t **esp_ble_uds_get_user_ctrl** (*esp_ble_uds_user_ctrl_t* *out_val)

Read the value of User Control characteristic.

Parameters **out_val** –**[in]** The pointer to store the User Control value.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_uds_set_user_ctrl** (*esp_ble_uds_user_ctrl_t* *in_val, bool need_send)

Set the User Control Point characteristic value.

Parameters
- **in_val** –**[in]** The pointer to store the User Control Value.
- **need_send** –**[in]** If set to true, the User Control Value will send to remote client.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

esp_err_t **esp_ble_uds_get_reg_user** (*esp_ble_uds_reg_user_t* *out_val)

Read the value of Register User characteristic.

Parameters **out_val** –**[in]** The pointer to store the Register User value.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_uds_set_reg_user** (*esp_ble_uds_reg_user_t* *in_val, bool need_send)

Set the Register User characteristic value.

Parameters
- **in_val** –**[in]** The pointer to store the Register User Value.
- **need_send** –**[in]** If set to true, the Register User Value will send to remote client.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

esp_err_t **esp_ble_uds_init** (void)

Initialization User Data Service.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

Structures

struct **esp_ble_uds_db_t**

Database Change Increment.

Public Members

uint8_t **len**

Database buffer length

uint8_t **db_buf**[BLE_UDS_ATT_VAL_LEN]

Database buffer

struct **esp_ble_uds_reg_user_t**

Register User Characteristic.

Public Members

uint8_t **first_segment**

0: False, 1: True

uint8_t **last_segment**

0: False, 1: True

uint8_t **rolling_segment_number**

When the Rolling segment number is equal to 63, the value is reset to 0 the next time it is incremented

struct *esp_ble_uds_reg_user_t*::[anonymous] **header**

The structure of the Segmentation Header field

uint8_t **user_name_present**

0: False, 1: True

uint8_t **user_name_truncated**

0: False, 1: True

struct *esp_ble_uds_reg_user_t*::[anonymous] **flag**

Defines the value of the flags field that a server shall use when the register user characteristic is indicate

uint8_t **user_index**

User ID Index

uint8_t **user_name**[BLE_UDS_ATT_VAL_LEN]

User name

struct **esp_ble_uds_user_ctrl_t**

User Control Point.

Public Members

uint8_t **op_code**

User control point opcode

uint8_t **param_len**

Parameter buffer length

uint8_t **parameter**[BLE_UDS_USER_CTRL_POINT_PARAM_LEN]

Parameter depend on opcode

Macros

BLE_UDS_ATT_VAL_LEN

Attribute value length

BLE_UDS_USER_CTRL_POINT_PARAM_LEN

User control point param length

BLE_UDS_UUID16

BLE_UDS_CHR_UUID16_DATABASH_CHG

BLE_UDS_CHR_UUID16_USER_INDEX

BLE_UDS_CHR_UUID16_USER_CTRL

BLE_UDS_CHR_UUID16_REG_USER

BLE_UDS_USER_CTRL_REG_NEW_OP

BLE_UDS_USER_CTRL_CONSENT_OP

BLE_UDS_USER_CTRL_DEL_DATA_OP

BLE_UDS_USER_CTRL_LIST_ALL_OP

BLE_UDS_USER_CTRL_DEL_USER_OP

BLE_UDS_USER_CTRL_RSP_OP

3.3.11 Weight Scale Service

This Weight Scale Service exposes weight and related data from a weight scale intended for consumer healthcare and sports/fitness applications.

Examples

bluetooth/ble_services/ble_wss.

API Reference

Header File

- components/bluetooth/ble_services/wss/include/esp_wss.h

Functions

esp_err_t **esp_ble_wss_get_measurement** (*esp_ble_wss_measurement_t* *out_val)

Read the weight measurement characteristic value.

Parameters **out_val** –**[in]** The pointer to store the weight measurement value.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong parameter

esp_err_t **esp_ble_wss_set_measurement** (*esp_ble_wss_measurement_t* *in_val, bool need_send)

Set the weight measurement characteristic value.

Parameters
- **in_val** –**[in]** The pointer to store the weight measurement.
- **need_send** –**[in]** send the weight measurement information to remote client.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

esp_err_t **esp_ble_wss_init** (void)

Initialization Weight Scale Service.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization

- ESP_FAIL on error

Structures

struct **esp_ble_wss_feature_t**

Weight Scale Feature.

Public Members

uint32_t **timestamp**

0: Don’t Support, 1: Support

uint32_t **user_id**

0: Don’t Support, 1: Support

uint32_t **bmi**

0: Don’t Support, 1: Support

uint32_t **weight**

0: Don’t Support, 1: Support

uint32_t **w_resolution**

If weight support, this filed should present

uint32_t **height**

0: Don’t Support, 1: Support

uint32_t **h_resolution**

If height support, this filed should present

struct **esp_ble_wss_measurement_t**

weight Measurement

Public Members

uint32_t **measurement_unit**

0: Kg & meter, 1: reference to weight and height resolution

uint32_t **time_present**

0: Don’t contain time information, 1: time stamp present

uint32_t **user_present**

0: Don’t contain user index, 1: contain user index

uint32_t **bmi_height_present**

0: Don’t contain BMI and height, 1: contain BMI and height

struct *esp_ble_wss_measurement_t*::[anonymous] **flag**

Flag

uint16_t **weight**

weight

uint16_t **year**

Year as defined by the Gregorian calendar, Valid range 1582 to 9999

uint8_t **month**

Month of the year as defined by the Gregorian calendar, Valid range 1 (January) to 12 (December)

uint8_t **day**

Day of the month as defined by the Gregorian calendar, Valid range 1 to 31

uint8_t **hours**

Number of hours past midnight, Valid range 0 to 23

uint8_t **minutes**

Number of minutes since the start of the hour. Valid range 0 to 59

uint8_t **seconds**

Number of seconds since the start of the minute. Valid range 0 to 59

struct *esp_ble_wss_measurement_t*::[anonymous] **timestamp**

The date and time

uint8_t **user_id**

User index

uint8_t **bmi**

BMI

uint16_t **height**

Height

uint8_t **weight_resolution**

Weight resolution

uint8_t **height_resolution**

Height resolution

Macros

BLE_WSS_UUID16

BLE_WSS_CHR_UUID16_WEIGHT_FEATURE

BLE_WSS_CHR_UUID16_WEIGHT_MEASUREMENT

BLE_WSS_TIMESTAMP_MASK

BLE_WSS_MULTI_USER_MASK

BLE_WSS_BMI_MASK

BLE_WSS_WEIGHT_RESOLUTION_OFSET

BLE_WSS_WEIGHT_RESOLUTION_NONE

BLE_WSS_WEIGHT_RESOLUTION_0P5_KG

BLE_WSS_WEIGHT_RESOLUTION_0P2_KG

BLE_WSS_WEIGHT_RESOLUTION_0P1_KG

BLE_WSS_WEIGHT_RESOLUTION_0P05_KG

BLE_WSS_WEIGHT_RESOLUTION_0P02_KG

BLE_WSS_WEIGHT_RESOLUTION_0P01_KG

BLE_WSS_WEIGHT_RESOLUTION_0P005_KG

BLE_WSS_HEIGHT_RESOLUTION_OFSET

BLE_WSS_HEIGHT_RESOLUTION_NONE

BLE_WSS_HEIGHT_RESOLUTION_0P01_M

BLE_WSS_HEIGHT_RESOLUTION_0P005_M

BLE_WSS_HEIGHT_RESOLUTION_0P001_M

BLE_WSS_MEASUREMENT_UINTS_FLAG

BLE_WSS_TIME_STAMP_FLAG

BLE_WSS_USER_ID_FLAG

BLE_WSS_BMI_FLAG

3.4 BLE Profiles

3.4.1 Alert Notification Profile

The Alert Notification profile allows a device like a watch to obtain information from a cellphone about incoming calls, missed calls and SMS/MMS messages. The information may include the caller ID for an incoming call or the sender' s ID for email/SMS/MMS but not the message. This profile also enables the client device to get information about the number of unread messages on the server device.

Examples

bluetooth/ble_profiles/ble_anp.

API Reference

Header File

- components/bluetooth/ble_profiles/std/ble_anp/include/esp_anp.h

Functions

esp_err_t **esp_ble_anp_get_new_alert** (uint8_t cat_id, uint8_t *cat_val)

Read the value of or check supported new alert category.

Attention 1. When cat_id is 0xFF, read the value of supported new alert category.
Attention 2. When cat_id isn' t 0xFF, check supported new alert category is enable or disable.

Parameters
- **cat_id** –**[in]** The ID of the category to read or check
- **cat_val** –**[out]** The value of read or check supported new alert category

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong category of the alert

esp_err_t **esp_ble_anp_set_new_alert** (uint8_t cat_id, *esp_ble_anp_option_t* option)

Request or recovery supported new alert notification to the given category.

Attention 1. When cat_id is 0xFF, recover for all supported new alert category to get the current message counts.
Attention 2. When cat_id isn' t 0xFF, request for a supported new alert category to get the current message counts.

Parameters
- **cat_id** –**[in]** The ID of the category to request or recover the notification to
- **option** –**[in]** Disable or enable supported new alert category

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong category of the alert

esp_err_t **esp_ble_anp_get_unr_alert** (uint8_t cat_id, uint8_t *cat_val)

Read the value of or check supported unread alert status category.

Attention 1. When cat_id is 0xFF, read the value of supported unread alert status category.
Attention 2. When cat_id isn' t 0xFF, check supported unread alert status category is enable or disable.

Parameters
- **cat_id** –**[in]** The ID of the category to read or check
- **cat_val** –**[out]** The value of read or check supported unread alert status category

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong category of the alert

esp_err_t **esp_ble_anp_set_unr_alert**(uint8_t cat_id, *esp_ble_anp_option_t* option)

Request or recovery supported unread alert status notification to the given category.

Attention 1. When cat_id is 0xFF, recover for all supported unread alert status category to get the current message counts.

Attention 2. When cat_id isn't 0xFF, request for an supported unread alert status category to get the current message counts.

Parameters
- **cat_id** –**[in]** The ID of the category to request or recover the notification to
- **option** –**[in]** Disable or enable supported unread alert status category

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong category of the alert

esp_err_t **esp_ble_anp_init**(void)

Initialization GATT Alert Notification Profile.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

esp_err_t **esp_ble_anp_deinit**(void)

Deinitialization GATT Alert Notification Profile.

Returns
- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

Structures

struct **esp_ble_anp_data_t**

The status of the new or unread alert.

Public Members

uint8_t **cat_id**

The predefined categories of unread alerts and messages

The predefined categories of new alerts and messages

uint8_t **count**

The number of unread alerts in the server ranging from 0 to 255

The number of new alerts in the server ranging from 0 to 255

struct *esp_ble_anp_data_t*::[anonymous]::[anonymous] **unr_alert_stat**

The status of unread alerts

uint8_t **cat_info**[BLE_ANP_INFO_STR_MAX_LEN]

The brief text information for the last alert

struct *esp_ble_anp_data_t*::[anonymous]::[anonymous] **new_alert_val**

The status of new alerts

union *esp_ble_anp_data_t*::[anonymous] **[anonymous]**

Alert notification status

Macros

BLE_ANP_UUID16

BLE_ANP_CHR_UUID16_SUP_NEW_ALERT_CAT

BLE_ANP_CHR_UUID16_NEW_ALERT

BLE_ANP_CHR_UUID16_SUP_UNR_ALERT_CAT

BLE_ANP_CHR_UUID16_UNR_ALERT_STAT

BLE_ANP_CHR_UUID16_ALERT_NOT_CTRL_PT

BLE_ANP_CAT_BM_NONE

BLE_ANP_CAT_BM_SIMPLE_ALERT

BLE_ANP_CAT_BM_EMAIL

BLE_ANP_CAT_BM_NEWS

BLE_ANP_CAT_BM_CALL

BLE_ANP_CAT_BM_MISSED_CALL

BLE_ANP_CAT_BM_SMS

BLE_ANP_CAT_BM_VOICE_MAIL

BLE_ANP_CAT_BM_SCHEDULE

BLE_ANP_CAT_ID_SIMPLE_ALERT

BLE_ANP_CAT_ID_EMAIL

BLE_ANP_CAT_ID_NEWS

BLE_ANP_CAT_ID_CALL

BLE_ANP_CAT_ID_MISSED_CALL

BLE_ANP_CAT_ID_SMS

BLE_ANP_CAT_ID_VOICE_MAIL

BLE_ANP_CAT_ID_SCHEDULE

BLE_ANP_CAT_NUM

BLE_ANP_CMD_EN_NEW_ALERT_CAT

BLE_ANP_CMD_EN_UNR_ALERT_CAT

BLE_ANP_CMD_DIS_NEW_ALERT_CAT

BLE_ANP_CMD_DIS_UNR_ALERT_CAT

BLE_ANP_CMD_NOT_NEW_ALERT_IMMEDIATE

BLE_ANP_CMD_NOT_UNR_ALERT_IMMEDIATE

BLE_ANP_INFO_STR_MAX_LEN

BLE_ANP_NEW_ALERT_MAX_LEN

Enumerations

enum **esp_ble_anp_option_t**

The option of the new or unread alert.

Values:

enumerator **BLE_ANP_OPT_ENABLE**

enumerator **BLE_ANP_OPT_DISABLE**

enumerator **BLE_ANP_OPT_RECOVER**

3.4.2 Bluetooth Low Energy MIDI Profile

The BLE MIDI profile enables the transmission of MIDI (Musical Instrument Digital Interface) messages over Bluetooth Low Energy. It is designed for low-latency, low-power communication between MIDI devices such as musical instruments, controllers, and mobile applications.

BLE MIDI allows devices to exchange standard MIDI messages, including note events, control changes, and system messages, while benefiting from the reduced power consumption and flexible connection model provided by Bluetooth Low Energy.

The profile follows the MIDI over BLE specification defined by the MIDI Manufacturers Association (MMA) and the Bluetooth SIG.

Examples

bluetooth/ble_profiles/ble_midi.

API Reference

Header File

- components/bluetooth/ble_profiles/third_party/ble_midi/include/esp_ble_midi.h

Functions

esp_err_t **esp_ble_midi_profile_init** (void)

Initialize the BLE MIDI profile.

This function must be called once before using any other BLE MIDI functions. It initializes internal synchronization primitives and state.

Returns
- ESP_OK on success
- ESP_ERR_NO_MEM if mutex creation fails
- ESP_OK if already initialized (idempotent)

esp_err_t **esp_ble_midi_profile_deinit** (void)

Deinitialize the BLE MIDI profile.

This function cleans up internal resources including mutex and clears all callbacks and state. After calling this function, esp_ble_midi_profile_init() must be called again before using any other BLE MIDI functions.

Returns
- ESP_OK on success
- ESP_OK if already deinitialized (idempotent)

esp_err_t **esp_ble_midi_register_rx_cb** (*esp_ble_midi_rx_cb_t* cb, void *user_ctx)

Register application RX callback to receive BLE-MIDI event packets.

Called from the BLE host task; keep the handler short or hand off to another task.

Note: esp_ble_midi_profile_init() must be called before registering callbacks.

Parameters
- **cb** –**[in]** RX callback; pass NULL to unregister
- **user_ctx** –**[in]** User context pointer to be passed to the callback

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if esp_ble_midi_profile_init() has not been called

- ESP_FAIL on other error

esp_err_t **esp_ble_midi_register_event_cb** (*esp_ble_midi_event_cb_t* cb)

Register parsed MIDI event callback.

Called once per decoded MIDI message (including SysEx and real-time). Runs in the BLE host task; keep the handler short or post to an app task.

Note: esp_ble_midi_profile_init() must be called before registering callbacks.

Parameters **cb** –**[in]** Event callback; pass NULL to unregister

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if esp_ble_midi_profile_init() has not been called
- ESP_FAIL on other error

esp_err_t **esp_ble_midi_send** (const uint8_t *data, uint16_t len)

Send BLE-MIDI event packet via notification.

data must be a complete BEP payload (packet header timestamp high bits and a per-message timestamp byte before each message). Most applications should prefer esp_ble_midi_send_raw_midi() or the helper APIs.

Note: esp_ble_midi_profile_init() must be called before sending.

Parameters
- **data** –**[in]** Pointer to packet buffer
- **len** –**[in]** Packet length

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid args
- ESP_ERR_INVALID_STATE if esp_ble_midi_profile_init() has not been called or notification is not enabled
- ESP_FAIL on other error

esp_err_t **esp_ble_midi_send_raw_midi** (const uint8_t *midi, uint8_t midi_len, uint16_t timestamp)

Build and send a BLE-MIDI Event Packet from raw MIDI bytes.

Packs the BLE-MIDI timestamps (packet header high bits + per-message low byte).

Parameters
- **midi** –**[in]** Pointer to MIDI message bytes (status + data)
- **midi_len** –**[in]** Length of MIDI message (1-3 bytes typical)
- **timestamp** –**[in]** 13-bit timestamp in milliseconds (rolls at 8192)

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on wrong parameter
- ESP_FAIL on other error

esp_err_t **esp_ble_midi_send_note_on** (uint8_t channel, uint8_t note, uint8_t velocity, uint16_t timestamp)

Send MIDI Note On.

Parameters
- **channel** –**[in]** 0-15
- **note** –**[in]** 0-127
- **velocity** –**[in]** 0-127
- **timestamp** –**[in]** 13-bit timestamp in ms

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on wrong parameter
- ESP_FAIL on other error

esp_err_t **esp_ble_midi_send_note_off**(uint8_t channel, uint8_t note, uint8_t velocity, uint16_t timestamp)

Send MIDI Note Off.

Parameters
- **channel** –**[in]** 0-15
- **note** –**[in]** 0-127
- **velocity** –**[in]** 0-127
- **timestamp** –**[in]** 13-bit timestamp in ms

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on wrong parameter
- ESP_FAIL on other error

esp_err_t **esp_ble_midi_send_cc**(uint8_t channel, uint8_t controller, uint8_t value, uint16_t timestamp)

Send MIDI Control Change.

Parameters
- **channel** –**[in]** 0-15
- **controller** –**[in]** 0-127
- **value** –**[in]** 0-127
- **timestamp** –**[in]** 13-bit timestamp in ms

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on wrong parameter
- ESP_FAIL on other error

esp_err_t **esp_ble_midi_send_pitch_bend**(uint8_t channel, uint16_t value, uint16_t timestamp)

Send MIDI Pitch Bend (14-bit value)

Parameters
- **channel** –**[in]** 0-15
- **value** –**[in]** 0-16383 (8192 is center)
- **timestamp** –**[in]** 13-bit timestamp in ms

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on wrong parameter
- ESP_FAIL on other error

esp_err_t **esp_ble_midi_send_multi**(const uint8_t *const msgs[], const uint8_t msg_lens[], const uint16_t timestamps[], uint16_t count, size_t *first_unsent_index)

Send multiple MIDI events aggregated into a single BLE-MIDI packet.

Aggregates multiple MIDI messages in one notification to improve throughput. Each message will be prefixed with its own BLE-MIDI timestamp byte.

Notes:

- The function packs as many messages as fit in an internal buffer (<= ATT MTU).
- If not all messages fit, this function sends multiple notifications.
- For SysEx, pass a complete 0xF0···0xF7 sequence. For automatic fragmentation, use esp_ble_midi_send_sysex(), or split by MTU and call this repeatedly.
- On failure: If first_unsent_index is provided, it will be set to the index of the first message that was not sent. On success, it will be set to count. This allows callers to determine which messages were successfully transmitted before the error occurred.

Parameters

- **msgs** –**[in]** Array of pointers to MIDI messages (status + data bytes)
- **msg_lens** –**[in]** Array of message lengths (1..n)
- **timestamps** –**[in]** Array of 13-bit timestamps for each message (ms resolution)
- **count** –**[in]** Number of messages in arrays
- **first_unsent_index** –**[out]** Optional output parameter: on failure, set to the index of the first message that was not sent; on success, set to count. Pass NULL if not needed.

Returns
- ESP_OK on success (all messages sent)
- ESP_ERR_INVALID_ARG on wrong parameter
- ESP_FAIL on other error (partial send may have occurred; check first_unsent_index)

esp_err_t **esp_ble_midi_send_sysex**(const uint8_t *sysex, uint16_t len, uint16_t timestamp)

Send a System Exclusive (SysEx) message with automatic multi-packet fragmentation.

The buffer must begin with 0xF0 and end with 0xF7. The function splits the SysEx into multiple BLE-MIDI Event Packets based on current ATT MTU.

Notes:

- Uses the current MTU to choose fragment sizes (accounting for ATT and BEP overhead).
- Real-time messages (0xF8..0xFF) are delivered as separate events on the receiver.

Parameters
- **sysex** –**[in]** Pointer to complete SysEx buffer (0xF0 ···0xF7)
- **len** –**[in]** Length of the SysEx buffer
- **timestamp** –**[in]** 13-bit timestamp in ms to use for all fragments

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on wrong parameter
- ESP_FAIL on other error

void **esp_ble_midi_on_bep_received**(const uint8_t *data, uint16_t len)

Profile entry for raw BLE-MIDI Event Packet received from GATT.

The service layer should call this to hand off incoming packets.

Parameters
- **data** –**[in]** Pointer to BEP payload buffer
- **len** –**[in]** Length of BEP payload

uint16_t **esp_ble_midi_get_timestamp_ms**(void)

Get 13-bit rolling timestamp in milliseconds for BLE-MIDI.

Returns a 1 ms resolution counter modulo 8192 suitable for BLE-MIDI timestamps. Intended for use as the `timestamp` parameter in send helpers.

Returns 13-bit timestamp in milliseconds (0..8191)

esp_err_t **esp_ble_midi_set_notify_enabled**(bool enabled)

Set notification enabled state (typically called when CCCD is updated).

This should be called when the client writes to the CCCD (Client Characteristic Configuration Descriptor) to enable/disable notifications. MIDI send functions will only work when notification is enabled.

Note: esp_ble_midi_profile_init() must be called before setting notify state.

Parameters **enabled** –**[in]** true if notification is enabled, false otherwise

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if esp_ble_midi_profile_init() has not been called

bool **esp_ble_midi_is_notify_enabled**(void)

Check if notification is currently enabled.

Note: esp_ble_midi_profile_init() must be called before checking notify state.

Returns true if notification is enabled, false otherwise (also returns false if not initialized)

Macros

BLE_MIDI_SERVICE_UUID128

BLE MIDI Service UUID (128-bit): 03B80E5A-EDE8-4B33-A751-6CE34EC4C700

BLE_MIDI_CHAR_UUID128

BLE MIDI IO Characteristic UUID (128-bit): 7772E5DB-3868-4112-A1A9-F2669D106BF3

BLE_MIDI_MAX_PKT_LEN

Maximum BLE-MIDI packet length (BLE 5.0+)

Used for buffer allocation and size validation in BLE-MIDI packets.

ESP_BLE_MIDI_TIMESTAMP_MAX

13-bit rolling timestamp (1 ms resolution)

Type Definitions

typedef void (*__esp_ble_midi_rx_cb_t__)(const uint8_t *data, uint16_t len, void *user_ctx)

RX callback for raw BLE-MIDI Event Packet (BEP) payloads.

The buffer contains the BEP payload as defined by the BLE-MIDI specification. For parsed per-message events, use esp_ble_midi_register_event_cb().

Param data **[in]** Pointer to BEP payload buffer
Param len **[in]** Length of BEP payload
Param user_ctx **[in]** User context pointer passed during registration

typedef void (*__esp_ble_midi_event_cb_t__)(uint16_t timestamp_ms, *esp_ble_midi_event_type_t* event_type, const uint8_t *msg, uint16_t msg_len)

Parsed event callback for a single decoded MIDI message.

- msg contains one complete MIDI message (status + data).
- SysEx is delivered as a complete 0xF0 ⋯0xF7 sequence (reassembled).
- Real-time messages (0xF8..0xFF) are delivered as 1-byte events and may interleave.
- On SysEx overflow (ESP_BLE_MIDI_EVENT_SYSEX_OVERFLOW), msg is NULL and msg_len is 0.

Param timestamp_ms **[in]** 13-bit rolling timestamp in milliseconds
Param event_type **[in]** Event type (normal message or overflow)
Param msg **[in]** Pointer to a single MIDI message (status + data), NULL on overflow
Param msg_len **[in]** Message length in bytes, 0 on overflow

Enumerations

enum **esp_ble_midi_event_type_t**

MIDI event types.

Values:

enumerator **ESP_BLE_MIDI_EVENT_NORMAL**

Normal MIDI message

enumerator **ESP_BLE_MIDI_EVENT_SYSEX_OVERFLOW**

SysEx buffer overflow detected

3.4.3 Heart Rate Profile

The Heart Rate Profile is used to enable a data collection device to obtain data from a Heart Rate Sensor that exposes the Heart Rate Service.

Examples

bluetooth/ble_profiles/ble_hrp.

API Reference

Header File

- components/bluetooth/ble_profiles/std/ble_hrp/include/esp_hrp.h

Functions

esp_err_t **esp_ble_hrp_get_location**(uint8_t *location)

Get the sensor location value of the device.

Parameters **location** –**[in]** The pointer to store the sensor location value

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery level

esp_err_t **esp_ble_hrp_get_ctrl**(uint8_t *cmd_id)

Get the control point value of the device.

Parameters **cmd_id** –**[in]** The pointer to store the control point value

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery level

esp_err_t **esp_ble_hrp_set_ctrl**(uint8_t cmd_id)

Set the control point value of the device.

Parameters **cmd_id** –**[in]** The control point value

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery level

esp_err_t **esp_ble_hrp_init**(void)

Initialization Heart Rate Profile.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

esp_err_t **esp_ble_hrp_deinit** (void)

Deinitialization Heart Rate Profile.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

Structures

struct **esp_ble_hrp_data_t**

Heart Rate Measurement Characteristic.

Public Members

uint8_t **format**

Heart rate value format flag

uint8_t **detected**

Sensor contact detected flag

uint8_t **supported**

Sensor contact supported flag

uint8_t **energy**

Energy expended present flag

uint8_t **interval**

RR-Interval present flag

uint8_t **reserved**

Reserved for future use flag

struct *esp_ble_hrp_data_t*::[anonymous] **flags**

Flags of heart rate measurement

uint8_t **u8**

8 bit resolution

uint16_t **u16**

16 bit resolution

union *esp_ble_hrp_data_t*::[anonymous] **heartrate**

Heart rate measurement value

uint16_t **energy_val**

Expended energy value

uint16_t **interval_buf**[BLE_HRP_CHR_MERSUREMENT_RR_INTERVAL_MAX_NUM]
The RR-Interval value represents the time between two R-Wave detections

Macros

BLE_HRP_UUID16

BLE_HRP_CHR_UUID16_MEASUREMENT

BLE_HRP_CHR_UUID16_BODY_SENSOR_LOC

BLE_HRP_CHR_UUID16_HEART_RATE_CNTL_POINT

BLE_HRP_CHR_MERSUREMENT_RR_INTERVAL_MAX_NUM

BLE_HRP_FLAGS_BM_NONE

BLE_HRP_FLAGS_BM_FORMAT

BLE_HRP_FLAGS_BM_SENSOR_CONTACT_DETECTED

BLE_HRP_FLAGS_BM_SENSOR_CONTACT_SUPPOTRED

BLE_HRP_FLAGS_BM_ENERGY

BLE_HRP_FLAGS_BM_RR_INTERVAL

BLE_HRP_FLAGS_BM_RFU

BLE_HRP_CHR_MERSUREMENT_FLAGS_FORMAT_U8

BLE_HRP_CHR_MERSUREMENT_FLAGS_FORMAT_U16

BLE_HRP_CHR_MERSUREMENT_FLAGS_NOT

BLE_HRP_CHR_MERSUREMENT_FLAGS_SET

BLE_HRP_CMD_RFU

BLE_HRP_CMD_RESET_ENERGY_EXPENDED

BLE_HRP_CMD_MAX

3.4.4 Health Thermometer Profile

The Health Thermometer Profile is used to enable a data collection device to obtain data from a thermometer sensor that exposes the Health Thermometer Service.

Examples

bluetooth/ble_profiles/ble_htp.

API Reference

Header File

- components/bluetooth/ble_profiles/std/ble_htp/include/esp_htp.h

Functions

esp_err_t **esp_ble_htp_get_temp_type** (uint8_t *temp_type)

Get the current temperature type value of the device.

Parameters **temp_type** –**[in]** The pointer to store the current temperature type value

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery level

esp_err_t **esp_ble_htp_get_measurement_interval** (uint16_t *interval_val)

Get the measurement interval value of the device.

Parameters **interval_val** –**[in]** The pointer to store the measurement interval value

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery level

esp_err_t **esp_ble_htp_set_measurement_interval** (uint16_t interval_val)

Set the measurement interval value of the device.

Parameters **interval_val** –**[in]** The measurement interval value

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong battery level

esp_err_t **esp_ble_htp_init** (void)

Initialization Health Thermometer Profile.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

esp_err_t **esp_ble_htp_deinit** (void)

Deinitialization Health Thermometer Profile.

Returns

- ESP_OK on successful
- ESP_ERR_INVALID_ARG on wrong initialization
- ESP_FAIL on error

Structures

struct **esp_ble_htp_data_t**

Temperature Measurement and Intermediate Temperature Characteristic.

Public Members

uint8_t **temperature_unit**

Temperature units flag

uint8_t **time_stamp**

Time stamp flag

uint8_t **temperature_type**

Temperature type flag

uint8_t **reserved**

Reserved for future use

struct *esp_ble_htp_data_t*::[anonymous] **flags**

Flags of temperature

uint32_t **celsius**

Celsius unit

uint32_t **fahrenheit**

Fahrenheit unit

union *esp_ble_htp_data_t*::[anonymous] **temperature**

Temperature value

uint16_t **year**

Year as defined by the Gregorian calendar, Valid range 1582 to 9999

uint8_t **month**

Month of the year as defined by the Gregorian calendar, Valid range 1 (January) to 12 (December)

uint8_t **day**

Day of the month as defined by the Gregorian calendar, Valid range 1 to 31

uint8_t **hours**

Number of hours past midnight, Valid range 0 to 23

uint8_t **minutes**

Number of minutes since the start of the hour. Valid range 0 to 59

uint8_t **seconds**

Number of seconds since the start of the minute. Valid range 0 to 59

struct *esp_ble_htp_data_t*::[anonymous] **timestamp**

The date and time

uint8_t **location**

The location of a temperature measurement

Macros

BLE_HTP_UUID16

BLE_HTP_CHR_UUID16_TEMPERATURE_MEASUREMENT

BLE_HTP_CHR_UUID16_TEMPERATURE_TYPE

BLE_HTP_CHR_UUID16_INTERMEDIATE_TEMPERATURE

BLE_HTP_CHR_UUID16_MEASUREMENT_INTERVAL

BLE_HTP_FLAGS_BM_NONE

BLE_HTP_FLAGS_BM_TEMPERATURE_UNITS

BLE_HTP_FLAGS_BM_TIME_STAMP

BLE_HTP_FLAGS_BM_TEMPERATURE_TYPE

BLE_HTP_FLAGS_BM_RFU

BLE_HTP_CHR_TEMPERATURE_UNITS_CELSIUS

BLE_HTP_CHR_TEMPERATURE_UNITS_FAHRENHEIT

BLE_HTP_CHR_TEMPERATURE_FLAGS_NOT

BLE_HTP_CHR_TEMPERATURE_FLAGS_SET

BLE_HTP_CHR_TEMPERATURE_TYPE_RFU

BLE_HTP_CHR_TEMPERATURE_TYPE_ARMPIT

BLE_HTP_CHR_TEMPERATURE_TYPE_BODY

BLE_HTP_CHR_TEMPERATURE_TYPE_EAR

BLE_HTP_CHR_TEMPERATURE_TYPE_FINGER

BLE_HTP_CHR_TEMPERATURE_TYPE_GAST_TRACT

BLE_HTP_CHR_TEMPERATURE_TYPE_MOUTH

BLE_HTP_CHR_TEMPERATURE_TYPE_RECTUM

BLE_HTP_CHR_TEMPERATURE_TYPE_TOE

BLE_HTP_CHR_TEMPERATURE_TYPE_TYMPANUM

BLE_HTP_CHR_TEMPERATURE_TYPE_MAX

3.4.5 Object Transfer Profile

The Object Transfer Profile enables a client to manage objects on a server and transfer object contents over an L2CAP connection-oriented channel.

Examples

bluetooth/ble_profiles/ble_otp/ble_otp_client. bluetooth/ble_profiles/ble_otp/ble_otp_server.

API Reference

Header File

- components/bluetooth/ble_profiles/std/ble_otp/include/esp_otp.h

Functions

esp_err_t **esp_ble_otp_client_resume_read_checksum**(uint16_t conn_handle, uint32_t offset, uint32_t length)

Resume read (checksum step)

Note: Feature bit `calculate_op` must be set. Caller should verify checksum and then call esp_ble_otp_client_read_object() with remaining offset.

Parameters
- **conn_handle** –**[in]** BLE connection handle
- **offset** –**[in]** Offset to start calculating from
- **length** –**[in]** Length to calculate

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_ERR_NOT_SUPPORTED if checksum not supported
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_resume_write_current_size**(uint16_t conn_handle, uint32_t total_size, *esp_ble_otp_write_mode_t* mode)

Resume write using Current Size method.

Note: Reads current size and resumes writing remaining bytes. Requires write_op support. For patching, use integrity method instead.

Parameters
- **conn_handle** –**[in]** BLE connection handle
- **total_size** –**[in]** Total object size to write

- **mode** –**[in]** Write mode

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_ERR_NOT_SUPPORTED if not supported
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_resume_write_checksum**(uint16_t conn_handle, uint32_t offset, uint32_t length)

Resume write (checksum step)

Note: Feature bit `calculate_op` must be set. Caller should verify checksum and then call esp_ble_otp_client_write_object() for failed regions.

Parameters

- **conn_handle** –**[in]** BLE connection handle
- **offset** –**[in]** Offset to start calculating from
- **length** –**[in]** Length to calculate

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_ERR_NOT_SUPPORTED if checksum not supported
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_discover_ots**(uint16_t conn_handle)

Discover OTS service on the connected peer device.

Note: Allowed when OTS is idle/failed. Discovery must complete before OACP/OLCP.

Parameters **conn_handle** –**[in]** BLE connection handle

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_read_feature**(uint16_t conn_handle, *esp_ble_ots_feature_t* *feature)

Read OTS Feature characteristic.

Note: Required before OACP/OLCP. Feature bits gate supported operations.

Parameters

- **conn_handle** –**[in]** BLE connection handle
- **feature** –**[out]** Pointer to store OTS Feature

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_select_first** (uint16_t conn_handle)

Select first object in the list.

Note: Allowed only when session is idle. Selection must be stable before OACP.

Parameters **conn_handle** –**[in]** BLE connection handle

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_select_last** (uint16_t conn_handle)

Select last object in the list.

Note: Allowed only when session is idle. Selection must be stable before OACP.

Parameters **conn_handle** –**[in]** BLE connection handle

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_select_previous** (uint16_t conn_handle)

Select previous object in the list.

Note: Allowed only when session is idle. Selection must be stable before OACP.

Parameters **conn_handle** –**[in]** BLE connection handle

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_select_next** (uint16_t conn_handle)

Select next object in the list.

Note: Allowed only when session is idle. Selection must be stable before OACP.

Parameters **conn_handle** –**[in]** BLE connection handle

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_select_by_id** (uint16_t conn_handle, const *esp_ble_ots_id_t* *object_id)

Select object by ID.

Note: Allowed only when session is idle. Selection must be stable before OACP. Requires OLCP Go To support. Filters are ignored by Go To and reset to No Filter.

Parameters
- **conn_handle** –**[in]** BLE connection handle
- **object_id** –**[in]** Object ID to select

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_select_by_index**(uint16_t conn_handle, uint32_t index)

Select object by index.

Parameters
- **conn_handle** –**[in]** BLE connection handle
- **index** –**[in]** Object index (0-based)

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_set_sort_order**(uint16_t conn_handle, uint8_t sort_order)

Set object list sort order.

Note: Allowed only when session is idle.

Parameters
- **conn_handle** –**[in]** BLE connection handle
- **sort_order** –**[in]** Sort order (BLE_OTS_OLCP_SORT_BY_*)

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_request_num_objects**(uint16_t conn_handle)

Request number of objects.

Note: Allowed only when session is idle.

Parameters **conn_handle** –**[in]** BLE connection handle

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_clear_mark**(uint16_t conn_handle)

Clear marking of objects.

Note: Allowed only when session is idle.

Parameters **conn_handle** –**[in]** BLE connection handle

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_set_filter**(uint16_t conn_handle, const *esp_ble_ots_filter_t* *filter)

Set object list filter.

Note: Allowed only when session is idle.

Parameters

- **conn_handle** –**[in]** BLE connection handle
- **filter** –**[in]** Filter type and parameter

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_read_object_info**(uint16_t conn_handle, *esp_ble_otp_object_info_t* *object_info)

Read current object information.

Note: Requires object selected. Updates metadata state to VALID. Call this after OLCP selection (e.g. select_first) before read_object, write_object, or delete_object so that metadata is valid.

Parameters

- **conn_handle** –**[in]** BLE connection handle
- **object_info** –**[out]** Pointer to store object information

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_write_name**(uint16_t conn_handle, const uint8_t *name, size_t name_len)

Write Object Name metadata.

Note: Requires object selected. Name must be non-empty and contain no ASCII control characters. Long write is not supported; if length exceeds ATT_MTU-3, return ESP_ERR_NOT_SUPPORTED.

Parameters

- **conn_handle** –**[in]** BLE connection handle

- **name** –**[in]** Object name buffer
- **name_len** –**[in]** Object name length

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_ERR_NOT_SUPPORTED if long write is required
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_write_prop**(uint16_t conn_handle, const *esp_ble_ots_prop_t* *prop)

Write Object Properties metadata.

Note: Requires object selected.

Parameters

- **conn_handle** –**[in]** BLE connection handle
- **prop** –**[in]** Object properties

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_discover_all_start**(uint16_t conn_handle)

Discover all objects (OLCP First)

Note: Requires OTS discovered and OLCP supported. Caller should iterate with esp_ble_otp_client_discover_next().

Parameters **conn_handle** –**[in]** BLE connection handle

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_ERR_NOT_SUPPORTED if OLCP not supported
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_discover_next**(uint16_t conn_handle)

Discover next object (OLCP Next)

Note: Requires OTS discovered and OLCP supported.

Parameters **conn_handle** –**[in]** BLE connection handle

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_ERR_NOT_SUPPORTED if OLCP not supported
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_discover_by_filter**(uint16_t conn_handle, const *esp_ble_ots_filter_t* *filter)

Discover objects by filter.

Note: Requires OTS discovered, OLCP supported, and Object List Filter exposed.

Parameters
- **conn_handle** –**[in]** BLE connection handle
- **filter** –**[in]** Filter to apply

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_ERR_NOT_SUPPORTED if OLCP or filter not supported
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_select_directory_listing**(uint16_t conn_handle)

Select the Directory Listing Object (Object ID 0x000000000000)

Note: Requires OTS discovered and OLCP Go-To supported.

Parameters **conn_handle** –**[in]** BLE connection handle

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_ERR_NOT_SUPPORTED if OLCP not supported
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_read_directory_listing**(uint16_t conn_handle)

Read Directory Listing Object contents.

Note: Selects Directory Listing Object (ID 0x000000000000), reads current size, and starts OACP Read for the full contents.

Parameters **conn_handle** –**[in]** BLE connection handle

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_ERR_NOT_SUPPORTED if OLCP not supported
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_create_object**(uint16_t conn_handle, uint16_t object_type, uint32_t object_size)

Create a new object.

Note: Feature bit `create_op` must be set. After success, client should write a valid Object Name (non-empty, no ASCII control characters). If disconnected before naming, the server may delete the object as malformed.

Parameters

- **conn_handle** –**[in]** BLE connection handle
- **object_type** –**[in]** Object type
- **object_size** –**[in]** Allocated object size

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_delete_object**(uint16_t conn_handle)

Delete current object.

Note: Requires object selected + metadata valid + server object property delete_prop set. Feature bit `delete_op` must be set. Server may respond with OACP RSP Invalid Object (e.g. object not found/deleted).

Parameters **conn_handle** –**[in]** BLE connection handle

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_ERR_NOT_SUPPORTED if object' s delete property is not set
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_read_object**(uint16_t conn_handle, uint32_t offset, uint32_t length)

Read object data via Object Transfer Channel.

Note: Requires: OTS discovered + object selected + metadata valid. Feature bit `read_op` must be set. After OLCP selection (e.g. select_first), metadata is UNKNOWN; call esp_ble_otp_client_read_object_info() first, or this returns ESP_ERR_INVALID_STATE.

Parameters

- **conn_handle** –**[in]** BLE connection handle
- **offset** –**[in]** Offset to start reading from
- **length** –**[in]** Length to read (0 = read all remaining data)

Returns

- ESP_OK on success (transfer will be initiated, wait for events)
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role, or metadata not valid
- ESP_ERR_NOT_SUPPORTED if read_op not supported
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_write_object**(uint16_t conn_handle, uint32_t offset, uint32_t length, *esp_ble_otp_write_mode_t* mode)

Write object data via Object Transfer Channel.

Note: Requires: OTS discovered + object selected + metadata valid. Feature bit `write_op` must be set. Object Properties must permit write. After OLCP selection, call esp_ble_otp_client_read_object_info() first if metadata not valid. Offset must be <= Current Size; Offset + Length must be <= Allocated Size, unless mode requests append/truncate/patch and the corresponding feature bit is supported. For resume (Current Size method), use the current size as the offset.

Parameters

- **conn_handle** –**[in]** BLE connection handle
- **offset** –**[in]** Offset to start writing at
- **length** –**[in]** Length to write
- **mode** –**[in]** Write mode

Returns

- ESP_OK on success (transfer will be initiated, wait for events)
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role, or metadata not valid
- ESP_ERR_NOT_SUPPORTED if write_op not supported or object properties do not permit write
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_calculate_checksum**(uint16_t conn_handle, uint32_t offset, uint32_t length)

Calculate checksum of object data.

Note: Feature bit `calculate_op` must be set. This API is used by resume procedures to verify partial data.

Parameters

- **conn_handle** –**[in]** BLE connection handle
- **offset** –**[in]** Offset to start calculating from
- **length** –**[in]** Length to calculate (0 = calculate all remaining data)

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_execute_object**(uint16_t conn_handle, const uint8_t *parameters, uint8_t param_len)

Execute object (if supported)

Note: Requires selected object + metadata valid + execute_prop enabled. Feature bit `execute_op` must be set. If committing a write, this is allowed after EOF even when session is busy.

Parameters

- **conn_handle** –**[in]** BLE connection handle
- **parameters** –**[in]** Execute parameters (optional, can be NULL)
- **param_len** –**[in]** Parameter length

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_abort**(uint16_t conn_handle)

Abort current object operation.

Note: ABORTED event is reported only after OACP Abort response.

Parameters **conn_handle** –**[in]** BLE connection handle

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in client role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_send_data**(const *esp_ble_otp_transfer_info_t* *transfer_info, const uint8_t *data, uint16_t data_len)

Send data over Object Transfer Channel (after read/write operation initiated)

Parameters
- **transfer_info** –**[in]** Transfer channel information (from event)
- **data** –**[in]** Data to send
- **data_len** –**[in]** Data length

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if transfer channel is not ready
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_client_disconnect_transfer_channel**(const *esp_ble_otp_transfer_info_t* *transfer_info)

Disconnect Object Transfer Channel.

Parameters **transfer_info** –**[in]** Transfer channel information (from event)

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_server_set_feature**(const *esp_ble_ots_feature_t* *feature)

Set OTS feature (Server only)

Parameters **feature** –**[in]** OTS Feature to set

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in server role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_server_register_oacp_callback**(esp_err_t (*callback)(uint8_t op_code, const uint8_t *parameter, uint16_t param_len, void *ctx), void *ctx)

Register callback for OACP operations (Server only)

Parameters
- **callback** –**[in]** Callback function to handle OACP operations Return ESP_OK to accept, ESP_ERR_* to reject with corresponding error code
- **ctx** –**[in]** User context passed to the callback

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in server role

esp_err_t **esp_ble_otp_server_register_olcp_callback**(esp_err_t (*callback)(uint8_t op_code, const uint8_t *parameter, uint16_t param_len, void *ctx), void *ctx)

Register callback for OLCP operations (Server only)

Parameters

- **callback** –**[in]** Callback function to handle OLCP operations Return ESP_OK to accept, ESP_ERR_* to reject with corresponding error code
- **ctx** –**[in]** User context passed to the callback

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in server role

esp_err_t **esp_ble_otp_server_send_oacp_response**(uint8_t req_op_code, uint8_t rsp_code, const uint8_t *rsp_parameter, uint8_t param_len)

Send OACP response (Server only)

Parameters

- **req_op_code** –**[in]** Request op code
- **rsp_code** –**[in]** Response code
- **rsp_parameter** –**[in]** Response parameter (can be NULL)
- **param_len** –**[in]** Parameter length

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in server role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_server_send_olcp_response**(uint8_t req_op_code, uint8_t rsp_code, const uint8_t *rsp_parameter, uint8_t param_len)

Send OLCP response (Server only)

Parameters

- **req_op_code** –**[in]** Request op code
- **rsp_code** –**[in]** Response code
- **rsp_parameter** –**[in]** Response parameter (can be NULL)
- **param_len** –**[in]** Parameter length

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if OTP is not initialized or not in server role
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_server_send_data**(const *esp_ble_otp_transfer_info_t* *transfer_info, const uint8_t *data, uint16_t data_len)

Send data over Object Transfer Channel (Server only)

Parameters

- **transfer_info** –**[in]** Transfer channel information (from event)
- **data** –**[in]** Data to send
- **data_len** –**[in]** Data length

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_INVALID_STATE if transfer channel is not ready
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_server_disconnect_transfer_channel**(const *esp_ble_otp_transfer_info_t* *transfer_info)

Disconnect Object Transfer Channel (Server only)

Parameters **transfer_info** –**[in]** Transfer channel information (from event)

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_init** (const *esp_ble_otp_config_t* *config)

Initialize OTP.

Parameters **config** –**[in]** OTP configuration

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG on invalid parameters
- ESP_ERR_NO_MEM if memory allocation failed
- ESP_FAIL on other error

esp_err_t **esp_ble_otp_deinit** (void)

Deinitialize OTP.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if OTP is not initialized
- ESP_FAIL on other error

Unions

union **esp_ble_otp_event_data_t**

#include <esp_otp.h> OTP Event Data Union.

Public Members

uint16_t **conn_handle**

BLE connection handle

uint8_t **op_code**

OACP op code

struct *esp_ble_otp_event_data_t*::[anonymous] **oacp_started**

OACP started event data

struct *esp_ble_otp_event_data_t*::[anonymous] **oacp_aborted**

OACP aborted event data

esp_ble_ots_feature_t **feature**

OTS Feature

struct *esp_ble_otp_event_data_t*::[anonymous] **ots_discovered**

OTS discovered event data

esp_ble_otp_object_info_t **object_info**

Selected object information

struct *esp_ble_otp_event_data_t*::[anonymous] **object_selected**

Object selected event data

esp_ble_ots_change_t **change**

Object change notification

struct *esp_ble_otp_event_data_t*::[anonymous] **object_changed**

Object changed event data

esp_ble_otp_oacp_rsp_t **response**

OACP response

struct *esp_ble_otp_event_data_t*::[anonymous] **oacp_response**

OACP response event data

esp_ble_otp_olcp_rsp_t **response**

OLCP response

struct *esp_ble_otp_event_data_t*::[anonymous] **olcp_response**

OLCP response event data

esp_ble_otp_transfer_info_t **transfer_info**

Transfer channel information

struct *esp_ble_otp_event_data_t*::[anonymous] **transfer_channel_connected**

Transfer channel connected event data

struct *esp_ble_otp_event_data_t*::[anonymous] **transfer_channel_disconnected**

Transfer channel disconnected event data

uint8_t **data**[BLE_OTP_EVENT_DATA_MAX_LEN]

Copy of received data

uint16_t **data_len**

Data length in this chunk

Data length sent

uint16_t **chunk_offset**

Byte offset of this chunk within the SDU (0 for single-chunk)

uint16_t **total_len**

Total SDU length (same for all chunks of one SDU)

struct *esp_ble_otp_event_data_t*::[anonymous] **transfer_data_received**

Transfer data received event data

struct *esp_ble_otp_event_data_t*::[anonymous] **transfer_data_sent**

Transfer data sent event data

struct *esp_ble_otp_event_data_t*::[anonymous] **transfer_eof**

Transfer EOF event data

bool **success**

Transfer success status

struct *esp_ble_otp_event_data_t*::[anonymous] **transfer_complete**

Transfer complete event data

esp_err_t **error**

Error code

struct *esp_ble_otp_event_data_t*::[anonymous] **transfer_error**

Transfer error event data

Structures

struct **esp_ble_otp_config_t**

OTP Configuration.

Public Members

esp_ble_otp_role_t **role**

OTP role (Client or Server)

uint16_t **psm**

PSM for Object Transfer Channel (default: BLE_OTP_PSM_DEFAULT)

uint16_t **l2cap_coc_mtu**

L2CAP CoC MTU for object transfer (default: BLE_OTP_L2CAP_COC_MTU_DEFAULT)

bool **auto_discover_ots**

Auto discover OTS service when connected (Client only, default: true)

struct **esp_ble_otp_object_info_t**

OTP Object Information.

Public Members

esp_ble_ots_id_t **object_id**

Object ID

uint8_t **object_name**[BLE_OTS_ATT_VAL_LEN]

Object name

uint16_t **object_type**

Object type

esp_ble_ots_size_t **object_size**

Object size

esp_ble_ots_prop_t **object_prop**

Object properties

esp_ble_ots_utc_t **first_created**

First created time (if available)

esp_ble_ots_utc_t **last_modified**

Last modified time (if available)

struct **esp_ble_otp_oacp_rsp_t**

OTP OACP Response Data.

Note: rsp_code uses BLE_OTS_OACP_RSP_* (e.g. Success, Invalid Object for object not found/deleted).

Public Members

uint8_t **req_op_code**

Request op code

uint8_t **rsp_code**

Response code (BLE_OTS_OACP_RSP_*)

uint8_t **rsp_parameter**[20]

Response parameter

struct **esp_ble_otp_olcp_rsp_t**

OTP OLCP Response Data.

Public Members

uint8_t **req_op_code**

Request op code

uint8_t **rsp_code**

Response code

uint8_t **rsp_parameter**[6]

Response parameter

uint32_t **num_objects**

Number of objects (for Request Number of Objects)

struct **esp_ble_otp_transfer_info_t**

OTP Transfer Channel Information.

Public Members

esp_ble_conn_l2cap_coc_chan_t **channel**
L2CAP CoC channel handle

uint16_t **conn_handle**
BLE connection handle

uint16_t **psm**
PSM

uint16_t **mtu**
MTU

esp_ble_ots_id_t **object_id**
Object ID being transferred

bool **is_read**
True if reading from server, false if writing to server

Macros

BLE_OTP_PSM_DEFAULT
Default PSM for Object Transfer Channel (Bluetooth SIG assigned)

BLE_OTP_L2CAP_COC_MTU_DEFAULT
Default L2CAP CoC MTU for object transfer

BLE_OTP_L2CAP_COC_MTU_MIN
Minimum L2CAP CoC MTU

BLE_OTP_EVENT_DATA_MAX_LEN
Max length of data copy in TRANSFER_DATA_RECEIVED event (avoids stack overflow in handler task)

BLE_OTP_OBJECT_SIZE_UNKNOWN
Object size unknown until metadata is received

BLE_OTP_EVENT_OTS_DISCOVERED
OTS service discovered

BLE_OTP_EVENT_OTS_DISCOVERY_FAILED
OTS service discovery failed

BLE_OTP_EVENT_OBJECT_SELECTED
Object selected (via OLCP)

BLE_OTP_EVENT_OBJECT_CHANGED
Object changed notification received

BLE_OTP_EVENT_OACP_STARTED
OACP procedure started

BLE_OTP_EVENT_OACP_ABORTED

OACP procedure aborted

BLE_OTP_EVENT_OACP_RESPONSE

OACP response received (rsp_code e.g. Success, Invalid Object / object not found, etc.)

BLE_OTP_EVENT_OLCP_RESPONSE

OLCP response received

BLE_OTP_EVENT_TRANSFER_CHANNEL_CONNECTED

Object Transfer Channel (L2CAP CoC) connected

BLE_OTP_EVENT_TRANSFER_CHANNEL_DISCONNECTED

Object Transfer Channel (L2CAP CoC) disconnected

BLE_OTP_EVENT_TRANSFER_DATA_RECEIVED

Object transfer data received

BLE_OTP_EVENT_TRANSFER_DATA_SENT

Object transfer data sent

BLE_OTP_EVENT_TRANSFER_EOF

Object content fully received/sent

BLE_OTP_EVENT_TRANSFER_COMPLETE

Read/Write completed after EOF and success

BLE_OTP_EVENT_TRANSFER_ERROR

Object transfer error

Enumerations

enum **esp_ble_otp_role_t**

OTP Role.

Values:

enumerator **BLE_OTP_ROLE_CLIENT**

OTP Client role

enumerator **BLE_OTP_ROLE_SERVER**

OTP Server role

enum **esp_ble_otp_write_mode_t**

OTP API rules (summary)

- OTS must be discovered before any OACP/OLCP.
- Feature bits gate OACP/OLCP operations.
- Metadata is set valid by esp_ble_otp_client_read_object_info(); call it after OLCP selection (e.g. select_first) before Read/Write/Delete when needed.

- Read/Write requires: OTS discovered + object selected + metadata valid.
- Execute requires selected object + metadata valid + execute_prop enabled. It is also used to commit write after EOF.
- Abort is reported only after OACP Abort response.

Error code rules:

- ESP_ERR_INVALID_STATE: state prerequisites not satisfied.
- ESP_ERR_NOT_SUPPORTED: feature bit disabled or operation unsupported.
- ESP_ERR_INVALID_ARG: invalid parameters.
- ESP_FAIL: unexpected stack or internal error.

Write Object Mode

Values:

enumerator **BLE_OTP_WRITE_MODE_OVERWRITE**

Overwrite within current size

enumerator **BLE_OTP_WRITE_MODE_TRUNCATE**

Truncate object after write

enumerator **BLE_OTP_WRITE_MODE_APPEND**

Append beyond allocated size

enumerator **BLE_OTP_WRITE_MODE_PATCH**

Patch within current size

3.5 BLE HCI Components

The BLE HCI component is used to directly operate the BLE Controller via the VHCI interface to achieve functionalities like broadcasting and scanning. Compared to initiating broadcasting and scanning through the Nimble or Bluedroid protocol stacks, using this component offers the following advantages:

- Lower memory usage
- Smaller firmware size
- Faster initialization process

3.5.1 How to Use BLE HCI

For Broadcasting Applications: 1.Initialize BLE HCI: Use `ble_hci_init()` function to initialize. 2.Set Local Random Address (Optional): If you need to use a random address for broadcasting, set it using the `ble_hci_set_random_address()` function. 3.Configure Broadcast Parameters: Use `ble_hci_set_adv_param()` function to configure broadcast parameters. 4.Set Broadcast Data: Use `ble_hci_set_adv_data()` function to specify the data to be broadcast. 5.Start Broadcasting: Use `ble_hci_set_adv_enable()` function.

For Scanning Applications: 1.Initialize BLE HCI: Use `ble_hci_init()` function to initialize. 2.Configure Scanning Parameters: Use `ble_hci_set_scan_param()` function to configure scanning parameters. 3.Enable Meta Event: Use `ble_hci_enable_meta_event()` function to enable interrupt events. 4.Register Scanning Event Function: Use `ble_hci_set_register_scan_callback()` function to register the interrupt event. 5.Start Scanning: Use the `ble_hci_set_scan_enable()` function.

3.5.2 API Reference

Header File

- components/bluetooth/ble_hci/include/ble_hci.h

Functions

esp_err_t **ble_hci_init** (void)

BLE HCI initialization.

Returns esp_err_t
- ESP_OK: succeed
- others: fail

esp_err_t **ble_hci_deinit** (void)

BLE HCI de-initialization.

Returns esp_err_t
- ESP_OK: succeed
- others: fail

esp_err_t **ble_hci_reset** (void)

BLE HCI reset controller.

Returns esp_err_t
- ESP_OK: succeed
- others: fail

esp_err_t **ble_hci_enable_meta_event** (void)

Enable BLE HCI meta event.

Returns esp_err_t

esp_err_t **ble_hci_set_adv_param** (*ble_hci_adv_param_t* *param)

Set BLE HCI advertising parameters.

Parameters **param** –: advertising parameters
Returns esp_err_t
- ESP_OK: succeed
- others: fail

esp_err_t **ble_hci_set_adv_data** (uint8_t len, uint8_t *data)

Set BLE HCI advertising data.

Parameters
- **len** –: advertising data length
- **data** –: advertising data

Returns esp_err_t
- ESP_OK: succeed
- others: fail

esp_err_t **ble_hci_set_adv_enable** (bool enable)

Set BLE HCI advertising enable.

Parameters **enable** –: true for enable advertising
Returns esp_err_t
- ESP_OK: succeed
- others: fail

esp_err_t **ble_hci_set_scan_param** (*ble_hci_scan_param_t* *param)

Set BLE HCI scan parameters.

Parameters **param** –: scan parameters

Returns esp_err_t

- ESP_OK: succeed
- others: fail

esp_err_t **ble_hci_set_scan_enable** (bool enable, bool filter_duplicates)

Set BLE HCI scan enable.

Parameters

- **enable** –: enable or disable scan
- **filter_duplicates** –: filter duplicates or not

Returns esp_err_t

- ESP_OK: succeed
- others: fail

esp_err_t **ble_hci_set_register_scan_callback** (*ble_hci_scan_cb_t* cb)

Set BLE HCI scan callback.

Parameters **cb** –: scan callback function pointer

Returns esp_err_t

- ESP_OK: succeed
- others: fail

esp_err_t **ble_hci_add_to_accept_list** (*ble_hci_addr_t* addr, *ble_hci_addr_type_t* addr_type)

Add BLE white list.

Parameters

- **addr** –: address to be added to white list
- **addr_type** –: address type to be added to white list

Returns esp_err_t

- ESP_OK: succeed
- others: fail

esp_err_t **ble_hci_clear_accept_list** (void)

Clear BLE white list.

Returns esp_err_t

- ESP_OK: succeed
- others: fail

esp_err_t **ble_hci_set_random_address** (*ble_hci_addr_t* addr)

Set BLE owner address.

Parameters **addr** –: owner address

Returns

- ESP_OK: succeed
- others: fail

Structures

struct **ble_hci_scan_result_t**

BLE scan result struct.

Public Members

ble_hci_search_evt_t **search_evt**

Search event type

ble_hci_dev_type_t **dev_type**

Device type

ble_hci_addr_t **bda**

Bluetooth device address which has been searched

ble_hci_addr_type_t **ble_addr_type**

Ble device address type

uint8_t **ble_adv**[ESP_BLE_ADV_DATA_LEN_MAX + ESP_BLE_SCAN_RSP_DATA_LEN_MAX]

Received EIR

uint8_t **adv_data_len**

Adv data length

uint8_t **scan_rsp_len**

Scan response length

int **rssi**

Searched device’s RSSI

struct **ble_hci_adv_param_t**

Ble adv parameters.

Public Members

uint16_t **adv_int_min**

Time = N × 0.625 ms Range: 0x0020 to 0x4000

uint16_t **adv_int_max**

Time = N × 0.625 ms Range: 0x0020 to 0x4000

ble_hci_adv_type_t **adv_type**

Advertising Type

ble_hci_addr_type_t **own_addr_type**

Own Address Type

ble_hci_addr_t **peer_addr**

Peer device bluetooth device address

ble_hci_addr_type_t **peer_addr_type**

Peer device bluetooth device address type, only support public address type and random address type

ble_hci_adv_channel_t **channel_map**

Advertising channel map

ble_hci_adv_filter_t **adv_filter_policy**

Advertising Filter Policy:

struct **ble_hci_scan_param_t**

Ble sccan parameters.

Public Members

ble_hci_scan_type_t **scan_type**

Scan Type

uint16_t **scan_interval**

Time = N × 0.625 ms Range: 0x0004 to 0x4000

uint16_t **scan_window**

Time = N × 0.625 ms Range: 0x0004 to 0x4000

ble_hci_addr_type_t **own_addr_type**

Own Address Type

ble_hci_adv_filter_t **filter_policy**

Scanning Filter Policy

Macros

ESP_BLE_ADV_DATA_LEN_MAX

Advertising data maximum length.

ESP_BLE_SCAN_RSP_DATA_LEN_MAX

Scan response data maximum length.

BLE_HCI_ADDR_LEN

BLE Address Length.

Type Definitions

typedef uint8_t **ble_hci_addr_t**[BLE_HCI_ADDR_LEN]

Bluetooth device address.

typedef void (*__ble_hci_scan_cb_t__)(*ble_hci_scan_result_t* *scan_result, uint16_t result_len)

BLE HCI scan callback function type.

Param scan_result : ble advertisement scan result
Param result_len : length of scan result

Enumerations

enum **ble_hci_search_evt_t**

Sub Event of BLE_HCI_BLE_SCAN_RESULT_EVT.

Values:

enumerator **BLE_HCI_SEARCH_INQ_RES_EVT**

Inquiry result for a peer device.

enumerator **BLE_HCI_SEARCH_INQ_CMPL_EVT**

Inquiry complete.

enumerator **BLE_HCI_SEARCH_DISC_RES_EVT**

Discovery result for a peer device.

enumerator **BLE_HCI_SEARCH_DISC_BLE_RES_EVT**

Discovery result for BLE GATT based service on a peer device.

enumerator **BLE_HCI_SEARCH_DISC_CMPL_EVT**

Discovery complete.

enumerator **BLE_HCI_SEARCH_DI_DISC_CMPL_EVT**

Discovery complete.

enumerator **BLE_HCI_SEARCH_SEARCH_CANCEL_CMPL_EVT**

Search cancelled

enumerator **BLE_HCI_SEARCH_INQ_DISCARD_NUM_EVT**

The number of pkt discarded by flow control

enum **ble_hci_addr_type_t**

BLE address type.

Values:

enumerator **BLE_ADDR_TYPE_PUBLIC**

Public Device Address

enumerator **BLE_ADDR_TYPE_RANDOM**

Random Device Address.

enumerator **BLE_ADDR_TYPE_RPA_PUBLIC**

Resolvable Private Address (RPA) with public identity address

enumerator **BLE_ADDR_TYPE_RPA_RANDOM**

Resolvable Private Address (RPA) with random identity address.

enum **ble_hci_dev_type_t**

Bluetooth device type.

Values:

enumerator **BLE_HCI_DEVICE_TYPE_BREDR**

enumerator **BLE_HCI_DEVICE_TYPE_BLE**

enumerator **BLE_HCI_DEVICE_TYPE_DUMO**

enum **ble_hci_adv_type_t**

BLE advertising type.

Values:

enumerator **ADV_TYPE_IND**

enumerator **ADV_TYPE_DIRECT_IND_HIGH**

enumerator **ADV_TYPE_SCAN_IND**

enumerator **ADV_TYPE_NONCONN_IND**

enumerator **ADV_TYPE_DIRECT_IND_LOW**

enum **ble_hci_adv_channel_t**

Advertising channel mask.

Values:

enumerator **ADV_CHNL_37**

enumerator **ADV_CHNL_38**

enumerator **ADV_CHNL_39**

enumerator **ADV_CHNL_ALL**

enum **ble_hci_scan_type_t**

Ble scan type.

Values:

enumerator **BLE_SCAN_TYPE_PASSIVE**

Passive scan

enumerator **BLE_SCAN_TYPE_ACTIVE**

Active scan

enum **ble_hci_adv_filter_t**

Ble adv filteer type.

Values:

enumerator **ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY**

Allow both scan and connection requests from anyone.

enumerator **ADV_FILTER_ALLOW_SCAN_WLST_CON_ANY**

Allow both scan req from White List devices only and connection req from anyone.

enumerator **ADV_FILTER_ALLOW_SCAN_ANY_CON_WLST**

Allow both scan req from anyone and connection req from White List devices only.

enumerator **ADV_FILTER_ALLOW_SCAN_WLST_CON_WLST**

Allow scan and connection requests from White List devices only.

3.6 BTHome Component

The BTHome component implements the BTHome V2 protocol, supporting sensor data reporting, binary sensor data reporting, and event data reporting. This component supports both encrypted and non-encrypted modes, and can be seamlessly integrated with smart home platforms like Home Assistant.

3.6.1 BTHome Usage

1. Initialize BTHome:
   - Use *bthome_create()* to create a BTHome instance
   - Use *bthome_register_callbacks()* to register callback functions
   - Use *bthome_set_encrypt_key()* to set encryption key (optional)
   - Use *bthome_set_peer_mac_addr()* to set peer MAC address
2. Configuration Storage:
   - Use `settings_store()` to store configuration
   - Use `settings_load()` to load configuration
3. Parse Advertising Data:
   - Use *bthome_parse_adv_data()* to parse advertising data
   - Use *bthome_free_reports()* to free report data

3.6.2 Example Code

```
// Create BTHome instance
bthome_handle_t bthome_recv;
ESP_ERROR_CHECK(bthome_create(&bthome_recv));

// Register callback functions
bthome_callbacks_t callbacks = {
    .store = settings_store,
    .load = settings_load,
};
ESP_ERROR_CHECK(bthome_register_callbacks(bthome_recv, &callbacks));

// Set encryption key
static const uint8_t encrypt_key[] = {0x23, 0x1d, 0x39, 0xc1, 0xd7, 0xcc, 0x1a,␣
↪0xb1, 0xae, 0xe2, 0x24, 0xcd, 0x09, 0x6d, 0xb9, 0x32};
ESP_ERROR_CHECK(bthome_set_encrypt_key(bthome_recv, encrypt_key));

// Set peer MAC address
```

(continues on next page)

(continued from previous page)

```
static const uint8_t peer_mac[] = {0x54, 0x48, 0xE6, 0x8F, 0x80, 0xA5};
ESP_ERROR_CHECK(bthome_set_peer_mac_addr(bthome_recv, peer_mac));

// Parse advertising data
bthome_reports_t *reports = bthome_parse_adv_data(bthome_recv, result.ble_adv,␣
↪result.adv_data_len);
if (reports != NULL) {
    // Process report data
    for (int i = 0; i < reports->num_reports; i++) {
        // Process each report
    }
    bthome_free_reports(reports);
}
```

3.6.3 API Reference

Header File

- components/bluetooth/ble_adv/bthome/include/bthome_v2.h

Functions

esp_err_t **bthome_create** (*bthome_handle_t* *handle)

Create a BTHome object.

Parameters **handle** –pointer to the bthome handle
Returns esp_err_t ESP_OK: success others: fail

esp_err_t **bthome_delete** (*bthome_handle_t* handle)

Delete a BTHome object.

Parameters **handle** –pointer to the bthome handle
Returns esp_err_t ESP_OK: success others: fail

esp_err_t **bthome_set_encrypt_key** (*bthome_handle_t* handle, const uint8_t *key)

set the encryption key used for encryption and decryption

Parameters
- **handle** –bthome handle
- **key** –pointer to the key

Returns esp_err_t ESP_OK: success others: fail

esp_err_t **bthome_set_local_mac_addr** (*bthome_handle_t* handle, uint8_t *mac)

set the local mac address used for encryption

Parameters
- **handle** –bthome handle
- **mac** –pointer to the local ble mac address

Returns esp_err_t ESP_OK: success others: fail

esp_err_t **bthome_set_peer_mac_addr** (*bthome_handle_t* handle, const uint8_t *mac)

set the peer mac address used for decryption

Parameters
- **handle** –bthome handle
- **mac** –pointer to the peer ble mac address

Returns esp_err_t ESP_OK: success others: fail

esp_err_t **bthome_register_callbacks** (*bthome_handle_t* handle, *bthome_callbacks_t* *callbacks)

register the callback function

Parameters
- **handle** –pointer to the bthome handle
- **callbacks** –pointer to the callback structure

Returns esp_err_t ESP_OK: success others: fail

esp_err_t **bthome_load_params** (*bthome_handle_t* handle)

load the parameters from the storage

Parameters **handle** –pointer to the bthome handle
Returns esp_err_t ESP_OK: success others: fail

void **bthome_free_reports** (*bthome_reports_t* *reports)

free the reports structure

Parameters **reports** –pointer to the reports need to be freed

uint8_t **bthome_payload_add_sensor_data** (uint8_t *buffer, uint8_t offset, *bthome_sensor_id_t* obj_id, uint8_t *data, uint8_t data_len)

Add sensor data to the payload.

Parameters
- **buffer** –pointer to the buffer
- **offset** –current offset of the buffer
- **obj_id** –sensor ID
- **data** –pointer to the data
- **data_len** –length of the data

Returns length of the data added to the buffer

uint8_t **bthome_payload_adv_add_bin_sensor_data** (uint8_t *buffer, uint8_t offset, *bthome_bin_sensor_id_t* obj_id, uint8_t data)

Add binary sensor data to the payload.

Parameters
- **buffer** –pointer to the buffer
- **offset** –current offset of the buffer
- **obj_id** –binary sensor ID
- **data** –data

Returns length of the data added to the buffer

uint8_t **bthome_payload_adv_add_evt_data** (uint8_t *buffer, uint8_t offset, *bthome_event_id_t* obj_id, uint8_t *evt, uint8_t evt_size)

Add event data to the payload.

Parameters
- **buffer** –pointer to the buffer
- **offset** –current offset of the buffer
- **obj_id** –event ID
- **evt_size** –event size
- **evt** –event value

Returns length of the data added to the buffer

bthome_reports_t ***bthome_parse_adv_data** (*bthome_handle_t* handle, uint8_t *adv, uint8_t len)

Parse the advertisement data.

Parameters
- **handle** –bthome handle
- **adv** –pointer to the advertisement data
- **len** –length of the advertisement data

Returns *bthome_reports_t* * pointer to the reports structure NULL if the parsing fails

uint8_t **bthome_make_adv_data** (*bthome_handle_t* handle, uint8_t *buffer, uint8_t *name, uint8_t name_len, *bthome_device_info_t* info, uint8_t *payload, uint8_t payload_len)

Create a BTHome advertisement data.

Parameters

- **handle** –bthome handle
- **buffer** –advertisement data buffer
- **name** –advertisement name
- **name_len** –length of the name
- **info** –bthome device info
- **payload** –pointer to the payload
- **payload_len** –length of the payload

Returns length of the advertisement data

Unions

union **bthome_device_info_t**

#include <bthome_v2.h> BTHome device info.

Public Members

uint8_t **encryption_flag**

bit 0: Encryption flag

uint8_t **reserved1**

bit 1: Reserved for future use

uint8_t **trigger_based_flag**

bit 2: Trigger based device flag

uint8_t **reserved2**

bit 3-4: Reserved for future use

uint8_t **bthome_version**

bit 5-7: BTHome Version

struct *bthome_device_info_t*::[anonymous] **bit**

Bit field representation of device info

uint8_t **all**

Byte representation of device info

Structures

struct **bthome_report_t**

BTHome report structure.

Public Members

uint8_t **id**

Object ID: One of the BTHome Sensor ID, Binary Sensor ID or Event ID

uint8_t **len**

Length of the data

uint8_t ***data**

Pointer to the data

struct **bthome_reports_t**

BTHome reports structure.

Public Members

uint8_t **num_reports**

Number of reports in the array

bthome_report_t **report**[BTHOME_REPORTS_MAX]

Array of reports

struct **bthome_callbacks_t**

BTHome callback functions.

Public Members

bthome_store_func_t **store**

Function pointer to store data

bthome_load_func_t **load**

Function pointer to load data

Macros

BTHOME_REPORTS_MAX

Type Definitions

typedef struct bthome_t ***bthome_handle_t**

bthome handle type

typedef void (***bthome_store_func_t**)(*bthome_handle_t* handle, const char *key, const uint8_t *data, uint8_t len)

BTHome store function.

typedef void (*`bthome_load_func_t`)(*bthome_handle_t* handle, const char *key, uint8_t *data, uint8_t len)

BTHome load function.

Enumerations

enum `bthome_sensor_id_t`

BTHome sensor ID.

Values:

enumerator `BTHOME_SENSOR_ID_PACKET`

enumerator `BTHOME_SENSOR_ID_BATTERY`

enumerator `BTHOME_SENSOR_ID_TEMPERATURE_PRECISE`

enumerator `BTHOME_SENSOR_ID_HUMIDITY_PRECISE`

enumerator `BTHOME_SENSOR_ID_PRESSURE`

enumerator `BTHOME_SENSOR_ID_ILLUMINANCE`

enumerator `BTHOME_SENSOR_ID_MASS`

enumerator `BTHOME_SENSOR_ID_MASSLB`

enumerator `BTHOME_SENSOR_ID_DEWPOINT`

enumerator `BTHOME_SENSOR_ID_COUNT`

enumerator `BTHOME_SENSOR_ID_ENERGY`

enumerator `BTHOME_SENSOR_ID_POWER`

enumerator `BTHOME_SENSOR_ID_VOLTAGE`

enumerator `BTHOME_SENSOR_ID_PM25`

enumerator `BTHOME_SENSOR_ID_PM10`

enumerator `BTHOME_SENSOR_ID_CO2`

enumerator `BTHOME_SENSOR_ID_TVOC`

enumerator `BTHOME_SENSOR_ID_MOISTURE_PRECISE`

enumerator **BTHOME_SENSOR_ID_HUMIDITY**

enumerator **BTHOME_SENSOR_ID_MOISTURE**

enumerator **BTHOME_SENSOR_ID_COUNT2**

enumerator **BTHOME_SENSOR_ID_COUNT4**

enumerator **BTHOME_SENSOR_ID_ROTATION**

enumerator **BTHOME_SENSOR_ID_DISTANCE**

enumerator **BTHOME_SENSOR_ID_DISTANCEM**

enumerator **BTHOME_SENSOR_ID_DURATION**

enumerator **BTHOME_SENSOR_ID_CURRENT**

enumerator **BTHOME_SENSOR_ID_SPD**

enumerator **BTHOME_SENSOR_ID_TEMPERATURE**

enumerator **BTHOME_SENSOR_ID_UV**

enumerator **BTHOME_SENSOR_ID_VOLUME1**

enumerator **BTHOME_SENSOR_ID_VOLUME2**

enumerator **BTHOME_SENSOR_ID_VOLUMEFR**

enumerator **BTHOME_SENSOR_ID_VOLTAGE1**

enumerator **BTHOME_SENSOR_ID_GAS**

enumerator **BTHOME_SENSOR_ID_GAS4**

enumerator **BTHOME_SENSOR_ID_ENERGY4**

enumerator **BTHOME_SENSOR_ID_VOLUME**

enumerator **BTHOME_SENSOR_ID_WATER**

enumerator **BTHOME_SENSOR_ID_TIMESTAMP**

enumerator **BTHOME_SENSOR_ID_ACCELERATION**

enumerator **BTHOME_SENSOR_ID_GYROSCOPE**

enumerator **BTHOME_SENSOR_ID_TEXT**

enumerator **BTHOME_SENSOR_ID_RAW**

enumerator **BTHOME_SENSOR_ID_VOLUME_STORAGE**

enum **bthome_bin_sensor_id_t**

BTHome binary sensor ID.

Values:

enumerator **BTHOME_BIN_SENSOR_ID_GENERIC**

enumerator **BTHOME_BIN_SENSOR_ID_POWER**

enumerator **BTHOME_BIN_SENSOR_ID_OPENING**

enumerator **BTHOME_BIN_SENSOR_ID_BATTERY**

enumerator **BTHOME_BIN_SENSOR_ID_BATTERY_CHARGING**

enumerator **BTHOME_BIN_SENSOR_ID_CARBON_MONOXIDE**

enumerator **BTHOME_BIN_SENSOR_ID_COLD**

enumerator **BTHOME_BIN_SENSOR_ID_CONNECTIVITY**

enumerator **BTHOME_BIN_SENSOR_ID_DOOR**

enumerator **BTHOME_BIN_SENSOR_ID_GARAGE_DOOR**

enumerator **BTHOME_BIN_SENSOR_ID_GAS**

enumerator **BTHOME_BIN_SENSOR_ID_HEAT**

enumerator **BTHOME_BIN_SENSOR_ID_LIGHT**

enumerator **BTHOME_BIN_SENSOR_ID_LOCK**

enumerator **BTHOME_BIN_SENSOR_ID_MOISTURE**

enumerator **BTHOME_BIN_SENSOR_ID_MOTION**

enumerator **BTHOME_BIN_SENSOR_ID_MOVING**

enumerator **BTHOME_BIN_SENSOR_ID_OCCUPANCY**

enumerator **BTHOME_BIN_SENSOR_ID_PLUG**

enumerator **BTHOME_BIN_SENSOR_ID_PRESENCE**

enumerator **BTHOME_BIN_SENSOR_ID_PROBLEM**

enumerator **BTHOME_BIN_SENSOR_ID_RUNNING**

enumerator **BTHOME_BIN_SENSOR_ID_SAFETY**

enumerator **BTHOME_BIN_SENSOR_ID_SMOKE**

enumerator **BTHOME_BIN_SENSOR_ID_SOUND**

enumerator **BTHOME_BIN_SENSOR_ID_TAMPER**

enumerator **BTHOME_BIN_SENSOR_ID_VIBRATION**

enumerator **BTHOME_BIN_SENSOR_ID_WINDOW**

enum **bthome_event_id_t**

BTHome event ID.

Values:

enumerator **BTHOME_EVENT_ID_BUTTON**

enumerator **BTHOME_EVENT_ID_DIMMER**

Chapter 4

Display

4.1 LCD Screen

4.1.1 LCD Introduction

Commonly LCD is a TFT-LCD (Thin Film Transistor Liquid Crystal Display). It is a common digital display technology used for displaying images and text. LCDs employ liquid crystal materials and polarization technology. When the liquid crystal molecules are influenced by an electric field, they change the polarization direction of light, thereby controlling the intensity of light and displaying images or text on the screen.

LCD has many advantages, such as low power consumption, long lifespan, high reliability, high clarity, compact size, high color reproduction, and strong anti-glare capability. Therefore, it has been widely used in various electronic devices, such as home appliances, portable devices, wearable devices, etc. At the same time, LCD technology continues to advance and improve, including different panel types such as IPS, VA, TN, and new LED backlight technologies, all of which further enhance the performance and user experience of LCD.

This guide contains the following sections:

- *Structure*: The main structure of the LCD module, primarily composed of a panel, backlight source, driver IC, and FPC.
- *Shape*: Common forms of the LCD module, including rectangular screens and circular screens.
- *Driver Interface*: The driver interface of the LCD module, including SPI, QSPI, I80, RGB, and MIPI-DSI.
- *Typical Connection Methods*: Typical ways to connect the LCD module, including the general pins of the LCD and various types of interface pins.
- *Frame Rate*: The frame rate of LCD applications, including rendering frame rate, interface frame rate, and screen refresh rate.

Terminology

Please refer to the *LCD Terms Table* 。

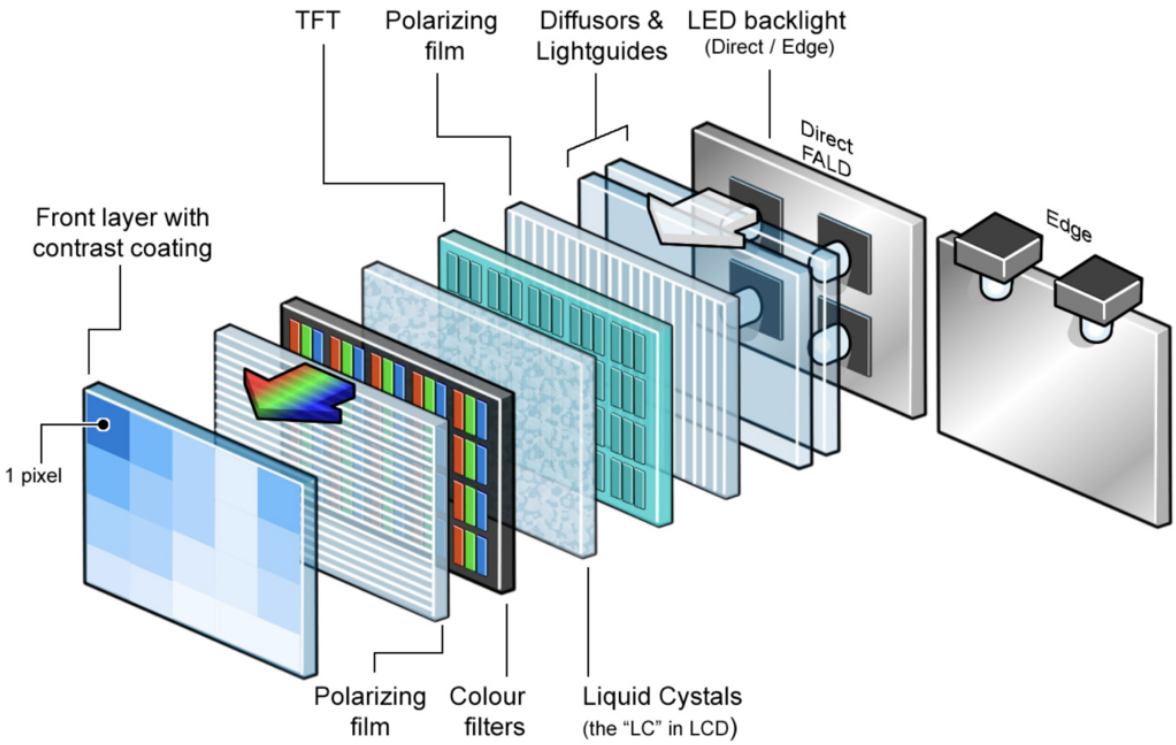


Fig. 1: TFT-LCD Hardware Diagram

Structure

To ensure the stable operation and convenient development of LCD, manufacturers typically encapsulate the LCD into an **integrated module** for user use. It mainly consists of the following four parts:

- **Panel**: The panel determines the color, viewing angle, and resolution of the LCD module. The price trend of the panel directly affects the module' s price, while the quality and technological aspects of the panel are important in shaping the module' s overall performance. Common panel types include IPS, VA, TN, etc.
- **Backlight Source**: Since liquid crystal molecules do not emit light on their own, a dedicated backlight source is required to illuminate the LCD screen and allow the liquid crystal molecules to produce different colors through their deflection. The role of the backlight source is to provide light energy, and its brightness can generally be controlled using PWM (Pulse Width Modulation).
- **Driver IC**: The driver IC communicates externally through specific interfaces, controlling the output voltage to twist the liquid crystals and induce changes in color levels and brightness. It typically consists of two parts: the control circuit, responsible for receiving signals from the main control chip and converting/processing image signals, and the driver circuit, responsible for outputting image signals and displaying them on the panel.
- **FPC (Flexible Printed Circuit)**: FPC serves as the external interface for the LCD module, connecting the driver IC, external drive circuit, and main control chip. Due to its outstanding flexibility and reliability, FPC addresses issues related to contact in traditional rigid circuit boards and provides improved resistance to vibration, thereby enhancing the stability and lifespan of the module.

Typically, the selection of an LCD module is primarily based on its **panel** and **driver IC**. For instance, considerations include the type and resolution of the panel, as well as the interface types and color formats supported by the driver IC. The driver IC, with a small footprint, is usually affixed at the connection point between the FPC and the panel, as illustrated.

Shape

When it comes to the physical shape of LCD panels, most are either rectangular or circular. The most commonly encountered type in daily life is the rectangular screen, while circular screens are often found in smaller-sized displays.

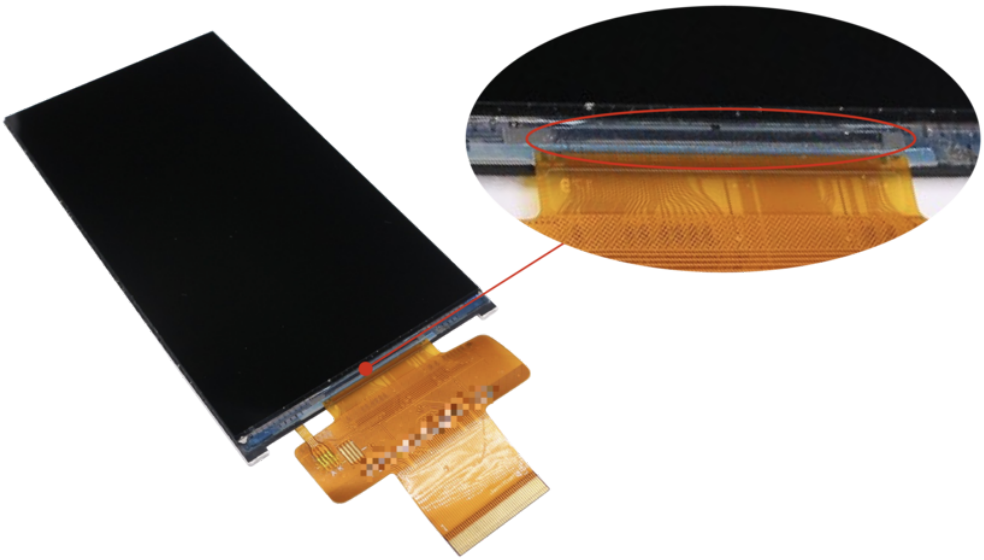


Fig. 2: LCD Module’s Driver IC

Fig. 3: Rectangular LCD Screen

Fig. 4: Circular LCD Screen

Characteristics and application scenarios for these shapes are as follows:

Type	Characteristics	Application Scenarios
Rectangular Screen	Large area, excellent display, capable of presenting more information, versatile applications	Mobile phones, tablets, control panels
Circular Screen	Stylish, lightweight, occupies less space, effectively utilizes device area	Smart wearables, electric vehicle dashboards, car display panels, smart home appliances, handheld smart devices

The **size** of an LCD panel is typically measured by the diagonal length, expressed in inches or centimeters, such as the commonly mentioned 1.28-inch and 3.5-inch screens. Besides the physical size, developers often pay more attention to the screen' s **resolution**. Resolution refers to the number of pixels the panel can display, representing the image precision: the more pixels it can display, the finer the picture, and the more information can be shown in the same screen area. Higher resolution also imposes greater performance demands on the main control chip, making it a crucial parameter.

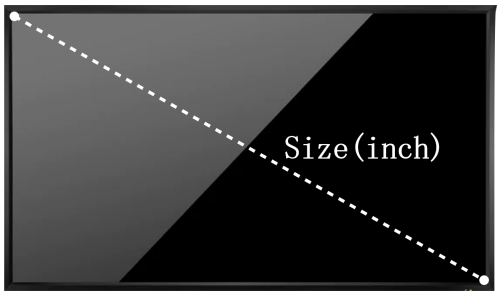


Fig. 5: Screen Size

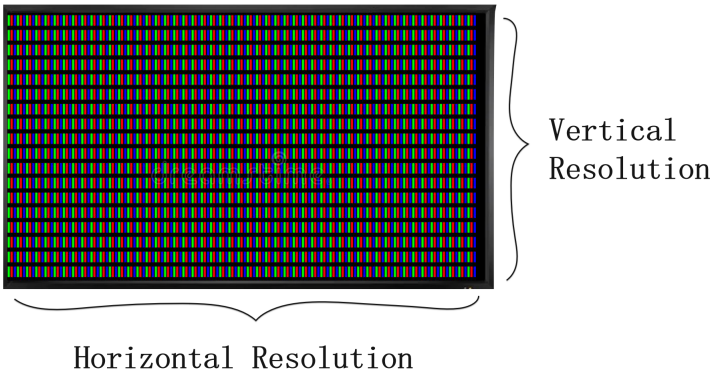


Fig. 6: Screen Resolution

The relationship between size and resolution is not a one-to-one correspondence, but there is a general proportional trend. For example, in most cases, a 2.4-inch or 2.8-inch screen commonly has a resolution of 320x240, while a 3.2-inch or 3.5-inch screen often has a resolution of 320x480. Larger screens may not necessarily have higher resolutions than smaller ones. Therefore, when selecting a screen, it is essential to determine the size and resolution based on the specific application scenarios and requirements.

Driver Interface

For developers, the focus is often on the LCD' s driver interface. Common interface types in the field of IoT include `SPI`, `QSPI`, `I80`, `RGB`, and `MIPI-DSI`. A comparison of parameters such as `IO count`, `parallel data bits`, `data transfer bandwidth`, and `Graphics RAM (GRAM) location` is presented below:

Parameter Comparison

Type	Description	IO Count	Parallel Bits	Data Bandwidth	GRAM Location
SPI	Serial interface based on the SPI bus protocol, typically using 4-wire or 3-wire modes	Minimum	1	Minimum	LCD
QSPI (Quad-SPI)	Extension of SPI interface, enables parallel transmission with 4 data lines	Fewer	4	Smaller	LCD or MCU
I80 (MCU, DBI)	Parallel interface based on the I80 bus protocol	More	8/16	Larger	LCD
RGB (DPI)	Parallel interface, usually paired with a 3-wire SPI interface	Maximum	8/16/18/24	Larger	MCU
MIPI-DSI	Serial interface using differential signal transmission, based on the high-speed, low-power, scalable serial interconnect D-PHY physical layer specification of MIPI	More	1/2/3/4	Maximum	LCD or MCU

Note:

- For the `QSPI` interface, different models of driver ICs may adopt different driving methods. For example, the *SPD2010* has built-in GRAM, and its driving method is similar to the `SPI/I80` interface, while the *ST77903* does not have built-in GRAM, and its driving method is similar to the `RGB` interface.
- For the `MIPI-DSI` interface, the use of Command mode requires the LCD to have built-in GRAM, while Video mode does not.

Summarizing as follows:

1. `SPI` The data transfer bandwidth of the `SPI` interface is small, making it suitable for low-resolution screens.
2. `QSPI` and `I80` interfaces have larger data transfer bandwidth, enabling support for higher-resolution screens. However, the `I80` interface requires the LCD to have built-in GRAM, leading to higher screen costs and making it challenging to achieve large screens.
3. The `RGB` interface is similar to the `I80` interface, but the `RGB` interface does not require LCD to have built-in GRAM, making it suitable for higher-resolution screens.
4. The `MIPI-DSI` interface is suitable for high-resolution, high-refresh-rate screens.

Interface Details The first step in driving an LCD is to determine its interface type. For most common driver ICs, such as *ST7789*, *GC9A01*, *ILI9341*, etc., they generally support multiple interfaces. However, when screen manufacturers package them into modules, they typically expose only one interface externally (RGB LCDs usually also use the SPI interface). Taking *GC9A01* as an example, its hardware block diagram is as follows:

The actual interface type of many LCD driver ICs is determined by the logic levels of their `IM[3:0]` pins. While most screens have these pin configurations fixed internally, some screens reserve these pins along with all interface pins, allowing users to configure them. Taking *ST7789* as an example, its interface type configuration is as follows:

Therefore, knowing just the model of the driver IC is not sufficient to determine the interface type of the screen. In such cases, you can consult the screen manufacturer, refer to the screen' s datasheet, or use the schematic combined with experience to make an informed judgment. Below is a pin comparison for various interfaces:

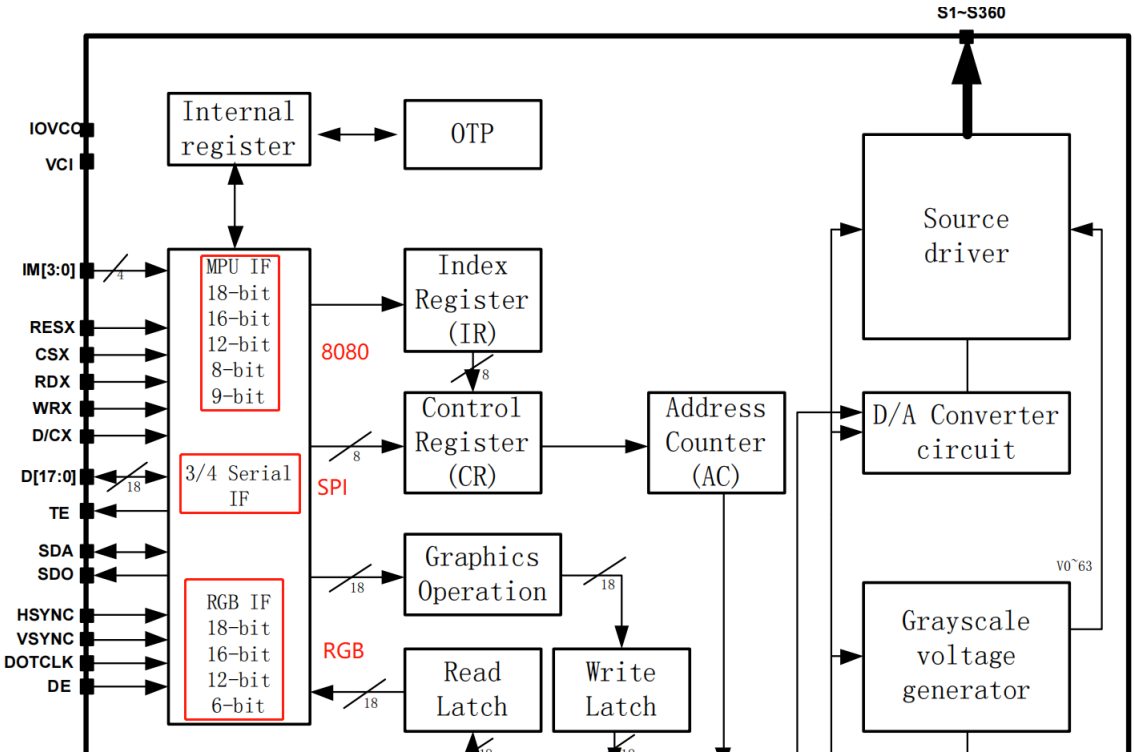


Fig. 7: Hardware Block Diagram of GC9A01

ST7789V supports 8/16/9/18 bit parallel data bus for 8080 series CPU, RGB serial interfaces. Selection of these interfaces are set by IM[3:0] pins as shown below.

IM3	IM2	IM1	IM0	Interface	Read Back Data Bus Selection
0	0	0	0	80-8bit parallel I/F	DB[7:0]
0	0	0	1	80-16bit parallel I/F	DB[15:0]
0	0	1	0	80-9bit parallel I/F	DB[8:0]
0	0	1	1	80-18bit parallel I/F	DB[17:0],
0	1	0	1	3-line 9bit serial I/F	SDA: in/out
				2 data lane serial I/F	SDA: in/out, WRX: in
0	1	1	0	4-line 8bit serial I/F	SDA: in/out
1	0	0	0	80-16bit parallel I/F Ⅱ	DB[17:10], DB[8:1]
1	0	0	1	80-8bit parallel I/F Ⅱ	DB[17:10]
1	0	1	0	80-18bit parallel I/F Ⅱ	DB[17:0],
1	0	1	1	80-9bit parallel I/F Ⅱ	DB[17:9]
1	1	0	1	3-line 9bit serial I/F Ⅱ	SDA: in/ SDO: out
1	1	1	0	4-line 8bit serial I/F Ⅱ	SDA: in/ SDO: out

Fig. 8: Interface Configuration of ST7789

Type	Pins
General LCD	RST (RESET), Backlight (LEDA, LEDK), TE (Tear Effect), Power (VCC, GND)
SPI	CS, SCK (SCL), SDA (MOSI), SDO (MISO), DC (RS)
QSPI	CS, SCK (SCL), SDA (DATA0), DATA1, DATA2, DATA3
I80	CS (CSX), RD (RDX), WR (WRX), DC (D/CX), D[15:0] (D[7:0])
RGB	CS, SCK (SCL), SDA (MOSI), HSYNC, VSYNC, PCLK, DE, D[23:0] (D[17:0]/D[7:0])

Detailed descriptions of commonly used interface types for LCDs are as follows:

- *SPI LCD Introduction*
- *RGB LCD Introduction*
- I80 LCD Introduction (To be updated)
- QSPI LCD Introduction (To be updated)

Typical Connection Methods

For the common LCD pins, the typical connection method is as follows:

- **RST (RESET)**: It is recommended to connect to a GPIO pin and according to the LCD driver IC' s datasheet, generate a reset timing sequence during power-on. In general, pull-up/pull-down resistors can also be used connected to the system power.
- **Backlight (LEDA、LEDK)**: It is recommended to connect LEDA(Anode) to the system power supply, and LEDK(Cathode) should be connected to the system power supply using switching devices. Control the on/off state through GPIO or use the LEDC peripheral to output PWM to adjust the backlight brightness.
- **TE (Tear Effect)**: Recommend connecting to GPIO and using GPIO interrupts to obtain the TE signal for achieving frame synchronization.
- **Power(VCC、GND)**: It is recommended to connect all to the corresponding system power sources and avoid leaving any pins floating.

For pins of different interface types, the MCU needs to adopt different connection methods. Below, we will introduce the typical connection methods for four interfaces: SPI, QSPI, I80, and RGB.

SPI Interface The hardware design of the LCD with the `SPI` interface can be referred to the development board ESP32-C3-LCDkit and its LCD sub-board, The typical connection diagram is as follows:

Note:

- `Interface I Mode` requires only the `SDA` data line, while `Interface II Mode` requires both `MISO & MOSI` data lines.
- In most cases, reading data from the LCD is not necessary, so the `MISO` connection may be omitted. If needed, please be aware that the maximum clock frequency for reading from SPI LCDs is often much lower than the writing frequency.
- Due to the `3-line Mode` (no D/C signal line), where each unit of data transfer (usually a byte) requires transmitting the D/C signal first (1-bit), and the current ESP' s SPI peripheral does not support direct transmission of 9-bit data, the commonly used `4-line Mode` shown in the above diagram is preferred.

QSPI Interface `QSPI` interface' s typical connection diagram is as follows:

Note:

- The `QSPI` interface connection may vary for different models of driver ICs. The above diagram is provided as an example using *ST77903*.
- When writing data, use four data lines: `SDA0` and `SDA[1:3]`. When reading data, only `SDA0` is used.

Fig. 9: Typical Connection Diagram for SPI Interface

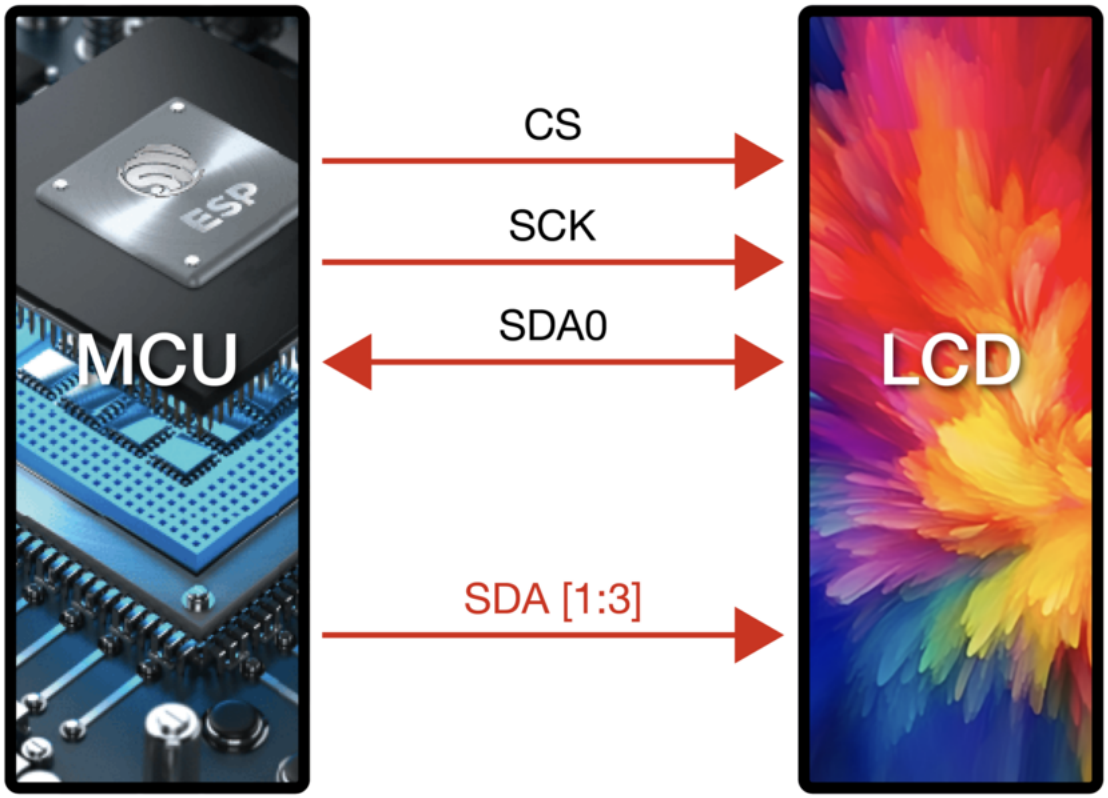


Fig. 10: Typical Connection Diagram for QSPI Interface

I80 Interface For the hardware design of the LCD with the `I80` interface, please refer to the development board ESP32-S3-LCD-EV-Board and its LCD sub-board (3.5' LCD_ZJY). The typical connection diagram is as follows:

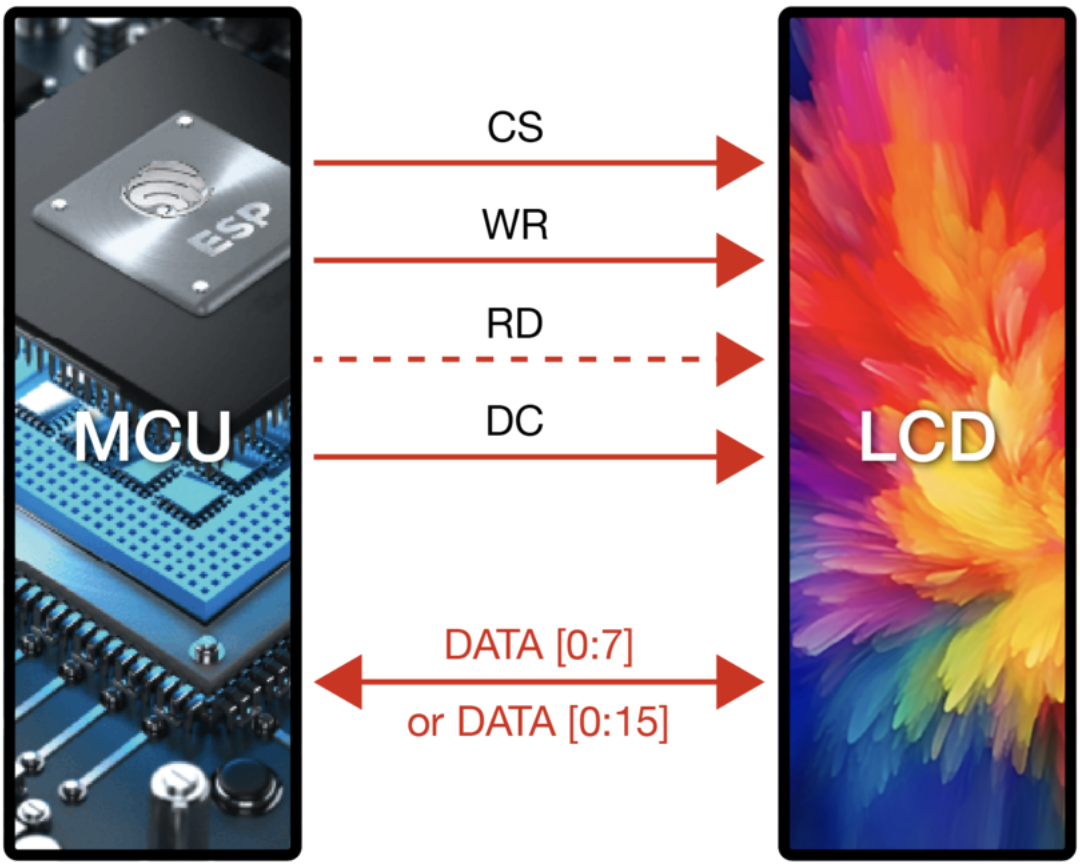


Fig. 11: Typical Connection Diagram for I80 Interface

Note:

- Dashed lines in the diagram represent optional pins.
- The I80 peripheral on ESP does not support using the `RD` signal for reading operations, so it needs to be pulled high during actual connections.

RGB Interface For the hardware design of the LCD with the `RGB` interface, please refer to the development board ESP32-S3-LCD-EV-Board and its LCD sub-board (3.95' LCD_QMZX). The typical connection diagram is as follows:

Note:

- Dashed lines in the diagram represent optional pins.
- `DE` is used for DE mode.
- `CS`, `SCK`, and `SDA` are 3-wire SPI interface pins used to send commands and parameters to configure the LCD. Some screens may not have these pins, and therefore, initialization configuration may not be necessary. Since the `3-wire SPI` interface can be used only for the initialization of the LCD and not for subsequent screen refresh, to save the number of I/O pins, `SCK` and `SDA` can be reused with any `RGB` interface pins.

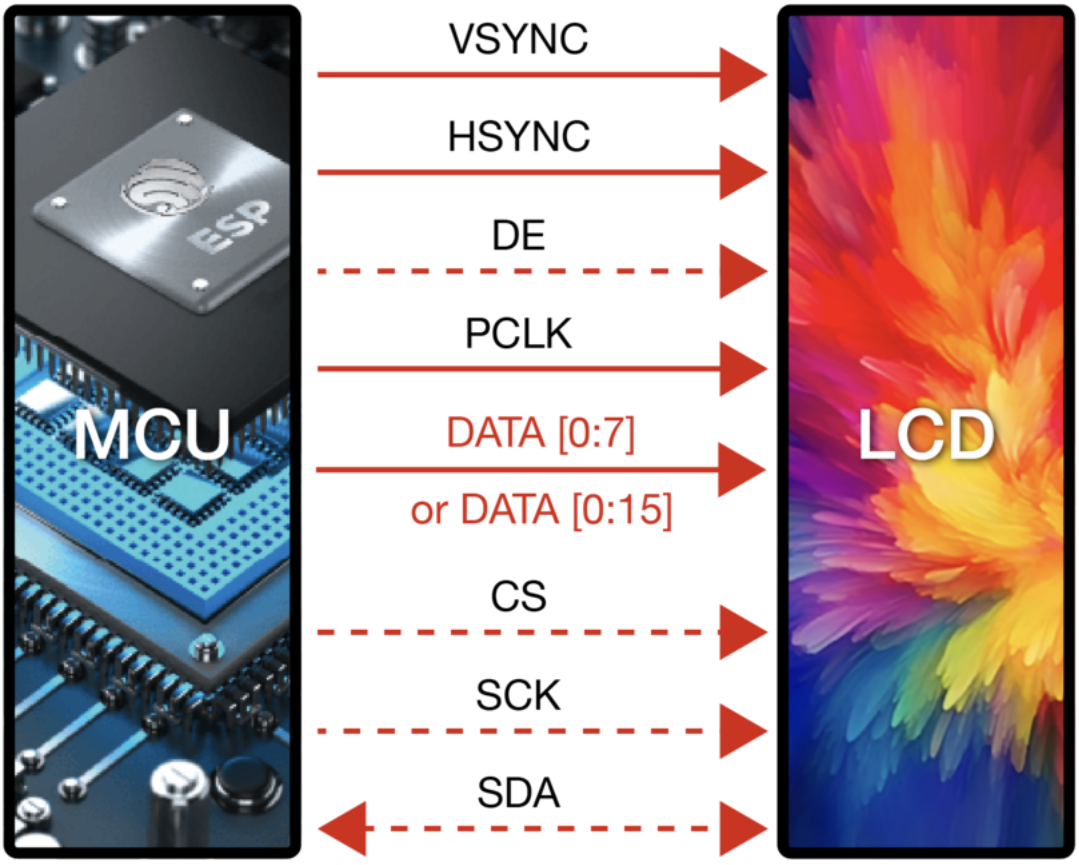


Fig. 12: Typical Connection Diagram for RGB Interface

Frame Rate

For LCD applications, animations on the screen are achieved by displaying multiple consecutive still images, known as **frames**. The **frame rate** is the speed at which new frames are displayed and is typically expressed as the number of frames that change per second, abbreviated as FPS. A higher frame rate means more frames are displayed per second, resulting in smoother and more realistic animation.

However, the display of a single frame is not completed all at once by the main controller; rather, it goes through multiple steps such as rendering, transmission, and display. Therefore, the frame rate not only depends on the performance of the main controller but also on factors such as the LCD interface type and refresh rate.

Rendering Rendering refers to the process by which the main controller calculates and generates image data. The speed of this process can be measured by the **rendering frame rate**.

The rendering frame rate depends on the performance of the main controller and is also influenced by the complexity of the animation. For example, animations with localized changes usually have a higher rendering frame rate than those with full-screen changes, and pure color fills typically have a higher rendering frame rate for layer blending. Therefore, the rendering frame rate is generally not fixed during image changes, as shown in the runtime FPS statistics of LVGL.

See the GIF for runtime FPS statistics of LVGL.

Transmission Transmission refers to the process in which the main controller transfers the rendered image data through the peripheral interface to the LCD driver IC. The speed of this process can be measured by the **interface frame rate**.

The interface frame rate depends on the LCD interface type and the data transfer bandwidth of the main controller. It is typically fixed after the initialization of the peripheral interface. It can be calculated using the following formula:

$$Interface\ Frame\ Rate = \frac{Data\ Transfer\ Bandwidth\ of\ the\ Interface}{Data\ Size\ of\ One\ Frame}$$

For SPI/I80 Interfaces:

$$Interface\ Frame\ Rate = \frac{Clock\ Frequency \times Number\ of\ Data\ Lines}{Color\ Depth \times Horizontal\ Resolution \times Vertical\ Resolution}$$

For RGB Interfaces:

$$Interface\ Frame\ Rate = \frac{Clock\ Frequency \times Number\ of\ Data\ Lines}{Color\ Depth \times Horizontal\ Period \times Vertical\ Period}$$

$$\begin{aligned} Horizontal\ Period =& Horizontal\ Pulse\ Width + Horizontal \\ & Horizontal\ Resolution + Horizontal\ F \\ Vertical\ Period =& Vertical\ Pulse\ Width + Vertical\ Back \\ & Vertical\ Resolution + Vertical\ Front \end{aligned}$$

Display Display refers to the process in which the LCD driver IC displays the received image data on the screen. The speed of this process can be measured by the **screen refresh rate**.

For LCDs with SPI/I80 interfaces, the screen refresh rate is determined by the LCD driver IC and can typically be set by sending specific commands, such as the *ST7789* command `FRCTRL2 (C6h)`. For LCDs with RGB interfaces, the screen refresh rate is determined by the main controller and is equivalent to the interface frame rate.

Note:

- If development needs to proceed without an LCD, the esp_lcd_usb_display component can be used to simulate the LCD display on a PC monitor via USB UVC, enabling application debugging. The corresponding example is available at usb_lcd_display.

4.1.2 LCD Terms Table

The section aims to introduce the meanings of LCD-related terms.

Term	Meaning
GRAM	Abbreviation for Graphic RAM, a storage area for saving graphic data.
TE	Timing signal used to indicate vertical or horizontal synchronization of screen refresh.
Porch	Blank time interval before displaying a line or frame of image data.

4.1.3 LCD Development Guide

This guide mainly contains the following sections:

- *Supported Interface Types*: Espressif' s support for different LCD interfaces in each series of chips.
- *Driver and Examples*: LCD driver and examples provided by Espressif Systems.
- *Development Framework*: Develop software and hardware framework for LCD.
- *Development Steps*: Detailed steps for developing LCD applications.
- *Common Problems*: Lists common problems in developing LCD applications.
- *Related Documents*: Links to relevant documentation are listed.

Terminology

Please refer to the *LCD Terms Table* 。

Supported Interface Types

Espressif Systems provides comprehensive support for all interface types introduced in the *LCD Overview - Driver Interface*. Specific support for each series of ESP chips within this section is as follows:

Soc	SPI(QSPI)	I80	RGB	MIPI-DSI
ESP32	Supported	Supported		
ESP32-C3	Supported			
ESP32-C6	Supported			
ESP32-S2	Supported	Supported		
ESP32-S3	Supported	Supported	Supported	
ESP32-P4	Supported	Supported	Supported	Supported

Driver and Examples

LCD Peripheral Driver is located in the *components/esp_lcd <https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/api-reference/peripherals/lcd.html>_* directory under **ESP-IDF**. Currently, it supports the `I2C`, `SPI (QSPI)`, `I80`, and `RGB` interfaces. The following table lists the LCD driver components officially ported by Espressif based on esp_lcd, and these components will continue to be updated:

接口	LCD 控制器
I2C	ssd1306, sh1107
SPI	axs15231b, st7789, nt35510, gc9b71, nv3022b, sh8601, spd2010, st77916, st77922, gc9a01, gc9d01, ili9341, ssd1681, st7796, gc9107
QSPI	axs15231b, gc9b71, sh8601, spd2010, st77903, st77916, st77922, co5300
I80	axs15231b, st7789, nt35510, ra8875, st7796
MIPI-DSI	co5300, ek79007, er88577, hx8399, ili9881c, jd9165, jd9365, st7123, st7701, st7703, st77916, st77922
3-wire SPI + RGB	st7701, st77903_rgb, st77922, gc9503, nv3052

Please note:

- The **st7789**, **nt35510**, and **ssd1306** components are maintained in the ESP-IDF. Other components can be found in the Espressif IDF Component Registry.
- Although the models of the **LCD driver IC** might be identical, **different screens** require specific configurations through initialization commands provided by their respective manufacturers. Most driver components support the customization of initialization commands during the initialization of the LCD device. If it is not supported, please refer to the method.

LCD Example is located in the examples/peripherals/lcd directory under **ESP-IDF** and the examples/display/lcd directory under **esp-iot-solution**. These serve as a reference for the usage of the LCD driver component.

Note:

- It is recommended to develop based on ESP-IDF release/v5.1 and above version branches, because lower versions may lack to support some parts of the important new features, especially for the `RGB` interface.
- For LCDs using the `3-wire SPI + RGB` interface, please refer to the example esp_lcd_st7701 - Example use.

Development Framework

Hardware Framework For SPI/I80 LCD, ESP can send **commands** to configure the LCD and transmit **regional color data** to refresh the screen through a single peripheral interface. The LCD driver IC will store the received color data in **full-screen size GRAM** and display **full-screen color data** on the panel at a fixed refresh rate. Importantly, these two processes are performed in asynchronously. The schematic diagram below illustrates the hardware driver framework of SPI/I80 LCD:

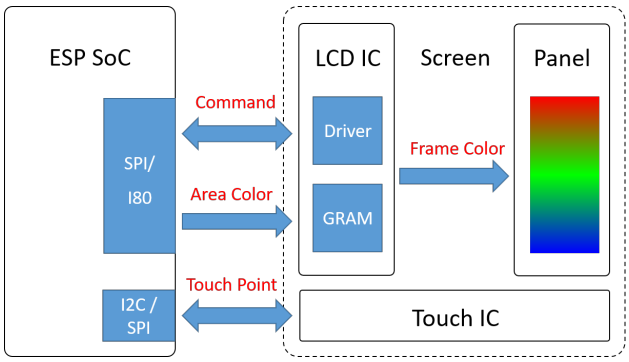


Fig. 13: Schematic diagram of hardware driver framework - SPI/I80 LCD

For most RGB LCDs, the ESP needs to use two different interfaces. On one hand, it utilizes the *3-wire SPI* interface to send commands for configuring the LCD. On the other hand, it uses the *RGB* interface to transmit **full-screen color data** for screen refresh. Since the LCD' s driver IC does not have a built-in Graphic RAM (GRAM), it directly

displays the received color data on the panel, making these two processes synchronous. The following is a schematic diagram of the hardware driving framework for RGB LCDs:

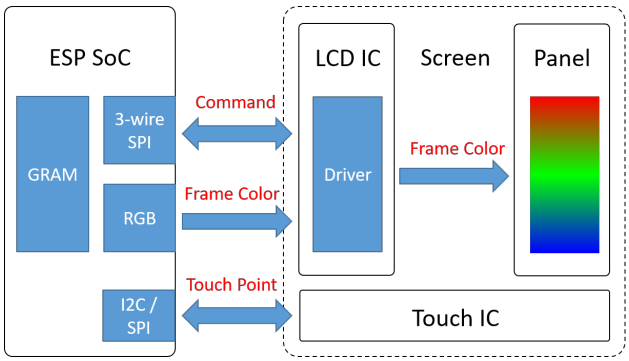


Fig. 14: Schematic diagram of hardware driver framework - RGB LCD

By comparing these two frameworks, it can be observed that, in contrast to SPI/I80 LCDs, RGB LCDs not only require the ESP to use two interfaces for transmitting commands and color data separately but also require that the ESP provides a full-screen-sized Graphic RAM (GRAM) for screen refresh. Due to the limited space in the on-chip SRAM, GRAM is typically placed in the PSRAM.

For QSPI LCDs, different models of driver ICs may require different driving methods. For example, the *SPD2010* IC has a built-in GRAM, and its driving method is similar to SPI/I80 LCDs. On the other hand, the *ST77903* IC does not have internal GRAM, and its driving method is similar to RGB LCDs. However, both of them use a single peripheral interface to transmit commands and color data. Below are schematic diagrams illustrating the hardware driving frameworks for these two types of QSPI LCDs:

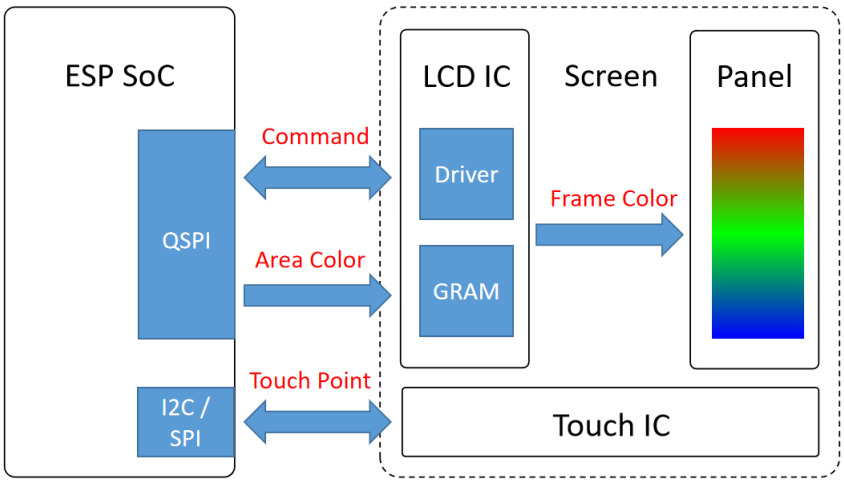


Fig. 15: Schematic diagram of hardware driver framework - QSPI LCD (with GRAM)

Software Framework The software development framework primarily consists of three layers: SDK (Software Development Kit), Driver, and APP (Application).

1. **SDK layer**: ESP-IDF serves as the foundational element of the framework. It not only includes `I2C`, `SPI (QSPI)`, `I80` and `RGB` required to drive LCD and other peripherals, it also provides unified APIs through the `esp_lcd` component to operate the interface and LCD, such as command and parameter transmission, LCD image refresh, inversion, mirroring and other functions.
2. **Driver layer**: Based on the APIs provided by the SDK, various device drivers can be implemented, and the porting of LVGL (GUI framework) can be implemented by initializing interface devices and LCD devices.
3. **APP layer**: Use the APIs provided by LVGL to implement various GUI functions, such as displaying pictures, animations, text, etc.

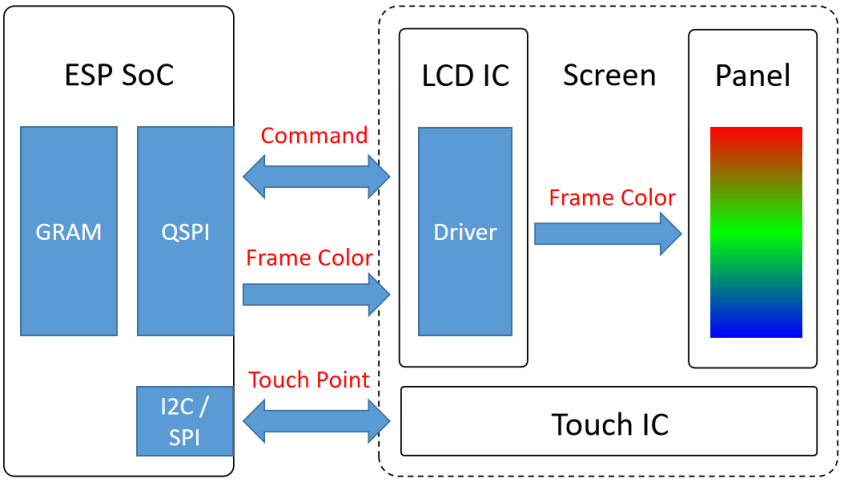


Fig. 16: Schematic diagram of hardware driver framework - QSPI LCD (without GRAM)

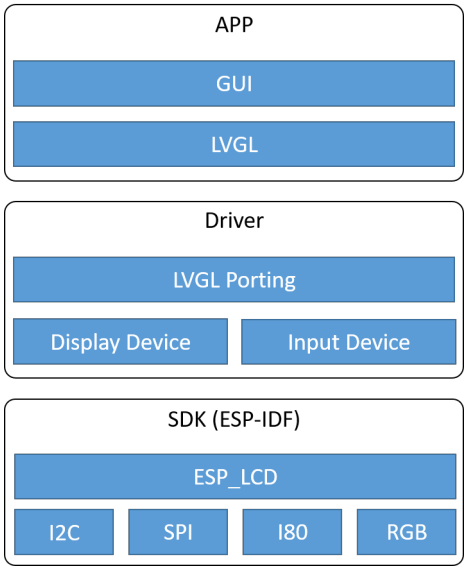


Fig. 17: Schematic diagram of software development framework

Development Steps

Initialize interface device First, initialize the peripherals corresponding to the LCD interface. Then, create the interface device and get its handle, the data type of the handle should be `esp_lcd_panel_io_handle_t`. In this way, unified interface common APIs can be used for data transmission.

Note: For LCDs that only use the `RGB` interface, there is no need to create its interface device, please refer directly to *LCD Initialization*.

Different types of LCD interfaces require the use of different peripherals. The following describes the device initialization process of several common interfaces:

- *SPI LCD Introduction - Initialize interface device*
- *RGB LCD Introduction - Initialize interface device*
- I80 LCD Introduction - Initialization interface device (to be updated)
- QSPI LCD Introduction - Initializing interface devices (to be updated)

For a more detailed description of this part, please refer to ESP-IDF Programming Guide.

Initialize LCD device Since different models of LCD driver ICs may have different commands (registers) and parameters, and different interface types may also use different data formats and driving methods, here first need to use interface common APIs for specific interfaces to port the target LCD driver, then create the LCD device and obtain the data type `esp_lcd_panel_handle_t` handle, ultimately enabling applications to pass unified LCD common APIs to operate the LCD device.

Note: For LCDs that only use the `RGB` interface, there is no need to port its driver components. Please refer directly to *LCD Initialization*.

Before porting the driver component, please first try to obtain the components of the target LCD driver IC directly from *LCD driver component*. If the component does not exist, it can also be porting based on an existing component with the same interface type. LCD drivers with different interface types may have different porting principles. The following describes the porting methods of LCD driver components with several common interfaces:

- *SPI LCD Introduction - Porting driver components*
- *RGB LCD Introduction - Porting driver components*
- I80 LCD Introduction - Porting driver component (to be updated)
- QSPI LCD Introduction - Porting driver component (to be updated)

Then, the LCD initialization can be realized by using the driver component. The LCD initialization of several common interfaces is explained below:

- *SPI LCD Introduction - Initialize LCD device*
- *RGB LCD Introduction - Initialize LCD device*
- I80 LCD Introduction - Initialize LCD device (To be updated)
- QSPI LCD Introduction - Initialize LCD device (To be updated)

For a more detailed description of this part, please refer to the ESP-IDF Programming Guide.

Porting LVGL (To be updated)

Design GUI (To be updated)

Common Problems

The following lists some common issues encountered during the development of LCD applications. Please click on the issues to navigate and view the solutions.

- How to use Arduino IDE to develop GUI for ESP series chips
- Maximum Resolution and Frame Rate Supported by ESP Series Chips for LCD
- How ESP series chips improve LCD rendering frame rate
- How to increase the PCLK (refresh frame rate) of RGB LCD with ESP32-S3
- How to solve the problem of screen offset or flickering when driving RGB LCD with ESP32-S3
- How to configure ESP32-S3R8 PSRAM 120M Octal(DDR)

Related Documents

- ESP-IDF Programming Guide- LCD
- ESP-FAQ - LCD
- LVGL Documentation

4.1.4 SPI LCD Introduction

Contents

- *Terminology*
- *Interface Mode*
  - *Interface I/II Mode*
  - *3/4-line Mode*
- *SPI LCD Driving Process:*
- *Initialization Interface Device*
  - *Initializing the Bus*
  - *Creating the Interface Device*
- *Porting Driver Components*
- *Initializing the LCD Device*
- *Related documentation*

Terminology

Please refer to the *LCD Terms Table* 。

Interface Mode

Different interface modes require the main control to adopt different wiring and driving methods. Below, taking *ST7789* as an example, we will introduce several common interface modes.

IM3	IM2	IM1	IM0	Interface	Read back selection
0	1	0	1	3-line serial interface Ⅰ	Via the read instruction (8-bit, 24-bit and 32-bit read parameter)
0	1	1	0	4-line serial interface Ⅰ	
1	1	0	1	3-line serial interface Ⅱ	
1	1	1	0	4-line serial interface Ⅱ	

Table 13 Selection of serial interface

Fig. 18: Mode Selection for SPI Interface

From the above figure, it can be seen that *ST7789* uses the `IM[3:0]` pins to select the configuration of `Interface I/II` and `3/4-line`, enabling four different interface modes. The following diagram shows the pin description for the *ST7789* SPI interface:

3-line serial interface Ⅰ

Pin Name	Description
CSX	Chip selection signal
DCX	Clock signal
SDA	Serial input/output data

4-line serial interface Ⅰ

Pin Name	Description
CSX	Chip selection signal
WRX	Data is regarded as a command when WRX is low Data is regarded as a parameter or data when WRX is high
DCX	Clock signal
SDA	Serial input/output data

3-line serial interface Ⅱ

Pin Name	Description
CSX	Chip selection signal
DCX	Clock signal
SDA	Serial input data
SDO	Serial output data

4-line serial interface Ⅱ

Pin Name	Description
CSX	Chip selection signal
WRX	Data is regarded as a command when WRX is low Data is regarded as a parameter or data when WRX is high

Fig. 19: Pin Description for SPI Interface

Note: SPI pin names: CS, SCK (SCL), SDA (MOSI), SDO (MISO), DC (RS).

Interface I/II Mode From the diagram, it can be observed that the main difference between `Interface I` and `Interface II` lies in whether only one data line is used for both data read and write (such as only using `MOSI`).

Mode	Whether only one data line is used for data read and write	ESP Support
Interface I	Yes	Yes
Interface II	No	Yes

3/4-line Mode From the diagram, it can be seen that the main difference between "3-line" and "4-line" lies in whether the D/C signal line is used.

Mode	Whether D/C signal line is used	ESP Support
3-line	No	No
4-line	Yes	Yes

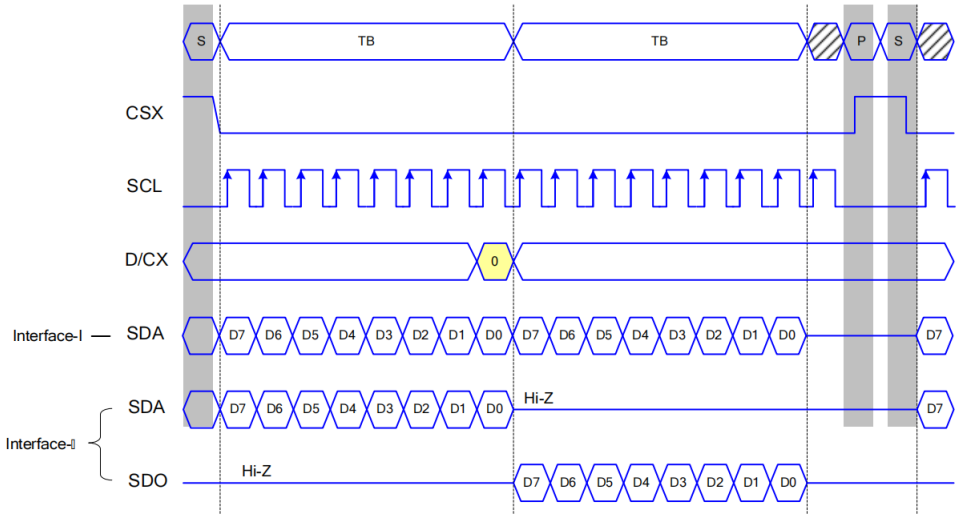


Fig. 20: Timing Diagram Comparison for Interface I/II Mode (4-line)

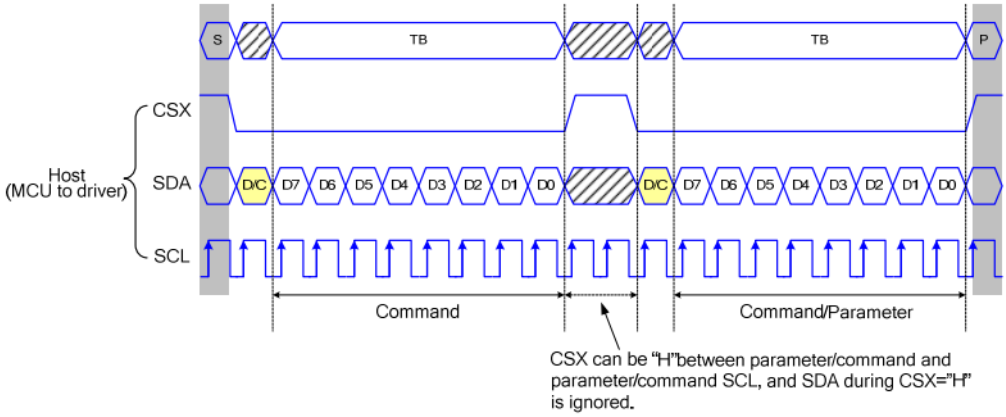


Figure 13 3-line serial interface write protocol (write to register with control bit in transmission)

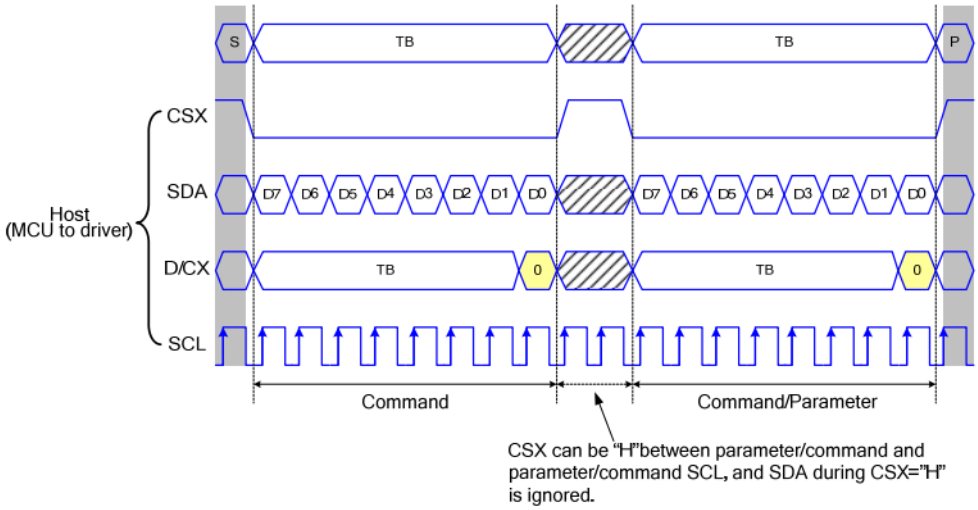


Figure 14 4-line serial interface write protocol (write to register with control bit in transmission)

Fig. 21: Timing Diagram Comparison for 3/4-line Mode (Interface I)

Note:

- `3-line` mode is sometimes referred to as `3-wire` or `9-bit` mode.
- While ESP' s SPI peripheral does not support the LCD' s `3-line` mode, it can be implemented through software emulation. For details, please refer to the esp_lcd_panel_io_additions component. This is typically used for initializing RGB LCDs.

SPI LCD Driving Process:

The SPI LCD driver process can be roughly divided into three parts: initializing the interface device, porting driver components, and initializing the LCD device.

Initialization Interface Device

Initializing the interface device involves first initializing the bus and then creating the interface device. The following is based on the spi_lcd_touch example from ESP-IDF release/v5.1, demonstrating how to initialize an SPI interface device.

Initializing the Bus Example Code:

```
#include "driver/spi_master.h"            // Dependent header files
#include "esp_check.h"

spi_bus_config_t buscfg = {
    .sclk_io_num = EXAMPLE_PIN_NUM_SCLK,  // IO number for connecting LCD SCK␣
↪(SCL) signal
    .mosi_io_num = EXAMPLE_PIN_NUM_MOSI,  // IO number for connecting LCD MOSI␣
↪(SDO, SDA) signal
    .miso_io_num = EXAMPLE_PIN_NUM_MISO,  // IO number for connecting LCD MISO␣
↪(SDI) signal; set to `-1` if data read from LCD is not required
    .quadwp_io_num = -1,                  // Must be set to `-1`
    .quadhd_io_num = -1,                  // Must be set to `-1`
    .max_transfer_sz = EXAMPLE_LCD_H_RES * 80 * sizeof(uint16_t), // Represents␣
↪the maximum number of bytes allowed for a single SPI transfer; usually set to␣
↪the screen size
};
ESP_ERROR_CHECK(spi_bus_initialize(LCD_HOST, &buscfg, SPI_DMA_CH_AUTO));
                                          // The 1st parameter represents the SPI␣
↪host ID used, consistent with subsequent interface device creation
                                          // The 3rd parameter represents the DMA␣
↪channel number used, set to `SPI_DMA_CH_AUTO` by default
```

If multiple devices are using the same SPI bus simultaneously, the bus only needs to be initialized once.

The following are explanations for some configuration parameters:

- If the LCD driver IC is configured in *Interface-I mode*, only set `mosi_io_num` as the data line IO, and set `miso_io_num` to -1.
- The SPI driver checks the size of the input data before transmitting data. If the number of bytes for a single transfer exceeds `max_transfer_sz`, an error will be reported. However, the **maximum number of bytes allowed for a single DMA transfer** depends not only on `max_transfer_sz` but is also limited by SPI_LL_DATA_MAX_BIT_LEN in ESP-IDF (different ESP series have different values), i.e., satisfying `max_transfer_sz <= MIN(max_transfer_sz, (SPI_LL_DATA_MAX_BIT_LEN / 8))`. Since the esp_lcd driver checks in advance whether the input data size exceeds the limit and performs **packetization** if it does, controlling SPI for multiple transfers, **max_transfer_sz is usually set to the screen size**.

Creating the Interface Device Example Code:

```
#include "esp_lcd_panel_io.h"       // Header file dependency

static bool example_on_color_trans_dome(esp_lcd_panel_io_handle_t panel_io, esp_
↪lcd_panel_io_event_data_t *edata, void *user_ctx)
{
        /* Callback function when color data transmission is completed; perform␣
↪operations here if needed */

    return false;
}

esp_lcd_panel_io_handle_t io_handle = NULL;
esp_lcd_panel_io_spi_config_t io_config = {
    .dc_gpio_num = EXAMPLE_PIN_NUM_LCD_DC,      // IO number connected to the LCD␣
↪DC (RS) signal; set to `-1` to disable
    .cs_gpio_num = EXAMPLE_PIN_NUM_LCD_CS,      // IO number connected to the LCD␣
↪CS signal; set to `-1` to disable
    .pclk_hz = EXAMPLE_LCD_PIXEL_CLOCK_HZ,      // SPI clock frequency (Hz), ESP␣
↪supports up to 80M (SPI_MASTER_FREQ_80M)
                                                // Determine the maximum value␣
↪based on the LCD driver IC data sheet
    .lcd_cmd_bits = EXAMPLE_LCD_CMD_BITS,       // Number of bits per LCD command,␣
↪should be a multiple of 8
    .lcd_param_bits = EXAMPLE_LCD_PARAM_BITS,   // Number of bits per LCD parameter,
↪ should be a multiple of 8
    .spi_mode = 0,                              // SPI mode (0-3); determine based␣
↪on the LCD driver IC data sheet and hardware configuration (e.g., IM[3:0])
    .trans_queue_depth = 10,                    // Queue depth for SPI device data␣
↪transmission; usually set to 10
    .on_color_trans_done = example_on_color_trans_dome,    // Callback function␣
↪after a single call to `esp_lcd_panel_draw_bitmap()` completes transmission
    .user_ctx = &example_user_ctx,              // User parameter passed to the␣
↪callback function
    .flags = {      // Parameters related to SPI timing; determine based on the LCD␣
↪driver IC data sheet and hardware configuration
        .sio_mode = 0,      // Read and write data through one data line (MOSI); 0:␣
↪Interface I type, 1: Interface II type
    },
};
ESP_ERROR_CHECK(esp_lcd_new_panel_io_spi((esp_lcd_spi_bus_handle_t)LCD_HOST, &io_
↪config, &io_handle));

/* The following functions can also be used to register the callback function for␣
↪color data transmission completion events */
// const esp_lcd_panel_io_callbacks_t cbs = {
//     .on_color_trans_done = example_on_color_trans_dome,
// };
// esp_lcd_panel_io_register_event_callbacks(io_handle, &cbs, &example_user_ctx);
```

Once the SPI bus is initialized, you can create the corresponding interface device. Each interface device corresponds to an SPI master device.

Note:: For a more detailed explanation of the ``SPI`` interface configuration parameters, please refer to the ESP-IDF Programming Guide.

By creating the interface device, you can obtain a handle of data type `esp_lcd_panel_io_handle_t`, which allows you to use the following General Interface APIs to send **commands** and **image data** to the LCD driver IC:

1. `esp_lcd_panel_io_tx_param()`: Used to send a single command and its associated parameters to the LCD. Internally, it uses the `spi_device_polling_transmit()` function for data transmission, and using this function will wait for the data transmission to complete before returning.
2. `esp_lcd_panel_io_tx_color()`: Used to send a single command and image data for LCD screen

refreshing. Inside the function, it uses `spi_device_polling_transmit()` to send commands and a small amount of parameters, and then uses `spi_device_queue_trans()` to send large amounts of image data in packets. The size of each packet is limited by the **maximum number of bytes allowed for a single DMA transfer in SPI**. This function pushes relevant data, including the image buffer address, into the queue, and the depth of the queue is specified by the `trans_queue_depth` parameter. Once the data is successfully pushed into the queue, the function immediately returns. Therefore, if you plan to modify the same image buffer in subsequent operations, you need to register a callback function to determine whether the previous transfer has been completed. If you don't do this, modifying on an incomplete transfer may lead to display errors due to data corruption.

Porting Driver Components

The basic principles of porting an SPI LCD driver component include the following three points:

1. Sending specified format commands and parameters based on the interface device handle of data type `esp_lcd_panel_io_handle_t`.
2. Implementing and creating an LCD device, then implementing various functions in the esp_lcd_panel_t structure through the registration of callback functions.
3. Implementing a function to provide an LCD device handle of data type `esp_lcd_panel_handle_t`, enabling the application to use LCD General APIs to operate the LCD device.

The following is an explanation of the implementation of various functions in `esp_lcd_panel_handle_t` and their corresponding relationships with LCD General APIs:

Function	LCD General APIs	Implementation Explanation
reset()	esp_lcd_panel_reset()	If the device is connected to a reset pin, perform a hardware reset through that pin. Otherwise, perform a software reset using the command `LCD_CMD_SWRESET (01h)`.
init()	esp_lcd_panel_init()	Initialize the LCD device by sending a series of commands and parameters.
del()	esp_lcd_panel_del()	Release resources occupied by the driver, including allocated memory and used IO.
draw_bitmap()	esp_lcd_panel_draw_bitmap()	First, send the starting and ending coordinates of the image using the commands `LCD_CMD_CASET (2Ah)` and `LCD_CMD_RASET (2Bh)`, then send the image data using the command `LCD_CMD_RAMWR (2Ch)`.
mirror()	esp_lcd_panel_mirror()	Set whether to mirror the X-axis and Y-axis of the screen using the command `LCD_CMD_MADCTL (36h)`.
swap_xy()	esp_lcd_panel_swap_xy()	Set whether to swap the X-axis and Y-axis of the screen using the command `LCD_CMD_MADCTL (36h)`.
set_gap()	esp_lcd_panel_set_gap()	Modify the starting and ending coordinates for drawing through software to achieve drawing offset.
invert_color()	esp_lcd_panel_invert_color()	Invert the color data of pixels using the commands `LCD_CMD_INVON (21h)` and `LCD_CMD_INVOFF (20h)` (0xF0F0 -> 0x0F0F).
disp_on_off()	esp_lcd_panel_disp_on_off()	Turn the screen display on or off using the commands `LCD_CMD_DISON (29h)` and `LCD_CMD_DISOFF (28h)`.

For most SPI LCDs, their driver IC commands and parameters are compatible with the explanations provided above. Therefore, porting can be completed through the following steps:

1. Choose an SPI LCD driver component in *LCD Driver Components* that is similar to the model you are targeting.
2. Consult the data sheet of the target LCD driver IC to verify the consistency of commands and parameters within each function of the selected component. If inconsistencies are identified, make appropriate modifications to the relevant code.
3. Although the models of the LCD driver IC might be identical, different screens require specific configurations through initialization commands provided by their respective manufacturers. Therefore, you need to modify the commands and parameters sent in the `init()` function. These initialization commands are usually stored in a static array in a specific format. Additionally, be careful not to include some special commands in the

initialization commands, such as `LCD_CMD_COLMOD (3Ah)` and `LCD_CMD_MADCTL (36h)`, as these commands are managed and used by the driver component.

4. Use the character search and replace feature in your editor to replace the LCD driver IC name in the component with the target name. For example, replace `gc9a01` with `st77916`.

Initializing the LCD Device

The following is an example code explanation using GC9A01:

```
#include "esp_lcd_panel_vendor.h"   // Dependent header files
#include "esp_lcd_panel_ops.h"
#include "esp_lcd_gc9a01.h"         // Header file of the target driver component

/**
* Used to store the initialization commands and parameters of the LCD driver IC
*/
// static const gc9a01_lcd_init_cmd_t lcd_init_cmds[] = {
// //  {cmd, { data }, data_size, delay_ms}
//     {0xfe, (uint8_t []){0x00}, 0, 0},
//     {0xef, (uint8_t []){0x00}, 0, 0},
//     {0xeb, (uint8_t []){0x14}, 1, 0},
//     ...
// };

/* Create the LCD device */
esp_lcd_panel_handle_t panel_handle = NULL;
// const gc9a01_vendor_config_t vendor_config = {  // Used to replace the␣
↪initialization commands and parameters in the driver component
//     .init_cmds = lcd_init_cmds,
//     .init_cmds_size = sizeof(lcd_init_cmds) / sizeof(gc9a01_lcd_init_cmd_t),
// };
esp_lcd_panel_dev_config_t panel_config = {
    .reset_gpio_num = EXAMPLE_PIN_NUM_LCD_RST,     // Connect the IO number of the␣
↪LCD reset signal, set to `-1` to indicate not using
    .rgb_ele_order = LCD_RGB_ELEMENT_ORDER_RGB,    // Element order of pixel color␣
↪(RGB/BGR),
                                                   // Usually controlled by the␣
↪command `LCD_CMD_MADCTL (36h)`
    .bits_per_pixel = EXAMPLE_LCD_BIT_PER_PIXEL,   // Bit depth of the color format␣
↪(RGB565: 16, RGB666: 18),
                                                   // usually controlled by the␣
↪command `LCD_CMD_COLMOD (3Ah)`
    // .vendor_config = &vendor_config,            // Used to replace the␣
↪initialization commands and parameters in the driver component
};
ESP_ERROR_CHECK(esp_lcd_new_panel_gc9a01(io_handle, &panel_config, &panel_handle));

/* Initialize the LCD device */
ESP_ERROR_CHECK(esp_lcd_panel_reset(panel_handle));
ESP_ERROR_CHECK(esp_lcd_panel_init(panel_handle));
// ESP_ERROR_CHECK(esp_lcd_panel_invert_color(panel_handle, true));   // Use these␣
↪functions as needed
// ESP_ERROR_CHECK(esp_lcd_panel_mirror(panel_handle, true, true));
// ESP_ERROR_CHECK(esp_lcd_panel_swap_xy(panel_handle, true));
// ESP_ERROR_CHECK(esp_lcd_panel_set_gap(panel_handle, 0, 0));
ESP_ERROR_CHECK(esp_lcd_panel_disp_on_off(panel_handle, true));
```

First, create an LCD device and obtain a handle of data type `esp_lcd_panel_handle_t` using the ported driver component. Then, use the LCD General APIs to initialize the LCD device.

Here are some explanations regarding the use of the `esp_lcd_panel_draw_bitmap()` function to refresh images on an SPI LCD:

- The number of bytes of the image buffer passed to this function can be greater than `max_transfer_sz`. In this case, the `esp_lcd` driver internally performs packetization based on the maximum number of bytes allowed for a single DMA transfer in SPI.
- Since this function transfers image data using DMA, which means that after the function call, data is still being transferred via DMA, it is not allowed to modify the currently used buffer area (such as rendering with LVGL). Therefore, it is necessary to determine whether the previous transfer has completed through bus initialization or by calling the callback functions registered with `esp_lcd_panel_io_register_event_callbacks()`.
- As the SPI driver currently does not support directly transferring data from PSRAM using DMA, it internally checks whether the data is stored in PSRAM. If it is, it will copy it to SRAM before transferring. Therefore, it is recommended to use SRAM as the image buffer for transfer (such as a buffer used for LVGL rendering). Otherwise, directly transferring large image data from PSRAM may lead to insufficient SRAM.

Related documentation

- ST7789 Datasheet

4.1.5 RGB LCD Introduction

Contents

- *Terminology*
- *Interface Mode*
  - *Mode Selection*
  - *DE Mode*
  - *SYNC Mode*
  - *Mode Comparison*
- *Color Formats*
- *RGB LCD Driver Process*
- *Initialization of Interface Devices*
- *Porting Driver Components*
- *Initialize LCD Device*
- *Related Documentation*

Terminology

Please refer to the *LCD Terms Table* 。

Interface Mode

Most RGB LCDs use the "SPI + RGB" interface mode. They require sending commands through the SPI interface to initialize the LCD. After initialization, it's possible to dynamically modify relevant configurations such as vertical/horizontal mirroring for greater flexibility. Some RGB LCDs only use the "RGB" interface, eliminating the need to send commands for LCD initialization. However, they cannot modify any configurations, making the driving method simpler. The following diagram illustrates the interface type selection for the ST7701S.

From the above figure, it can be observed that *ST7701S* selects the configuration of the `SPI + RGB` interface through the `IM[3:0]` pins. Typically, LCDs of this type choose the `3-wire SPI + RGB` interface, corresponding to the `RGB+9b_SPI(rise/fall)` in the figure. Here, `9b_SPI` represents the *3-line mode* of the SPI interface (generally referred to as `3-wire`), and `rise/fall` indicates the effective edge of the `SCL` signal. `rise` signifies the rising edge is effective (SPI modes 0/3), while `fall` represents the falling edge (SPI modes 1/2)

The following diagram depicts the pin descriptions for the *ST7701S* `SPI` and `RGB` interfaces:

IM3	IM2	IM1	IM0	Interface	Data pins
0	0	0	1	RGB+8b_SPI(fall)	D[0~23]
	0	1	0	RGB+9b_SPI(fall)	D[0~23]
	0	1	1	RGB+16b_SPI(rise)	D[0~23]
	1	0	1	MIPI	HSSI_D1_P/N,HSSI_D0_P/N
	1	1	0	MIPI+16b_SPI(rise)	HSSI_D1_P/N,HSSI_D0_P/N
1	0	0	1	RGB+8b_SPI(rise)	D[0~23]
	0	1	0	RGB+9b_SPI(rise)	D[0~23]
	0	1	1	RGB+16b_SPI(fall)	D[0~23]
	1	0	1	MIPI	HSSI_D1_P/N,HSSI_D0_P/N
	1	1	0	MIPI+16b_SPI(fall)	HSSI_D1_P/N,HSSI_D0_P/N

Table 10 Interface Type Selection

Fig. 22: Interface Type Selection for ST7701S

3-line serial interface (9 bits)

Pin Name	Description
CSX	Chip selection signal
SCL	Serial input CLK
SDA	Serial input data
SDO	Serial output data

4-line serial interface (8 bits)

Pin Name	Description
CSX	Chip selection signal
DCX	Data is regarded as a command when SCL is low Data is regarded as a parameter or data when SCL is high
SCL	Clock signal
SDA	Serial input data
SDO	Serial output data

Fig. 23: Pin Description for ST7701S SPI Interface

Symbol	Name	Description
PCLK	Pixel clock	Pixel clock for capturing pixels at display interface
HS	Horizontal sync	Horizontal synchronization timing signal
VS	Vertical sync	Vertical synchronization timing signal
DE	Data enable	Data enable signal (assertion indicates valid pixels)
DB[23:0]	Pixel data	Pixel data in 16-bit,18-bit and 24-bit format

Table 11 The interface signals of RGB interface

Fig. 24: Pin Description for ST7701S RGB Interface

Note: RGB pin names: CS, SCK (SCL), SDA (MOSI), HSYNC, VSYNC, PCLK, DE, D[23:0] (D[17:0]/D[7:0]).

For LCDs using the `SPI + RGB` interfaces, you can typically configure the `RGB` interface to operate in either `DE mode` or `SYNC mode` through commands. The following section illustrates these two modes using *ST7701S* as an example.

ST7701S supports two kinds of RGB interface, DE mode and HV mode. The table shown below uses command C3h to select RGB interface mode.

DE/Sync	RGB Mode
0	DE mode
1	HV mode

Fig. 25: Mode Selection for ST7701S RGB Interface

Mode Selection From the diagram, it can be observed that *ST7701S* can configure the mode of RGB using the command `C3h`. It is important to note that LCD driver ICs of different models may use different commands for configuration, for instance, *GC9503* is configured using the command `B0h`.

DE Mode

SYNC Mode

Mode Comparison By comparing the timing diagrams of `DE Mode` and `SYNC Mode`, the main differences lie in whether the DE signal line is used and the configuration requirements for the blanking porch. Summarized in the table below:

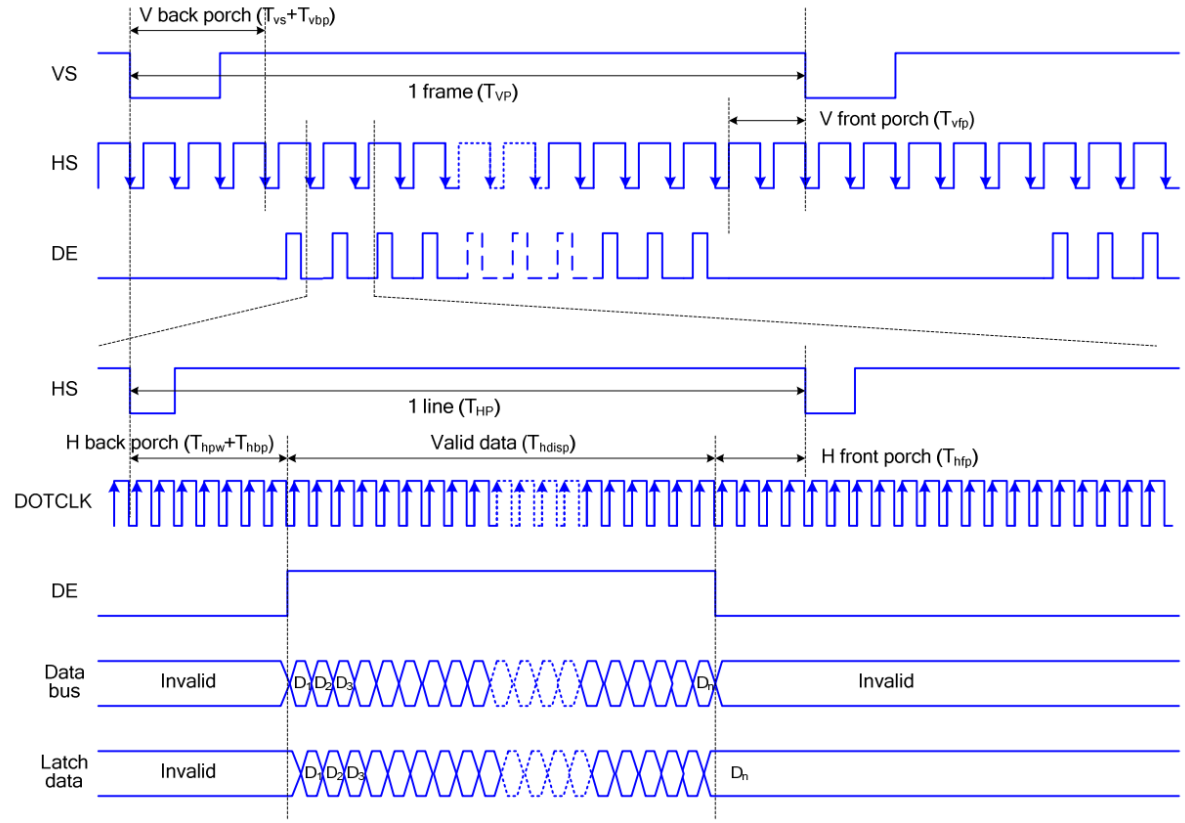
Mode	Use of DE Signal Line	Configuration of Blanking Porch Register	ESP Support
DE Mode	Yes	No	Yes
SYNC Mode	No	Yes	Yes

Color Formats

Most RGB LCDs support various color (input data) formats, including `RGB565`, `RGB666`, `RGB888`, etc. Typically, the `COLMOD (3Ah)` command can be used to configure the color format. The following diagram illustrates the color format configuration for *ST7701S*:

From the above diagram, it can be seen that *ST7701S* supports three color formats: `16-bit RGB565`, `18-bit RGB666`, and `24-bit RGB888`. Here, `N-bit` indicates the number of data lines in the interface, and the selection is made through the `COLMOD (3Ah): VIPF[2:0]` and `COLCTRL (CDh): MDT` commands. **Note that command configuration must be consistent with the hardware interface**. For example, if the LCD module provides only 18 data lines, the software must not configure the color format as `24-bit RGB888`. In such a case, the configuration for `16-bit RGB565` is only possible when the data lines are `D[21:16], D[13:8], D[5:0]`.

In addition, the bit depth of color formats is not necessarily equal to the effective number of data lines in the interface. The following diagrams illustrate the interface type selection and color format configuration for *ST77903*:



Note: The setting of front porch and back porch in host must match that in IC as this mode.

Figure 23 Timing Chart of Signals in RGB Interface DE Mode

Fig. 26: Timing Diagram for ST7701S DE Mode

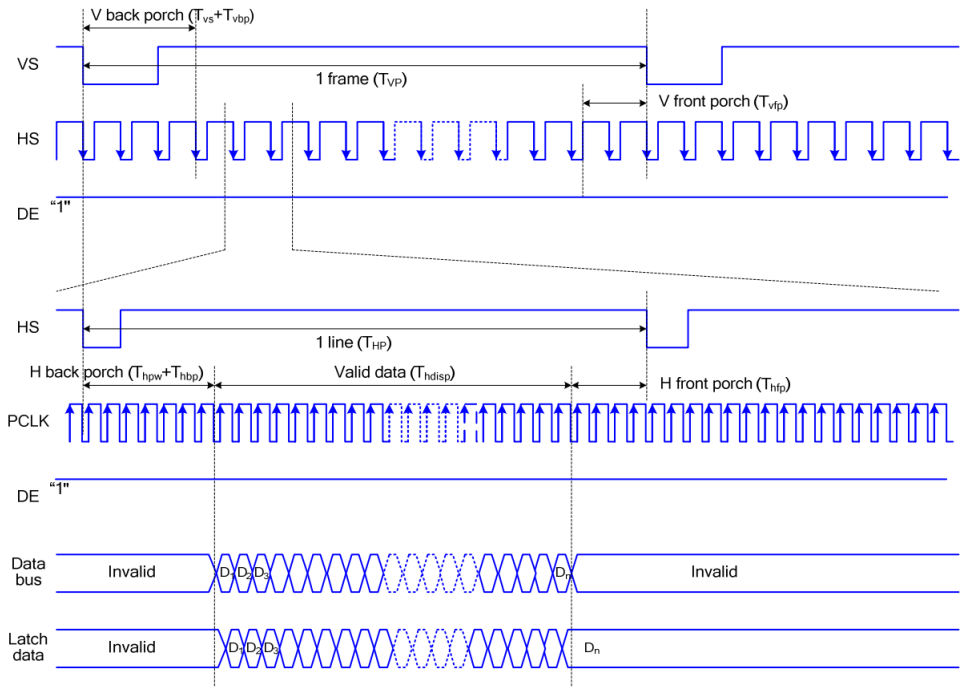


Figure 24 Timing chart of RGB interface HV mod

Fig. 27: Timing Diagram for ST7701S SYNC Mode

Pad name	24 bits configuration VIPF[3:0]=0111	18 bits configuration VIPF[3:0]=0110		16 bits configuration VIPF[3:0]=0101
		MDT=0	MDT=1	
DB[23]	R7	Not used	Not used	Not used
DB[22]	R6	Not used	Not used	Not used
DB[21]	R5	R5	Not used	Not used
DB[20]	R4	R4	Not used	R4
DB[19]	R3	R3	Not used	R3
DB[18]	R2	R2	Not used	R2
DB[17]	R1	R1	R5	R1
DB[16]	R0	R0	R4	R0
DB[15]	G7	Not used	R3	Not used
DB[14]	G6	Not used	R2	Not used
DB[13]	G5	G5	R1	G5
DB[12]	G4	G4	R0	G4
DB[11]	G3	G3	G5	G3
DB[10]	G2	G2	G4	G2
DB[09]	G1	G1	G3	G1
DB[08]	G0	G0	G2	G0
DB[07]	B7	Not used	G1	Not used
DB[06]	B6	Not used	G0	Not used
DB[05]	B5	B5	B5	Not used
DB[04]	B4	B4	B4	B4
DB[03]	B3	B3	B3	B3
DB[02]	B2	B2	B2	B2
DB[01]	B1	B1	B1	B1
DB[00]	B0	B0	B0	B0

Table 12 The interface color mapping of RGB interface

Fig. 28: Color Format Configuration for ST7701S

IM	3Ah	RGB Interface Mode	Data pins
1,0	101	3-SPI with RGB565	DB[7:2]
1,0	110	3-SPI with RGB666	DB[7:2]
1,0	111	3-SPI with RGB888	DB[7:0]
1,1	101	4-SPI with RGB565	DB[7:2]
1,1	110	4-SPI with RGB666	DB[7:2]
1,1	111	4-SPI with RGB888	DB[7:0]

Fig. 29: Interface Type Selection for ST77903 RGB Interface

serial RGB 565

Pin	1st Data	2nd Data	3rd Data	...	(3N+1)th Data
DAP[0]	x	x	x	...	x
DAP[1]	x	x	x	...	x
DAP[2]	x	1'G0	x	...	x
DAP[3]	1'R0	1'G1	1'B0	...	N'R0
DAP[4]	1'R1	1'G2	1'B1	...	N'R1
DAP[5]	1'R2	1'G3	1'B2	...	N'R2
DAP[6]	1'R3	1'G4	1'B3	...	N'R3
DAP[7]	1'R4	1'G5	1'B4	...	N'R4

serial RGB 666

Pin	1st Data	2nd Data	3rd Data	...	(3N+1)th Data
DAP[0]	x	x	x	...	x
DAP[1]	x	x	x	...	x
DAP[2]	1'R0	1'G0	1'B0	...	N'R0
DAP[3]	1'R1	1'G1	1'B1	...	N'R1
DAP[4]	1'R2	1'G2	1'B2	...	N'R2
DAP[5]	1'R3	1'G3	1'B3	...	N'R3
DAP[6]	1'R4	1'G4	1'B4	...	N'R4
DAP[7]	1'R5	1'G5	1'B5	...	N'R5

serial RGB 888

Pin	1st Data	2nd Data	3rd Data	...	(3N+1)th Data
DAP[0]	1'R0	1'G0	1'B0	...	N'R0
DAP[1]	1'R1	1'G1	1'B1	...	N'R1
DAP[2]	1'R2	1'G2	1'B2	...	N'R2
DAP[3]	1'R3	1'G3	1'B3	...	N'R3
DAP[4]	1'R4	1'G4	1'B4	...	N'R4
DAP[5]	1'R5	1'G5	1'B5	...	N'R5
DAP[6]	1'R6	1'G6	1'B6	...	N'R6
DAP[7]	1'R7	1'G7	1'B7	...	N'R7

Fig. 30: Color Format Configuration for ST77903

From the above diagrams, it can be observed that *ST77903* supports three color formats: `6-bit RGB565`, `6-bit RGB666`, and `8-bit RGB888`, with bit depths of `16-bit`, `18-bit`, and `24-bit`, respectively. While most LCDs' `RGB` interfaces can parallelly transmit color data for a single pixel within a single clock cycle, interfaces like ST77903 require multiple clock cycles to transmit color data for a single pixel, earning them the name **Serial RGB Interface (SRGB)**.

Note: Although ESP32-S3 only supports `16-bit RGB565` and `8-bit RGB888` color formats, it can be configured to drive LCDs with `18-bit RGB666` or `24-bit RGB888` color formats through special hardware connections. For the connection details, please refer to the development board ESP32-S3-LCD-EV-Board and its LCD Subboard 2 (3.95' LCD_QMZX) and LCD Subboard 3 schematics.

RGB LCD Driver Process

The RGB LCD driver process can be roughly divided into three parts: initializing interface devices, porting driver components, and initializing the LCD device.

Initialization of Interface Devices

Here is the code explanation for creating a 3-wire SPI interface device using the esp_lcd_panel_io_additions component:

```
#include "esp_check.h"        // Header file dependency
#include "esp_lcd_panel_io.h"
#include "esp_lcd_panel_io_additions.h"

esp_lcd_panel_io_3wire_spi_config_t io_config = {
    .line_config = {
        .cs_io_type = IO_TYPE_GPIO,                     // Set to `IO_TYPE_
↪EXPANDER` to use IO expander pins; otherwise, use GPIO
        .cs_gpio_num = EXAMPLE_LCD_IO_SPI_CS,           // GPIO number connected to␣
↪the LCD CS signal
        // .cs_expander_pin = EXAMPLE_LCD_IO_SPI_CS,    // Expander IO chip pin␣
↪number connected to the LCD CS signal
        .scl_io_type = IO_TYPE_GPIO,                    // Set to `IO_TYPE_
↪EXPANDER` to use IO expander pins; otherwise, use GPIOO
        .scl_gpio_num = EXAMPLE_LCD_IO_SPI_SCK,         // GPIO number connected to␣
↪the LCD SCK (SCL) signal
        // .scl_expander_pin = EXAMPLE_LCD_IO_SPI_SCK, // Expander IO chip pin␣
↪number connected to the LCD SCK (SCL) signal
        .sda_io_type = IO_TYPE_GPIO,                    // Set to `IO_TYPE_
↪EXPANDER` to use IO expander pins; otherwise, use GPIO
        .sda_gpio_num = EXAMPLE_LCD_IO_SPI_SDO,         // GPIO number connected to␣
↪the LCD MOSI (SDO, SDA) signal
        // .sda_expander_pin = EXAMPLE_LCD_IO_SPI_SDO, // Expander IO chip pin␣
↪number connected to the LCD MOSI (SDO, SDA) signal
        .io_expander = NULL,                            // If using IO expander␣
↪pins, pass the initialized device handle
    },
    .expect_clk_speed = PANEL_IO_3WIRE_SPI_CLK_MAX,     // Expected SPI clock␣
↪frequency; due to software simulation, there may be a significant error
                                                        // Default set to `PANEL_IO_
↪3WIRE_SPI_CLK_MAX`
    .spi_mode = 0,                  // SPI mode (0-3); determine based on the LCD␣
↪driver IC data sheet and hardware configuration (e.g., IM[3:0])
    .lcd_cmd_bytes = 1,             // Number of bytes per LCD command (1-4);␣
↪usually set to `1`
    .lcd_param_bytes = 1,           // Number of bytes per LCD parameter (1-4);␣
↪usually set to `1`
```

(continues on next page)

(continued from previous page)

```
    .flags = {
        .use_dc_bit = 1,            // Default set to `1`
        .del_keep_cs_inactive = 1,  // Default set to `1`
    },
}
esp_lcd_panel_io_handle_t io_handle = NULL;
ESP_ERROR_CHECK(esp_lcd_new_panel_io_3wire_spi(&io_config, &io_handle));
```

For LCDs that only use the RGB interface, as they do not support the transmission of commands and parameters, there is no need to initialize an interface device. Please refer directly to *Initializing the LCD device*.

For LCDs using both the 3-wire SPI and RGB interface, only the creation of the `3-wire SPI` interface device is required. Since ESP' s SPI peripheral does not directly support the transmission of 9-bit data, and this interface is only used for transmitting commands and parameters with a small data volume, and the bandwidth and timing requirements for data transmission are not high, GPIO or IO expander chip pins (such as TCA9554) can be used to simulate the SPI protocol through software.

Creating the interface device provides a handle of data type `esp_lcd_panel_io_handle_t`. Subsequently, you can use `esp_lcd_panel_io_tx_param()` to send **commands** to the LCD driver IC.

Porting Driver Components

For LCDs using only the RGB interface, the RGB interface driver already implements the functionalities specified in the esp_lcd_panel_t structure through registered callback functions. Additionally, it provides the function `esp_lcd_new_rgb_panel()` to create an LCD device with the data type `esp_lcd_panel_handle_t`, allowing the application to use the LCD Generic APIs to operate the LCD device. Therefore, no driver component porting is needed for this type of LCD; please refer directly to *Initializing the LCD device*.

For LCDs using both the 3-wire SPI and RGB interface, in addition to the RGB interface driver, you also need to send commands and parameters through the `3-wire SPI` interface. The basic principles for implementing this LCD driver component include the following three points:

1. Send commands and parameters in the specified format through the interface device with data type `esp_lcd_panel_io_handle_t`.
2. Create an LCD device using the function `esp_lcd_new_rgb_panel()`, then use the registered callback functions to **save and override some** functionalities in the device.
3. Implement a function to provide a handle of data type `esp_lcd_panel_handle_t` for the LCD device, allowing the application to use the LCD Generic APIs to operate the LCD device.

Here is the explanation of the functions implemented for `esp_lcd_panel_handle_t` and their corresponding relationships with the RGB interface driver and LCD general APIs:

Function	RGB Interface Driver	LCD General APIs	Implementation Description
reset()	rgb_panel_reset()	esp_lcd_panel_reset()	If the device is connected to a reset pin, perform a hardware reset through that pin; otherwise, perform a software reset using the command `LCD_CMD_SWRESET (01h)` and finally reset the `RGB` interface using `rgb_panel_reset()`.
init()	rgb_panel_init()	esp_lcd_panel_init()	If the `3-wire SPI` interface is not sharing pins with the `RGB` interface, initialize the LCD device by sending a series of commands and parameters; otherwise, initialization should be done during LCD creation, and finally, initialize the `RGB` interface using `rgb_panel_init()`.
del()	rgb_panel_del()	esp_lcd_panel_del()	Release the resources occupied by the driver, including allocated memory and used IO, and use `rgb_panel_del()` to delete the `RGB` interface.
draw_bitmap()	rgb_panel_draw_bitmap()	esp_lcd_panel_draw_bitmap()	Send image data using `rgb_panel_draw_bitmap()` without saving and overwriting.
mirror()	rgb_panel_mirror()	esp_lcd_panel_mirror()	Mirror the X and Y axes either through commands or using `rgb_panel_mirror()` based on user configuration.
swap_xy()	rgb_panel_swap_xy()	esp_lcd_panel_swap_xy()	Swap X and Y axes through software using `rgb_panel_swap_xy()` without saving and overwriting.
set_gap()	rgb_panel_set_gap()	esp_lcd_panel_set_gap()	Modify the starting and ending coordinates for drawing through software using `rgb_panel_set_gap()` without saving and overwriting.
invert_color()	rgb_panel_invert_color()	esp_lcd_panel_invert_color()	Invert pixel color data bitwise through hardware using `rgb_panel_invert_color()` without saving and overwriting (0xF0F0 -> 0x0F0F).
disp_on_off()	rgb_panel_disp_on_off()	esp_lcd_panel_disp_on_off()	Implement the on/off control of LCD display based on user configuration. If `disp_gpio_num` is not configured, control can be achieved using LCD commands `LCD_CMD_DISON (29h)` and `LCD_CMD_DISOFF (28h)`. Additionally, if `disp_gpio_num` is configured, control can be achieved by calling the function `rgb_panel_disp_on_off()`.

For the majority of RGB LCDs, the commands and parameters of their driver IC are compatible with the implementation details mentioned above. Therefore, the porting process can be completed using the following steps:

1. Choose an RGB LCD driver component in the *LCD Driver Components* that is similar to the model you are working with.
2. Refer to the datasheet of the target LCD driver IC to confirm whether its commands and parameters used by various functions in the selected component are consistent. If not, modify the relevant code accordingly.
3. Even if the model of the LCD driver IC is the same, screens from different manufacturers often require configuration with their own set of initialization commands. Therefore, modify the commands and parameters sent in the `init()` function. These initialization commands are typically stored in a static array in a specific format. Additionally, ensure not to include commands controlled by the driver IC, such as `LCD_CMD_COLMOD (3Ah)`, in the initialization commands to ensure successful initialization of the LCD device.
4. Use the search and replace function in your editor to replace the LCD driver IC name in the component with the target name. For example, replace `gc9503` with `st7701`.

Initialize LCD Device

Below is an example code explanation using the rgb_panel code from ESP-IDF release/v5.1:

```
#include "esp_check.h"        // Dependent header file
#include "esp_lcd_panel_ops.h"
#include "esp_lcd_panel_rgb.h"

esp_lcd_panel_handle_t panel_handle = NULL;
esp_lcd_rgb_panel_config_t panel_config = {   // Configuration parameters for the␣
↪RGB interface
    .data_width = EXAMPLE_LCD_DATA_WIDTH,               // Data line width of the␣
↪RGB interface, e.g., `16-bit RGB565`: 16, `8-bit RGB888`: 8
    .bits_per_pixel = EXAMPLE_LCD_BIT_PER_PIXEL,        // Number of bits for the␣
↪color format, may not be equal to the data line width of the RGB interface,
                                                        // e.g., `16-bit RGB565`:␣
↪16, `8-bit RGB888`: 24
    .psram_trans_align = 64,                            // Set to `64` by default
    .num_fbs = EXAMPLE_LCD_NUM_FB,                      // Number of frame buffers␣
↪for the RGB interface, set to `1` by default, greater than `1` for multiple␣
↪buffering to prevent tearing
    .bounce_buffer_size_px = 10 * EXAMPLE_LCD_H_RES,    // Used to increase the␣
↪data transfer bandwidth of the RGB interface, usually set to `10 * EXAMPLE_LCD_H_
↪RES`
    .clk_src = LCD_CLK_SRC_DEFAULT,                     // Set to `LCD_CLK_SRC_
↪DEFAULT` by default
    .disp_gpio_num = EXAMPLE_PIN_NUM_DISP_EN,           // Pin number connected to␣
↪the LCD DISP signal, can be set to `-1` to disable
    .pclk_gpio_num = EXAMPLE_PIN_NUM_PCLK,              // Pin number connected to␣
↪the LCD PCLK signal
    .vsync_gpio_num = EXAMPLE_PIN_NUM_VSYNC,            // Pin number connected to␣
↪the LCD VSYNC signal
    .hsync_gpio_num = EXAMPLE_PIN_NUM_HSYNC,            // Pin number connected to␣
↪the LCD HSYNC signal
    .de_gpio_num = EXAMPLE_PIN_NUM_DE,                  // Pin number connected to␣
↪the LCD DE signal, can be set to `-1` to disable
    .data_gpio_nums = {                                 // Pin numbers connected␣
↪to the LCD D[15:0] signals, the valid quantity is specified by `data_width`,
                                                        // set to D[7:0] for 8-bit
        EXAMPLE_PIN_NUM_DATA0,
        EXAMPLE_PIN_NUM_DATA1,
        EXAMPLE_PIN_NUM_DATA2,
        EXAMPLE_PIN_NUM_DATA3,
        EXAMPLE_PIN_NUM_DATA4,
        EXAMPLE_PIN_NUM_DATA5,
        EXAMPLE_PIN_NUM_DATA6,
        EXAMPLE_PIN_NUM_DATA7,
        EXAMPLE_PIN_NUM_DATA8,
        EXAMPLE_PIN_NUM_DATA9,
        EXAMPLE_PIN_NUM_DATA10,
        EXAMPLE_PIN_NUM_DATA11,
        EXAMPLE_PIN_NUM_DATA12,
        EXAMPLE_PIN_NUM_DATA13,
        EXAMPLE_PIN_NUM_DATA14,
        EXAMPLE_PIN_NUM_DATA15,
    },
    .timings = {        // The following are parameters related to RGB timing,␣
↪which need to be determined based on the datasheet of the LCD driver IC and␣
↪hardware configuration
        .pclk_hz = EXAMPLE_LCD_PIXEL_CLOCK_HZ,
        .h_res = EXAMPLE_LCD_H_RES,
        .v_res = EXAMPLE_LCD_V_RES,
        .hsync_back_porch = 40,         // In DE mode, parameters related to HSYNC␣
↪and VSYNC can be adjusted according to the desired refresh rate
        .hsync_front_porch = 20,        // In SYNC mode, parameters related to␣
↪HSYNC and VSYNC need to be consistent with the configuration in the software␣
↪initialization command
```

(continues on next page)

(continued from previous page)

```
        .hsync_pulse_width = 1,
        .vsync_back_porch = 8,
        .vsync_front_porch = 4,
        .vsync_pulse_width = 1,
        .flgas = {      // Since some LCDs can configure these parameters through␣
↪hardware pins, make sure they are consistent with the configuration, but usually␣
↪set to `0`
          .hsync_idle_low = 0,     // Level when the HSYNC signal is idle, 0: high␣
↪level, 1: low level
          .vsync_idle_low = 0,     // Level when the VSYNC signal is idle, 0: high␣
↪level, 1: low level
          .de_idle_high = 0,       // Level when the DE signal is idle, 0: high␣
↪level, 1: low level
          .pclk_active_neg = 0,    // Effective edge of the clock signal, 0: rising␣
↪edge, 1: falling edge
          .pclk_idle_high = 0,     // Level when the PCLK signal is idle, 0: high␣
↪level, 1: low level
        },
    },
    .flags.fb_in_psram = 1,        // Set to `1` by default
};
ESP_ERROR_CHECK(esp_lcd_new_rgb_panel(&panel_config, &panel_handle));
ESP_ERROR_CHECK(esp_lcd_panel_reset(panel_handle));
ESP_ERROR_CHECK(esp_lcd_panel_init(panel_handle));

/* The following functions can be called as needed */
// ESP_ERROR_CHECK(esp_lcd_panel_invert_color(panel_handle, true));   // Invert␣
↪pixel color data bitwise through hardware (0xF0F0 -> 0x0F0F)
// ESP_ERROR_CHECK(esp_lcd_panel_mirror(panel_handle, true, true));   // Mirror X␣
↪and Y axes through software
// ESP_ERROR_CHECK(esp_lcd_panel_swap_xy(panel_handle, true));        // Swap X␣
↪and Y axes through software
// ESP_ERROR_CHECK(esp_lcd_panel_set_gap(panel_handle, 0, 0));        // Modify␣
↪the starting and ending coordinates for drawing through software to achieve␣
↪drawing offset
// ESP_ERROR_CHECK(esp_lcd_panel_disp_on_off(panel_handle, true));    // Control␣
↪the on/off of LCD display through the `disp_gpio_num` pin,
                                                                      // only␣
↪available when the pin is set and not equal to `-1`, otherwise an error will be␣
↪reported
```

Timing Parameter Correspondence:

The correspondence between the timing parameters in the above code and the symbols in the *DE Mode* and *SYNC Mode* timing diagrams is as follows:

Table 1: Horizontal Timing Parameter Correspondence

Code Parameter Name	Timing Diagram Symbol	Description
`hsync_back_porch`	Thbp	Horizontal sync back porch
`hsync_front_porch`	Thfp	Horizontal sync front porch
`hsync_pulse_width`	Thpw	Horizontal sync pulse width

Table 2: Vertical Timing Parameter Correspondence

Code Parameter Name	Timing Diagram Symbol	Description
`vsync_back_porch`	Tvbp	Vertical sync back porch
`vsync_front_porch`	Tvfp	Vertical sync front porch
`vsync_pulse_width`	Tvs	Vertical sync pulse width

For LCDs using both 3-wire SPI and RGB interfaces, start by creating an LCD device and obtaining a handle of

data type `esp_lcd_panel_handle_t` using the `esp_lcd_new_rgb_panel()` function from the RGB Interface Driver. Then, use the LCD General APIs to initialize the LCD device.

For configuration parameters and explanations of certain functions related to the `RGB` interface, please refer to *RGB Configuration Parameters and Function Descriptions*.

Below is an example code explanation using the ST7701S driver component:

```
#include "esp_check.h"          // Header file dependency
#include "esp_lcd_panel_ops.h"
#include "esp_lcd_panel_rgb.h"
#include "esp_lcd_panel_vendor.h"
#include "esp_lcd_st7701.h"     // Header file for the target driver component

/**
* Holds initialization commands and parameters for the LCD driver IC
*/
// static const st7701_lcd_init_cmd_t lcd_init_cmds[] = {
// //   cmd   data        data_size  delay_ms
//    {0xFF, (uint8_t []){0x77, 0x01, 0x00, 0x00, 0x13}, 5, 0},
//    {0xEF, (uint8_t []){0x08}, 1, 0},
//    {0xFF, (uint8_t []){0x77, 0x01, 0x00, 0x00, 0x10}, 5, 0},
//    {0xC0, (uint8_t []){0x3B, 0x00}, 2, 0},
//     ...
// };

/* Create LCD device */
esp_lcd_rgb_panel_config_t rgb_config = {   // Configuration parameters for the␣
↪RGB interface
    .data_width = EXAMPLE_LCD_DATA_WIDTH,               // Data line width of the␣
↪RGB interface, e.g., `16-bit RGB565`: 16, `8-bit RGB888`: 8
    .bits_per_pixel = EXAMPLE_LCD_BIT_PER_PIXEL,        // Bit depth of the color␣
↪format, may differ from the data line width of the RGB interface,
                                                        // e.g., `16-bit RGB565`:␣
↪16, `8-bit RGB888`: 24
    .psram_trans_align = 64,                            // Set to `64` by default
    .num_fbs = EXAMPLE_LCD_NUM_FB,                      // Number of frame buffers␣
↪for the RGB interface, set to `1` by default, greater than `1` for multi-
↪buffering to prevent tearing
    .bounce_buffer_size_px = 10 * EXAMPLE_LCD_H_RES,    // Used to improve data␣
↪transfer bandwidth of the RGB interface, usually set to `10 * EXAMPLE_LCD_H_RES`
    .clk_src = LCD_CLK_SRC_DEFAULT,                     // Set to `LCD_CLK_SRC_
↪DEFAULT` by default
    .disp_gpio_num = EXAMPLE_PIN_NUM_DISP_EN,           // Pin number for␣
↪connecting the LCD DISP signal, set to -1 to indicate not using
    .pclk_gpio_num = EXAMPLE_PIN_NUM_PCLK,              // Pin number for␣
↪connecting the LCD PCLK signal
    .vsync_gpio_num = EXAMPLE_PIN_NUM_VSYNC,            // Pin number for␣
↪connecting the LCD VSYNC signal
    .hsync_gpio_num = EXAMPLE_PIN_NUM_HSYNC,            // Pin number for␣
↪connecting the LCD HSYNC signal
    .de_gpio_num = EXAMPLE_PIN_NUM_DE,                  // Pin number for␣
↪connecting the LCD DE signal, set to -1 to indicate not using
    .data_gpio_nums = {                                 // Pin numbers for␣
↪connecting LCD D[15:0] signals, the valid quantity is specified by `data_width`,
                                                        // for 8-bit, set D[7:0]␣
↪is enough
        EXAMPLE_PIN_NUM_DATA0,
        EXAMPLE_PIN_NUM_DATA1,
        EXAMPLE_PIN_NUM_DATA2,
        EXAMPLE_PIN_NUM_DATA3,
        EXAMPLE_PIN_NUM_DATA4,
        EXAMPLE_PIN_NUM_DATA5,
```

(continues on next page)

(continued from previous page)

```
        EXAMPLE_PIN_NUM_DATA6,
        EXAMPLE_PIN_NUM_DATA7,
        EXAMPLE_PIN_NUM_DATA8,
        EXAMPLE_PIN_NUM_DATA9,
        EXAMPLE_PIN_NUM_DATA10,
        EXAMPLE_PIN_NUM_DATA11,
        EXAMPLE_PIN_NUM_DATA12,
        EXAMPLE_PIN_NUM_DATA13,
        EXAMPLE_PIN_NUM_DATA14,
        EXAMPLE_PIN_NUM_DATA15,
    },
    .timings = {        // The following are parameters related to RGB timing,␣
↪which need to be determined based on the data sheet of the LCD driver IC and the␣
↪configuration of software and hardware
        .pclk_hz = EXAMPLE_LCD_PIXEL_CLOCK_HZ,
        .h_res = EXAMPLE_LCD_H_RES,
        .v_res = EXAMPLE_LCD_V_RES,
        .hsync_back_porch = 40,         // In DE mode, parameters related to HSYNC␣
↪and VSYNC can be adjusted according to the desired refresh rate
        .hsync_front_porch = 20,        // In SYNC mode, parameters related to␣
↪HSYNC and VSYNC need to be consistent with the configuration in the software␣
↪initialization command
        .hsync_pulse_width = 1,
        .vsync_back_porch = 8,
        .vsync_front_porch = 4,
        .vsync_pulse_width = 1,
        .flgas = {      // Since some LCDs can configure these parameters through␣
↪hardware pins or software commands, make sure they are consistent with the␣
↪configuration,
          .hsync_idle_low = 0,     // Level of HSYNC signal when idle, 0: high␣
↪level, 1: low level
          .vsync_idle_low = 0,     // Level of VSYNC signal when idle, 0 means high␣
↪level, 1: low level
          .de_idle_high = 0,       // Level of DE signal when idle, 0: high level,␣
↪1: low level
          .pclk_active_neg = 0,    // Effective edge of the clock signal, 0: rising␣
↪edge, 1: falling edge
          .pclk_idle_high = 0,     // Level of PCLK signal when idle, 0: high level,
↪ 1: low level
        },
    },
    .flags.fb_in_psram = 1,       // Set to `1` by default
};
st7701_vendor_config_t vendor_config = {
    .rgb_config = &rgb_config,    // Configuration parameters for the RGB interface
    // .init_cmds = lcd_init_cmds,    // Used to replace the initialization␣
↪commands and parameters in the driver component
    // .init_cmds_size = sizeof(lcd_init_cmds) / sizeof(st7701_lcd_init_cmd_t),
    .flags = {          // Configuration parameters for the LCD driver IC
        .mirror_by_cmd = 1,       // If `1`, use LCD command to implement␣
↪mirroring function (esp_lcd_panel_mirror()), if `0`, implement through software
        .enable_io_multiplex = 0, // If `1`, automatically delete the interface␣
↪device when deleting the LCD device, all parameters named `*_by_cmd` should be␣
↪set to `0`,
                                  // if `0`, do not delete. If the pins of the 3-
↪wire SPI interface are multiplexed with the RGB interface, set this parameter to␣
↪`1`
    },
};
const esp_lcd_panel_dev_config_t panel_config = {
    .reset_gpio_num = EXAMPLE_LCD_IO_RST,           // IO number for connecting␣
↪the LCD reset signal, set to `-1` to indicate not using
```

(continues on next page)

(continued from previous page)

```
    .rgb_ele_order = LCD_RGB_ELEMENT_ORDER_RGB,      // Element order of pixel␣
↪color (RGB/BGR),
                                                     // generally controlled by␣
↪command `LCD_CMD_MADCTL(36h)`
    .bits_per_pixel = EXAMPLE_LCD_BIT_PER_PIXEL,     // Bit depth of the color␣
↪format (RGB565: 16, RGB666: 18, RGB888: 24),
                                                     // generally controlled by␣
↪command `LCD_CMD_COLMOD(3Ah)`
    .vendor_config = &vendor_config,                 // Configuration parameters␣
↪for the RGB interface and LCD driver IC
};
esp_lcd_panel_handle_t panel_handle = NULL;
ESP_ERROR_CHECK(esp_lcd_new_panel_st7701(io_handle, &panel_config, &panel_handle));

/* Initialize LCD device */
ESP_ERROR_CHECK(esp_lcd_new_rgb_panel(&panel_config, &panel_handle));
ESP_ERROR_CHECK(esp_lcd_panel_reset(panel_handle));
ESP_ERROR_CHECK(esp_lcd_panel_init(panel_handle));
// The following functions can be used as needed
// ESP_ERROR_CHECK(esp_lcd_panel_invert_color(panel_handle, true));
// ESP_ERROR_CHECK(esp_lcd_panel_mirror(panel_handle, true, true));
// ESP_ERROR_CHECK(esp_lcd_panel_swap_xy(panel_handle, true));
// ESP_ERROR_CHECK(esp_lcd_panel_set_gap(panel_handle, 0, 0));
// ESP_ERROR_CHECK(esp_lcd_panel_disp_on_off(panel_handle, true));
```

For LCDs using both 3-wire SPI and RGB interfaces, start by creating an LCD device using the ported driver component and obtaining a handle of data type `esp_lcd_panel_handle_t`. Then, use the LCD General APIs to initialize the LCD device.

For more detailed information on the configuration parameters for the `RGB` interface, please refer to the ESP-IDF Programming Guide. Below are some instructions on using the `esp_lcd_panel_draw_bitmap()` function to refresh RGB LCD images:

- This function refreshes the image data in the frame buffer through memory copy. In other words, after calling this function, the image data in the frame buffer is already updated. The `RGB` interface itself refreshes the LCD by obtaining image data from the frame buffer through DMA, and these two processes are asynchronous.
- The function checks whether the value of the passed parameter `color_data` is the internal frame buffer address of the `RGB` interface. If it is, the memory copy operation mentioned above will not be performed. Instead, the DMA transfer address of the `RGB` interface is directly set to this buffer address, achieving the switching function in the case of multiple frame buffers.

In addition to the LCD common APIs, the RGB interface driver also provides some functions for special features. Here are usage instructions for some commonly used functions:

- `esp_lcd_rgb_panel_set_pclk()`: Dynamically modifies the clock frequency, can be used after LCD initialization.
- `esp_lcd_rgb_panel_restart()`: Resets data transfer, used to restore normal operation when the screen is offset.
- `esp_lcd_rgb_panel_get_frame_buffer()`: Gets the address of the frame buffer. The available quantity is determined by the configuration parameter `num_fbs` and is used for multi-buffering to prevent tearing.
- `esp_lcd_rgb_panel_register_event_callbacks()`: Registers callback functions for various events. Example code and explanations are as follows:

```
static bool example_on_vsync_event(esp_lcd_panel_handle_t panel, const␣
↪esp_lcd_rgb_panel_event_data_t *edata, void *user_ctx)
{
    /* Perform some operations here */

    return false;
}
```

(continues on next page)

(continued from previous page)

```
static bool example_on_bounce_event(esp_lcd_panel_handle_t panel,␣
↪const esp_lcd_rgb_panel_event_data_t *edata, void *user_ctx)
{
    /* Perform some operations here */

    return false;
}

esp_lcd_rgb_panel_event_callbacks_t cbs = {
    .on_vsync = example_on_vsync_event,                 // Callback␣
↪function when a frame of image is refreshed
    .on_bounce_frame_finish = example_on_bounce_event,  // Callback␣
↪function when a frame of image is transferred through the Bounce␣
↪Buffer mechanism
                                                        // Note that the RGB␣
↪interface has not completed the transmission of this frame at this␣
↪time
};
ESP_ERROR_CHECK(esp_lcd_rgb_panel_register_event_callbacks(panel_
↪handle, &cbs, &example_user_ctx));
```

Related Documentation

- ST7701S Datasheet
- ST77903 Datasheet
- GC9503 Datasheet

4.1.6 MIPI DSI LCD Detailed Guide

Table of Contents

- *Glossary*
  - *MIPI-DSI Related Terms*
- *MIPI-DSI Interface and Protocol Introduction*
  - *MIPI-DSI Interface Overview*
  - *MIPI-DSI Screen Refresh Principle*
  - *MIPI-DSI Protocol*
- *Color Format*
  - *Description*
  - *Common Color Formats and Data Types*
- *MIPI-DSI Driver Flow*
  - *DSI, DBI, and DPI*
  - *Initialize DSI Bus*
  - *Configure DBI Interface*
  - *Port Driver Component*
  - *Initialize LCD Device*
- *Common Issues*
  - *Issue: No image display on screen with no other error logs*
  - *Issue: Screen initialization commands cannot be sent normally*
  - *Issue: Screen image distortion and misalignment*
  - *Issue: Screen flashes blue screen with log: "can't fetch data from external memory fast enough, underrun happens"*

- *Issue: How to resolve screen tearing*
- *Issue: Frame rate optimization configuration*
- *Issue: How to drive default 4-lane screens*
- *Issue: How to confirm the lane bit rate range supported by the screen*

• *Related Documentation and Examples*

Glossary

Please refer to *LCD Terms Table*.

MIPI-DSI Related Terms

Abbreviation	Full Name	Description
MIPI-DSI	MIPI Display Serial Interface	High-speed serial interface standard developed by MIPI Alliance, specifically designed for connecting processors and display devices
DCS	Display Command Set	Standardized display command set defined by MIPI-DSI for controlling display modules
DBI	Display Bus Interface	Encapsulates DSI' s Command Mode into esp-lcd' s control IO layer
DPI	Display Pixel Interface	Encapsulates DSI' s Video Mode into esp-lcd' s data panel layer
HS	High-Speed Mode	High-speed mode, mainly used for transmitting display frame data, typically 80 Mbps ~ 1.5 Gbps/Lane
LP	Low-Power Mode	Low-power mode, typically used for sending DCS commands or controlling link states, maximum rate approximately 10 MHz
HS-TX/HS-RX	High-Speed Transmitter/Receiver	High-speed transmitter/receiver
LP-TX/LP-RX	Low-Power Transmitter/Receiver	Low-power transmitter/receiver
LP-CD	Low-Power Contention Detector	Low-power contention detector
BTA	Bus Turnaround	Bus turnaround for bidirectional communication
ULPS	Ultra Low Power State	Ultra low power state for maximum energy savings
LPDT	Low Power Data Transmission	Low power data transmission mode
ESC	Escape Mode	Escape mode, low-power operation mode of MIPI-DSI
SoT	Start of Transmission	Start of transmission signal
EoT	End of Transmission	End of transmission signal
DI	Data Identifier	Data identifier for identifying packet types
ECC	Error Correction Code	Error detection code for data integrity verification
DT	Data Type	Data type for identifying pixel data format
HSYNC	Horizontal Synchronization	Horizontal synchronization signal
VSYNC	Vertical Synchronization	Vertical synchronization signal
DMA	Direct Memory Access	Direct memory access for efficient data transfer
PSRAM	Pseudo Static RAM	Pseudo static random access memory
RGB565	Red Green Blue 565	16-bit color format, 5 bits red, 6 bits green, 5 bits blue
RGB666	Red Green Blue 666	18-bit color format, 6 bits each for red, green, and blue
RGB888	Red Green Blue 888	24-bit color format, 8 bits each for red, green, and blue

MIPI-DSI Interface and Protocol Introduction

MIPI-DSI Interface Overview MIPI-DSI (MIPI Display Serial Interface) is a high-speed serial interface standard developed by the MIPI Alliance, specifically designed for connecting processors and display devices. It features:

- High-speed transmission: Each data channel can support transmission rates up to several Gbps
- Low power consumption: Uses low-voltage differential signaling and enters low-power mode during idle or low-speed transmission
- Pin reduction: Significantly reduces the number of signal lines compared to parallel RGB interfaces
- Scalability: Supports 1 to 4 data channels
- High compatibility: Supports various resolutions and display devices

Currently, ESP32-P4 chips feature MIPI-DSI interfaces, supporting 2-lane × 1.5Gbps and using Video mode for video stream output.

MIPI-DSI Screen Refresh Principle MIPI-DSI is a commonly used display interface standard for connecting processors and displays. It supports two main data transmission modes:

1. Command Mode
2. Video Mode

ESP32-P4 currently only supports Video mode for video stream output.

In Video Mode, the host transmits data in continuous video frames, strictly synchronized with the display' s refresh cycle (such as VSync). The display does not require frame buffer memory, simplifying hardware design. This mode is widely used for high-resolution, high-refresh-rate display scenarios.

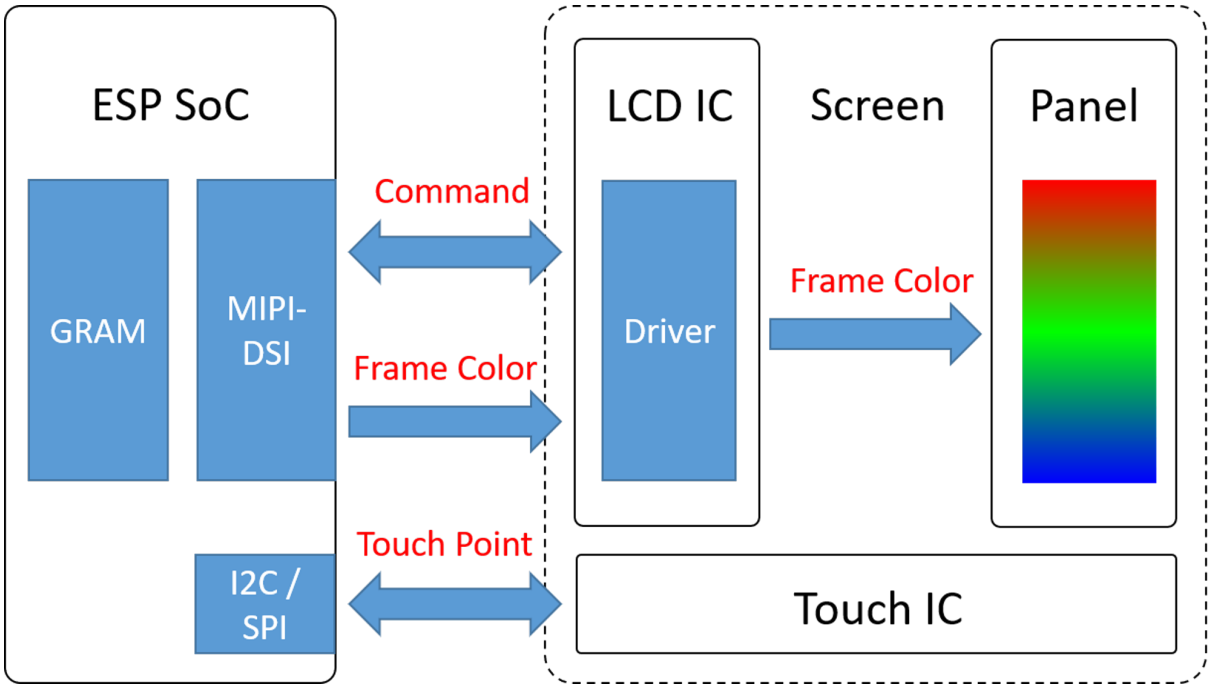


Fig. 31: MIPI-DSI Screen Refresh Process

The figure above shows the MIPI-DSI screen refresh process, where:

- Command data: MIPI-DCS commands are transmitted to the LCD controller through DBI for initializing registers and controlling display functions.
- Frame color data: In MIPI-DSI Video mode, pixel data is continuously output from the host-side frame buffer (internal/external RAM) to the LCD panel for display through DMA, while the LCD driver IC does not require GRAM.
- Touch data: Touch signals are detected by the touchscreen and transmitted back to the host system through I^2C or SPI interfaces for processing.

MIPI-DSI Protocol MIPI-DSI can be roughly divided into Physical Layer, Protocol Layer, and Display Command Layer (DCS Layer), each with specific functions that work together to complete command and pixel data transmission from the host to the display.

The collaborative workflow of each layer can be summarized as shown in the following figure:

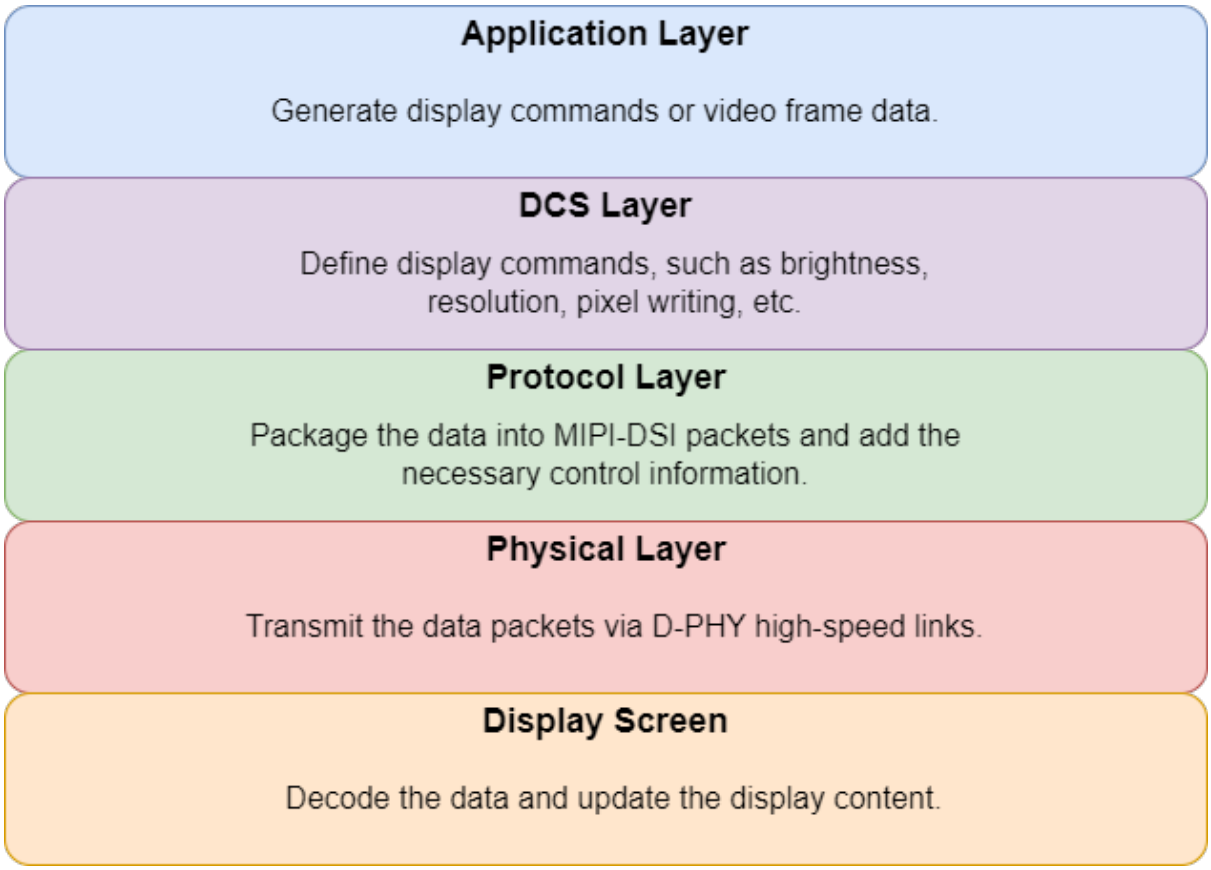


Fig. 32: MIPI-DSI Protocol Layers

Physical Layer

- Function: Responsible for actual data transmission, serving as the bottom layer of the protocol stack. Uses MIPI D-PHY to provide high-speed differential signal transmission, supporting multi-channel data transmission.
- Transmission modes:
  - High-Speed Mode: Mainly used for transmitting display frame data, can also transmit commands, typically 80 Mbps ~ 1.5 Gbps/Lane.
  - Low-Power Mode: Typically used for sending DCS commands (in command mode) or controlling link states, maximum rate approximately 10 MHz (specific depends on chip support).
- Configuration:
  - Clock Lane: Provides synchronization signals.
  - Data Lane: Transmits actual display data.
- Detailed explanation of Lane Module, i.e., D-PHY:
  - Types

Lane Type/Transmission Role	Master	Slave
Unidirectional Clock Lane	HS-TX, LP-TX	HS-RX, LP-RX
Unidirectional Data Lane	HS-TX, LP-TX	HS-RX, LP-RX
Bidirectional Data Lane	HS-TX, LP-TX, HS-RX, LP-RX, LP-CD	HS-TX, LP-TX, HS-RX, LP-RX, LP-CD

Note: Module function descriptions:

- **LP-TX/LP-RX**: Low-power transmitter/receiver
- **HS-TX/HS-RX**: High-speed transmitter/receiver
- **LP-CD**: Low-power contention detector

Clock lanes are typically unidirectional, provided by Master and received by Slave. Bidirectional lanes and LP-CD are mainly used for low-power communication in command mode.

- States
  - LP States (LP-00, LP-01, LP-10, LP-11): Use single-ended signals, mainly for low-speed command transmission, control signals, initialization, or standby states. Different LP states are used for encoding and link control.
  - HS States (HS-0, HS-1): Use differential signals, mainly for high-speed transmission of video data or frame content. HS-0 and HS-1 correspond to logical levels on differential pairs, only valid on HS-TX/HS-RX.
- Typical voltages
  - LP: Typical voltage approximately 0 –1.2 V
  - HP: Typical differential voltage approximately 100 –300 mV
- Operation modes
  - Escape Mode: Only available in Low-Power Mode, MIPI-DSI’ s low-power operation mode, used for special data transmission or triggering specific functions. Supports the following functions:

Function Name	Description	Command
LPDT (Low Power Data Transmission)	Transmit data in low-power mode	0x87
ULPS (Ultra Low Power State)	Enter ultra low power state for maximum energy savings	0x84
Trigger Command	Trigger specific events, such as state switching or mode adjustment	0x85

    * Entry process (signal state machine transition): LP-11 → LP-10 → LP-00 → LP-01 → LP-00
    * Exit process: LP-10 → LP-11
  - High-Speed (Burst) Mode: Always in High-Speed Mode, used for high-speed serial data transmission, suitable for high bandwidth requirements.
    * Entry process: LP-11 → LP-01 → LP-00 → SoT (Start of Transmission) → HSD (80Mbps ~ 1.5Gbps)
    * Exit process: EoT (End of Transmission) → LP-11
  - Control Mode: Default in Low-Power Mode, used for command transmission in stop state and bidirectional communication (BTA, Bus Turnaround).
    * BTA transmission process: LP-11 → LP-10 → LP-00 → LP-10 → LP-00 (bus turnaround state transition)
    * BTA receive process: LP-00 → LP-10 → LP-11

Important Signal Descriptions

During MIPI-DSI communication, two critical Stop signals ensure proper bus state management:

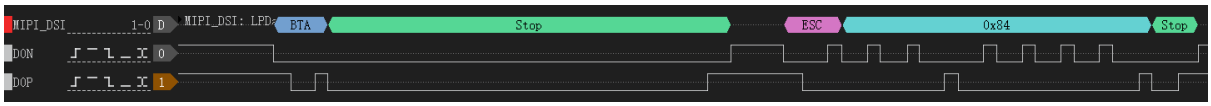


Fig. 33: MIPI-DSI Stop Signal Timing Diagram

1. **BTA-Stop Signal**
   Sent by the slave after completing Bus Turnaround response, indicating:
   - Bus turnaround operation completed successfully
   - Slave has released bus control

- Host can regain bus control

2. **ESC-0X84-Stop Signal**
   Used to indicate Lane stop signal transmission, specific functions include:
   - Enter system into ultra low power state (ULPS)
   - Corresponds to 0x84 command in Escape Mode
   - Achieve maximum energy management

Tip: These Stop signals are key mechanisms in the MIPI-DSI protocol for ensuring communication reliability and power management.

Protocol Layer

- Function: Implements data packaging and parsing, defines transmission formats (such as DCS and video data packets), and provides error detection mechanisms to ensure data correctness and reliability.
- Data frame structure
  - Short Packet:
    - Length: Fixed length of 4 bytes
    - Composition: Data Identifier (DI), 1 byte; Frame data, 2 bytes; Error detection (ECC), 1 byte.

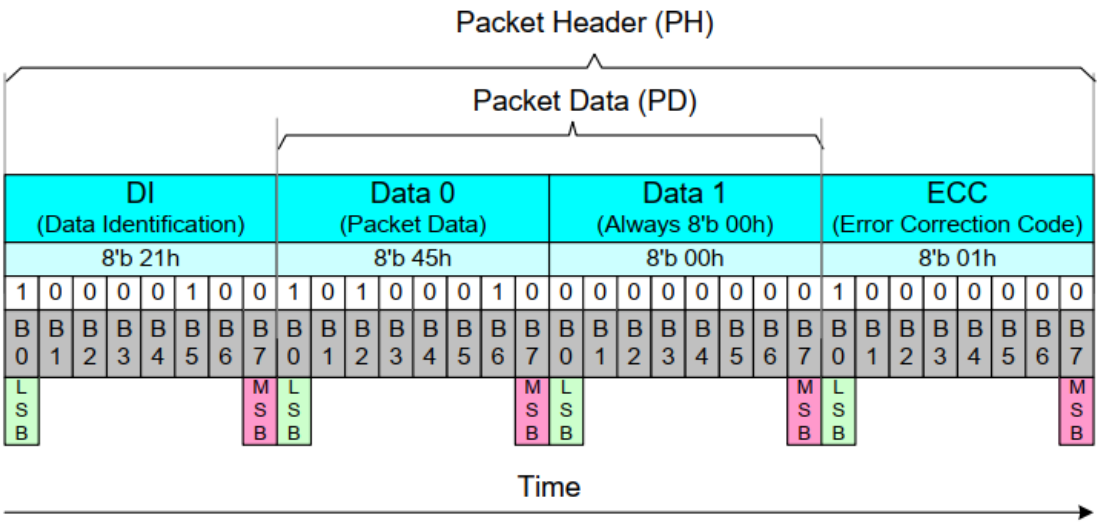


Fig. 34: MIPI-DSI Short Packet Composition

    - Example:

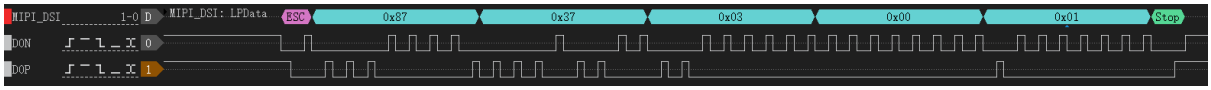


Fig. 35: MIPI-DSI Short Packet Timing

    The key fields in order are:
    - ESC: Enter Escape mode
    - 0x87: LPDT (Low Power Data Transmission mode)
    - 0x37: Data identifier, corresponding to DCS command "Set Maximum Return Packet Size"; Function: Set the maximum size of return data packets when the host receives data from the slave
    - 0x03, 0x00: Frame data, indicating setting the maximum return packet size to 3 bytes
    - 0x01: ECC (Error Correction Code)
  - Long Packet:
    - Length: Variable length, 6 to 65541 bytes.

* Composition: Frame header (Data Identifier (DI) 1 byte; Data Count (WC) 2 bytes; Error detection (ECC) 1 byte); Data payload (0 to 65535 bytes); Frame tail (Checksum 2 bytes).

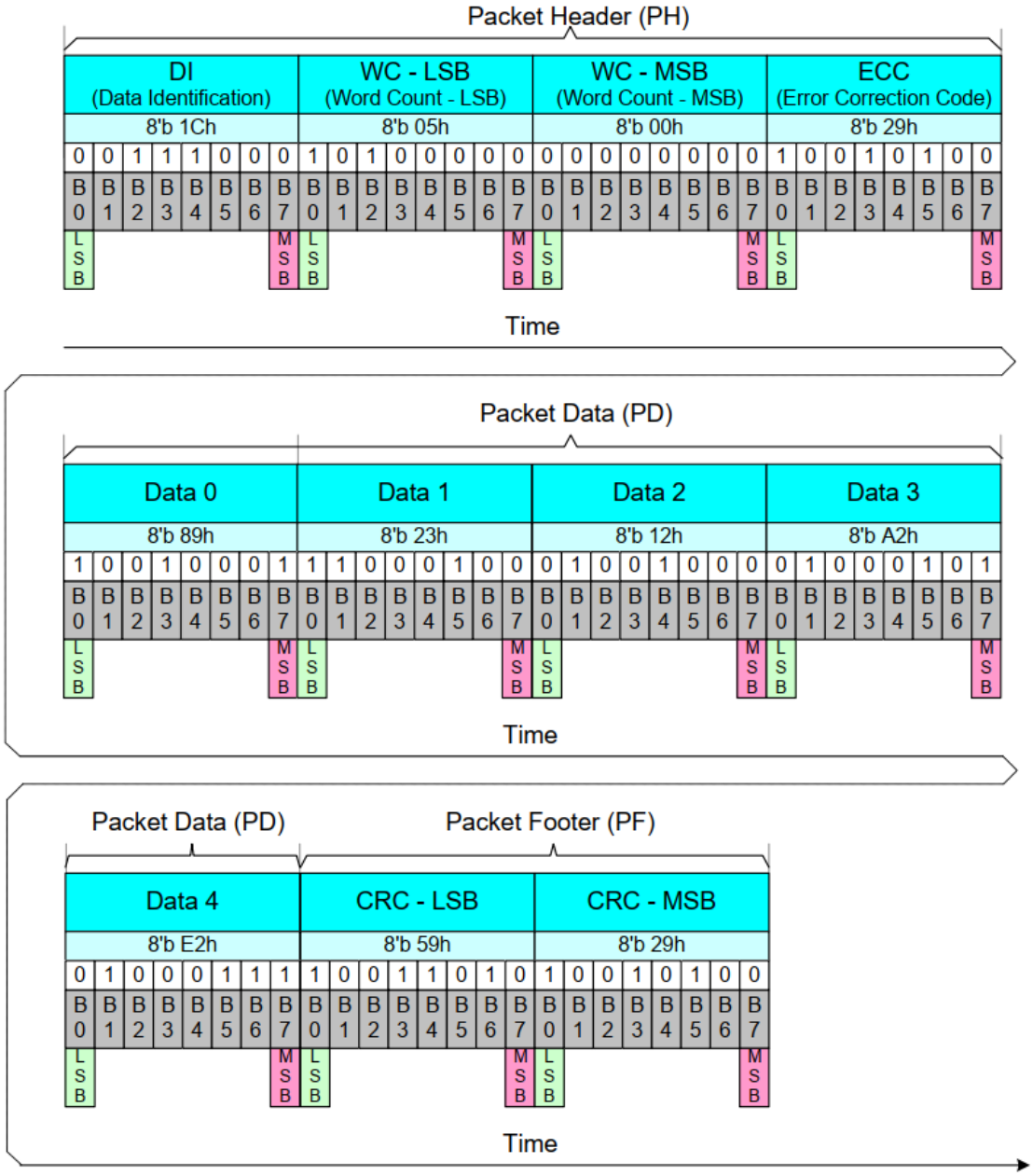


Fig. 36: MIPI-DSI Long Packet Composition

* Example:

The key fields in order are:

* ESC: Enter Escape mode
* 0x87: LPDT (Low Power Data Transmission mode)
* 0x39: Data identifier, corresponding to DCS command "DCS Write Long"
* 0x03, 0x00: Data count, indicating sending 3 bytes of data commands
* 0x09: ECC (Error Correction Code)
* 0xB6, 0xB2, 0xB2: Actual data commands being sent
* 0xEF, 0xFA: Checksum field

- Common Data Identifiers (DI)

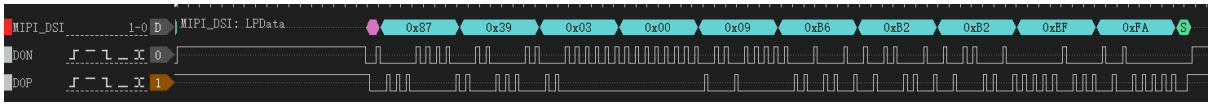


Fig. 37: MIPI-DSI Long Packet Timing

Data ID	Name	Description
0x05	DCS Short Write, 0 Param	Send a DCS command with no parameters (e.g., Sleep Out *0x11*)
0x15	DCS Short Write, 1 Param	Send a DCS command with 1 parameter
0x39	DCS Long Write	Send a DCS command with multiple bytes of data (commonly used for memory writes)
0x06	DCS Short Read, 0 Param	Read a DCS status or register with no parameters
0x16	DCS Short Read, 1 Param	Read a DCS status or register with 1 parameter
0x37	DCS Read Response	DCS read data returned by the display
0x03	Generic Short Write, 0 Param	Send generic command with no parameters
0x13	Generic Short Write, 1 Param	Send generic command with 1 parameter
0x23	Generic Short Write, 2 Param	Send generic command with 2 parameters
0x29	Generic Long Write	Send generic command with multiple bytes of data
0x04	Generic Read, 0 Param	Generic read command with no parameters
0x14	Generic Read, 1 Param	Generic read command with 1 parameter
0x24	Generic Read, 2 Param	Generic read command with 2 parameters

Display Command Layer (DCS Layer)

- Function: DCS (Display Command Set) is a standardized display command set defined by MIPI-DSI. The host interacts with the display through these commands to implement basic display module control (such as power on/off, sleep, backlight, display on/off), display parameter configuration (such as pixel format, display mode, address setting), and frame buffer access (such as writing image data, reading status information). DCS commands are transmitted on the DSI bus in the form of short write, long write, and read commands, serving as the main control interface between the application layer and display hardware.
- Common commands:

Command Code	Name	Function Description
0x01	Software Reset	Software reset
0x11	Sleep Out	Exit sleep mode
0x28	Display Off	Turn off display
0x29	Display On	Turn on display
0x2A	Column Address Set	Set column address range
0x2B	Page Address Set	Set page address range
0x2C	Memory Write	Write frame buffer data
0x2E	Memory Read	Read frame buffer data
0x36	Address Mode	Set memory scan direction and flip mode
0x3A	Pixel Format Set	Set pixel format (e.g., RGB565/888)

Color Format

Most MIPI-DSI LCDs support multiple input color formats, such as **RGB565, RGB666, RGB888**. The host can configure the display color depth through **DCS command 0x3A (Pixel Format Set)**, for example:

- When setting to **RGB565 (16-bit)**, send `Command: 0x3A`, `Parameter: 0x55` (specific parameter values need to refer to the LCD controller datasheet).

	Display image data (24 bits)																							
	R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0
24-bit	R[7]	R[6]	R[5]	R[4]	R[3]	R[2]	R[1]	R[0]	G[7]	G[6]	G[5]	G[4]	G[3]	G[2]	G[1]	G[0]	B[7]	B[6]	B[5]	B[4]	B[3]	B[2]	B[1]	B[0]
18-bit EPF[1:0]=00 (Note 1)	R[5]	R[4]	R[3]	R[2]	R[1]	R[0]	0	0	G[5]	G[4]	G[3]	G[2]	G[1]	G[0]	0	0	B[5]	B[4]	B[3]	B[2]	B[1]	B[0]	0	0
18-bit EPF[1:0]=01 (Note 2)	R[5]	R[4]	R[3]	R[2]	R[1]	R[0]	1	1	G[5]	G[4]	G[3]	G[2]	G[1]	G[0]	1	1	B[5]	B[4]	B[3]	B[2]	B[1]	B[0]	1	1
18-bit EPF[1:0]=10	R[5]	R[4]	R[3]	R[2]	R[1]	R[0]	R[5]	R[4]	G[5]	G[4]	G[3]	G[2]	G[1]	G[0]	G[5]	G[4]	B[5]	B[4]	B[3]	B[2]	B[1]	B[0]	B[5]	B[4]
16-bit EPF[1:0]=00 (Note 3)	R[4]	R[3]	R[2]	R[1]	R[0]	0	0	0	G[5]	G[4]	G[3]	G[2]	G[1]	G[0]	0	0	B[4]	B[3]	B[2]	B[1]	B[0]	0	0	0
16-bit EPF[1:0]=01 (Note 4)	R[4]	R[3]	R[2]	R[1]	R[0]	1	1	1	G[5]	G[4]	G[3]	G[2]	G[1]	G[0]	1	1	B[4]	B[3]	B[2]	B[1]	B[0]	1	1	1
16-bit EPF[1:0]=10	R[4]	R[3]	R[2]	R[1]	R[0]	R[4]	R[3]	R[2]	G[5]	G[4]	G[3]	G[2]	G[1]	G[0]	G[5]	G[4]	B[4]	B[3]	B[2]	B[1]	B[0]	B[4]	B[3]	B[2]

Fig. 38: MIPI-DSI Color Format

Pixel data is typically transmitted through **Long Packets**, which contain pixel data and corresponding Data Types. For example, RGB565 format pixel streams generally use **DT = 0x0E (Packed Pixel Stream, 16-bit)**.

Description

- **16-bit and 18-bit** formats are commonly used to reduce bandwidth and storage usage
- **24-bit** format is suitable for high-quality display scenarios

Common Color Formats and Data Types

Color Format	Bit Depth	DCS Parameter (Example)	Data Type (DT)
RGB565	16-bit	0x55	0x0E
RGB666	18-bit	0x66	0x1E / 0x2E
RGB888	24-bit	0x77	0x3E

MIPI-DSI Driver Flow

The MIPI-DSI LCD driver flow can be roughly divided into four parts: initializing the DSI bus, configuring the DBI interface, porting the driver component, and initializing the LCD device.

DSI, DBI, and DPI

- DSI: Responsible for low-level initialization of PHY, Host, and Bridge, serving as the common "bus" for DBI and DPI.
- DBI: Encapsulates DSI's Command Mode into esp-lcd's control IO layer
- DPI: Encapsulates DSI's Video Mode into esp-lcd's data panel layer, responsible for continuous refresh from frame buffer to screen

Initialize DSI Bus

```
esp_lcd_dsi_bus_handle_t dsi_bus = NULL;
esp_lcd_dsi_bus_config_t bus_config = {
    .bus_id = 0,                                    // DSI controller number
    .num_data_lanes = 2,                            // Number of data lanes (1-2)
    .phy_clk_src = MIPI_DSI_PHY_CLK_SRC_DEFAULT,    // PHY clock source
    .lane_bit_rate_mbps = 1300,                     // Bit rate per lane (Mbps)
};
ESP_ERROR_CHECK(esp_lcd_new_dsi_bus(&bus_config, &dsi_bus));
```

The number of data lanes (num_data_lanes) depends on the number of lanes used and supported by the LCD IC. Some LCD ICs can configure the number of lanes used through registers, while some LCD ICs have a fixed number of lanes. It's worth noting that ESP32-P4 supports a maximum of 2 lanes, supporting both 1-lane and 2-lane LCD channel configurations.

Calculation method for bit rate per lane (lane_bit_rate_mbps):

- Example calculation:
  - Resolution: 800×1280 (hspw=4, hbp=20, hfp=20; vspw=4, vbp=20, vfp=20)
  - Refresh rate: 60Hz
  - Color depth: 24bpp
  - Number of data lanes: 2
- Calculate raw data rate
  - pixel_clock = horizontal_total_pixels × vertical_total_pixels × refresh_rate
  - raw_data_rate = pixel_clock × bits_per_pixel
    Example calculation:
    - pixel_clock = (800+20+20+4) × (1280+20+20+4) × 60 = 62.4MHz

    * raw_data_rate = 62.4M × 24 = 1497.6Mbps
- Consider DSI protocol overhead
  - 8b/10b encoding: actual data rate needs to be multiplied by 1.25
  - Protocol header and ECC: additional overhead of approximately 10-15%
  - protocol_data_rate = raw_data_rate × 1.25 × 1.15
    Example calculation:
    * protocol_data_rate = 1497.6 × 1.25 × 1.15 = 2153.55Mbps
- Distribute to each data lane
  - lane_bit_rate = protocol_data_rate ÷ num_data_lanes
    Example calculation:
    * lane_bit_rate = 2153.55 ÷ 2 = 1076.78Mbps
- Reserve bandwidth margin (recommended 20%)
  - final_bit_rate = lane_bit_rate × 1.2
    Example calculation:
    * final_bit_rate = 1076.78 × 1.2 ≈ 1292Mbps

Therefore, it is recommended to set lane_bit_rate_mbps to final_bit_rate, which is around *1300* in the example calculation scenario. Note that on the ESP32P4 platform, lane_bit_rate_mbps cannot exceed *1500*, and the recommended minimum should not be below *480*.

Configure DBI Interface

```
esp_lcd_panel_io_handle_t io_handle = NULL;
esp_lcd_dbi_io_config_t io_config = {
    .virtual_channel = 0,            // Virtual channel number (0-3), most LCDs␣
↪only support channel `0`, usually set to `0`
    .lcd_cmd_bits = 8,               // Command bit width
    .lcd_param_bits = 8,             // Parameter bit width
};
ESP_ERROR_CHECK(esp_lcd_new_panel_io_dbi(dsi_bus, &io_config, &io_handle));
```

The command and parameter bit widths must match the panel specifications, and virtual channels are used to distinguish multiple devices.

Port Driver Component The basic principles for porting MIPI-DSI LCD driver components include the following three points:

1. Send commands and parameters in specified formats based on interface device handles of type `esp_lcd_dbi_io_config_t`.
2. Implement and create an LCD device, then implement the functions in the esp_lcd_panel_t structure by registering callback functions.
3. Implement a function to provide LCD device handles of type `esp_lcd_panel_handle_t`, enabling applications to use LCD General APIs to operate the LCD device

Below is the implementation description of each function in `esp_lcd_panel_handle_t` and its correspondence with LCD General APIs:

For most MIPI-DSI LCDs, their driver IC commands and parameters are compatible with the implementation descriptions above, so porting can be completed through the following steps:

1. Select a MIPI-DSI LCD driver component with a similar model from the LCD Driver Components.
2. Confirm whether the commands and parameters used in each function of the selected component are consistent with the target LCD driver IC by consulting the target LCD driver IC datasheet. If not, modify the relevant code.
3. Even for LCD driver ICs of the same model, screens from different manufacturers usually require their own initialization command configurations. Therefore, it is necessary to modify the commands and parameters sent in the initialization function `init()`. These initialization commands are usually stored in a static array in a specific format. Additionally, be careful not to include special commands in the initialization commands, such as `LCD_CMD_COLMOD(3Ah)` and `LCD_CMD_MADCTL(36h)`, which are managed and used by the driver component.

4. Use the editor's character search and replace function to replace the LCD driver IC name in the component with the target name, such as replacing `ek79007` with `ili9881`

Initialize LCD Device The following code example uses EK79007 as an example:

```
#include "esp_lcd_panel_vendor.h"   // Required header file
#include "esp_lcd_panel_ops.h"
#include "esp_lcd_ek79007.h"        // Target driver component header file

// static const ek79007_lcd_init_cmd_t lcd_init_cmds[] = {
//  {cmd, { data }, data_size, delay_ms}
// {0xE0, (uint8_t []){0x00}, 1, 0},
// {0xE1, (uint8_t []){0x93}, 1, 0},
// {0xE2, (uint8_t []){0x65}, 1, 0},
// {0xE3, (uint8_t []){0xF8}, 1, 0},
//     ...
// };

    ESP_LOGI(TAG, "MIPI DSI PHY Powered on");
    esp_ldo_channel_handle_t ldo_mipi_phy = NULL;
    esp_ldo_channel_config_t ldo_mipi_phy_config = {
        .chan_id = 3,
        .voltage_mv = 2500,
    };
    ESP_ERROR_CHECK(esp_ldo_acquire_channel(&ldo_mipi_phy_config, &ldo_mipi_phy));

    ESP_LOGI(TAG, "Initialize MIPI DSI bus");
    esp_lcd_dsi_bus_handle_t mipi_dsi_bus = NULL;
    esp_lcd_dsi_bus_config_t bus_config = EK79007_PANEL_BUS_DSI_2CH_CONFIG();
    ESP_ERROR_CHECK(esp_lcd_new_dsi_bus(&bus_config, &mipi_dsi_bus));

    ESP_LOGI(TAG, "Install panel IO");
    esp_lcd_panel_io_handle_t mipi_dbi_io = NULL;
    esp_lcd_dpi_panel_config_t dpi_config = {
        .virtual_channel = 0,
        .dpi_clk_src = MIPI_DSI_DPI_CLK_SRC_DEFAULT,
        .dpi_clock_freq_mhz = 60,                   // Pixel clock frequency
        .in_color_format = LCD_COLOR_FMT_RGB888,    // Color format
        .video_timing = {
            .h_size = 800,                          // Horizontal resolution
            .v_size = 1280,                         // Vertical resolution
            .hsync_pulse_width = 4,                 // HSYNC pulse width
            .hsync_back_porch = 20,                 // Horizontal back porch
            .hsync_front_porch = 20,                // Horizontal front porch
            .vsync_pulse_width = 4,                 // VSYNC pulse width
            .vsync_back_porch = 20,                 // Vertical back porch
            .vsync_front_porch = 20,                // Vertical front porch
        },
        .flags.use_dma2d = true,                    // Use 2D DMA acceleration
    };
    ESP_ERROR_CHECK(esp_lcd_new_panel_io_dbi(mipi_dsi_bus, &dbi_config, &mipi_dbi_
↪io));

    ESP_LOGI(TAG, "Install EK79007S panel driver");
    esp_lcd_panel_handle_t panel_handle = NULL;
    ek79007_vendor_config_t vendor_config = {
        .flags = {
            .use_mipi_interface = 1,
        },
        .mipi_config = {
            .dsi_bus = mipi_dsi_bus,
            .dpi_config = &dpi_config,
```

(continues on next page)

(continued from previous page)

```
        },
    };
    const esp_lcd_panel_dev_config_t panel_config = {
        .reset_gpio_num = EXAMPLE_LCD_IO_RST,            // Set to -1 if not use
        .rgb_ele_order = LCD_RGB_ELEMENT_ORDER_RGB,      // Implemented by LCD␣
↪command `36h`
        .bits_per_pixel = EXAMPLE_LCD_BIT_PER_PIXEL,     // Implemented by LCD␣
↪command `3Ah` (16/18/24)
        .vendor_config = &vendor_config,
    };
    ESP_ERROR_CHECK(esp_lcd_new_panel_ek79007(mipi_dbi_io, &panel_config, &panel_
↪handle));
    ESP_ERROR_CHECK(esp_lcd_panel_reset(panel_handle));
    ESP_ERROR_CHECK(esp_lcd_panel_init(panel_handle));
    ESP_ERROR_CHECK(esp_lcd_panel_disp_on_off(panel_handle, true));
```

Timing Parameter Description:

The timing parameter definitions for MIPI-DSI are consistent with the *SYNC Mode* of the RGB interface in terms of timing diagrams. Both use HSYNC and VSYNC synchronization signals and require configuration of blanking porch parameters. For detailed explanations of timing parameters, the correspondence between parameters and timing diagram symbols, and configuration methods, please refer to the "Timing Parameter Correspondence" section in *RGB LCD Introduction*.

Note that ESP32P4 requires a stable 2.5V power supply for the MIPI DSI PHY. The clock frequency calculation refers to the pixel_clock calculation method mentioned above. Color format, resolution, pulse width, and front/back porch parameters must strictly follow the panel datasheet requirements.

Then create an LCD device through the ported driver component and obtain a handle of type `esp_lcd_panel_handle_t`, then use LCD General APIs to initialize the LCD device.

For more detailed explanations of `MIPI-DSI` interface configuration parameters, please refer to the ESP-IDF Programming Guide. Below are some explanations about using the function `esp_lcd_panel_draw_bitmap()` to refresh LCD images:

- This function refreshes image data in the frame buffer through memory copy, meaning that after the function call completes, the image data in the frame buffer has also been updated. The `MIPI-DSI` interface itself uses DMA to fetch image data from the frame buffer to refresh the LCD, and these two processes are asynchronous.
- This function will check whether the incoming parameter `color_data` value is the internal frame buffer address of the `MIPI-DSI` interface. If so, it will not perform the above memory copy operation, but directly set the DMA transfer address of the `MIPI-DSI` interface to that buffer address, thus achieving switching functionality in cases with multiple frame buffers.

In addition to LCD General APIs, the MIPI-DSI Interface Driver also provides some special function functions. Below are usage instructions for some commonly used functions:

- esp_lcd_dpi_panel_get_frame_buffer(): Get frame buffer addresses, available quantity determined by configuration parameter `num_fbs`, used for multi-buffer anti-tearing.
- esp_lcd_dpi_panel_set_pattern(): Set predefined patterns to the screen for testing or debugging purposes.
- esp_lcd_dpi_panel_set_color_conversion(): Set color conversion configuration for DPI panel.
- esp_lcd_dpi_panel_register_event_callbacks(): Register callback functions for various events, example code and explanation as follows:

```
static bool example_on_color_trans_event(esp_lcd_panel_handle_t panel, const esp_
↪lcd_dpi_panel_event_callbacks_t *edata, void *user_ctx)
{
    /* Operations can be performed here */

    return false;
}

static bool example_on_refresh_event(esp_lcd_panel_handle_t panel, const esp_lcd_
↪dpi_panel_event_callbacks_t *edata, void *user_ctx)
```

(continues on next page)

(continued from previous page)

```
{
    /* Operations can be performed here */

    return false;
}

esp_lcd_dpi_panel_event_callbacks_t cbs = {
    .on_color_trans_done = example_on_color_trans_event,   // Triggered when user
↪'s color buffer is copied to internal frame buffer
    .on_refresh_done = example_on_refresh_event,               // Triggered when␣
↪internal frame buffer content completes refresh and displays on screen
};
ESP_ERROR_CHECK(esp_lcd_dpi_panel_register_event_callbacks(panel_handle, &cbs, &
↪example_user_ctx));
```

Common Issues

Issue: No image display on screen with no other error logs Check if SPIRAM SPEED is configured to 200M. Driving MIPI-DSI interface screens requires higher PSRAM bandwidth.

Issue: Screen initialization commands cannot be sent normally The chip's DSI data pins (DSI_DATAN0 and DSI_DATAP0) and clock pins (DSI_CLKN and DSI_CLKP) are not correctly connected to the screen's data pins and clock pins.

Issue: Screen image distortion and misalignment This situation is generally caused by mismatch between lane_bit_rate_mbps and pixel_clock, or mismatch between lane_bit_rate_mbps and screen required rate. You can adjust the lane_bit_rate_mbps value and test, specifically refer to the pixel_clock calculation method mentioned above.

Issue: Screen flashes blue screen with log: "can't fetch data from external memory fast enough, underrun happens" This issue occurs due to insufficient PSRAM bandwidth. You can consider reducing lane_bit_rate_mbps and pixel_clock values. In RGB888 color format scenarios, you can consider changing to RGB565 format. Also consider configuring the following options to improve PSRAM bandwidth:

- `CONFIG_SPIRAM_XIP_FROM_PSRAM=y`
- `CONFIG_CACHE_L2_CACHE_256KB=y`
- `CONFIG_CACHE_L2_CACHE_LINE_128B=y`
- `COMPILER_OPTIMIZATION_PERF=y`

Issue: How to resolve screen tearing Refer to *LCD Screen Tearing Detailed Guide <https://github.com/espressif/esp-iot-solution/blob/master/docs/zh_CN/display/lcd/lcd_screen_tearing.rst>*, related examples can be found in mipi_dsi_avoid_tearing.

Issue: Frame rate optimization configuration Consider configuring the following options to optimize frame rate:

- `CONFIG_SPIRAM_XIP_FROM_PSRAM=y`
- `CONFIG_CACHE_L2_CACHE_256KB=y`
- `CONFIG_CACHE_L2_CACHE_LINE_128B=y`
- `COMPILER_OPTIMIZATION_PERF=y`
- **LVGL related configurations:**
  - `CONFIG_LV_MEMCPY_MEMSET_STD=y`
  - `CONFIG_LV_MEM_CUSTOM=y`

Issue: How to drive default 4-lane screens Most LCD ICs support adjusting the number of lanes through registers, but the register addresses and operations differ between ICs. You need to confirm whether the lane count can be adjusted based on the specific IC' s datasheet or directly consult the screen manufacturer. For example, in EK79007, you can configure the 0xB2 register to choose between 2-lane/4-lane.

RB2h: Panel Control Register

Address (MIPI I/F)	B2h					Access Attribute			R/W
						Number of Parameter(s)			1
Parameter	D[7]	D[6]	D[5]	D[4]	D[3]	D[2]	D[1]	D[0]	Default Value
	—	NBW (0)	—	2Lane_EN (0)	—	—	—	—	00h
Description	NBW: Normally black or normally white setting. NBW="1" : Normally black. NBW="0" : Normally white(default). 2Lane_EN: 2 lane/4 lane setting 2Lane_EN="0":4 lane(default) 2Lane_EN="1":2 lane								

Fig. 39: Lane Number Configuration

Issue: How to confirm the lane bit rate range supported by the screen In high-speed serial communication, UI usually refers to the time unit of signal bits (bit) on the physical channel, which is one bit time (bit period). For example, if the frame rate or link rate is 1 Gbps, then one UI ≈ 1 nanosecond (ns). If the rate is 500 Mbps, then UI ≈ 2 ns, and so on. Therefore, you can calculate the lane bit rate range based on the AC ELECTRICAL CHARACTERISTIC - High speed transmission in the LCD IC datasheet. As shown in the figure, the lane bit rate (transmission rate of a single Data Lane) ranges from 20Mbps to 500Mbps.

15.3.3. High speed transmission

Parameter	Symbol	Min	Typ	Max	Units
UI instantaneous	UI_{INST}	2	-	12.5	ns
Data to Clock Skew(measured at transmitter)	$T_{SKEW(TX)}$	-0.15	-	0.15	UI_{INST}
Data to Clock Setup time(measured at receiver)	$T_{SETUP(RX)}$	0.15	-	-	UI_{INST}
Data to Clock Hold time(measured at receiver)	$T_{HOLD(RX)}$	0.15	-	-	UI_{INST}
20%~80% rise time and fall time	T_R, T_F	150	-		ps
		-	-	0.3	UI_{INST}

Fig. 40: Speed Range

Related Documentation and Examples

- MIPI DSI Specification
- ESP-IDF MIPI DSI LCD Programming Guide
- ESP LCD Driver Library
- ESP LCD Example Code
- ESP LCD FAQ

4.1.7 Detailed Explanation of LCD Screen Tearing

This section aims to introduce the principle of screen tearing on LCD displays and the corresponding solutions.

Terminology

Please refer to the *LCD Terms Table* 。

The principle of screen tearing

Phenomenon Screen tearing, also known as tearing effect, is a common issue in LCD applications, typically occurring when there is a full-screen or large area change in the GUI. The phenomenon involves displaying different parts of several frames simultaneously on the LCD, causing noticeable visual discontinuities in the image, significantly degrading the visual experience of the GUI. Below are the visual effects of running the LVGL Music example with and without screen tearing.

See the visual effect of screen tearing occurring.

See the visual effect diagram without screen tearing.

Reason Next, a series of assumptions will be set and combined with schematic diagrams to detail the reasons for screen tearing. For convenience, the screen refresh process here is simplified into two steps: **write** and **read**. Writing refers to the process where the main controller writes the rendered color data into the frame buffer (GRAM), while reading refers to the continuous process where the screen reads color data from GRAM and displays it on the panel. Below is a simplified schematic diagram of the screen refresh process.

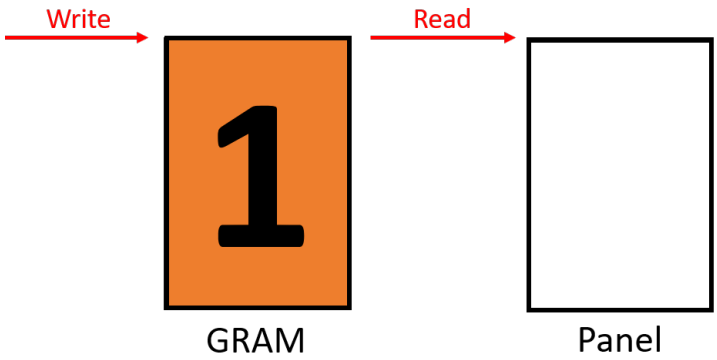


Fig. 41: Simplified screen refresh schematic diagram

Note: Some LCDs can control the direction of main controller writing and screen reading through commands, such as the `36h` command of **ST7789**. When the directions are inconsistent, screen tearing may also occur. Subsequently, explanations will only be provided for cases where the reading and writing directions are consistent.

When the main controller does not perform any synchronization operations during writing, meaning the initial position and time of writing are both unknown, screen tearing may occur if the speeds of writing and reading are not equal.

1. Assuming that the writing speed is slower than the reading speed (using a speed ratio of 1:2 as an example), during the process of the main controller writing the second frame of the image, the reading position will surpass the writing position. This results in the screen only reading the front half of the second frame of the image, leading to a torn display image. Below is a schematic diagram of the demonstration process.

See the schematic diagram of asynchronous writing and reading with a speed ratio of 1:2.

2. Assuming that the writing speed is faster than the reading speed (using a speed ratio of 2:1 as an example), during the process of the screen reading the first frame of the image, the writing position will surpass the reading position. This results in the screen reading the back half of the second frame of the image, leading to a torn display image. Below is a schematic diagram of the demonstration process.

See the schematic diagram of asynchronous writing and reading with a speed ratio of 2:1.

When the main controller performs synchronization operations during writing, meaning the initial position and time of writing are synchronized with reading, screen tearing may still occur if the speeds of writing and reading do not match.

1. Assuming that the writing speed is less than half of the reading speed (using a speed ratio of 1:3 as an example), during the process of the main controller writing the second frame of the image, the reading position will surpass the writing position. This results in the screen only reading the front half of the second frame of the image, leading to a torn display image. Below is a schematic diagram of the demonstration process.

See the schematic diagram of synchronous writing and reading with a speed ratio of 1:3.

2. Assuming that the writing speed is greater than or equal to half of the reading speed (using a speed ratio of 1:2 as an example), during the process of the main controller writing the second frame of the image, the reading position will not overlap with the writing position. This allows the screen to read the complete second frame of the image, ensuring that the display image does not tear. Below is a schematic diagram of the demonstration process.

See the schematic diagram of synchronous writing and reading with a speed ratio of 1:2.

Based on the assumptions above, the main reasons for screen tearing include the following two points:

1. Simultaneous operation of writing and reading to the same GRAM
2. The initial states of writing and reading are not synchronized or their speeds do not match

Methods to prevent screen tearing

After understanding the causes of screen tearing, methods to prevent tearing can be implemented from two perspectives: **GRAM** and the synchronization of **reading and writing states and speeds**. Due to the different *refresh mechanisms* and *GRAM positions* of LCDs with different interface types, it is necessary to select the recommended anti-tearing methods based on the specific interface type. The table below shows the positions of GRAM and the corresponding anti-tearing methods for different interface types.

Interface types	GRAM position	Anti-tearing methods
RGB, MIPI-DSI (video mode), QSPI (without internal GRAM)	main controller	*Methods based on multiple GRAM*
SPI, I80, QSPI (with internal GRAM)	LCD	*The method based on TE signal*

The method based on multiple GRAM This method is suitable for the situation where GRAM is in the main controller, and requires the main controller to be able to freely adjust the target GRAM for screen reading. The working principle is: by adding additional GRAM to avoid writing and reading operating on the same GRAM simultaneously. The following introduces the anti-tearing method based on dual GRAM, and the schematic diagram of the demonstration process is as follows.

See the schematic diagram of anti-tearing implemented based on dual GRAM.

From the diagram, it can be seen that initially the main controller is ready to write the second frame image into GRAM2, while the screen is ready to read the first frame image from GRAM1. After the main controller completes the writing, it first needs to set the screen to read from GRAM2 for the next frame, and then wait for the screen to finish reading the current frame image. After the screen finishes reading, it then starts reading the second frame

image from GRAM2, while the main controller also starts writing the third frame image into GRAM1. Therefore, writing and reading will not operate on the same GRAM simultaneously, thus avoiding screen tearing.

Here is the relevant sample code implemented based on LVGL:

1. rgb_avoid_tearing
2. qspi_without_ram

Note: To optimize display performance, an additional GRAM can be added on top of using two GRAMs. In this case, after the main controller completes writing one frame, it does not need to wait for the screen to finish reading one frame, but can directly start writing the next frame. For how to implement the anti-tearing method with three GRAMs, please refer to the *sample code*.

The method based on the TE signal This method is suitable for the situation where GRAM is inside the LCD, and requires the LCD to provide an external TE signal pin. The working principle is: controlling the initial state of writing through the TE signal to keep it synchronized with reading, while ensuring that the writing speed is not less than half of the reading speed, thereby avoiding overlap between writing and reading at the middle position of the GRAM. The following introduces the anti-tearing method based on the TE signal, and the schematic diagram of the demonstration process is as follows.

See the schematic diagram of anti-tearing implemented based on the TE signal.

From the diagram, it can be seen that initially the main controller is waiting for the TE signal, while the screen is preparing to enter the blanking area (Porch). When the screen starts reading the first frame image from the GRAM, it sends the TE signal to the main controller. Upon receiving the TE signal, the main controller starts writing the second frame image to the GRAM, ensuring that the ratio of writing speed to reading speed is 2:3. Therefore, writing and reading will not overlap at the middle position of the GRAM, thus avoiding screen tearing.

Here is the relevant sample code implemented based on LVGL:

1. lcd_with_te

Note:

1. Some LCDs can control the switch of the TE signal and trigger timing parameters through commands, such as the `35h` and `44h` commands of **ST7789**. To ensure the effectiveness of the above method, users need to set the corresponding parameters according to the data sheet of the specific LCD driver IC, so that the TE signal is enabled and triggered at the appropriate position.
2. Some LCDs can control the direction of writing by the main controller and reading by the screen through commands, such as the `36h` command of **ST7789**. When the directions are inconsistent, the above method to prevent screen tearing will be ineffective. Users need to set the corresponding parameters according to the data sheet of the specific LCD driver IC to ensure that the direction of writing and reading is consistent.

4.1.8 LCD Application Solution

Introduction to LCD Solutions

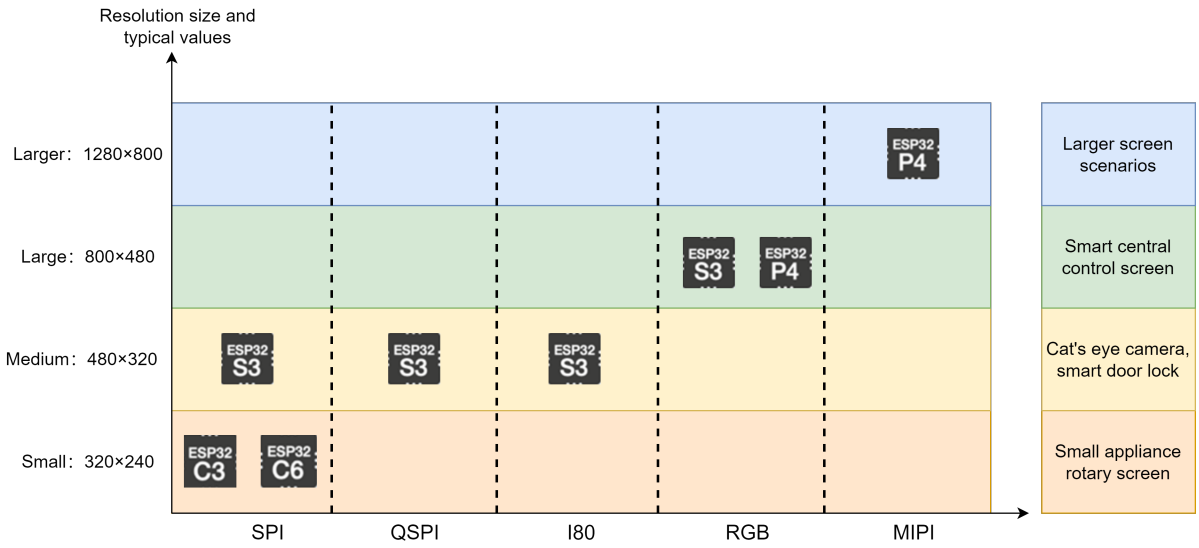
Espressif' s HMI Smart Screen (LCD) solution features outstanding performance and scalability, and can be paired with different ESP main control chips. This solution performs well in various application scenarios such as smart home control, household appliance screens, medical equipment, industrial control, and children' s education. Advantages include high-performance graphic visualization, low memory usage, and a comprehensive screen adaptation solution that supports high-performance JPEG decoding and frame rate optimization. Below is a detailed introduction to the LCD solution

- **Outstanding Graphic Visualization:** Leveraging the high-performance graphic processing capabilities of ESP chips and deep collaboration with the LVGL official, it excels in LVGL compatibility. This solution offers exquisite visual effects, low memory usage, and can be easily integrated into product designs.
- **Simple UI Design:** The UI editor Squareline Studio supports quick and easy design and development of UI for embedded devices. It is easy to port and allows UI implementation without code, minimizing development time.
- **Rich Software and Hardware References:** Provides comprehensive LCD software and hardware development materials, including detailed guides and examples. In addition, HMI development boards designed for various HMI application scenarios can help developers get started quickly.
- **Comprehensive Screen Adaptation:** Supports various operation methods, including touch, knobs, etc. It also supports various peripheral interfaces such as RGB, SPI, etc. Multiple LCD driver ICs and Touch driver ICs have been adapted and organized into components to meet the needs of different user groups.

Furthermore, Espressif also supports the following features to further ensure a smoother interactive experience in LCD application scenarios:

- **Frame Rate Optimization and Tear Prevention Technology:** Ensures smooth and consistent image display through carefully optimized frame rate control and tear prevention technology.
- **High-Performance JPEG Decoding:** Supports efficient image processing to ensure a smooth multimedia experience.
- **Voice Wake-up and Recognition:** Integration of advanced voice recognition technology provides users with a more convenient way of interaction.

The following is an overview of a flowchart scheme:

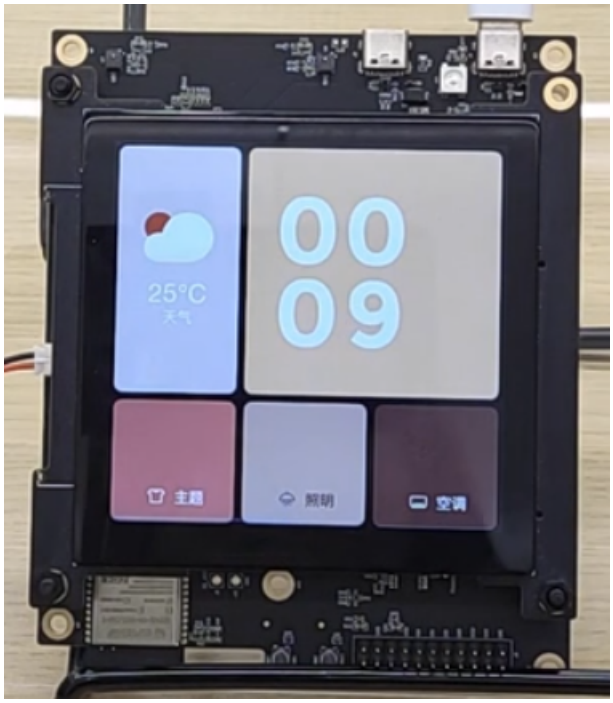


Common Applications of LCDs:

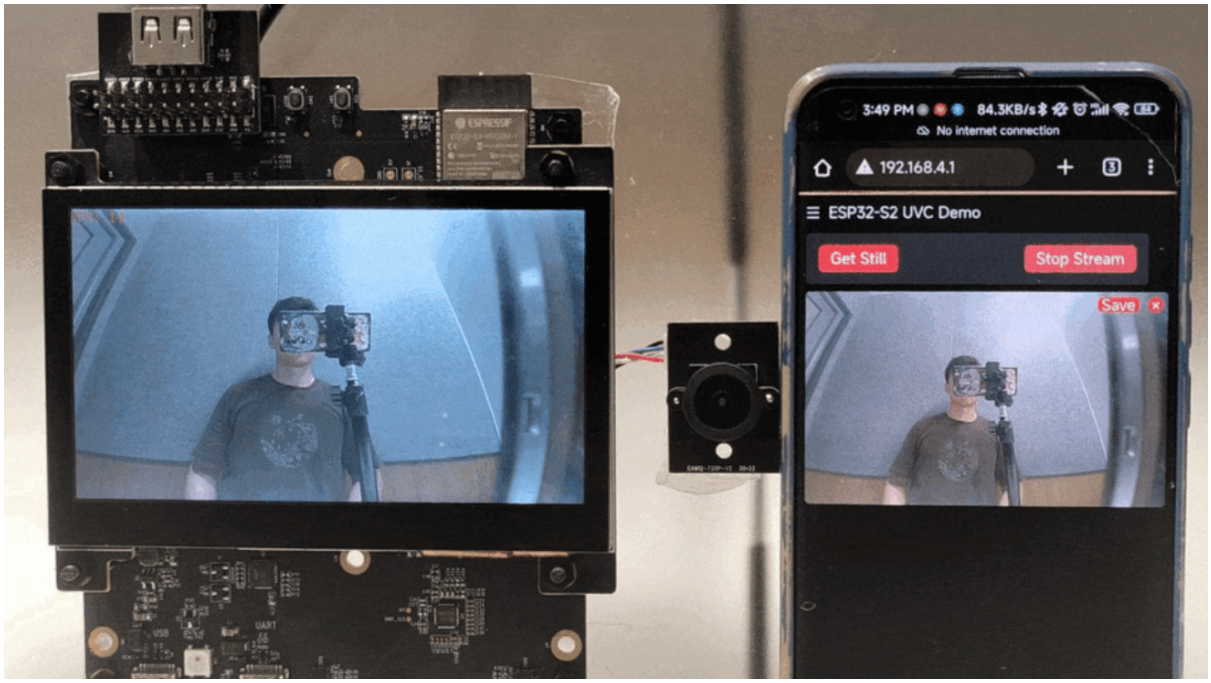
Espressif' s LCD solutions are widely used in various fields, including but not limited to:

- **Rotary Knob Screen Solution:** Targeted at smart home appliance products, it is the preferred upgrade for traditional segment code screens and black-and-white screens. It supports Wi-Fi, Bluetooth, and expansion interfaces for functions such as serial communication. A typical application scenario is the rotary knob screen in small home appliance applications.

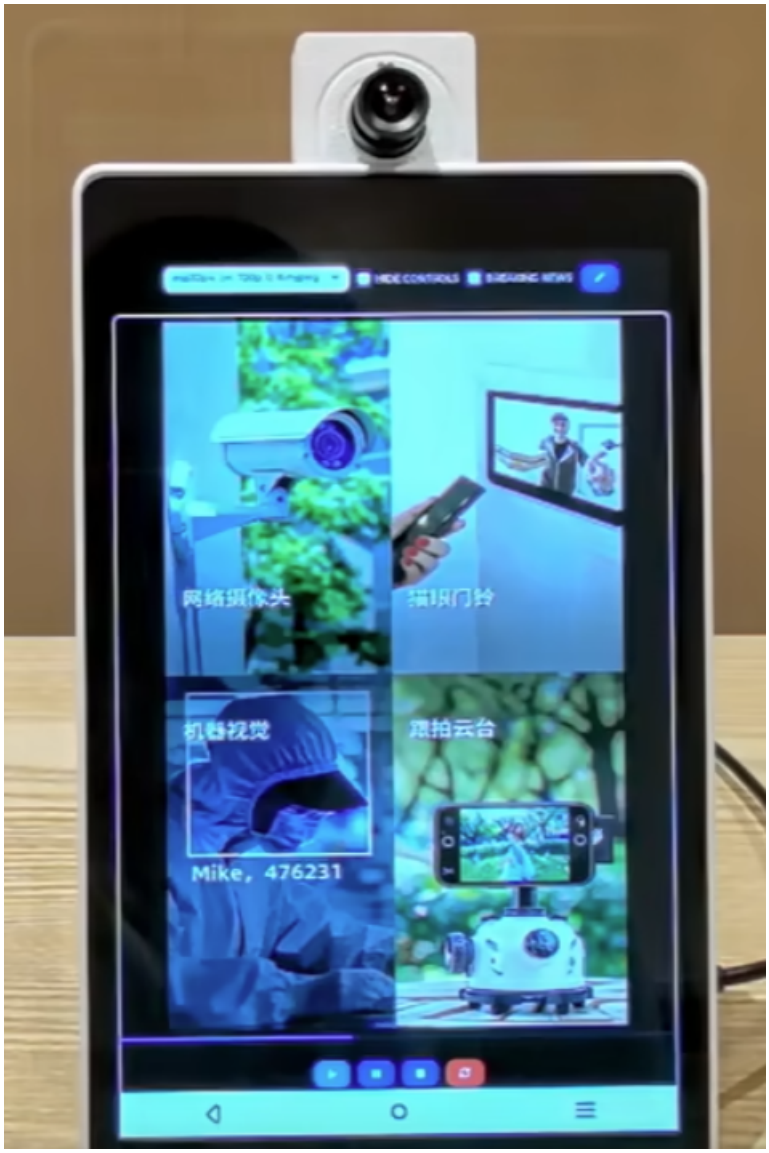
- **Central Control Screen Solution:** Integrates Wi-Fi, BLE, offline voice, RGB LCD display, supports offline command words, and continuous voice recognition. Suitable for the upgrade and iteration of traditional 86 panels, it constructs a smart home control center that integrates device control, switch panels, temperature control panels, smart remote controls, etc.



Visual Voice Solution: Utilizes native USB to connect to a universal USB camera, achieving camera data stream reading, JPEG decoding, and real-time display on an RGB interface screen on a single SoC, without the need for additional USB chips. Suitable for scenarios such as cat' s eyes, smart doorbell locks, electronic endoscopes, etc.



High-Performance Multimedia Solution: This solution uses the ESP32-P4 chip, supporting MIPI-CSI and MIPI-DSI interfaces, suitable for various scenarios requiring high-resolution cameras and displays. The chip also integrates hardware accelerators for various media encoding and compression protocols, hardware pixel processing accelerator (PPA), and 2D-DMA, making it suitable for a wide range of multimedia applications.



In summary:

Table 3: Overview of LCD Solutions

Solution Category	Solution Name	Main Control	Screen	Features	Application Scenarios
Rotary Screen	2.1-inch Rotary Screen Solution (Qiming)	ESP32-S3	2.1-inch, 480 x 480 RGB interface screen	Supports Wi-Fi, Bluetooth, and expandable interfaces for serial communication, buttons, USB cameras, and other functions	Ideal for smart home appliances, preferred upgrade for traditional segment displays and black-and-white screens
Rotary Screen	1.28-inch Rotary Screen Solution	ESP32-C3	1.28-inch, 240 x 240 SPI interface screen	Rotary encoder with push-button switch, compact design	Application scenarios include rotary screens and small-sized displays in small home appliances
Central Control Screen	Smart Voice Touch Panel (86 Box) Solution	ESP32-S3	3.95-inch, 480 x 480 RGB interface screen	Integrated with Wi-Fi, BLE, offline voice recognition, RGB LCD display, supports offline command words and continuous voice recognition	Upgraded iteration for traditional 86 panels, serving as the central hub for smart home control integrating device control, switch panels, temperature control panels, and smart remote controls
Central Control Screen	Smart Central Control Switch Solution	ESP32-S3	7-inch, 800 x 480 RGB interface screen	Multi-touch screen for gesture recognition, supports Wi-Fi CSI human body proximity sensing, used for building home visual panels	Quick switch control in smart homes, such as scene mode switching and light control
Visual Voice Solution	Visual Voice Solution	ESP32-S3	4.3-inch, 800 x 480 RGB interface screen	Utilizes native USB for connecting to a generic USB camera, achieving camera data stream reading, JPEG decoding, and real-time display on an RGB interface screen on a single SoC without the need for additional USB chips. Local video decoding and screen refresh resolution can reach 800 x 480 @15 FPS	Applications include cat' s eye cameras, smart doorbells, smart door locks, electronic endoscopes, and more.
High-Performance Multimedia Solution	High-Performance Multimedia Solution	ESP32-P4	8-inch, 800 x 1280 MIPI-DSI interface screen	Supporting MIPI-CSI and MIPI-DSI interfaces, it is suitable for various scenarios requiring high-resolution cameras and displays. It incorporates hardware accelerators for multiple media encoding and compression protocols, a hardware pixel processing accelerator (PPA), and 2D-DMA, making it ideal for diverse multimedia applications.	Designed for high-performance multimedia applications

LCD Reference Solutions

ESP-BOX

Description:

A home appliance control platform developed for voice assistants, touchscreen controllers, sensors, infrared controllers, and smart Wi-Fi gateways.

Hardware:

- Development Board: ESP32-S3-BOX-3

Related Links:

- Code Repository: esp-box
- Related Video: ESP32-S3-BOX-3 Surprise Unboxing!

Features:

- Based on LVGL GUI framework
- Dual-microphone far-field voice interaction, offline AI recognition in Chinese and English, supports over 200 voice commands
- Integrated end-to-end AIoT development framework ESP-RainMaker
- Pmod™ compatible connector supports peripheral modules for expanding sensors, infrared controllers, etc.
- PSRAM requirement: 8-bit (8M)

ESP32-C3 Rotary Screen

Description:

Circular rotary screen solution, integrating common scenarios such as washing machines, dimmers, and thermostats.

Hardware:

- Development Board: ESP32-C3-LCDkit

Related Links:

- Code Repository: esp32-c3-lcdkit
- **Related Videos:**
  - ESP32-C3 Rotary Screen Demo
  - ESP32-C3-LCDKit Rotary Screen Development Board

Features:

- Based on LVGL GUI framework
- Circular screen UI display (non-touch), controlled by a rotary encoder

Intelligent Voice Touch Panel (86 Box)

Description:

Can be used for the upgrade iteration of traditional 86 panels, building an intelligent home control center integrating device control, switch panel, temperature control panel, smart remote control, etc.

Hardware:

- Development Board: ESP32-S3-LCD-EV-Board
- Screen: LCD Subboard 2 (480x480)

Related Links:

- Code Repository: esp32-s3-lcd-ev-board/86-box Smart Panel Example
- Related Video: ESP32-S3 Smart Voice Touch Panel

Features:

- Based on LVGL GUI framework
- Dual microphone far-field voice interaction, Chinese and English AI offline voice recognition, supports over 200 voice commands
- PSRAM requires 8 lines (R8) and enables 120M

Electronic Visual Doorbell

Description:

Utilizes native USB interface to connect with a generic USB camera, achieving camera data stream reading, JPEG decoding, and real-time display on an RGB interface screen on a single SoC without the need for additional USB chips. The local video decoding and screen refresh resolution can reach 800x480@15 FPS.

Hardware:

- Development Board: ESP32-S3-LCD-EV-Board
- Screen: LCD Subboard 3 (800x480)

Related Links:

- Code Repository: esp32-s3-lcd-ev-board/USB Camera LCD Example
- Related Video: ESP32-S3 Driving RGB Interface Screen + USB CDC Camera Demo

Features:

- USB camera data stream reading, requires support for Bulk mode
- JPEG decoding
- 800x480 RGB LCD display
- PSRAM requires 8 lines (R8) and enables 120M

Smart Central Control Switch

Description:

Recognizes gestures such as double-finger tapping and patting through a multi-touch screen, which can be used for quick switch control in smart homes, such as scene mode switching and light control. Combined with Wi-Fi CSI human body proximity sensing function, it can also achieve automatic on-off control of screen proximity wake-up and screen-off.

Hardware:

- Development Board: ESP32-S3-LCD-EV-Board
- Screen: 7-inch, RGB interface, 800x480 resolution

Related Links:

- Related Video: ESP32-S3 Driving Super Large RGB Interface Screen

Features:

- 7-inch large LCD screen, supports multi-touch
- Wi-Fi CSI human body proximity sensing
- PSRAM requires 8 lines (R8) and enables 120M

High-Performance Multimedia Solution

Description:

Supporting MIPI-CSI and MIPI-DSI interfaces, it is suitable for various scenarios requiring high-resolution cameras and displays, with hardware accelerators for multiple media encoding and compression protocols, a hardware pixel processing accelerator (PPA), and 2D-DMA, making it suitable for a wide range of multimedia applications.

Hardware:

- Development Board: ESP32-P4_Function_EV_Board
- Screen: 8-inch 800 x 1280 LCD screen (IC: ILI9881C)

Related Links:

- Related Video: Challenge: Building a Smartphone with ESP32-P4

Features:

- Supports MIPI-DSI and MIPI-CSI interfaces
- Hardware accelerators for various media encoding and compression protocols
- Hardware pixel processing accelerator (PPA) and 2D-DMA

LCD Reference Materials

- LCD Software References
  - ESP LCD Driver Library
  - Arduino LCD Driver Library
  - ESP LCD Driver Documentation
  - ESP LCD Examples
  - ESP-BOX AIoT Development Framework
- LCD Solutions & Development Guides
  - ESP-HMI Smart Display Solution
  - Quick Start GUI (Part 1)
  - Quick Start GUI (Part 2)
  - ESP LCD Development Guide
- Purchase Links for LCD Development Boards
  - ESP32-S3-LCD-EV-Board Purchase Link
  - ESP32-S3-BOX Purchase Link
  - ESP32-S3-BOX-Lite Purchase Link
  - ESP32-C3-LCDkit Purchase Link
- Module/Development Board References and Options
  - ESP32-S3 Datasheet
  - ESP32-S3-WROOM-1 Datasheet
  - ESP32-C3 Datasheet
  - ESP32-C3-MINI-1 Datasheet
  - Espressif Product Selection Tool

4.1.9 GUI Optimization Solutions

ESP LVGL Adapter

ESP LVGL Adapter is a high-performance LVGL adaptation layer designed specifically for the entire ESP32 chip family. It provides deep optimizations for screen tearing and frame rate, and seamlessly integrates advanced features such as image decoding, FreeType font rendering, and Dummy Draw mode, delivering a true out-of-the-box experience for developers.

Component Link: https://components.espressif.com/components/espressif/esp_lvgl_adapter

Related Example:

- https://github.com/espressif/esp-iot-solution/tree/master/examples/display/gui/lvgl_common_demo
- https://github.com/espressif/esp-iot-solution/tree/master/examples/display/gui/lvgl_dummy_draw
- https://github.com/espressif/esp-iot-solution/tree/master/examples/display/gui/lvgl_decode_image
- https://github.com/espressif/esp-iot-solution/tree/master/examples/display/gui/lvgl_freetype_font
- https://github.com/espressif/esp-iot-solution/tree/master/examples/display/gui/lvgl_multi_screen
- https://github.com/espressif/esp-iot-solution/tree/master/examples/display/gui/lvgl_mono_demo

ESP LVGL EAF Player

esp_lv_eaf_player is a lightweight and efficient EAF animation player designed specifically for LVGL v8/v9. It enables seamless integration of compressed animation sequences into LVGL projects. The EAF format supports multiple compression methods including RLE, Huffman coding, and JPEG compression, providing high-quality animations with minimal memory footprint.

Features:

- Multiple compression support: RLE, Huffman, JPEG
- Optimized for embedded: Minimal memory footprint and efficient decoding
- LVGL integration: Seamless integration with LVGL widget system

- Animation control: Play, pause, restart, and loop control

Applicable Scenarios:

- Users using LVGL v8/v9 who need to play high-quality animations
- Efficient animation playback on resource-constrained platforms

EAF File Conversion: Use the online conversion tool to convert GIF or other animation formats to EAF format: https://esp32-gif.espressif.com/

Component Link: https://components.espressif.com/components/espressif/esp_lv_eaf_player

Related Example:

- https://github.com/espressif/esp-iot-solution/tree/master/examples/display/gui/lvgl_eaf_player

Thorvg Component

Supports Lottie animation and TVG vector graphics parsing and rendering, noglic version supported

Component Link: https://components.espressif.com/components/espressif/thorvg

Related Example:

- Lottie: https://components.espressif.com/components/espressif/thorvg/versions/0.13.8/examples/thorvg-example

tinyGL 3D Graphics Library

- tinyGL is a lightweight 3D graphics library that supports a subset of OpenGL ES 1.1, suitable for ESP32-P4 platform.

Applicable Scenarios:

- Need to implement 3D graphics rendering.

Component Link: https://github.com/espressif2022/TinyGL

SDL3 Development Library

SDL is a cross-platform development library designed to provide low-level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL/Direct3D/Metal/Vulkan. It is used by video playback software, emulators, and many popular games including Valve' s award-winning games and Humble Bundle games.

Applicable Scenarios:

- Users with emulator development, game porting, and cross-platform application development requirements.

Component Link: https://components.espressif.com/components/georgik/sdl/versions/3.1.2~9

Related Example: https://github.com/georgik/esp32-sdl3-test/tree/main

esp_lv_lottie_player

esp_lv_lottie_player is a lightweight and efficient Lottie animation player designed specifically for LVGL v8/v9. It can seamlessly integrate vector-based animations into LVGL projects. By leveraging ThorVG as the rendering engine, this module ensures high performance and flexibility for modern embedded GUI applications.

Applicable Scenarios:

- Users using LVGL v8/v9 who need to play Lottie animations

Component Link: https://components.espressif.com/components/espressif/esp_lv_lottie_player

Related Example:

- https://github.com/espressif/esp-iot-solution/tree/master/examples/display/gui/lvgl_lottie_player

esp_emote_gfx

esp_emote_gfx is a lightweight animation rendering library for resource-constrained platforms, supporting conversion of GIF to custom EAF animation format for playback, with advantages of fast decoding, small memory footprint, and flexible display, suitable for efficient dynamic graphics playback needs in HMI scenarios.

Features:

- Supports text scrolling effects
- Supports simultaneous playback of multiple animations with independent frame rate control
- Adapts to multiple screen sizes

Applicable Scenarios:

- Guide animations, boot animations, emoticons and text rendering on various screens
- Dynamic UI component display on resource-constrained platforms (ESP32-C2/C3, etc.)

Component Link: https://components.espressif.com/components/espressif2022/esp_emote_gfx/

esp_emote_expression

esp_emote_expression is an expression and interface description system built on esp_emote_gfx, providing complete resource parsing, loading and lifecycle management capabilities, and supporting third-party custom widget extensions. It can be used as a foundational UI framework for AI interactive devices.

The system covers common UI scenarios for AI conversations, including:

- Expression and animation display
- Text rendering (system prompts, user messages, etc.)
- Status icons (speaking, listening, etc.)
- QR code display
- Popups and notification components

Applicable Scenarios:

- Foundational UI framework for AI interactive devices
- Embedded devices requiring complete UI resource management and widget extensions

Integrated Repositories:

- xiaozhi: https://github.com/78/xiaozhi-esp32
- rainmaker agent: https://github.com/espressif/esp-agents-firmware

Quick Start:

- gfx-gen-tool: https://gfx-gen-tool.pages.dev/
- Local preset assets: https://components.espressif.com/components/espressif2022/esp_emote_assets

Component Link: https://components.espressif.com/components/espressif2022/esp_emote_expression

4.2 Digital Tube

Digital tube and dot matrix LEDs are common display solutions in embedded systems, which occupy fewer pins and memory resources than LCD displays and are simpler to implement, making them more suitable for application scenarios with single display requirements such as timing, counting, status display and etc.

The digital tube and LED display drivers adapted to ESP-IoT-Solution are shown in the following table:

Name	Features	Interface	Driver	Datasheet
CH450	Digital tube display driving chip, supports 6-bit digital tube	I2C	ch450	CH450
HT16C21	20×4/16×8 LCD controller, supports RAM mapping	I2C	ht16c21	HT16C21
IS31FL3XXX	Dot matrix LED controller	I2C	is31fl3xxx	IS31FL3XXX

4.2.1 CH450 Driver

CH450 is a digital tube display driving chip that can be used to drive a 6-bit digital tube or a 48-dot LED matrix and can communicate with ESP32 via the `I2C` interface.

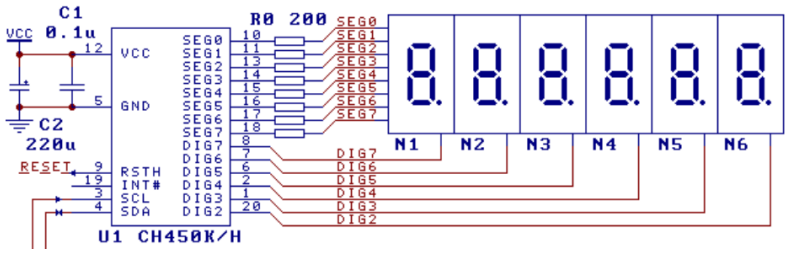


Fig. 42: CH450 Typical Application Circuit

This driver encapsulates the basic operations of CH450, and users can directly call `ch450_write()` or `ch450_write_num()` to display numbers on the digital tube.

Example

```
i2c_bus_handle_t i2c_bus = NULL;
ch450_handle_t seg = NULL;
i2c_config_t conf = {
    .mode = I2C_MODE_MASTER,
    .sda_io_num = I2C_MASTER_SDA_IO,
    .sda_pullup_en = GPIO_PULLUP_ENABLE,
    .scl_io_num = I2C_MASTER_SCL_IO,
    .scl_pullup_en = GPIO_PULLUP_ENABLE,
    .master.clk_speed = I2C_MASTER_FREQ_HZ,
};
i2c_bus = i2c_bus_create(I2C_MASTER_NUM, &conf);
seg = ch450_create(i2c_bus);

for (size_t i = 0; i < 10; i++) {
    for (size_t index = 0; index < 6; index++) {
        ch450_write_num(seg, index, i);
    }
    vTaskDelay(1000 / portTICK_PERIOD_MS);
}

ch450_delete(seg);
i2c_bus_delete(&i2c_bus);
```

4.2.2 HT16C21 Driver

HT16C21 is a LCD control/driver chip which supports RAM mapping and can be used to drive `20 x 4` or `16 x 8` segmented LCDs. The chip communicates with ESP32 via the `I2C` interface.

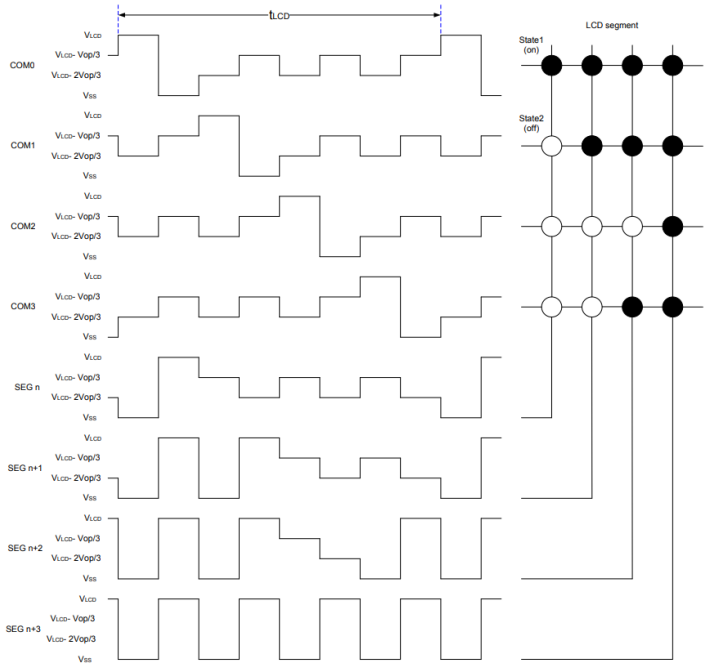


Fig. 43: HT16C21 Typical Drive Model

This driver encapsulates the basic operations of HT16C21. After creating an example using `ht16c21_create`, users can configure its parameters via `ht16c21_param_config` and then call `ht16c21_ram_write` directly to write data.

Example

```
i2c_bus_handle_t i2c_bus = NULL;
ht16c21_handle_t seg = NULL;
uint8_t lcd_data[8] = { 0x10, 0x20, 0x30, 0x50, 0x60, 0x70, 0x80 };

i2c_config_t conf = {
    .mode = I2C_MODE_MASTER,
    .sda_io_num = I2C_MASTER_SDA_IO,
    .sda_pullup_en = GPIO_PULLUP_ENABLE,
    .scl_io_num = I2C_MASTER_SCL_IO,
    .scl_pullup_en = GPIO_PULLUP_ENABLE,
    .master.clk_speed = I2C_MASTER_FREQ_HZ,
};
i2c_bus = i2c_bus_create(I2C_MASTER_NUM, &conf);
seg = ht16c21_create(i2c_bus, HT16C21_I2C_ADDRESS_DEFAULT);

ht16c21_config_t ht16c21_conf = {
    .duty_bias = HT16C21_4DUTY_3BIAS;
    .oscillator_display = HT16C21_OSCILLATOR_ON_DISPLAY_ON;
    .frame_frequency = HT16C21_FRAME_160HZ;
    .blinking_frequency = HT16C21_BLINKING_OFF;
    .pin_and_voltage = HT16C21_VLCD_PIN_VOL_ADJ_ON;
    .adjustment_voltage = 0;
};
```

(continues on next page)

(continued from previous page)

```
ht16c21_param_config(seg, &ht16c21_conf);
ht16c21_ram_write(seg, 0x00, lcd_data, 8);

ht16c21_delete(seg);
i2c_bus_delete(&i2c_bus);
```

4.2.3 IS31FL3XXX Driver

The IS31FL3XXX series chips can be used to drive dot matrix LED screens with different sizes. The IS31FL3218 supports 18 constant current channels, with each channel controlled by an independent PWM. It has a maximum output current of 38 mA and can directly drive LEDs for display. The IS31FL3736 supports more channels and can compose a maximum size of LED matrix as `12 x 8`. With each channel driven by an 8-bit PWM driver, the IS31FL3736 can support up to 256 gradients.

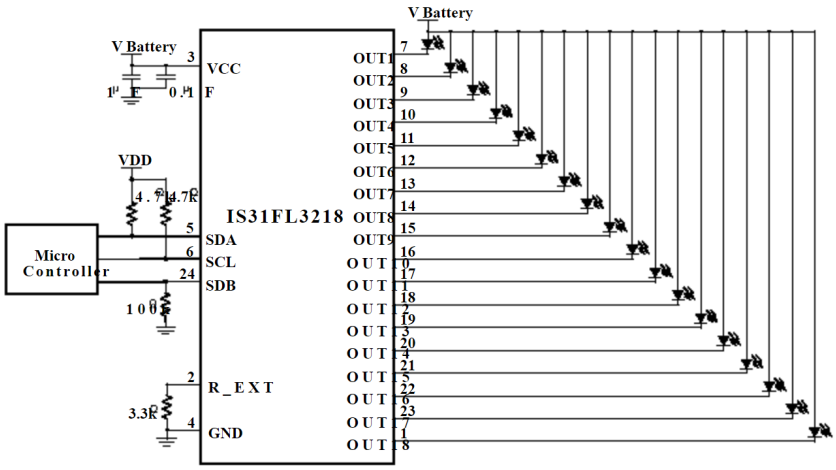


Fig. 44: IS31FL3218 Typical Application Circuit

This driver encapsulates the basic operations of IS31FL3XXX. The example is shown in the next section.

IS31FL3218 Example

```
i2c_bus_handle_t i2c_bus = NULL;
is31fl3218_handle_t fxled = NULL;
i2c_config_t conf = {
    .mode = I2C_MODE_MASTER,
    .sda_io_num = I2C_MASTER_SDA_IO,
    .sda_pullup_en = GPIO_PULLUP_ENABLE,
    .scl_io_num = I2C_MASTER_SCL_IO,
    .scl_pullup_en = GPIO_PULLUP_ENABLE,
    .master.clk_speed = I2C_MASTER_FREQ_HZ,
};
i2c_bus = i2c_bus_create(I2C_MASTER_NUM, &conf);
fxled = is31fl3218_create(i2c_bus);
is31fl3218_channel_set(fxled, 0x00ff, 128); // set PWM 1 ~ PWM 8 duty cycle 50%
is31fl3218_delete(fxled);
i2c_bus_delete(&i2c_bus);
```

4.3 LED Indicator

This guide includes the following content:

Table of Contents

- *Supported Indicator Light Types*
- *Defining Blinking Type*
  - *Control On/Off*
  - *Controlling Brightness*
  - *Controlling Color*
  - *Controlling Index*
- *Predefined Blinking Priorities*
- *Control Indicator Blinks*
- *Custom light blink*
- *Adjustment of Gamma*
- *Drive Level Setting*
  - *API Reference*
  - *Header File*
  - *Functions*
  - *Macros*

As one of the simplest output peripherals, LED indicators can indicate the current operating state of the system by blinking in different types. ESP-IoT-Solution provides an LED indicator component with the following features:

- Can define multiple groups of different blink types
- Can define the priority of blink types
- Can set up multiple indicators
- The LEDC driver supports adjusting brightness, gradient, and color.

4.3.1 Supported Indicator Light Types

Driver Type	Description	On/Off	Brightness	Breathing	Color	Color Gradient	Index
GPIO	Control indicator lights through GPIO	√	×	×	×	×	×
LEDC	Control indicator lights through PWM (single channel)	√	√	√	×	×	×
RGB LED	Control indicator lights through PWM (three channels)	√	√	√	√	√	×
LED Strips	Control lights like WS2812 through RMT/SPI	√	√	√	√	√	√

4.3.2 Defining Blinking Type

Control On/Off

The blink step structure `blink_step_t` defines the type of a step, indicator state and the state duration. Multiple steps are combined into one blink type, and different blink types can be used to identify different system states. The blink type is defined as follows:

Example 1. define a blink loop: turn on 0.05 s, turn off 0.1 s, and loop.

```
const blink_step_t test_blink_loop[] = {
    {LED_BLINK_HOLD, LED_STATE_ON, 50},                // step1: turn on LED 50 ms
    {LED_BLINK_HOLD, LED_STATE_OFF, 100},              // step2: turn off LED 100 ms
    {LED_BLINK_LOOP, 0, 0},                            // step3: loop from step1
};
```

Example 2. define a blink loop: turn on 0.05 s, turn off 0.1 s, turn on 0.15 s, turn off 0.1 s, and stop blink.

```
const blink_step_t test_blink_one_time[] = {
    {LED_BLINK_HOLD, LED_STATE_ON, 50},                // step1: turn on LED 50 ms
    {LED_BLINK_HOLD, LED_STATE_OFF, 100},              // step2: turn off LED 100 ms
    {LED_BLINK_HOLD, LED_STATE_ON, 150},               // step3: turn on LED 150 ms
    {LED_BLINK_HOLD, LED_STATE_OFF, 100},              // step4: turn off LED 100 ms
    {LED_BLINK_STOP, 0, 0},                            // step5: stop blink (off)
};
```

```
typedef enum {
    BLINK_TEST_BLINK_ONE_TIME, /**< test_blink_one_time */
    BLINK_TEST_BLINK_LOOP,     /**< test_blink_loop */
    BLINK_MAX,                 /**< INVALID type */
} led_indicator_blink_type_t;

blink_step_t const * led_indicator_blink_lists[] = {
    [BLINK_TEST_BLINK_ONE_TIME] = test_blink_one_time,
    [BLINK_TEST_BLINK_LOOP] = test_blink_loop,
    [BLINK_MAX] = NULL,
};
```

Controlling Brightness

For drivers supporting brightness control, the indicator light' s brightness can be controlled in the following ways:

Example 1: Defining a brightness setting: Setting the indicator light to 50% brightness for 0.5 seconds.

```
const blink_step_t test_blink_50_brightness[] = {
    {LED_BLINK_BRIGHTNESS, LED_STATE_50_PERCENT, 500},   // step1: set to half␣
↪brightness 500 ms
    {LED_BLINK_STOP, 0, 0},                              // step2: stop blink (50%␣
↪brightness)
};
```

Example 2: Defining a looping blink: Gradually turning on for 0.5s, then gradually turning off for 0.5s, repeating the sequence.

```
const blink_step_t test_blink_breathe[] = {
    {LED_BLINK_HOLD, LED_STATE_OFF, 0},                  // step1: set LED off
    {LED_BLINK_BREATHE, LED_STATE_ON, 500},              // step2: fade from off␣
↪to on 500ms
    {LED_BLINK_BREATHE, LED_STATE_OFF, 500},             // step3: fade from on to␣
↪off 500ms
    {LED_BLINK_LOOP, 0, 0},                              // step4: loop from step1
};
```

Example 3: Defining a blink: Gradually brightening from 50% to 100% brightness for 0.5s.

```
const blink_step_t test_blink_breathe_2[] = {
    {LED_BLINK_BRIGHTNESS, LED_STATE_50_PERCENT, 0},     // step1: set to half␣
↪brightness 0 ms
    {LED_BLINK_BREATHE, LED_STATE_ON, 500},              // step2: fade from off␣
↪to on 500ms
    {LED_BLINK_STOP, 0, 0},                              // step3: stop blink (100
↪% brightness)
};
```

Controlling Color

For drivers supporting color control, we can use *LED_BLINK_RGB*, *LED_BLINK_RGB_RING*, *LED_BLINK_HSV*, *LED_BLINK_HSV_RING* to control the color.

- *LED_BLINK_RGB*: Controls color via RGB, where R takes 8 bits (0-255), G takes 8 bits (0-255), and B takes 8 bits (0-255).
- *LED_BLINK_RGB_RING*: Controls color gradient through RGB, transitioning from the previous color to the current set color. Use RGB value interpolation method.
- *LED_BLINK_HSV*: Controls color via HSV, where H takes 9 bits (0-360), S takes 8 bits (0-255), and V takes 8 bits (0-255).
- *LED_BLINK_HSV_RING*: Controls color gradient through HSV, transitioning from the previous color to the current set color. Use HSV value interpolation method.

Example 1: Defining a color setting to display red on the indicator light.

```
const blink_step_t test_blink_rgb_red[] = {
    {LED_BLINK_RGB, SET_RGB(255, 0, 0), 0},                // step1: set to red␣
↪color 0 ms
    {LED_BLINK_STOP, 0, 0},                                // step2: stop blink␣
↪(red color)
};
```

Example 2: Defining a color gradient, transitioning the indicator light from red to blue and looping the sequence.

```
const blink_step_t test_blink_rgb_red_blue[] = {
    {LED_BLINK_RGB, SET_RGB(0xFF, 0, 0), 0},               // step1: set to red␣
↪color 0 ms
    {LED_BLINK_RGB_RING, SET_RGB(0, 0, 0xFF), 4000},       // step2: fade from red␣
↪to blue 4000ms
    {LED_BLINK_RGB_RING, SET_RGB(0xFF, 0, 0), 4000},       // step3: fade from␣
↪blue to red 4000ms
    {LED_BLINK_LOOP, 0, 0},                                // step4: loop from␣
↪step1
};
```

Display color gradient using RGB interpolation. The effect is as follows.

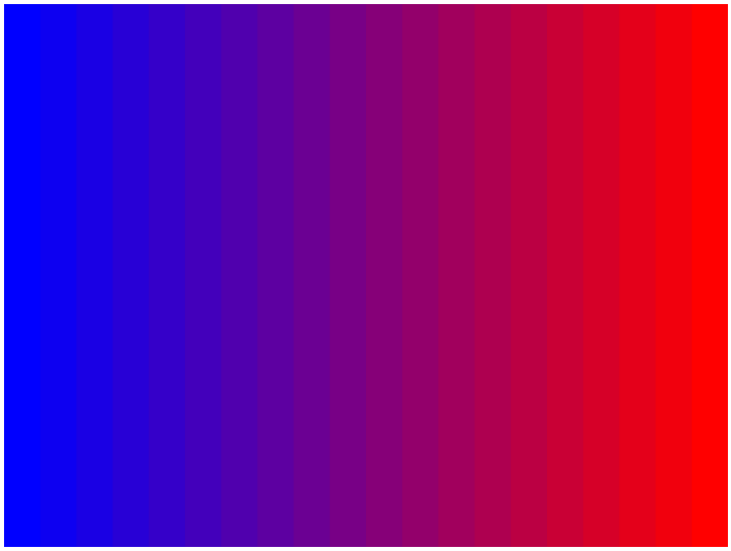


Fig. 45: RGB Gradient

Additionally, the driver supports setting colors through HSV similarly to RGB.

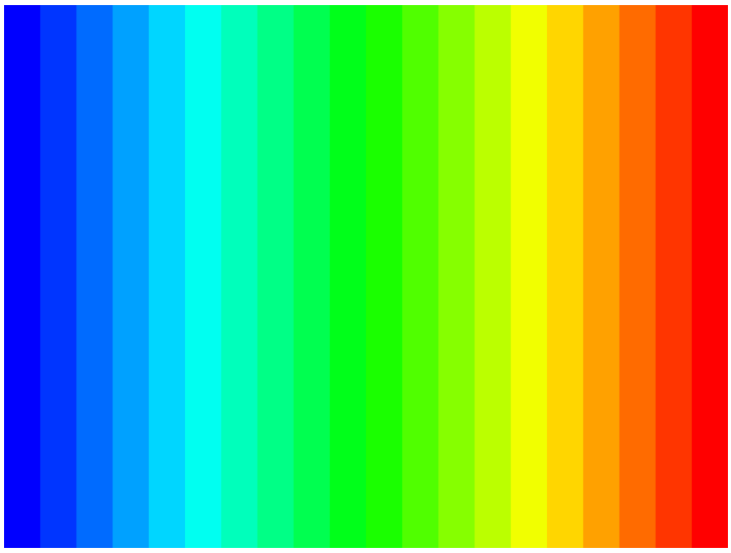
Example 3: Defining a color sequence to display red for 0.5s, green for 0.5s, blue for 0.5s, and then stop.

```
const blink_step_t test_blink_hsv_colors[] = {
    {LED_BLINK_HSV, SET_HSV(0, 255, 255), 500},            // step1: set color to␣
↪red 500 ms
    {LED_BLINK_HSV, SET_HSV(120, 255, 255), 500},          // step2: set color to␣
↪green 500 ms
    {LED_BLINK_HSV, SET_HSV(240, 255, 255), 500},          // step3: set color to␣
↪blue 500 ms
    {LED_BLINK_STOP, 0, 0},                                // step4: stop blink␣
↪(blue color)
};
```

Example 4: Defining a color gradient, transitioning the indicator light from red to blue and looping the sequence using HSV.

```
const blink_step_t test_blink_hsv_red_blue[] = {
    {LED_BLINK_HSV, SET_HSV(0, 255, 255), 0},              // step1: set to red␣
↪color 0 ms
    {LED_BLINK_HSV_RING, SET_HSV(240, 255, 255), 4000},    // step2: fade from red␣
↪to blue 4000ms
    {LED_BLINK_HSV_RING, SET_HSV(0, 255, 255), 4000},      // step3: fade from␣
↪blue to red 4000ms
    {LED_BLINK_LOOP, 0, 0},                                // step4: loop from␣
↪step1
};
```

Using HSV interpolation to display color gradient, the effect is as follows. This method creates a more vibrant color gradient.



Controlling Index

For drivers supporting index control, we can manipulate the state of each light on the strip using macros like INSERT_INDEX, SET_IHSV, SET_IRGB. Setting a value of MAX_INDEX:127 indicates setting all the lights.

Example 1: Defining a color pattern where light at index 0 displays red, index 1 displays green, index 2 displays blue, and then exits.

```
const blink_step_t test_blink_index_setting1[] = {
    {LED_BLINK_RGB, SET_IRGB(0, 255, 0, 0), 0},      // step1: set index 0 to red␣
↪color 0 ms
    {LED_BLINK_RGB, SET_IRGB(1, 0, 255, 0), 0},      // step2: set index 1 to␣
↪green color 0 ms
```

(continues on next page)

(continued from previous page)

```
    {LED_BLINK_RGB, SET_IRGB(2, 0, 0, 255), 0},      // step3: set index 2 to blue␣
↪color 0 ms
    {LED_BLINK_LOOP, 0, 0},                           // step4: loop from step1
};
```

Example 2: Defining a color pattern where all lights breathe continuously.

```
const blink_step_t test_blink_all_breath[] = {
    {LED_BLINK_BRIGHTNESS, INSERT_INDEX(MAX_INDEX, LED_STATE_OFF), 0},      //␣
↪step1: set all leds to off 0 ms
    {LED_BLINK_BREATHE, INSERT_INDEX(MAX_INDEX, LED_STATE_ON), 1000},      //␣
↪step2: set all leds fade to on 1000 ms
    {LED_BLINK_BREATHE, INSERT_INDEX(MAX_INDEX, LED_STATE_OFF), 1000},     //␣
↪step3: set all leds fade to off 1000 ms
    {LED_BLINK_LOOP, 0, 0},                                                // step4:␣
↪loop from step1
};
```

4.3.3 Predefined Blinking Priorities

These examples demonstrate controlling individual or all lights on the strip using specific indexes or macros like MAX_INDEX.

For the same indicator, a high-priority blink can interrupt an ongoing low-priority blink, which will resume execution after the high-priority blink stop. The blink priority can be adjusted by configuring the enumeration member order of the blink type `led_indicator_blink_type_t`, the smaller order value the higher execution priority.

For instance, in the following example, `test_blink_one_time` has higher priority than `test_blink_loop`, and should blink first:

```
typedef enum {
    BLINK_TEST_BLINK_ONE_TIME, /**< test_blink_one_time */
    BLINK_TEST_BLINK_LOOP,     /**< test_blink_loop */
    BLINK_MAX,                 /**< INVALID type */
} led_indicator_blink_type_t;
```

4.3.4 Control Indicator Blinks

Create an indicator by specifying an IO and a set of configuration information.

```
led_indicator_gpio_config_t led_indicator_gpio_config = {
     .gpio_num = 1,              /**< num of GPIO */
     .is_active_level_high = 1,
 };
 led_indicator_config_t config = {
     .blink_lists = led_indicator_get_sample_lists(),
     .blink_list_num = led_indicator_get_sample_lists_num(),
 };
 esp_err_t ret = led_indicator_new_gpio_device(&config, &led_indicator_gpio_config,
↪ &led_handle);
```

Start/stop blinking: control your indicator to start/stop a specified type of blink by calling corresponding functions. The functions are returned immediately after calling, and the blink process is controlled by the internal timer. The same indicator can perform multiple blink types in turn based on their priorities.

```
led_indicator_start(led_handle, BLINK_TEST_BLINK_LOOP); // call to start, the␣
↪function not block
```

(continues on next page)

(continued from previous page)

```
/*
*......
*/

led_indicator_stop(led_handle, BLINK_TEST_BLINK_LOOP); // call stop
```

Delete an indicator: you can also delete an indicator to release resources if there are no further operations required.

```
led_indicator_delete(&led_handle);
```

Preempt operation: You can flash the specified type directly at any time.

```
led_indicator_preempt_start(led_handle, BLINK_TEST_BLINK_LOOP);
```

Stop preempt: You can use the stop queueing function to cancel the blinking mode that is being queued.

```
led_indicator_preempt_stop(led_handle, BLINK_TEST_BLINK_LOOP);
```

Note: This component supports thread-safe operations. You can share the LED indicator handle `led_indicator_handle_t` with global variables, or use `led_indicator_get_handle` to get the handle in other threads via the LED' s IO number for operation.

4.3.5 Custom light blink

```
static blink_step_t const *led_blink_lst[] = {
    [BLINK_DOUBLE] = double_blink,
    [BLINK_TRIPLE] = triple_blink,
    [BLINK_NUM] = NULL,
};

led_indicator_gpio_config_t led_indicator_gpio_config = {
    .gpio_num = 1,              /**< num of GPIO */
    .is_active_level_high = 1,
};

led_indicator_config_t config = {
    .blink_lists = led_blink_lst,,
    .blink_list_num = BLINK_MAX,,
};
```

By defining `led_blink_lst[]` to achieve the custom indicator.

4.3.6 Adjustment of Gamma

The way human eyes perceive brightness is not linear but has certain nonlinear characteristics. Under normal conditions, the human eye is more sensitive to darker areas and less sensitive to brighter areas. However, on digital display devices such as monitors, the brightness values of images are usually encoded in a linear manner. This leads to issues of brightness distortion or loss of details when converting the linearly encoded brightness values to the perceived brightness by the human eye. To address this problem, gamma correction is applied to the image. Gamma correction involves adjusting the brightness values nonlinearly to correct the image display. By applying a gamma value (typically ranging from 2.2 to 2.4), the linearly encoded brightness values are mapped to a nonlinear brightness curve that better matches the perception of the human eye. This improves the visibility of details in darker areas and enhances the overall visual accuracy and balance of the image.

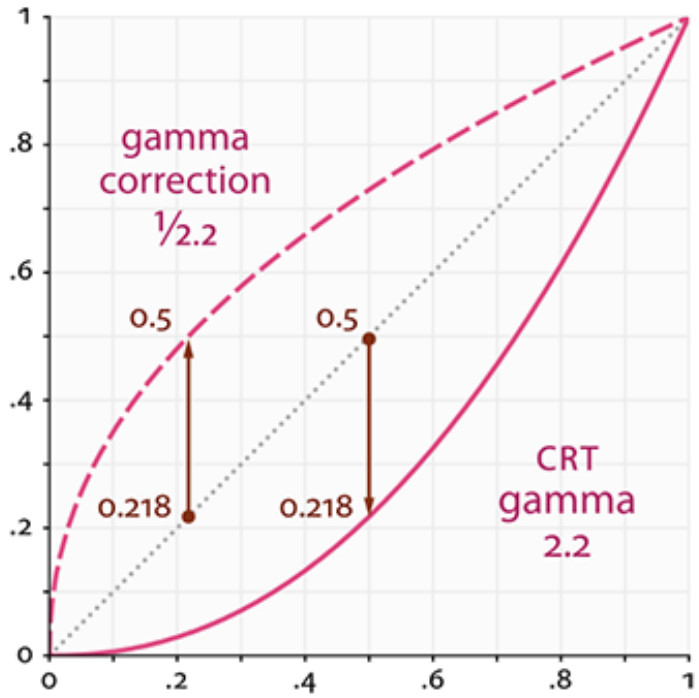


Fig. 46: Gamma Curve

```
float gamma = 2.3;
led_indicator_new_gamma_table(gamma);
```

The default gamma table is 2.3, and a new gamma table can be generated using the *led_indicator_new_gamma_table()* function.

4.3.7 Drive Level Setting

For different hardware configurations, it might involve either common anode or common cathode connections. You can adjust the *is_active_level_high* in the settings to either *true* or *false* to configure the drive level.

API Reference

Header File

- components/led/led_indicator/include/led_indicator.h

Functions

esp_err_t **led_indicator_delete**(led_indicator_handle_t handle)

delete the LED indicator and release resource

Parameters **handle** –pointer to LED indicator handle

Returns esp_err_t

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK Success
- ESP_FAIL Delete fail

esp_err_t **led_indicator_start** (led_indicator_handle_t handle, int blink_type)

start a new blink_type on the LED indicator. if multiple blink_type started simultaneously, it will be executed according to priority.

Parameters
- **handle** –LED indicator handle
- **blink_type** –predefined blink type

Returns esp_err_t
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NOT_FOUND no predefined blink_type found
- ESP_OK Success

esp_err_t **led_indicator_stop** (led_indicator_handle_t handle, int blink_type)

stop a blink_type. you can stop a blink_type at any time, no matter it is executing or waiting to be executed.

Parameters
- **handle** –LED indicator handle
- **blink_type** –predefined blink type

Returns esp_err_t
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NOT_FOUND no predefined blink_type found
- ESP_OK Success

esp_err_t **led_indicator_preempt_start** (led_indicator_handle_t handle, int blink_type)

Immediately execute an action of any priority. Until the action is executed, or call led_indicator_preempt_stop().

Parameters
- **handle** –LED indicator handle
- **blink_type** –predefined blink type

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail
- ESP_ERR_INVALID_ARG if parameter is invalid

esp_err_t **led_indicator_preempt_stop** (led_indicator_handle_t handle, int blink_type)

Stop the current preemptive action.

Parameters
- **handle** –LED indicator handle
- **blink_type** –predefined blink type

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail
- ESP_ERR_INVALID_ARG if parameter is invalid

uint8_t **led_indicator_get_brightness** (led_indicator_handle_t handle)

Get the current brightness value of the LED indicator.

Parameters **handle** –LED indicator handle

Returns uint8_t Current brightness value: 0-255 if handle is null return 0

esp_err_t **led_indicator_set_on_off** (led_indicator_handle_t handle, bool on_off)

Set the LED indicator on or off.

Note: If you have an RGB/Strips type of light, this API will control the last LED index you set, and the color will be displayed based on the last color you set.

Parameters
- **handle** –LED indicator handle.

- **on_off** –true: on, false: off

Returns esp_err_t
- ESP_OK: Success
- ESP_FAIL: Failure
- ESP_ERR_INVALID_ARG: Invalid parameter

esp_err_t **led_indicator_set_brightness**(led_indicator_handle_t handle, uint32_t brightness)

Set the brightness for the LED indicator.

Parameters
- **handle** –LED indicator handle.
- **brightness** –Brightness value to set (0 to 255). You can control a specific LED by specifying the index using `SET_IB`, and set it to MAX_INDEX 127 to control all LEDs. This feature is only supported for LEDs of type LED_RGB_MODE. Index: (0-126), set (127) to control all.

Returns esp_err_t
- ESP_OK: Success
- ESP_FAIL: Failure
- ESP_ERR_INVALID_ARG: Invalid parameter

uint32_t **led_indicator_get_hsv**(led_indicator_handle_t handle)

Get the HSV color of the LED indicator.

Note: Index settings are only supported for LED_RGB_MODE.

Parameters **handle** –LED indicator handle.

Returns HSV color value H: 0-360, S: 0-255, V: 0-255

esp_err_t **led_indicator_set_hsv**(led_indicator_handle_t handle, uint32_t ihsv_value)

Set the HSV color for the LED indicator.

Note: Index settings are only supported for LED_RGB_MODE.

Parameters
- **handle** –LED indicator handle.
- **ihsv_value** –HSV color value to set. I: 0-126, set 127 to control all H: 0-360, S: 0-255, V: 0-255

Returns esp_err_t
- ESP_OK: Success
- ESP_FAIL: Failure
- ESP_ERR_INVALID_ARG: Invalid parameter

uint32_t **led_indicator_get_rgb**(led_indicator_handle_t handle)

Get the RGB color of the LED indicator.

Note: Index settings are only supported for LED_RGB_MODE.

Parameters **handle** –LED indicator handle.

Returns RGB color value (0xRRGGBB) R: 0-255, G: 0-255, B: 0-255

esp_err_t **led_indicator_set_rgb**(led_indicator_handle_t handle, uint32_t irgb_value)

Set the RGB color for the LED indicator.

Note: Index settings are only supported for LED_RGB_MODE.

Parameters
- **handle** –LED indicator handle.
- **irgb_value** –RGB color value to set (0xRRGGBB). I: 0-126, set 127 to control all R: 0-255, G: 0-255, B: 0-255

Returns esp_err_t
- ESP_OK: Success
- ESP_FAIL: Failure
- ESP_ERR_INVALID_ARG: Invalid parameter

esp_err_t **led_indicator_set_color_temperature**(led_indicator_handle_t handle, const uint32_t temperature)

Set the color temperature for the LED indicator.

Note: Index settings are only supported for LED_RGB_MODE.

Parameters
- **handle** –LED indicator handle.
- **temperature** –Color temperature of LED (0xIITTTTTT) I: 0-126, set 127 to control all, TTTTTT: 0-1000000

Returns esp_err_t
- ESP_OK: Success
- ESP_FAIL: Failure
- ESP_ERR_INVALID_ARG: Invalid parameter

Macros

LED_INDICATOR_CHECK(a, str, action)

LED_INDICATOR_CHECK_WARNING(a, str, action)

4.4 LCD Tools

4.4.1 ESP LVGL ADAPTER

`esp_lvgl_adapter` provides a runtime-managed LVGL integration for ESP-IDF projects. It unifies display registration, tearing control, thread-safe LVGL access, and optional integration with filesystem, image decoding, and FreeType fonts. It supports LVGL v8 and v9.

Features

- **Unified display registration**: Supports MIPI DSI / RGB / QSPI / SPI / I2C / I80 interfaces
- **Tearing control**: Multiple tearing modes
- **Thread safety**: Lock/unlock APIs for safe multi-threaded LVGL access
- **Optional modules** (via Kconfig):
  - Filesystem bridge (`esp_lv_fs`)

  - Image decoder (`esp_lv_decoder`)
  - FreeType vector font loader
  - FPS statistics, Dummy Draw (headless rendering)
- **Multi-display support**: Manage multiple displays simultaneously

Add to Project

Via Component Service This component is available from Espressif's component service. Add it to your project with:

```
idf.py add-dependency "espressif/esp_lvgl_adapter"
```

In `idf_component.yml` Alternatively, declare it in your component manifest:

```yaml
dependencies:
  espressif/esp_lvgl_adapter: "*"
```

Requirements

- **ESP-IDF**: >= 5.5
- **LVGL**: 8.x or 9.x (auto-detected)
- **Optional dependencies**: Required only if corresponding Kconfig options are enabled

Quick Start

Here's a minimal example showing how to initialize the adapter and register a display:

```
#include "esp_lv_adapter.h"  // Includes display & input adapters

void app_main(void)
{
    // Step 0: Create your esp_lcd panel and (optionally) panel_io with esp_lcd␣
↪APIs
    esp_lcd_panel_handle_t panel = /* ... */;
    esp_lcd_panel_io_handle_t panel_io = /* ... or NULL */;

    // Step 1: Initialize the adapter
    esp_lv_adapter_config_t cfg = ESP_LV_ADAPTER_DEFAULT_CONFIG();
    ESP_ERROR_CHECK(esp_lv_adapter_init(&cfg));

    // Step 2: Register a display (choose macro by interface)
    esp_lv_adapter_display_config_t disp_cfg = ESP_LV_ADAPTER_DISPLAY_RGB_DEFAULT_
↪CONFIG(
        panel,           // LCD panel handle
        panel_io,        // LCD panel IO handle (can be NULL for some interfaces)
        800,             // Horizontal resolution
        480,             // Vertical resolution
        ESP_LV_ADAPTER_ROTATE_0 // Rotation
    );
    lv_display_t *disp = esp_lv_adapter_register_display(&disp_cfg);
    assert(disp != NULL);

    // Step 3: (Optional) Register input device(s)
    // Create touch handle using esp_lcd_touch API (implementation omitted here)
    // esp_lcd_touch_handle_t touch_handle = /* ... */;
    // esp_lv_adapter_touch_config_t touch_cfg = ESP_LV_ADAPTER_TOUCH_DEFAULT_
↪CONFIG(disp, touch_handle);
```

(continues on next page)

(continued from previous page)

```
    // lv_indev_t *touch = esp_lv_adapter_register_touch(&touch_cfg);
    // assert(touch != NULL);

    // Step 4: Start the adapter task
    ESP_ERROR_CHECK(esp_lv_adapter_start());

    // Step 5: Draw with LVGL (guarded by adapter lock for thread safety)
    if (esp_lv_adapter_lock(-1) == ESP_OK) {
        lv_obj_t *label = lv_label_create(lv_scr_act());
        lv_label_set_text(label, "Hello LVGL!");
        lv_obj_center(label);
        esp_lv_adapter_unlock();
    }
}
```

Usage

This section guides you through integrating the LVGL adapter with ESP-IDF projects.

Recommended SDKCONFIG Settings Add these configurations to your project's `sdkconfig.defaults` for optimal LVGL performance:

Core LVGL Configurations

```
# FreeRTOS tick rate (improves timer precision)
CONFIG_FREERTOS_HZ=1000

# LVGL OS integration
CONFIG_LV_OS_FREERTOS=y

# Use standard C library functions
CONFIG_LV_USE_CLIB_MALLOC=y
CONFIG_LV_USE_CLIB_STRING=y
CONFIG_LV_USE_CLIB_SPRINTF=y

# Display refresh rate (in milliseconds, 15ms ≈ 66 FPS)
CONFIG_LV_DEF_REFR_PERIOD=15

# Enable style caching for better performance
CONFIG_LV_OBJ_STYLE_CACHE=y
```

Multi-core Optimization (ESP32-S3/P4) For chips with multiple cores, enable parallel rendering:

```
# Use 2 drawing threads for better performance
CONFIG_LV_DRAW_SW_DRAW_UNIT_CNT=2
```

PSRAM Optimization (When Available) For devices with PSRAM, optimize cache and XIP settings:

ESP32-S3:

```
CONFIG_SPIRAM_XIP_FROM_PSRAM=y
CONFIG_ESP32S3_INSTRUCTION_CACHE_32KB=y
CONFIG_ESP32S3_DATA_CACHE_64KB=y
CONFIG_ESP32S3_DATA_CACHE_LINE_64B=y
```

ESP32-P4:

```
CONFIG_SPIRAM_XIP_FROM_PSRAM=y
CONFIG_CACHE_L2_CACHE_256KB=y
CONFIG_CACHE_L2_CACHE_LINE_128B=y
```

Decision Flow Before starting, make these decisions:

1. **Pick interface type**: Determine your display interface (MIPI DSI / RGB / QSPI / SPI / I2C / I80)
2. **Pick default macro**: Select corresponding `ESP_LV_ADAPTER_DISPLAY_XXX_DEFAULT_CONFIG` macro
3. **Pick appropriate tearing mode** (RGB/MIPI DSI only):
   - Compute required frame buffer count (`num_fbs`)
   - Pass `num_fbs` to esp_lcd panel config
4. **Tune buffer size**: Adjust `buffer_height` based on memory vs throughput needs

Display Registration and Tearing

Interface Types and Default Macros Choose the configuration macro based on your display interface:

Interface Type	Default Config Macro	Default Tearing Mode
MIPI DSI	`ESP_LV_ADAPTER_DISPLAY_MIPI_DEFAULT_CONFIG(..)`	`ESP_LV_ADAPTER_TEAR_AVOID_MODE_DEFAULT`
RGB	`ESP_LV_ADAPTER_DISPLAY_RGB_DEFAULT_CONFIG(..)`	`ESP_LV_ADAPTER_TEAR_AVOID_MODE_DEFAULT`
SPI/I2C/I80/QSPI (with PSRAM)	`ESP_LV_ADAPTER_DISPLAY_SPI_WITH_PSRAM_DEFAULT_CONFIG(..)`	`ESP_LV_ADAPTER_TEAR_AVOID_MODE_DEFAULT` (i.e., `NONE`)
SPI/I2C/I80/QSPI (without PSRAM)	`ESP_LV_ADAPTER_DISPLAY_SPI_WITHOUT_PSRAM_DEFAULT_CONFIG(..)`	`ESP_LV_ADAPTER_TEAR_AVOID_MODE_DEFAULT` (i.e., `NONE`)

Note:

- Only MIPI DSI and RGB support tearing modes
- SPI/I2C/I80/QSPI are collectively called "OTHER" interfaces in the adapter and support only `NONE` mode

Computing Frame Buffer Count For RGB/MIPI DSI, compute required frame buffers early before hardware initialization:

```
uint8_t num_fbs = esp_lv_adapter_get_required_frame_buffer_count(
    ESP_LV_ADAPTER_TEAR_AVOID_MODE_DEFAULT_RGB,  // Tearing mode
    ESP_LV_ADAPTER_ROTATE_0                      // Rotation
);
// Pass num_fbs to your esp_lcd panel config
```

Frame Buffer Count Rules:

- 90°/270° rotation or triple-buffer modes: **3** frame buffers
- Double-buffer modes: **2** frame buffers
- Single-buffer mode: **1** frame buffer

Tearing Modes: Selector (RGB/MIPI DSI) Choose the appropriate tearing mode based on your use case:

Tearing Mode	Use Case	Frame Buffers	Memory
`TRIPLE_PARTIAL`	90°/270° rotation, High-res smooth UI	3	High
`TRIPLE_FULL`	Full-screen/large-area updates, Plenty of RAM	3	High
`DOUBLE_FULL`	Large-area updates, Tighter RAM	2	Medium
`DOUBLE_DIRECT`	Small-area updates, Widget/UI deltas	2	Medium
`NONE`	Static UI, Ultra-low RAM	1	Low

Warning:

- RGB/MIPI DSI with `TEAR_AVOID_MODE_NONE` **forbids rotation** (any non-zero rotation is rejected)
- OTHER (SPI/I2C/I80/QSPI) interfaces support `NONE` only; for rotation, configure panel orientation (swap XY/mirror) during LCD initialization and adjust touch mapping accordingly

Memory Estimation Full-frame buffer size ≈ `horizontal_res` × `vertical_res` × `bytes_per_pixel`

Example (RGB565 format, 2 bytes/pixel):

- 800×480 resolution: ≈ 750 KB
- Triple buffering: ≈ 2.2 MB
- **Conclusion**: PSRAM is typically required

Buffer Tuning

Default `buffer_height` Values Default `buffer_height` for each interface (see `esp_lv_adapter_display.h`):

Interface Type	Default `buffer_height`	Description
MIPI DSI	50	Medium stripe height
RGB	50	Medium stripe height
OTHER with PSRAM	`ver_res` (full height)	Full-height partial refresh, better throughput
OTHER without PSRAM	10	Small stripe to save RAM

Tuning Tips

- **Increase** `buffer_height`:
  - ✅ Improves throughput (fewer flushes, larger stripes)
  - ❌ Increases RAM usage
- **Decrease** `buffer_height`:
  - ✅ Saves RAM
  - ❌ More flushes, may reduce performance

Thread-Safe LVGL Access LVGL APIs are **not thread-safe**. Guard all LVGL calls with the adapter's lock:

```
if (esp_lv_adapter_lock(-1) == ESP_OK) {
    lv_obj_t *btn = lv_btn_create(lv_scr_act());
    lv_obj_set_size(btn, 100, 50);
    esp_lv_adapter_unlock();
}
```

Input Devices

Touch The adapter provides a default configuration macro `ESP_LV_ADAPTER_TOUCH_DEFAULT_CONFIG` to simplify touch device registration:

```
// Register touch device using default configuration macro
esp_lv_adapter_touch_config_t touch_cfg = ESP_LV_ADAPTER_TOUCH_DEFAULT_CONFIG(disp,
↪ touch_handle);
lv_indev_t *touch = esp_lv_adapter_register_touch(&touch_cfg);
assert(touch != NULL);
```

Parameters:

- `disp`: Associated LVGL display device
- `touch_handle`: Touch handle created via `esp_lcd_touch` API
- Default scale factors: x = 1.0, y = 1.0

Encoder/Knob Requires Kconfig option `ESP_LV_ADAPTER_ENABLE_KNOB`. See `esp_lv_adapter_input.h` for `esp_lv_adapter_encoder_config_t`.

Navigation Buttons Requires Kconfig option `ESP_LV_ADAPTER_ENABLE_BUTTON`. Use `esp_lv_adapter_register_navigation_buttons()`.

Filesystem Bridge (Optional) If `ESP_LV_ADAPTER_ENABLE_FS` is enabled, mount a memory-mapped asset partition as an LVGL drive. Indexing comes from `esp_mmap_assets`.

```
#include "esp_lv_fs.h"
#include "esp_mmap_assets.h"

// 1. Create mmap assets handle
mmap_assets_handle_t assets;
const mmap_assets_config_t mmap_cfg = {
    .partition_label = "assets_A",
    .max_files = MMAP_DRIVE_A_FILES,
    .checksum  = MMAP_DRIVE_A_CHECKSUM,
    .flags = { .mmap_enable = true },
};
ESP_ERROR_CHECK(mmap_assets_new(&mmap_cfg, &assets));

// 2. Mount as LVGL filesystem
esp_lv_fs_handle_t fs_handle;
const fs_cfg_t fs_cfg = {
    .fs_letter = 'A',           // Drive letter (A to Z)
    .fs_assets = assets,
    .fs_nums = MMAP_DRIVE_A_FILES,
};
ESP_ERROR_CHECK(esp_lv_adapter_fs_mount(&fs_cfg, &fs_handle));

// 3. Use file paths in LVGL, e.g., "A:my_image.png"
lv_obj_t *img = lv_img_create(lv_scr_act());
lv_img_set_src(img, "A:my_image.png");
```

Depends on esp_lv_fs and esp_mmap_assets components.

Image Decoding (Optional) Enable `ESP_LV_ADAPTER_ENABLE_DECODER` to integrate `esp_lv_decoder` with LVGL. Use LVGL image widgets as usual; load from the mounted drive:

```
// Use LVGL image widget to load images
lv_obj_t *img = lv_img_create(lv_scr_act());
lv_img_set_src(img, "I:red_png.png");  // I: is the decoder drive letter
```

Supported Formats:

- Standard: JPG, PNG, QOI
- Split formats: SJPG, SPNG, SQOI (optimized for embedded systems)
- Hardware acceleration: JPEG, PJPG (if chip supports)

For details, refer to esp_lv_decoder.

FreeType Fonts (Optional) Enable `ESP_LV_ADAPTER_ENABLE_FREETYPE` and ensure LVGL FreeType is enabled. Then initialize fonts from file or memory:

```
#include "esp_lv_adapter.h" // FreeType APIs under this header

// Load font from file
esp_lv_adapter_ft_font_handle_t fh;
esp_lv_adapter_ft_font_config_t cfg = {
    .name = "F:my_font.ttf",          // Font file path (Note: no slash after colon␣
↪for mmap assets)
    .size = 30,                       // Font size
    .style = ESP_LV_ADAPTER_FT_FONT_STYLE_NORMAL,
};
ESP_ERROR_CHECK(esp_lv_adapter_ft_font_init(&cfg, &fh));
const lv_font_t *font30 = esp_lv_adapter_ft_font_get(fh);

// Use the font
lv_obj_set_style_text_font(label, font30, 0);
```

Stack Configuration:

LVGL Version	Required Setting	Explanation
v8	`CONFIG_ESP_MAIN_TASK_STACK_SIZE=32768`	Font initialization runs on calling thread
v9	`CONFIG_LV_DRAW_THREAD_STACK_SIZE=32768`	Font rendering runs on draw threads

Note:

- LVGL v8: Does not support LVGL virtual filesystem (`lv_fs`), use direct file paths or memory buffers
- LVGL v9: Supports virtual filesystem paths (e.g., `"F:font.ttf"`)

FPS Statistics and Dummy Draw

FPS Statistics Requires `ESP_LV_ADAPTER_ENABLE_FPS_STATS`.

```
// Enable FPS statistics
ESP_ERROR_CHECK(esp_lv_adapter_fps_stats_enable(disp, true));

// Get current FPS
uint32_t fps;
if (esp_lv_adapter_get_fps(disp, &fps) == ESP_OK) {
    printf("Current FPS: %lu\n", fps);
}
```

Dummy Draw Dummy Draw mode bypasses LVGL rendering for direct display control:

```
ESP_ERROR_CHECK(esp_lv_adapter_set_dummy_draw(disp, true));
ESP_ERROR_CHECK(esp_lv_adapter_dummy_draw_blit(disp, 0, 0, 800, 480, framebuffer,␣
↪true));
ESP_ERROR_CHECK(esp_lv_adapter_set_dummy_draw(disp, false));
```

Immediate Refresh Force a synchronous refresh:

```
esp_lv_adapter_refresh_now(disp);
```

Configuration (Kconfig)

Configure via `idf.py menuconfig`:

Option	Description
`ESP_LV_ADAPTER_ENABLE_FS`	Enable filesystem bridge (`esp_lv_fs`)
`ESP_LV_ADAPTER_ENABLE_DECODER`	Enable image decoder (`esp_lv_decoder`)
`ESP_LV_ADAPTER_ENABLE_FREETYPE`	Enable FreeType vector font support
`ESP_LV_ADAPTER_ENABLE_FPS_STATS`	Enable FPS monitoring APIs
`ESP_LV_ADAPTER_ENABLE_BUTTON`	Enable navigation button input support
`ESP_LV_ADAPTER_ENABLE_KNOB`	Enable rotary encoder input support

Limitations & Notes

Interface and Tearing Mode Support

Interface Type	Supported Tearing Modes
RGB / MIPI DSI	`NONE`, `DOUBLE_FULL`, `TRIPLE_FULL`, `DOUBLE_DIRECT`, `TRIPLE_PARTIAL`
OTHER (SPI/I2C/I80/QSPI)	`NONE` only

Rotation Limitations

- **RGB/MIPI DSI**:
  - `TEAR_AVOID_MODE_NONE` **forbids rotation** (any non-zero rotation is rejected)
- **OTHER (SPI/I2C/I80/QSPI)**:
  - Adapter does not apply 90°/270° rotation
  - For rotation, configure panel orientation (swap XY/mirror) during LCD initialization
  - Must also adjust touch coordinate mapping

Buffering and Render Modes

- **Mode Mapping**:
  - `DOUBLE_FULL` / `TRIPLE_FULL` → FULL render mode
  - `DOUBLE_DIRECT` → DIRECT render mode
  - Others (incl. `TRIPLE_PARTIAL`) → PARTIAL render mode
- **LVGL Draw Buffer Count**:
  - `TRIPLE_PARTIAL`: 1 buffer
  - `TRIPLE_FULL` / `DOUBLE_*`: 2 buffers
  - FULL/DIRECT default: 2 buffers
  - Otherwise: 1 buffer
- **OTHER + NONE**:
  - No panel frame buffers requested
  - Adapter sends partial stripes from LVGL draw buffer

PPA/DMA2D Acceleration

- With PPA acceleration, set `LV_DRAW_SW_DRAW_UNIT_CNT == 1` for best behavior
- Rotation uses PPA when possible; otherwise CPU copy with cache-friendly blocks
- **Effect**:
  - PPA reduces CPU load
  - Usually does not deliver noticeable FPS gains

Deinitialization

Recommended cleanup order when shutting down UI:

```
// 1. Unregister input devices
esp_lv_adapter_unregister_touch(touch);
// esp_lv_adapter_unregister_encoder(encoder);
// esp_lv_adapter_unregister_navigation_buttons(buttons);

// 2. Unregister display(s)
esp_lv_adapter_unregister_display(disp);

// 3. Unmount filesystem(s) (if used)
esp_lv_adapter_fs_unmount(fs_handle);
// Release mmap assets
mmap_assets_del(assets);

// 4. Deinitialize adapter
// Note: FreeType fonts are auto-cleaned if enabled
esp_lv_adapter_deinit();
```

Compatibility

- **ESP-IDF**: >= 5.5
- **LVGL**: >= 8, < 10 (v8/v9 supported)

ESP-IDF Patches

PPA Freeze Fix (ESP32-P4) If you encounter display freeze when using `ESP_LV_ADAPTER_TEAR_AVOID_MODE_TRIPLE_PARTIAL` mode with screen rotation enabled on ESP32-P4, you need to apply the following patch:

Target Version: ESP-IDF release/v5.5 (commit `62beeae461bd3692c2028f96a93c84f11291e155`)

Patch File: `0001-bugfix-lcd-Fixed-PPA-freeze.patch`

How to Apply:

In your ESP-IDF repository root directory:

```
cd $IDF_PATH
git apply /path/to/esp_lvgl_adapter/0001-bugfix-lcd-Fixed-PPA-freeze.patch
```

Issue Description:

- This patch fixes a display freeze issue on ESP32-P4 when using `ESP_LV_ADAPTER_TEAR_AVOID_MODE_TRIPLE_PARTIAL` mode with rotation enabled, caused by PPA hardware acceleration
- Only affects scenarios that meet ALL of the following conditions:
  - Using ESP32-P4 chip
  - Using `TRIPLE_PARTIAL` tear avoidance mode
  - Screen rotation enabled (90°/270°)

- Not required for other configuration combinations or chips

Verify Patch Applied:

Check if `components/esp_driver_ppa/src/ppa_srm.c` contains the following code:

```
PPA.sr_byte_order.sr_macro_bk_ro_bypass = 1;
```

Glossary

Term	Description
Panel frame buffers	Hardware frame buffers owned by the panel driver
LVGL draw buffer	RAM used by LVGL to render stripes/frames before flush
Partial refresh	Updating a sub-region (dirty area) of the screen instead of full screen
Direct mode	LVGL renders directly to the frame buffer (reduced copies; suited to small-area updates)
PPA	Pixel Processing Accelerator used for hardware-accelerated rotate/scale/memcpy-like operations
Tearing	Visual artifact when screen updates while being scanned; mitigated by buffering strategies

Examples

Example	Description
`lvgl_common_demo`	Unified LVGL demo supporting multiple LCD interfaces
`lvgl_multi_screen`	Multi-display management
`lvgl_decode_image`	Image decoder showcase
`lvgl_freetype_font`	FreeType font loading and usage
`lvgl_dummy_draw`	Dummy Draw switching

Troubleshooting

FreeType Crashes or Renders Incorrectly

Solutions:

- Ensure LVGL FreeType is enabled (`CONFIG_LV_USE_FREETYPE=y`)
- LVGL v8: Set `CONFIG_ESP_MAIN_TASK_STACK_SIZE=32768`
- LVGL v9: Set `CONFIG_LV_DRAW_THREAD_STACK_SIZE=32768`

Screen Tearing or Flicker

Possible Causes:

- Inappropriate tearing mode
- Incorrect frame buffer count
- Improper rotation config

Solutions:

1. Choose appropriate tearing mode based on use case
2. Use `esp_lv_adapter_get_required_frame_buffer_count()` to compute correct frame buffer count

Decoder or FS Not Working

Possible Causes:

- Kconfig options not enabled
- Component dependencies missing
- Asset partition not flashed

Solutions:

1. Verify corresponding Kconfig options
2. Ensure dependencies like `esp_mmap_assets` are added
3. Verify asset partition is properly packed and flashed

LVGL Object API Crashes **Possible Cause**:

- Accessing LVGL APIs without lock in multi-threaded environment

Solution:

```
// All LVGL API calls must be protected by lock
if (esp_lv_adapter_lock(-1) == ESP_OK) {
    // LVGL API calls
    esp_lv_adapter_unlock();
}
```

API Reference

Header File

- components/display/tools/esp_lvgl_adapter/include/esp_lv_adapter.h

Functions

esp_err_t **esp_lv_adapter_init** (const *esp_lv_adapter_config_t* *config)

Initialize the LVGL adapter.

This function initializes the LVGL library and creates necessary resources. Must be called before any other LVGL adapter functions.

Parameters **config** –**[in]** Pointer to LVGL adapter configuration

Returns

- ESP_OK: Success
- ESP_ERR_INVALID_STATE: Adapter already initialized
- ESP_ERR_INVALID_ARG: Invalid configuration parameter
- ESP_ERR_NO_MEM: Memory allocation failed

esp_err_t **esp_lv_adapter_start** (void)

Start the LVGL adapter task.

Creates and starts the LVGL task for handling timer events.

Returns

- ESP_OK: Success
- ESP_ERR_INVALID_STATE: Adapter not initialized
- ESP_ERR_NO_MEM: Task creation failed

esp_err_t **esp_lv_adapter_deinit** (void)

Deinitialize the LVGL adapter.

Stops the LVGL task and releases all resources.

Returns

- ESP_OK: Success

bool **esp_lv_adapter_is_initialized** (void)

Check if the LVGL adapter is initialized.

Returns

- true: Adapter is initialized
- false: Adapter is not initialized

esp_err_t **esp_lv_adapter_lock** (int32_t timeout_ms)

Acquire the LVGL adapter lock.

Must be called before accessing LVGL objects from user code.

Parameters **timeout_ms** –**[in]** Timeout in milliseconds, use -1 for infinite wait

Returns

- ESP_OK: Lock acquired successfully
- ESP_ERR_INVALID_STATE: Adapter not initialized
- ESP_ERR_TIMEOUT: Failed to acquire lock within timeout

void **esp_lv_adapter_unlock** (void)

Release the LVGL adapter lock.

esp_err_t **esp_lv_adapter_refresh_now** (lv_display_t *disp)

Force LVGL to refresh the display immediately.

This function triggers an immediate refresh of the LVGL display, forcing all dirty areas to be redrawn without waiting for the next timer period. Useful when you need to update the display synchronously.

Parameters **disp** –**[in]** Pointer to LVGL display object (NULL for default display)

Returns

- ESP_OK: Success
- ESP_ERR_INVALID_STATE: Adapter not initialized

esp_err_t **esp_lv_adapter_pause** (int32_t timeout_ms)

Pause LVGL worker.

Stops the LVGL worker task and waits for acknowledgement. This API is intended for advanced use cases requiring custom control over the LVGL rendering loop.

Note: You MUST pair this call with esp_lv_adapter_resume()

Note: Do NOT call esp_lv_adapter_sleep_prepare() while manually paused

Note: Handle timeout errors appropriately

Parameters **timeout_ms** –**[in]** Timeout in ms, -1 for infinite

Returns

- ESP_OK: Success
- ESP_ERR_TIMEOUT: Timeout waiting for pause acknowledgement
- ESP_ERR_INVALID_STATE: Adapter not initialized

esp_err_t **esp_lv_adapter_resume** (void)

Resume LVGL worker.

Resumes the LVGL worker task that was previously paused. Do not call this if the adapter was paused by esp_lv_adapter_sleep_prepare().

Returns

- ESP_OK: Success
- ESP_ERR_INVALID_STATE: Adapter not initialized

esp_err_t **esp_lv_adapter_sleep_prepare** (void)

Prepare all displays for sleep.

Automatically detaches all LCD panels while preserving LVGL display objects and UI state. This allows you to safely delete LCD hardware handles to release DMA and framebuffer resources.

Note: This function automatically pauses the LVGL worker and waits for all pending flush operations to complete.

Note: After calling this function, you can safely call esp_lcd_panel_del() for each display's panel handle.

Note: Touch inputs remain registered. If you need to power down touch hardware, suspend it manually before calling this API.

Returns

- ESP_OK: Success
- ESP_ERR_INVALID_STATE: Adapter not initialized or already sleeping

esp_err_t **esp_lv_adapter_sleep_recover**(lv_display_t *disp, esp_lcd_panel_handle_t panel, esp_lcd_panel_io_handle_t panel_io)

Recover a display from sleep.

Re-binds a new LCD panel to an existing LVGL display, automatically updating all framebuffer references. The UI elements are preserved.

Note: LCD hardware must be initialized before calling this function.

Note: This function can be called multiple times to recover multiple displays. When all displays are recovered, the adapter automatically resumes the LVGL worker.

Note: This function automatically triggers a full screen refresh for the display.

Parameters

- **disp** –**[in]** LVGL display handle to rebind
- **panel** –**[in]** New LCD panel handle
- **panel_io** –**[in]** New LCD panel IO handle (can be NULL for RGB/MIPI DSI)

Returns

- ESP_OK: Success
- ESP_ERR_INVALID_STATE: Adapter not initialized or not sleeping
- ESP_ERR_INVALID_ARG: Invalid handles
- ESP_FAIL: Failed to rebind panel

esp_err_t **esp_lv_adapter_set_dummy_draw**(lv_display_t *disp, bool enable)

Enable or disable dummy draw mode for a display.

In dummy draw mode, LVGL rendering is bypassed and the application takes direct control of the display. Use esp_lv_adapter_dummy_draw_blit() to update the screen.

Note: Manages locking internally. Disabling triggers a full screen refresh.

Parameters

- **disp** –**[in]** Pointer to LVGL display object
- **enable** –**[in]** true to enable dummy draw, false to disable

Returns

- ESP_OK: Success
- ESP_ERR_INVALID_STATE: Adapter not initialized
- ESP_ERR_INVALID_ARG: Invalid display pointer

bool **esp_lv_adapter_get_dummy_draw_enabled** (lv_display_t *disp)

Get the current dummy draw mode state for a display.

This function retrieves whether dummy draw mode is currently enabled. In dummy draw mode, LVGL rendering is bypassed and the application has direct control over the framebuffer.

Parameters **disp** –**[in]** Pointer to LVGL display object

Returns
- true: Dummy draw mode is enabled
- false: Dummy draw mode is disabled (or adapter not initialized/invalid display)

esp_err_t **esp_lv_adapter_set_dummy_draw_callbacks** (lv_display_t *disp, const *esp_lv_adapter_dummy_draw_callbacks_t* *cbs, void *user_ctx)

Register dummy draw callbacks for a display.

Callbacks allow applications to hook into dummy draw transitions and VSYNC notifications (useful for self-refresh flows). Registering new callbacks replaces any previously registered ones. Pass NULL to remove callbacks.

Parameters
- **disp** –Display handle
- **cbs** –Pointer to callback collection (NULL to clear)
- **user_ctx** –User context pointer passed back to callbacks

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_STATE: Adapter not initialized
- ESP_ERR_INVALID_ARG: Invalid display handle

esp_err_t **esp_lv_adapter_dummy_draw_blit** (lv_display_t *disp, int x_start, int y_start, int x_end, int y_end, const void *frame_buffer, bool wait)

Blit a framebuffer region to display in dummy draw mode.

Directly sends framebuffer to display hardware, bypassing LVGL.

Parameters
- **disp** –**[in]** Pointer to LVGL display object
- **x_start** –**[in]** Starting X coordinate (inclusive)
- **y_start** –**[in]** Starting Y coordinate (inclusive)
- **x_end** –**[in]** Ending X coordinate (exclusive)
- **y_end** –**[in]** Ending Y coordinate (exclusive)
- **frame_buffer** –**[in]** Pointer to framebuffer data
- **wait** –**[in]** true to wait for completion, false to return immediately

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_STATE: Adapter not initialized or dummy draw not enabled
- ESP_ERR_INVALID_ARG: Invalid parameters

Structures

struct **esp_lv_adapter_config_t**

LVGL adapter configuration structure.

Public Members

uint32_t **task_stack_size**

LVGL task stack size in bytes

uint32_t **task_priority**

LVGL task priority

int **task_core_id**

LVGL task core ID (-1 for no affinity)

uint32_t **tick_period_ms**

LVGL tick period in milliseconds

uint32_t **task_min_delay_ms**

Minimum LVGL task delay in milliseconds

uint32_t **task_max_delay_ms**

Maximum LVGL task delay in milliseconds

bool **stack_in_psram**

Allocate LVGL task stack in PSRAM when available

struct **esp_lv_adapter_dummy_draw_callbacks_t**

Dummy draw callback collection.

All callbacks are optional; set any member to NULL to omit it.

Note: The on_vsync callback is executed in interrupt context on most panels, so implementations must be ISR-safe.

Public Members

void (*__on_enable__)(lv_display_t *disp, void *user_ctx)

Called when dummy draw mode is enabled

void (*__on_disable__)(lv_display_t *disp, void *user_ctx)

Called when dummy draw mode is disabled

void (*__on_color_trans_done__)(lv_display_t *disp, bool in_isr, void *user_ctx)

Called when color transfer is done

void (*__on_vsync__)(lv_display_t *disp, bool in_isr, void *user_ctx)

Called on VSYNC signal (ISR context)

Macros

ESP_LV_ADAPTER_DEFAULT_STACK_SIZE

Default configuration values for LVGL adapter.

Default task stack size (8KB)

ESP_LV_ADAPTER_DEFAULT_TASK_PRIORITY

Default task priority

ESP_LV_ADAPTER_DEFAULT_TASK_CORE_ID

Default task core ID (-1 for no affinity)

ESP_LV_ADAPTER_DEFAULT_TICK_PERIOD_MS

Default tick period in milliseconds

ESP_LV_ADAPTER_DEFAULT_TASK_MIN_DELAY_MS

Default minimum task delay in milliseconds

ESP_LV_ADAPTER_DEFAULT_TASK_MAX_DELAY_MS

Default maximum task delay in milliseconds

ESP_LV_ADAPTER_DEFAULT_CONFIG()

Default configuration for LVGL adapter.

4.4.2 ESP LV DECODER

Allow the use of images in LVGL. Besides that it also allows the use of a custom format, called Split image, which can be decoded in more optimal way on embedded systems.

Referencing the implementation of SJPG.

Features

- Supports both standard and custom split image formats, including JPG, PNG, and QOI.
- Decoding standard image requires RAM equivalent to the full uncompressed image size (recommended for devices with more RAM).
- Split image is a custom format based on standard image formats, specifically designed for LVGL.
- File reads are implemented for both file storage and C-arrays.
- Split images are decoded in segments, so zooming and rotating are not supported.

Usage

The esp_mmap_assets component is required. It automatically packages and converts images to your desired format during compilation.

Converting JPG to SJPG

```
spiffs_create_partition_assets(
     my_spiffs_partition
     my_folder
     FLASH_IN_PROJECT
     MMAP_FILE_SUPPORT_FORMAT ".jpg"
     MMAP_SUPPORT_SJPG
     MMAP_SPLIT_HEIGHT 16)
```

Converting PNG to SPNG

```
spiffs_create_partition_assets(
     my_spiffs_partition
     my_folder
     FLASH_IN_PROJECT
     MMAP_FILE_SUPPORT_FORMAT ".png"
     MMAP_SUPPORT_SPNG
     MMAP_SPLIT_HEIGHT 16)
```

Converting PNG、JPG to QOI

```
spiffs_create_partition_assets(
     my_spiffs_partition
     my_folder
     FLASH_IN_PROJECT
     MMAP_FILE_SUPPORT_FORMAT ".jpg,.png"
     MMAP_SUPPORT_QOI)
```

Converting PNG、JPG to SQOI

```
spiffs_create_partition_assets(
     my_spiffs_partition
     my_folder
     FLASH_IN_PROJECT
     MMAP_FILE_SUPPORT_FORMAT ".jpg,.png"
     MMAP_SUPPORT_QOI
     MMAP_SUPPORT_SQOI
     MMAP_SPLIT_HEIGHT 16)
```

Application Examples

Register Decoder Register the decoder function after LVGL starts.

```
esp_lv_decoder_handle_t decoder_handle = NULL;
esp_lv_decoder_init(&decoder_handle); //Initialize this after lvgl starts
```

API Reference

Header File

- components/display/tools/esp_lv_decoder/include/esp_lv_decoder.h

Functions

esp_err_t **esp_lv_decoder_init** (*esp_lv_decoder_handle_t* *ret_handle)

Register the decoder functions in LVGL.

Parameters **ret_handle** –Pointer to the handle where the decoder handle will be stored

Returns

- ESP_OK on success
- ESP_ERR_* error codes on failure

esp_err_t **esp_lv_decoder_deinit** (*esp_lv_decoder_handle_t* handle)

Deinitialize the decoder handle.

Parameters **handle** –The handle to be deinitialized

Returns

- ESP_OK on success
- ESP_ERR_* error codes on failure

Type Definitions

typedef void ***esp_lv_decoder_handle_t**

Type of handle for the decoder.

4.4.3 ESP LV FS

esp_lv_fs allows LVGL to use filesystems for accessing assets. This component integrates with *esp_mmap_assets* to efficiently manage file operations and supports multiple partitions.

Features

- Integrates with *esp_mmap_assets* for filesystem creation.
- Supports standard file operations: fopen, fclose, fread, ftell, and fseek.
- Uses the *esp_partition_read* API for efficient file access.
- Supports multiple partitions.

Dependencies

The esp_mmap_assets component is required. It provides file offset index relationship.

Application Examples

Register Filesystem Register the Filesystem after LVGL starts.

```
#include "esp_lv_fs.h"
#include "esp_mmap_assets.h"

esp_lv_fs_handle_t fs_drive_a_handle;
mmap_assets_handle_t mmap_drive_a_handle;

const mmap_assets_config_t asset_cfg = {
    .partition_label = "assets_A",
    .max_files = MMAP_DRIVE_A_FILES,
    .checksum = MMAP_DRIVE_A_CHECKSUM,
    .flags = {
        .mmap_enable = true,
    }
};
mmap_assets_new(&asset_cfg, &mmap_drive_a_handle);

const fs_cfg_t fs_drive_a_cfg = {
    .fs_letter = 'A',
    .fs_assets = mmap_drive_a_handle,
    .fs_nums = MMAP_DRIVE_A_FILES
};
esp_lv_fs_desc_init(&fs_drive_a_cfg, &fs_drive_a_handle);
```

API Reference

Header File

- components/display/tools/esp_lv_fs/include/esp_lv_fs.h

Functions

esp_err_t **esp_lv_fs_desc_init** (const *fs_cfg_t* *cfg, *esp_lv_fs_handle_t* *ret_handle)

Initialize file descriptors for the filesystem.

This function sets up the filesystem by initializing file descriptors based on the provided configuration. It allocates necessary memory and populates the file descriptors with information about the assets.

Parameters
- **cfg** –**[in]** Pointer to the filesystem configuration structure.
- **ret_handle** –**[out]** Pointer to the handle that will hold the initialized filesystem instance.

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_NO_MEM: Memory allocation failed

esp_err_t **esp_lv_fs_desc_deinit** (*esp_lv_fs_handle_t* handle)

Deinitialize the filesystem and clean up resources.

This function cleans up the filesystem by freeing allocated memory for file descriptors and other resources associated with the provided handle.

Parameters **handle** –**[in]** Handle to the filesystem instance to be deinitialized.

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid argument

Structures

struct **fs_cfg_t**

Configuration structure for the filesystem.

Public Members

char **fs_letter**

Filesystem letter identifier

int **fs_nums**

Number of filesystem instances

mmap_assets_handle_t **fs_assets**

Handle to memory-mapped assets

Type Definitions

typedef void ***esp_lv_fs_handle_t**

Filesystem handle type definition.

4.4.4 ESP MMAP ASSETS

This module is mainly used to package resources (such as images, fonts, etc.) and map them directly for user access. At the same time, it also integrates rich image preprocessing functions.

Features

Adding Import File Types
- Support for various file formats such as `.bin`, `.jpg`, `.ttf`, etc.

Multiple Access Modes
- **Partition Mode**: Access assets from ESP32 partition (default)
- **Memory Mapping Mode**: Direct memory access for maximum performance
- **File System Mode**: Access assets from file system for development and testing

Enable Split JPG
- Need to use `SJPG` to parse. Refer to the LVGL SJPG.

Enable Split PNG
- Need to use `SPNG` to parse. See the component esp_lv_decoder.

Set Split Height
- Set the split height, depends on `MMAP_SUPPORT_SJPG`, `MMAP_SUPPORT_SPNG` or `MMAP_SUPPORT_SQOI`.

Support QOI
- Supports converting images from JPG and PNG formats to QOI.
- Supports split qoi, which requires the use of `esp_lv_decoder` for parsing.

Supporting multiple partitions
- The spiffs_create_partition_assets function allows you to mount multiple partitions, each with its own processing logic.

Supporting LVGL Image Converter
- By enabling `MMAP_SUPPORT_RAW` and related configurations, JPG and PNG can be converted to `Bin` so that they can be used in `LVGL`.

CMake Options

The following options are supported. These options allow you to configure various aspects of image handling.

```
set(options
    FLASH_IN_PROJECT,
    FLASH_APPEND_APP,
    MMAP_SUPPORT_SJPG,
    MMAP_SUPPORT_SPNG,
    MMAP_SUPPORT_QOI,
    MMAP_SUPPORT_SQOI,
    MMAP_SUPPORT_RAW,
    MMAP_RAW_DITHER,
    MMAP_RAW_BGR_MODE)

set(one_value_args
    MMAP_FILE_SUPPORT_FORMAT,
    MMAP_SPLIT_HEIGHT,
    MMAP_RAW_FILE_FORMAT,
    IMPORT_INC_PATH,
    COPY_PREBUILT_BIN)
```

Option Explanations

General Options

- `FLASH_IN_PROJECT`: Users can opt to have the image automatically flashed together with the app binaries, partition tables, etc. on `idf.py flash`.
- `FLASH_APPEND_APP`: Enables appending binary data (`bin`) to the application binary (`app_bin`).
- `IMPORT_INC_PATH`: Target path for generated include files. Defaults to referencing component location.
- `COPY_PREBUILT_BIN`: Copies pre-generated binary files to target directory. This option allows you to use externally generated asset binaries instead of building them from source files.

- MMAP_FILE_SUPPORT_FORMAT: Specifies supported file formats (e.g., .png, .jpg, .ttf).
- MMAP_SPLIT_HEIGHT: Defines height for image splitting to reduce memory usage. Depends on:
  - MMAP_SUPPORT_SJPG
  - MMAP_SUPPORT_SPNG
  - MMAP_SUPPORT_SQOI

General demo

```
spiffs_create_partition_assets(
   my_spiffs_partition
   my_folder
   FLASH_IN_PROJECT
   MMAP_FILE_SUPPORT_FORMAT ".jpg,.png,.ttf"
)
```

Pre-built binary assets demo

```
spiffs_create_partition_assets(
   my_spiffs_partition
   "${ASSETS_DIR}"
   FLASH_IN_PROJECT
   COPY_PREBUILT_BIN "${ASSETS_DIR}/prebuilt.bin"
)
```

Supported Image Formats

- MMAP_SUPPORT_SJPG: Enables support for SJPG format.
- MMAP_SUPPORT_SPNG: Enables support for SPNG format.
- MMAP_SUPPORT_QOI: Enables support for QOI format.
- MMAP_SUPPORT_SQOI: Enables support for SQOI format. Depends on:
  - MMAP_SUPPORT_QOI

Image Splitting Demo

```
spiffs_create_partition_assets(
   my_spiffs_partition
   my_folder
   FLASH_IN_PROJECT
   MMAP_FILE_SUPPORT_FORMAT ".jpg"
   MMAP_SUPPORT_SJPG
   MMAP_SPLIT_HEIGHT 16
)
```

LVGL Bin Support

- MMAP_SUPPORT_RAW: Converts images to LVGL-supported **Binary** data.
  References:
    - LVGL v8: Use detailed reference
    - LVGL v9: Use detailed reference
- MMAP_RAW_FILE_FORMAT: Specifies file format for RAW images.
  - LVGL v8: {true_color, true_color_alpha, true_color_chroma, indexed_1, indexed_2, indexed_4, indexed_8, alpha_1, alpha_2, alpha_4, alpha_8, raw, raw_alpha, raw_chroma}
  - LVGL v9: Not used.
- MMAP_RAW_COLOR_FORMAT: Specifies color format for RAW images.
  - LVGL v8: {RGB332, RGB565, RGB565SWAP, RGB888}
  - LVGL v9: {L8, I1, I2, I4, I8, A1, A2, A4, A8, ARGB8888, XRGB8888, RGB565, RGB565A8, ARGB8565, RGB888, AUTO, RAW, RAW_ALPHA}

- `MMAP_RAW_DITHER`: Enables **dithering** for RAW images.
  - LVGL v8: Requires dithering.
  - LVGL v9: Not used.
- `MMAP_RAW_BGR_MODE`: Enables **BGR mode** for RAW images.
  - LVGL v8: Not used.
  - LVGL v9: Not used.

LVGL v9 demo

```
spiffs_create_partition_assets(
    .........
    MMAP_FILE_SUPPORT_FORMAT ".png"
    MMAP_SUPPORT_RAW
    MMAP_RAW_COLOR_FORMAT "ARGB8888"
)
```

LVGL v8 demo

```
spiffs_create_partition_assets(
    .........
    MMAP_FILE_SUPPORT_FORMAT ".png"
    MMAP_SUPPORT_RAW
    MMAP_RAW_FILE_FORMAT "true_color_alpha"
    MMAP_RAW_COLOR_FORMAT "RGB565SWAP"
)
```

Application Examples

Generate Header(mmap_generate_my_spiffs_partition.h) This header file is automatically generated and includes essential definitions for memory-mapped assets.

```
#include "mmap_generate_my_spiffs_partition.h"

#define TOTAL_MMAP_FILES      2
#define MMAP_CHECKSUM         0xB043

enum MMAP_FILES {
   MMAP_JPG_JPG = 0,    /*!< jpg.jpg */
   MMAP_PNG_PNG = 1,    /*!< png.png */
};
```

Create Assets Handle The assets config ensures consistency with `mmap_generate_my_spiffs_partition.h`. It sets the `max_files` and `checksum`, verifying the header and memory-mapped binary file.

Partition Mode (Default)

```
mmap_assets_handle_t asset_handle;

const mmap_assets_config_t config = {
   .partition_label = "my_spiffs_partition",
   .max_files = MMAP_MY_FOLDER_FILES, //Get it from the compiled .h
   .checksum = MMAP_MY_FOLDER_CHECKSUM, //Get it from the compiled .h
   .flags = {
      .mmap_enable = false,  // Use partition mode
      .use_fs = false,       // Not using file system
      .app_bin_check = true,
   }
```

(continues on next page)

(continued from previous page)

```
};

ESP_ERROR_CHECK(mmap_assets_new(&config, &asset_handle));
```

Memory Mapping Mode

```
const mmap_assets_config_t config = {
    .partition_label = "my_spiffs_partition",
    .max_files = MMAP_MY_FOLDER_FILES,
    .checksum = MMAP_MY_FOLDER_CHECKSUM,
    .flags = {
        .mmap_enable = true,    // Enable memory mapping
        .use_fs = false,        // Not using file system
        .app_bin_check = true,
    }
};

ESP_ERROR_CHECK(mmap_assets_new(&config, &asset_handle));
```

File System Mode

```
const mmap_assets_config_t config = {
    .partition_label = "/spiffs/assets.bin",  // File path instead of partition name
    .max_files = MMAP_MY_FOLDER_FILES,
    .checksum = MMAP_MY_FOLDER_CHECKSUM,
    .flags = {
        .mmap_enable = false,  // Disable memory mapping
        .use_fs = true,        // Use file system
        .app_bin_check = true,
    }
};

ESP_ERROR_CHECK(mmap_assets_new(&config, &asset_handle));
```

Assets Usage You can use the enum defined in `mmap_generate_my_spiffs_partition.h` to get asset information.

```
const char *name = mmap_assets_get_name(asset_handle, MMAP_JPG_JPG);
const void *mem = mmap_assets_get_mem(asset_handle, MMAP_JPG_JPG);
int size = mmap_assets_get_size(asset_handle, MMAP_JPG_JPG);
int width = mmap_assets_get_width(asset_handle, MMAP_JPG_JPG);
int height = mmap_assets_get_height(asset_handle, MMAP_JPG_JPG);

ESP_LOGI(TAG, "Name:[%s], Mem:[%p], Size:[%d bytes], Width:[%d px], Height:[%d px]
↪", name, mem, size, width, height);
```

API Reference

Header File

- components/display/tools/esp_mmap_assets/include/esp_mmap_assets.h

Functions

esp_err_t **mmap_assets_new**(const *mmap_assets_config_t* *config, *mmap_assets_handle_t* *ret_item)

Create a new asset instance.

Parameters
- **config** –**[in]** Pointer to the asset configuration structure.
- **ret_item** –**[out]** Pointer to the handle of the newly created asset instance.

Returns
- ESP_OK: Success
- ESP_ERR_NO_MEM: Insufficient memory
- ESP_ERR_NOT_FOUND: Can't find partition
- ESP_ERR_INVALID_SIZE: File num mismatch
- ESP_ERR_INVALID_CRC: Checksum mismatch

esp_err_t **mmap_assets_del**(*mmap_assets_handle_t* handle)

Delete an asset instance.

Parameters **handle** –**[in]** Asset instance handle.

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid argument

const uint8_t ***mmap_assets_get_mem**(*mmap_assets_handle_t* handle, int index)

Get the memory of the asset at the specified index.

Parameters
- **handle** –**[in]** Asset instance handle.
- **index** –**[in]** Index of the asset.

Returns Pointer to the asset memory, or NULL if index is invalid.

size_t **mmap_assets_copy_mem**(*mmap_assets_handle_t* handle, size_t offset, void *dest_buffer, size_t size)

Copy a portion of an asset's memory to a destination buffer.

Parameters
- **handle** –**[in]** Asset instance handle.
- **offset** –**[in]** Offset within the asset's memory from which to start copying.
- **dest_buffer** –**[out]** Pointer to the destination buffer where the memory will be copied.
- **size** –**[in]** Number of bytes to copy from the asset's memory to the destination buffer.

Returns The actual number of bytes copied, or 0 if the operation fails.

size_t **mmap_assets_copy_by_index**(*mmap_assets_handle_t* handle, int index, void *dest_buffer, size_t size)

Copy asset data by index to a destination buffer (one-step operation).

This function combines get_mem and copy_mem operations into a single call. It retrieves the asset at the specified index and copies its data to the destination buffer.

Parameters
- **handle** –**[in]** Asset instance handle.
- **index** –**[in]** Index of the asset.
- **dest_buffer** –**[out]** Pointer to the destination buffer where the asset data will be copied.
- **size** –**[in]** Number of bytes to copy from the asset to the destination buffer.

Returns The actual number of bytes copied, or 0 if the operation fails.

const char ***mmap_assets_get_name**(*mmap_assets_handle_t* handle, int index)

Get the name of the asset at the specified index.

Parameters
- **handle** –**[in]** Asset instance handle.
- **index** –**[in]** Index of the asset.

Returns Pointer to the asset name, or NULL if index is invalid.

int **mmap_assets_get_size**(*mmap_assets_handle_t* handle, int index)

Get the size of the asset at the specified index.

Parameters
- **handle** –**[in]** Asset instance handle.

- **index** –**[in]** Index of the asset.

Returns Size of the asset, or -1 if index is invalid.

int **mmap_assets_get_width** (*mmap_assets_handle_t* handle, int index)

Get the width of the asset at the specified index.

Parameters
- **handle** –**[in]** Asset instance handle.
- **index** –**[in]** Index of the asset.

Returns Width of the asset, or -1 if index is invalid.

int **mmap_assets_get_height** (*mmap_assets_handle_t* handle, int index)

Get the height of the asset at the specified index.

Parameters
- **handle** –**[in]** Asset instance handle.
- **index** –**[in]** Index of the asset.

Returns Height of the asset, or -1 if index is invalid.

int **mmap_assets_get_stored_files** (*mmap_assets_handle_t* handle)

Get the number of stored files in the memory-mapped asset.

This function returns the total number of assets stored in the memory-mapped file associated with the given handle. This can be useful for iterating over all assets or validating the file contents.

Parameters **handle** –**[in]** Asset instance handle. This handle is used to identify the memory-mapped file instance containing the assets.

Returns The number of stored assets. Returns the total count of assets if the handle is valid, otherwise returns 0.

Structures

struct **mmap_assets_config_t**

Asset configuration structure, contains the asset table and other configuration information.

Public Members

const char ***partition_label**

Label of the partition containing the assets, or file path when use_fs is enabled

int **max_files**

Maximum number of assets supported

uint32_t **checksum**

Checksum of the asset table for integrity verification

unsigned int **mmap_enable**

Flag to indicate if memory-mapped I/O is enabled

unsigned int **use_fs**

Flag to use file system (partition_label as file path) instead of partition

unsigned int **app_bin_check**

Flag to enable app header and bin file consistency check

unsigned int **full_check**

Flag to enable self-consistency check

unsigned int **metadata_check**

Flag to enable metadata verification

unsigned int **reserved**

Reserved for future use

struct *mmap_assets_config_t*::[anonymous] **flags**

Configuration flags

Type Definitions

typedef struct mmap_assets_t ***mmap_assets_handle_t**

Asset handle type, points to the asset.

Type of asset handle

Chapter 5

USB Host & Device

5.1 USB Host & Device

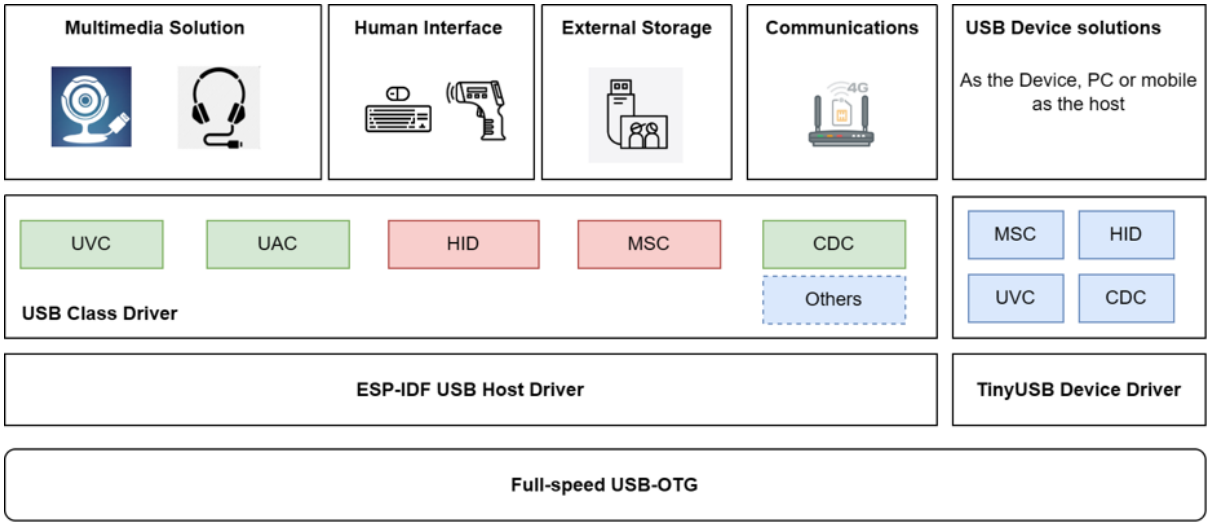
5.1.1 ESP USB Peripheral Introduction

USB Introduction

USB (Universal Serial Bus) is a universal bus standard used to connect hosts and peripheral devices. A USB host can establish a connection with USB devices through a USB interface, enabling functions such as data transfer and power supply.

USB-IF (USB Implementers Forum) is the organization responsible for establishing USB standards. It defines USB standards, including USB 1.1, USB 2.0, USB 3.0, and others. USB-IF specifies protocols for the physical layer, data link layer, transport layer, session layer, presentation layer, and more for the USB interface. It also defines USB Device Class standards, with common device classes such as HID (Human Interface Device), MSC (Mass Storage Class), CDC (Communication Device Class), Audio, Video, and more.

Espressif' s ESP32-S2/S3/C3/P4 chips come with built-in USB-OTG or USB-Serial-JTAG peripherals, supporting a variety of USB applications. These include USB multimedia applications, USB communication applications, USB storage applications, USB human interface applications, and more.



USB Electrical Properties The electrical properties of the Type-A USB interface are as follows:

Pin	Name	Cable color	Description
1	VBUS	**Red**	+5V
2	D-	**White**	Data- （0 or 3.3V）
3	D+	**Green**	Data+ （0 or 3.3V）
4	GND	**Black**	Ground

- For *self-powered devices, an additional IO is required to check the VBUS voltage*, to detect whether the device is unplugged.
- Reversing the D- D+ connection will not damage the hardware, but the host will be unable to recognize it.

USB-OTG Full-Speed Controller Introduction **USB OTG Full-Speed Controller**: refers to a controller that simultaneously supports USB-OTG, USB Host, and USB Device modes, with the capability for mode negotiation and switching. It supports two speeds: Full-speed (12Mbps) and Low-speed (1.5Mbps), and is compatible with both USB 1.1 and USB 2.0 protocols.

Starting from ESP-IDF version 4.4, it includes USB Host and USB Device protocol stacks, as well as various device class drivers, supporting user secondary development.

For more information, please refer to the *USB-OTG Controller Introduction*.

USB-OTG High-speed Controller Introduction **USB OTG High-speed Controller**is a controller that simultaneously supports USB-OTG, USB Host, and USB Device modes, with the capability for mode negotiation and switching. It supports three speeds: High-speed (480Mbps), Full-speed (12Mbps), and Low-speed (1.5Mbps), and is compatible with USB 2.0 protocol.

Starting from ESP-IDF version 5.5, the originally independent USB PHY implementation of ESP32-P4 has been merged into the shared PHY module with S2/S3, unified driving structure, simplified maintenance.

For more information, please refer to the *USB-OTG Controller Introduction*.

Introduction to USB-Serial-JTAG Controller **USB-Serial-JTAG Controller**: A dedicated USB controller with both USB Serial and USB JTAG capabilities. It supports firmware download, log printing, CDC transmission, and JTAG debugging through the USB interface. Secondary development such as modifying USB functions or descriptors is not supported.

For more information, please refer to the *USB-Serial-JTAG Controller Introduction*.

USB Full-Speed PHY Introduction **USB Full-Speed PHY**: Also known as USB Full-Speed Transceiver, it is used for converting USB controller digital signals to USB bus signal levels and providing bus driving capability. The internal USB Full-speed PHY is connected to external fixed IO pins.

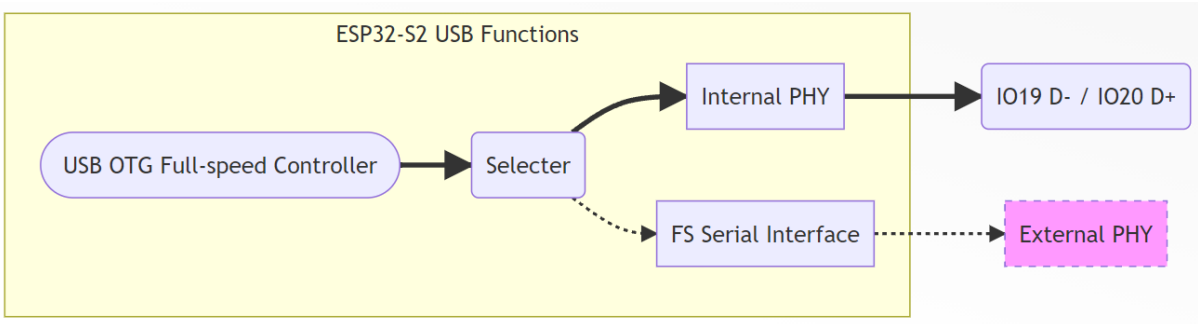
For more information, please refer to the *USB-PHY Introduction*.

ESP32-S/P/C Series USB Peripheral Support

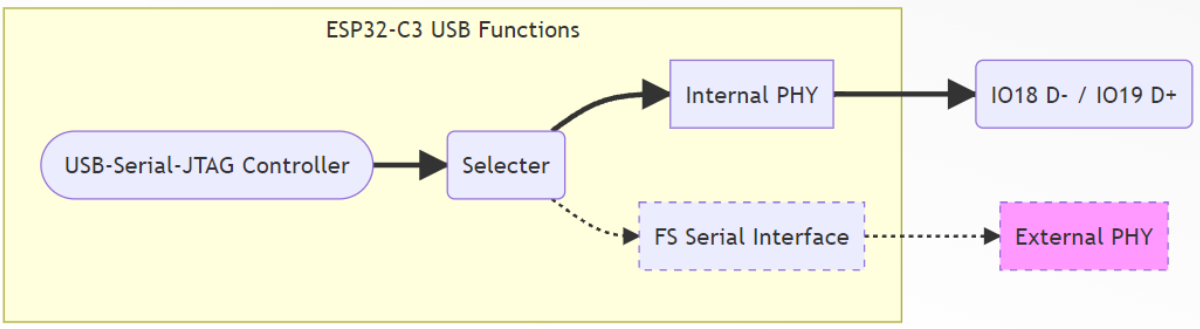
	USB OTG High-Speed	USB OTG Full-Speed	USB-Serial-JTAG	Full-Speed PHY	High-Speed PHY
ESP32-P4	√	√	√	√	√
ESP32-S3	X	√	√	√	X
ESP32-S2	X	√	X	√	X
ESP32-C6	X	X	√	√	X
ESP32-C3	X	X	√	√	X
ESP32-C2	X	X	X	X	X
ESP32	X	X	X	X	X
ESP8266	X	X	X	X	X

- √ : Supported
- X : Not Supported

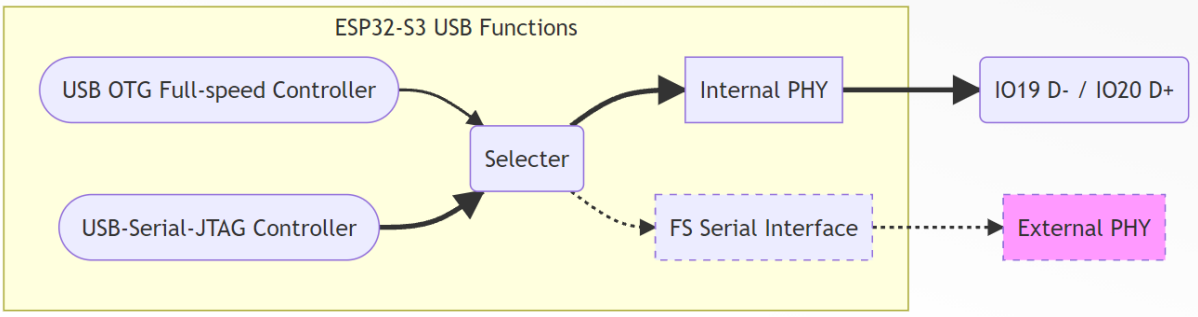
ESP32-S2 USB Function Overview The ESP32-S2 features an integrated **USB OTG Full-Speed Controller** and **USB Full-Speed PHY**. The internal architecture is as follows:



ESP32-C3 USB Function Overview The ESP32-C3 comes equipped with a built-in **USB-Serial-JTAG Controller** and **USB Full-Speed PHY**. The internal architecture is outlined below:

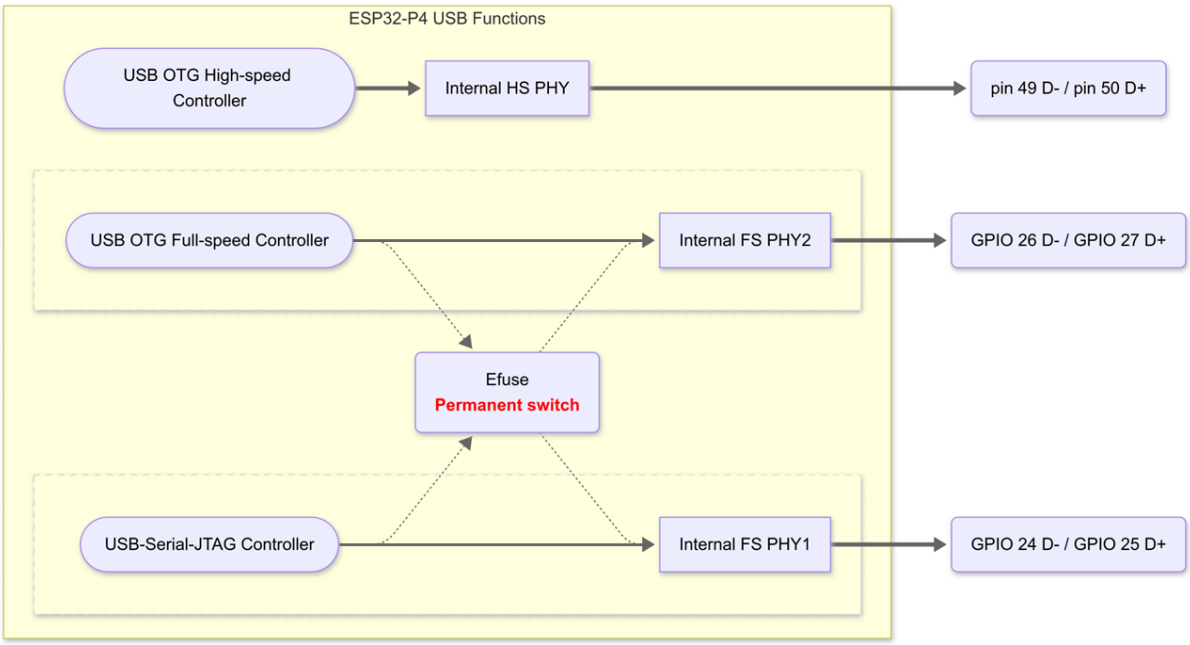


ESP32-S3 USB Function Overview The ESP32-S3 is equipped with two built-in USB controllers. **USB OTG Full-Speed Controller** and **USB-Serial-JTAG Controller**, Additionally, there is an integrated USB Full-speed PHY. The internal USB PHY is initially connected to the **USB-Serial-JTAG** controller by default. It can be modified through eFuse burning to change the default configuration or dynamically switched through register configuration. It is also possible to enable both controllers simultaneously by adding an external PHY. For detailed information on switching the internal USB PHY, refer to *USB PHY Switching*.



ESP32-P4 USB Function Overview ESP32-P4 is equipped with three built-in USB controllers. **USB OTG High-speed Controller** , **USB OTG Full-speed Controller** and **USB-Serial-JTAG Controller**, Additionally, there is an integrated USB High-speed PHY and two USB Full-speed PHYs. The default connection is FS_PHY1 connected to the USB Serial/JTAG controller, and FS_PHY2 connected to OTG_FS. The user can change the connection relationship through EFUSE_USB_PHY_SEL.

- 0: FS_PHY1 connected to the USB Serial/JTAG controller, and FS_PHY2 connected to OTG_FS.
- 1: FS_PHY2 connected to the USB Serial/JTAG controller, and FS_PHY1 connected to OTG_FS.



5.1.2 USB-OTG Peripheral Introduction

The ESP32-S2/S3/P4 and other chips come with built-in USB On-The-Go (USB-OTG) peripherals. They include a USB controller and USB PHY, supporting connection to a PC via a USB cable, enabling USB Host and USB Device functionalities.

USB-OTG Transfer Rate

For the ESP32-S2/S3/P4, the USB On-The-Go (USB-OTG) Full Speed bus transfer rate is 12 Mbps. However, due to the presence of checksum and synchronization mechanisms in USB transfers, the actual effective transfer rate will be lower than 12 Mbps. The specific values depend on the transfer type and are outlined in the table below:

Transfer Types	Control	Interrupt	Bulk	Isochronous
Application Scenarios	Device Initialization and Management	Mouse and Keyboard	Printers and Bulk Storage	Streaming Audio and Video
Supports Low Speed	Yes	Yes	No	No
Checksum and Retransmission	Yes	Yes	Yes	No
Ensuring Transfer Speed	No	No	No	Yes
Utilizing Fixed Bandwidth	Yes (10%)	Yes (90%)	No	Yes (90%)
Reducing Latency Time	No	Yes	No	Yes
Maximum Transfer Size	64 Bytes	64 Bytes	64 Bytes	~512 Bytes
Number of Packets Transferred Per Millisecond	•	1	19	1
Theoretical Effective Rate	•	64000 Bytes/s	1216000 Bytes/s	512000 Bytes/s

For ESP32-P4 USB-OTG High Speed bus transfer rate is 480 Mbps, compared to Full Speed, there are some differences in the maximum transfer size, number of packets transferred per millisecond, and theoretical effective rate, as shown in the table below:

Transfer Types	Control	Interrupt	Bulk	Isochronous
Maximum Transfer Size	64 Bytes	1024 Bytes	512 Bytes	1024 Bytes
Number of Packets per Microframe	•	6	13	7
Theoretical Effective Rate	•	49152000 Bytes/s	53248000 Bytes/s	57344000 Bytes/s

- Calculation Formula for Transfer Rate: Transfer Rate (Bytes/s) = Maximum Packet Size * Number of Packets per Microframe * 8000
- Control transfers are used for transmitting device control information and involve multiple stages. The effective transfer rate needs to be calculated based on the implementation of the protocol stack.

USB-OTG Peripheral Built-in Features

Using USB OTG Console for Firmware Download and Logging For chips like ESP32-S2/S3 with built-in USB On-The-Go (USB-OTG) peripherals, the ROM Code contains the functionality of USB Communication Device Class (CDC). This feature can be utilized as an alternative to UART interfaces, enabling functions such as logging, console access, and firmware downloads.

1. Since the USB OTG Console is initially disabled, follow these steps to perform the first firmware download:
    1. Enable the USB OTG Console feature in `menuconfig` and then compile the firmware.

   2. Manually pull down the `Boot` pin of the chip and connect the chip to the PC via USB to enter download mode. The PC will detect a new serial port device, listed as `COM*` on Windows, `/dev/ttyACM*` on Linux, and `/dev/cu*` on MacOS.
   3. Use the `esptool` utility (or directly use `idf.py flash`) to configure the corresponding serial port and download the firmware.
2. After the initial download, the USB OTG Console functionality will be automatically enabled. You can then connect to the PC via USB, and the PC will detect a new serial port device, listed as `COM*` on Windows, `/dev/ttyACM*` on Linux, and `/dev/cu*` on MacOS. Log data will be printed from this virtual serial port.
3. Users no longer need to manually pull down the Boot control pin. To download firmware, use the esptool utility (or directly use idf.py flash) to configure the corresponding serial port. During the download, esptool will automatically reset the device and switch it to download mode using the USB control protocol.

   For more detailed information, please refer to USB OTG Console

Download firmware using USB OTG DFU For chips like ESP32-S2/S3/P4 with built-in USB On-The-Go (USB-OTG) peripherals, the ROM Code contains the functionality of USB DFU (Device Firmware Upgrade). This feature enables the implementation of a standard DFU download mode.

1. To download firmware using DFU, users need to manually enter download mode each time by pulling down the Boot control pin of the chip and connecting it to the PC via USB.
2. Run the command `idf.py dfu` in the project directory to generate the DFU firmware, and then use `idf.py dfu-flash` to download the firmware.
3. If there are multiple DFU devices, users can use `idf.py dfu-list` to view the DFU device list and then use `idf.py dfu-flash --path <path>` to specify the download port.

   For more detailed information, please refer to Device Firmware Upgrade via USB.

USB Host development using USB-OTG peripherals

USB-OTG peripherals support USB Host functionality, allowing users to connect directly to external USB devices through the USB interface. Starting from ESP-IDF version 4.4, the USB Host Driver is supported. Users can refer to the ESP-IDF USB Host documentation to develop USB Class Drivers.

Additionally, Espressif officially supports USB Host drivers for HID, MSC, CDC, UVC, and other device classes. Users can utilize these drivers directly for application development.

For more details on the USB Host solution, please refer to *USB Host Solution*.

USB Device development using USB-OTG peripherals

USB-OTG peripherals support USB Device functionality, and Espressif has officially adapted the `TinyUSB` stack. Users can develop USB standard devices or custom devices based on the open-source `TinyUSB`, such as HID, MSC, CDC, ECM, UAC, and more.

For more details on the USB Device solution, please refer to *USB Device Solution*.

Dynamic Switching of USB-OTG Peripheral Modes

The USB-OTG peripheral supports dynamic switching of its mode. Users can achieve dynamic switching by dynamically registering the USB Host Driver or USB Device Driver.

- Example Code: USB OTG Manual Switching usb/otg/usb_host_device_mode_manual_switch

Using USB-OTG to Proactively Disconnect from the Host

You can proactively disconnect from the host by pulling the USB OTG `VBUSVALID` signal low, simulating the disconnection of the USB OTG peripheral. Note that if you need to reattach the device, you must wait at least **10ms** before doing so. Please refer to *USB Device Self Power*.

5.1.3 USB-Serial-JTAG Peripheral Introduction

ESP32 series chips with integrated USB-Serial-JTAG peripheral (such as ESP32-S3, ESP32-C3, ESP32-P4) feature both USB-to-serial and USB-to-JTAG functionality. This peripheral provides direct USB connectivity to a host PC, supporting firmware download, JTAG-based debugging, and system console output. For detailed information on the USB-Serial-JTAG peripheral architecture, please refer to the ESP32-C3 Technical Reference Manual - USB Serial/JTAG Controller.

USB-Serial-JTAG peripheral driver

- For Linux and MacOS systems, No need to manually install drivers.
- For Windows 10 and above, drivers will be automatically installed when connected to the internet.
- For Windows 7/8 systems, manual driver installation is necessary. The driver can be downloaded from: esp32-usb-jtag-2021-07-15. Alternatively, users can use the ESP-IDF Windows Installer, selecting the USB-Serial-JTAG driver during installation.

Note: If the above driver installation fails, you can use the Zadig tool to install the driver. When using Zadig, please select drivers as follows:

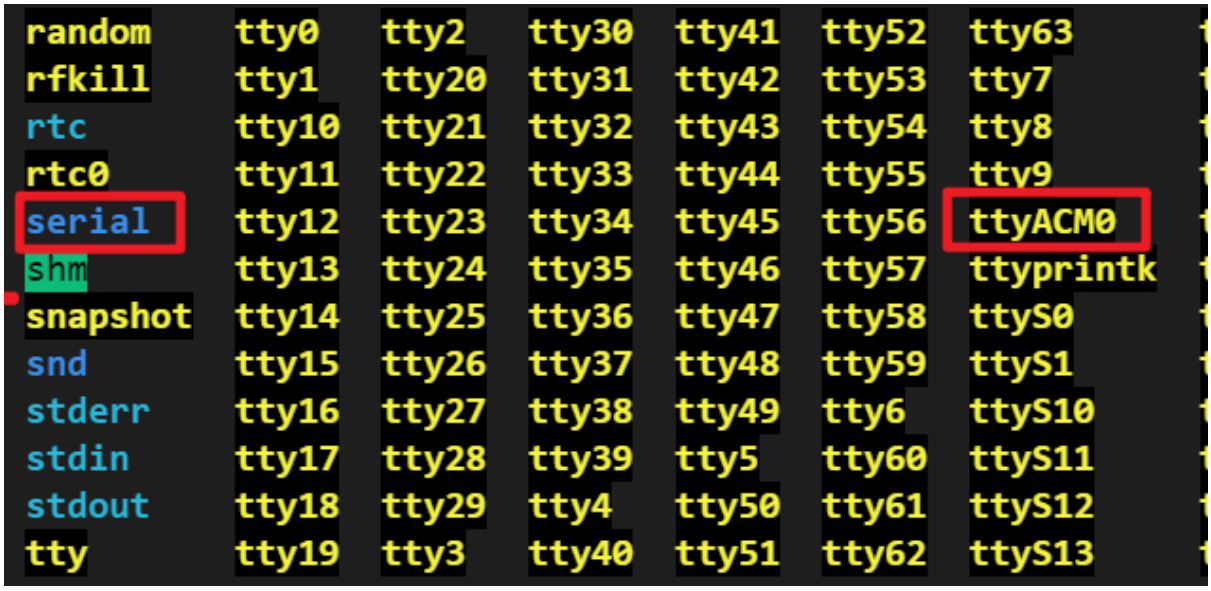
- For `USB JTAG/serial debug unit (Interface 0)`, select `usbser` or `USB Serial (CDC)` driver
- For `USB JTAG/serial debug unit (Interface 2)`, select `WinUSB` driver

USB-Serial-JTAG peripheral built-in functionality

Upon connecting the USB-Serial-JTAG peripheral to a PC, the Device Manager will show the addition of two devices:

For Windows as shown in the following figure:

For Linux as shown in the following figure:



Note: If the devices are not displayed correctly, please refer to the driver installation section above.

Use USB-Serial-JTAG to download firmware

- By default, the USB-Serial-JTAG download function is enabled. You can directly connect it to the PC using a USB cable and then use the esptool tool (or directly use idf.py flash) to configure the serial port corresponding to the USB-Serial-JTAG device (COM* for Windows, /dev/ttyACM* for Linux, /dev/cu* for MacOS) for firmware download. During the download process, esptool automatically resets the device and switches it to download mode through the USB control protocol.
- If the USB pins corresponding to USB-Serial-JTAG are used for other functions in the application, such as being used as a regular GPIO or other peripheral IO, USB-Serial-JTAG will be unable to establish a connection with the USB host. Therefore, it cannot switch the device to download mode via USB. In such cases, users must manually switch the device to download mode using the Boot control pin and then use esptool for firmware download.
- To avoid using the USB pins corresponding to USB-Serial-JTAG for other functions in the application, which would prevent automatic entry into download mode via USB, users need to expose the Boot control pin in hardware design.
- By default, when downloading different chips through the USB interface, the COM number will increment, which may cause inconvenience for mass production. Users can refer to *Prevent Windows from incrementing COM numbers based on USB device serial number* for a solution.

Debugging code using USB-Serial-JTAG

The `USB-Serial-JTAG` supports debugging code through the `JTAG` interface. Users only need to connect it to the PC using a USB cable and then use the `OpenOCD` tool for code debugging. Please refer to the configuration guide for setting up the built-in JTAG interface on ESP32-C3 at: Configure Built-in JTAG Interface on ESP32-C3.

Print system LOG using USB-Serial-JTAG

- Users can enable the USB-Serial-JTAG LOG feature by configuring the state in `menuconfig-> Component config → ESP System Settings → Channel for console secondary output`.
- Once the LOG feature is enabled, you can connect the device directly to the PC using a USB cable and then use `idf.py monitor` or other serial port tools to open the serial port corresponding to the USB-Serial-JTAG device (`COM*` for Windows, `/dev/ttyACM*` for Linux, `/dev/cu*` for MacOS) to print system logs.

- The `USB-Serial-JTAG` will only print logs after the host is connected. If the host is not connected, `USB-Serial-JTAG` will not be initialized, and logs will not be printed.
- The LOG feature of `USB-Serial-JTAG` cannot be used in sleep modes (including deep sleep and light sleep modes). If it is necessary to print logs in sleep mode, the `UART` interface can be used.

Using USB-Serial-JTAG pins as normal GPIO

If users need to use the USB pins corresponding to USB-Serial-JTAG for other functions in the application, such as using them as regular GPIO, it is important to note that the USB D+ interface has a default pull-up resistor, which keeps the USB D+ pin at a high level. Therefore, it is necessary to disable this pull-up resistor when using it as a GPIO.

- Starting from ESP-IDF v4.4, the GPIO driver defaults to disabling the USB D+ pull-up resistor. Users do not need additional configuration when using the GPIO driver.
- Users can also modify the register value `USB_SERIAL_JTAG.conf0.dp_pullup = 0;` to disable the USB D+ pull-up resistor.

It is important to note that the pull-up resistor on the USB D+ pin is present at power-up. Before software disables the pull-up resistor, the USB D+ pin has already been pulled high, causing it to be in a high-level state during the initial phase when used as a GPIO. If users need the USB D+ pin to be immediately low after power-up, it is necessary to design the hardware to pull down the USB D+ pin through an external circuit.

If users wish to again use the USB-Serial-JTAG function, please refer to the following code:

```
#include "soc/soc_caps.h"
#include "soc/usb_serial_jtag_reg.h"
#include "hal/usb_serial_jtag_ll.h"

    SET_PERI_REG_MASK(USB_SERIAL_JTAG_CONF0_REG, USB_SERIAL_JTAG_PAD_PULL_
↪OVERRIDE);
    CLEAR_PERI_REG_MASK(USB_SERIAL_JTAG_CONF0_REG, USB_SERIAL_JTAG_DP_PULLUP);
    SET_PERI_REG_MASK(USB_SERIAL_JTAG_CONF0_REG, USB_SERIAL_JTAG_DP_PULLDOWN);
    vTaskDelay(pdMS_TO_TICKS(10));
#if USB_SERIAL_JTAG_LL_EXT_PHY_SUPPORTED
    usb_serial_jtag_ll_phy_enable_external(false);  // Use internal PHY
    usb_serial_jtag_ll_phy_enable_pad(true);        // Enable USB PHY pads
#else // USB_SERIAL_JTAG_LL_EXT_PHY_SUPPORTED
    usb_serial_jtag_ll_phy_set_defaults();          // External PHY not supported.␣
↪Set default values.
#endif // USB_WRAP_LL_EXT_PHY_SUPPORTED
    CLEAR_PERI_REG_MASK(USB_SERIAL_JTAG_CONF0_REG, USB_SERIAL_JTAG_DP_PULLDOWN);
    SET_PERI_REG_MASK(USB_SERIAL_JTAG_CONF0_REG, USB_SERIAL_JTAG_DP_PULLUP);
    CLEAR_PERI_REG_MASK(USB_SERIAL_JTAG_CONF0_REG, USB_SERIAL_JTAG_PAD_PULL_
↪OVERRIDE);
```

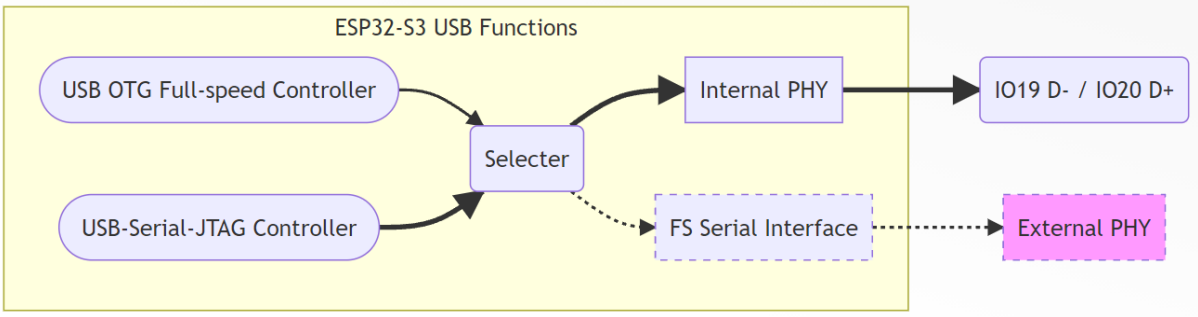
5.1.4 USB PHY/Transceiver Introduction

The function of the USB PHY/Transceiver is to convert the digital signals from the USB controller into USB bus signal levels, providing bus driving capability, and detecting receive errors, among other functions. Chips like ESP32-S2/S3/P4 have a built-in USB Full-speed PHY, allowing users to directly use the USB D+ D- pins specified by the chip for communication with an external USB system. Additionally, ESP32-P4 has a built-in USB High-Speed PHY. ESP32-S2/S3 retains an external PHY extension interface, allowing users to connect an external PHY when needed.

Use the internal PHY

The ESP32-S2/S3/C3 internally integrates a USB PHY, eliminating the need for an external PHY chip. It can directly connect to external USB hosts or devices through the USB D+/D- pins. However, for chips with two integrated USB

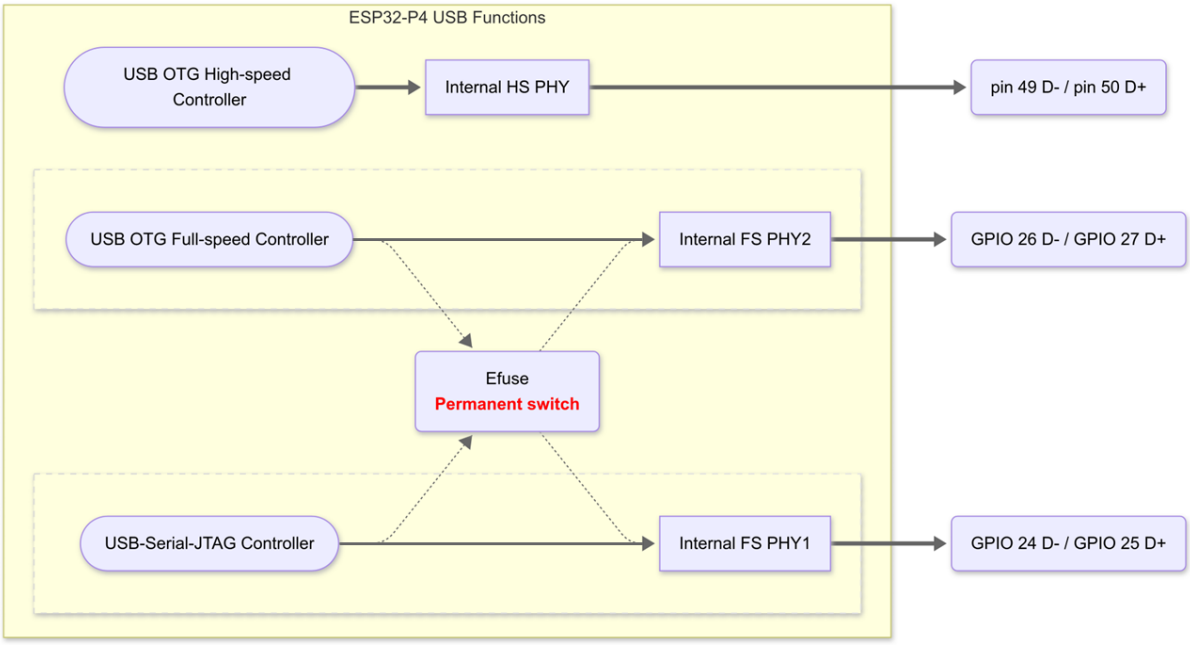
controllers, such as the ESP32-S3 with built-in USB-OTG and USB-Serial-JTAG, both controllers share a single internal PHY, allowing only one to operate at a time.



The internal USB-PHY corresponds to fixed GPIO pins, as shown in the table below:

	D+	D-
ESP32-S2	20	19
ESP32-S3	20	19
ESP32-C3	19	18
ESP32-C6	13	12

For ESP32-P4, it internally integrates two USB Full-speed PHYs and one USB High-speed PHY.



The internal USB-PHY corresponds to fixed GPIO pins, as shown in the table below:

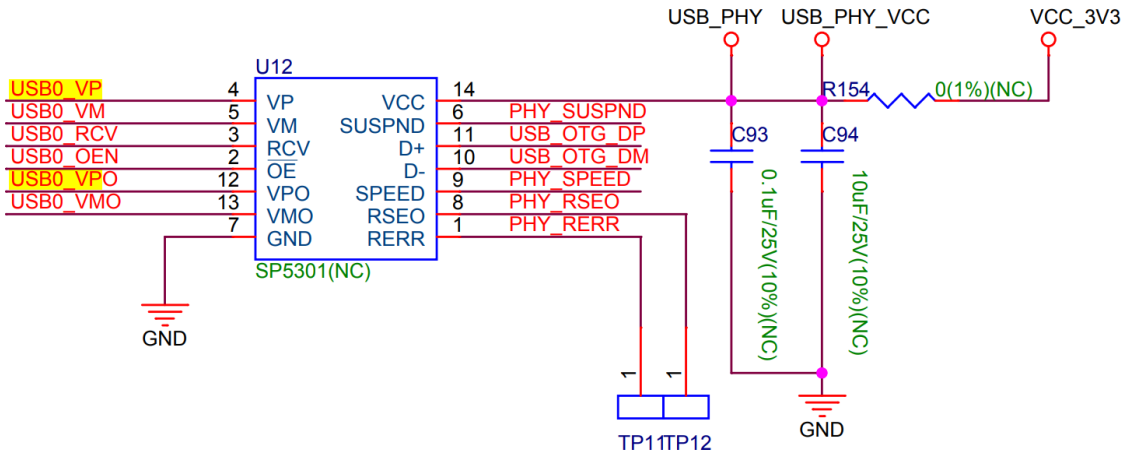
	D+	D-
ESP32-P4 FS_PHY1	25	24
ESP32-P4 FS_PHY2	27	26
ESP32-P4 HS_PHY	pin 50	pin 49

The default connection is FS_PHY1 connected to the USB Serial/JTAG controller, and FS_PHY2 connected to OTG_FS. The user can change the connection relationship through EFUSE_USB_PHY_SEL.

Use an external PHY (ESP32-S2/S3)

By adding an external PHY, it is possible to enable the simultaneous operation of both USB-OTG and USB-Serial-JTAG.

ESP32S2/S3 supports SP5301 or equivalent USB PHY. A typical circuit diagram for an external PHY is as follows:



Using an external USB PHY will occupy a minimum of 6 GPIO pins.

USB PHY default configuration

1. For chips that feature both USB-OTG and USB-Serial-JTAG peripherals simultaneously, the default configuration is for USB-Serial-JTAG to be connected to the internal USB-PHY. Users can directly download or debug through the USB interface without additional configuration.
2. If there is a need to develop a USB-OTG application using USB Host Driver or the TinyUSB protocol stack, during protocol stack initialization, the USB-PHY connection will automatically switch to USB-OTG, and users do not need to perform additional configuration. In USB-OTG mode, if users wish to utilize the download functionality of USB-Serial-JTAG, they need to manually boot into download mode.

Modify the default configuration of the USB PHY

Method 1: Switch the USB-PHY connection to USB-OTG by configuring the registers.

- The USB Host Driver or TinyUSB stack internally switches the connection of the internal USB-PHY to USB-OTG by configuring USB PHY registers. For more information, please refer to the USB PHY Configuration API.

Method 2: Switch the default connection of USB-PHY to USB-OTG by burning the efuse `usb_phy_sel` bit to 1.

- This efuse bit should only be burned if the user needs to use USB-OTG functionality in Boot mode. Once burned, when the chip enters Boot mode, it will utilize the download functionality provided by USB-OTG, such as USB DFU.
- Note that once the efuse bit is burned to 1, it cannot be restored to 0. When the USB-PHY default connection is switched to USB-OTG, and the chip enters Boot mode, USB-OTG functionality will be active, and USB-Serial-JTAG functionality will be unavailable.
- Note: For ESP32-S3 modules and development boards produced before DateCode 2219 (PW No. earlier than PW-2022-06-XXXX), because EFUSE_DIS_USB_OTG_DOWNLOAD_MODE (BLK0 B19[7]) has already been burned to 1 (USB OTG download disabled), if users burn the efuse_usb_phy_sel bit to 1, it will result in both USB-Serial-JTAG and USB-OTG download functionalities being disabled when the chip enters Boot mode.

5.1.5 USB VID and PID

VID and PID are unique identifiers for USB devices, used to distinguish between different USB devices. Generally, VID and PID are assigned by USB-IF, which is the USB Implementers Forum.

In the following scenarios, you can exempt from applying for VID and PID

- If your product operates in USB Host mode, you do not need to apply for VID and PID.
- If your product operates in USB Device mode, and you plan to use Espressif's VID (0x303A) and develop a USB standard device based on the TinyUSB protocol stack, you do not need to apply for a PID. You can use the default PID provided by TinyUSB.

To apply for a VID (Vendor ID) and PID (Product ID)

If your product requires the use of USB Device mode, you can apply for VID (Vendor ID) or PID (Product ID) following the process outlined below.

- If your product requires the use of a VID assigned by USB-IF, you need to first register as a USB-IF member and then follow the USB-IF process to apply for VID and PID.
- If your product is considering the use of Espressif's VID, you can directly apply for a PID (free of charge). For the application process, please refer to Applying for Espressif PID.

Note: Utilizing Espressif's VID and TinyUSB's default PID does not imply compliance with USB specifications. You still need to undergo USB certification for your product.

USB Certification

USB Certification is managed by the USB Implementers Forum (USB-IF) and is designed to ensure that products comply with USB specifications, ensuring interoperability and compatibility between devices.

USB USB certification is an optional process, but it is mandatory in the following scenarios:

- If a product claims to comply with USB specifications and uses the official USB logo.
- If a product intends to use the USB logo, trademark, or references USB certification in promotional materials.

For specific certification processes and requirements, please refer to https://www.usb.org or contact the USB-IF Authorized Testing Laboratory.

In addition, for detailed steps and firmware related to USB signal quality testing, please refer to *USB Signal Quality Test*

5.1.6 USB Host Solution

ESP32-S2/S3/P4 chips have built-in USB-OTG peripherals, which support USB host mode. Based on the USB host stack and various USB host class drivers provided by ESP-IDF, they can connect to a variety of USB devices through the USB interface. The following describes the USB Host solutions supported by ESP32-S2/S3/P4 chips.

ESP USB Camera Video Solution

Supports the connection of a camera module through the USB interface, enabling the acquisition and transmission of MJPEG format video streams, with a maximum resolution of 480*800@15fps（Full-Speed，ESP32-S2/S3）and 1080*1920@60fps（High-Speed，ESP32-P4）. Ideal for applications such as cat's eye doorbells, smart door locks, endoscopes, rearview cameras, and other scenarios.

Features:

- Quick Start
- Hot Plug Support
- Cameras that support UVC1.1/1.5 Specifications
- Automatic Descriptor Parsing
- Dynamic Resolution Configuration
- MJPEG Video Stream Transmission
- Bulk or Isochronous Transfer Modes

Hardware:

- Chips: ESP32-S2, ESP32-S3
- Peripherals: USB-OTG
- USB Camera: Supports MJPEG format, with a bulk transfer mode of 800*480@15fps, or an isochronous transfer mode of 480*320@15fps. For camera limitations, refer to the usb_stream API documentation.

For ESP32-P4, please use the *usb_host_uvc* component, which supports MJPEG, YUY2, H264, and H265 formats.

links:

- usb_stream component
- usb_host_uvc component
- usb_stream API reference
- USB Camera Demo video
- Example Code: USB Camera + WiFi Image Transmission: usb/host/usb_camera_mic_spk
- Example Code: USB Camera + Local Screen Display with LCD: usb/host/usb_camera_lcd_display

ESP USB Audio Solution

Supports connecting USB audio devices through the USB interface, enabling PCM format audio stream acquisition and transmission. It can simultaneously support multiple channels of 48KHz 16bit speakers and multiple channels of 48KHz 16bit microphones. Also supports Type-C interface headphones, suitable for audio playback scenarios. It can operate simultaneously with UVC, making it suitable for scenarios such as doorbell intercoms.

Features:

- Quick Start
- Hot Swap
- Automatic Parsing Descriptors
- PCM Audio Format
- Dynamic Modification of Sampling Rate
- Multi-Channel Speakers
- Multi-Channel Microphone
- Support Volume and Mute Control
- Support Simultaneous Work with USB Camera

Hardware:

- Chips: ESP32-S2, ESP32-S3, ESP32-P4
- Peripherals: USB-OTG
- USB Audio Devices: Supports PCM format

Links:

- usb_stream component
- usb_host_uac component

- usb_stream API reference
- USB Audio Demo video
- Example Code: MP3 Music Player + USB Headphones: usb/host/usb_audio_player

ESP USB 4G Networking Solutions

Supports connecting 4G Cat.1 or Cat.4 modules via the USB interface, enabling PPP dial-up for internet access. It also supports sharing the internet via Wi-Fi SoftAP hotspot for other devices. Suitable for IoT gateways, MiFi mobile hotspots, smart energy storage, advertising lightboxes, and other scenarios.

Features:

- Quick Start
- Hot Plug
- Modem+AT Dual Interface
- PPP Standard Protocol
- 4G to Wi-Fi Hotspot Support
- NAPT (Network Address and Port Translation) Support
- Power Management Support
- Automatic Network Recovery
- SIM Card Detection and Signal Quality Monitoring
- Web-based Configuration Interface

Hardware:

- Chips: ESP32-S2, ESP32-S3, ESP32-P4
- Peripherals: USB-OTG
- 4G Modules: Supports Cat.1, Cat.4, and other network standard 4G modules, requiring module support for the PPP/ECM/RNDIS protocol.

Links:

- USB 4G Demo video
- iot_usbh_modem component
- iot_usbh_ecm 组件
- iot_usbh_rndis 组件
- Example code: 4G PPP dial-up usb/host/usb_cdc_4g_module
- Example code: 4G RNDIS usb/host/usb_rndis_4g_module
- Example code: 4G ECM usb/host/usb_ecm_4g_module

ESP USB Storage Solution

Supports connecting standard USB flash drives via the USB interface (compatible with USB 3.1/3.0/2.0 protocols), and can mount the USB flash drive to the FatFS file system for file read and write operations. Suitable for outdoor advertising billboards, attendance machines, mobile speakers, recorders, and other application scenarios.

Features:

- Compatible with USB 3.1/3.0/2.0 Flash Drives
- Default Support for Up to 32GB
- Hot Plug
- Support for Fat32/exFAT Formats
- File System Read and Write
- USB Flash Drive Over-The-Air (OTA) Update

Hardware:

- Chips: ESP32-S2, ESP32-S3, ESP32-P4
- Peripherals: USB-OTG
- USB Flash Drive: Formatted as Fat32 by default, with support for USB drives up to 32GB. Drives larger than 32GB require exFAT file system support.

Links:

- usb_host_msc component
- USB Flash Drive OTA component
- Mount USB Flash Drive + File System Access Example

ESP USB Hub Solution

Supports connecting multiple USB devices through USB Hub, enabling simultaneous work of multiple devices. Suitable for multiple USB devices working together, such as dual camera video acquisition, audio and video synchronization processing, peripheral expansion and data storage.

Features:

- Supports connecting multiple USB devices through USB Hub
- Supports hot plug

Hardware:

- Chips: ESP32-S2, ESP32-S3, ESP32-P4
- Peripherals: USB-OTG

Links:

- USB Hub Dual Camera Demo

5.1.7 USB Device Solution

ESP32-S2/S3/P4/etc. has a USB-OTG peripheral that supports USB device mode, which can be connected to a PC or other USB host devices via USB. By combining the device TinyUSB protocol stack and device class drivers, it can be used to develop various USB devices, such as HID devices, CDC devices, and composite devices. The following describes the related solutions based on USB devices.

USB Audio Device Solution

The USB audio device solution is based on the UAC 2.0 (USB Audio Class) protocol standard, which allows Espressif SoCs to function as audio devices, providing convenient and high-quality audio transmission capabilities. For example, it can be used as a microphone or speaker connected to a computer or other USB audio-enabled devices for audio input and output.

Features:

- Supports UAC 2.0
- Supports various audio formats and sampling rates
- Supports audio input and output

Hardware:

- Chip: ESP32-S2, ESP32-S3, ESP32-P4
- Peripheral: USB-OTG

Links:

- ESP-BOX USB Speaker with Display

USB Video Device Solution

The USB Video device solution is based on the UVC (USB Video Class) protocol standard, which allows Espressif SoCs to function as video devices, providing convenient and high-quality video transmission capabilities. It can be applied to USB doorbell cameras or USB + Wi-Fi dual-mode network cameras.

Features:

- Supports UVC 1.5
- Supports synchronous and bulk transfer modes
- Supports functioning as a virtual camera device

Hardware:

- Chip: ESP32-S2, ESP32-S3, ESP32-P4
- Peripheral: USB-OTG

Links:

- USB Network Camera

USB Mass Storage Device Solution

The USB mass storage device solution is based on the MSC (Mass Storage Class) protocol standard. It can also be combined with Wi-Fi functionality to build wireless shared storage devices such as USB wireless flash drives, card readers, digital music players, and digital media players.

Features:

- USB-Wi-Fi bidirectional access
- Multiple device connections
- Emulates a USB flash drive

Hardware:

- Chip: ESP32-S2, ESP32-S3, ESP32-P4
- Peripheral: USB-OTG

Links:

- USB + Wi-Fi Wireless Flash Drive

USB HID Device Solution

The USB HID device solution is based on the HID (Human Interface Device) protocol standard. It can function as a USB keyboard, mouse, gamepad, and other devices to enable human-computer interaction. When combined with wireless features such as Wi-Fi, Bluetooth, and ESP-Now, it can also be used to build wireless HID devices.

Features:

- Supports various HID devices
- Supports custom HID devices
- Supports both USB HID and BLE HID modes

Hardware:

- Chip: ESP32-S2, ESP32-S3, ESP32-P4
- Peripheral: USB-OTG

Links:

- USB HID Keyboard and Mouse Example
- USB HID Surface Dial Example
- USB Custom Keyboard Example

USB Drag-and-Drop OTA Upgrade

Based on esp-tinyuf2 as a virtual USB flash drive, this solution supports drag-and-drop UF2 firmware onto the flash drive for OTA updates. It also supports mapping NVS data to files on the flash drive, allowing modification of NVS by modifying the files.

Features:

- OTA updates by dragging and dropping UF2 firmware
- Modifying NVS through virtual file manipulation

Hardware:

- Chip: ESP32-S2, ESP32-S3
- Peripheral: USB-OTG

Links:

- Read/Write NVS with USB Flash Drive
- Virtual USB Flash Drive UF2 OTA Upgrade

USB Extended Screen Solution

The USB extended screen solution allows a device to function as an additional display through a USB connection. It supports data transmission over a single USB cable, including audio, touch information, and video. This solution can be applied in various scenarios such as computer monitors, signature pads, and extended display setups.

Host-side Driver Since USB 2.0 does not support HDMI transmission, the host must transmit image data to the device. Currently, this solution only supports the Windows platform, utilizing the Windows driver model, [*IDD (Indirect Display Driver)*](https://learn.microsoft.com/en-us/windows-hardware/drivers/display/indirect-display-driver-model-overview). This driver captures the desktop image from Windows. Due to USB' s speed limitations, the image is first compressed into formats like JPEG, and then transmitted through the USB vendor interface. Each image frame is prepended with a 16-byte header containing details such as image width, height, format, length, and compression type.

Features

- Supports image transmission via USB
- Supports audio transmission via USB
- Supports touch data transmission via USB

Hardware

- Chips: ESP32-S2, ESP32-S3, ESP32-P4
- Peripheral: USB-OTG

Links

- P4 USB Extended Screen Example

5.1.8 Self-Powered USB Device Solutions

According to the USB protocol requirements, self-powered USB devices must detect the 5V VBUS voltage to determine if the device is unplugged, thereby enabling hot-plugging. For host-powered devices, since the device shuts down immediately when the host VBUS is disconnected, there is no need to implement this logic.

There are generally two methods for USB device VBUS detection: detection by USB PHY hardware, or **detection by software with the help of ADC/GPIO**.

The internal USB PHY of ESP32S2/S3/P4 does not support hardware detection logic, this function needs to be implemented by software with the help of ADC/GPIO. Among them, using the GPIO detection method is the simplest. The implementation is as follows:

For ESP-IDF 4.4 and earlier versions:

1. On the hardware side, you need to allocate an additional I/O (except for special pins) connected to ESP32S2/S3 through a voltage divider with two resistors (e.g., two 100KΩ). (Note: ESP32S2/S3 I/O maximum input voltage is 3.3v.)
2. After the `tinyusb_driver_install`, it is necessary to call the `usbd_vbus_detect_gpio_enable` function to enable VBUS detection. The implementation code for this function is as follows. Please copy it to the required location for invocation:

```
/**
 *
 * @brief For USB Self-power device, the VBUS voltage must be monitored to achieve␣
↪hot-plug,
 *        The simplest solution is detecting GPIO level as voltage signal.
 *        A divider resistance Must be used due to ESP32S2/S3/P4 has no 5V␣
↪tolerate pin.
 *
 *   5V VBUS ⍰⍰⍰⍰⍰⍰⍰⍰    ⍰⍰⍰⍰⍰⍰⍰⍰   GND
 *    ⍰⍰⍰⍰⍰⍰⍰⍰ 100K ⍰⍰⍰⍰⍰⍰ 100K ⍰⍰⍰⍰⍰⍰⍰
 *           ⍰⍰⍰⍰⍰⍰⍰⍰ ⍰   ⍰⍰⍰⍰⍰⍰⍰⍰
 *                     ⍰           GPIOX
```

(continues on next page)

(continued from previous page)

```
 *                    ⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
 *        The API Must be Called after tinyusb_driver_install to overwrite the␣
↪default config.
 * @param gpio_num, The gpio number used for vbus detect
 *
 */
static void usbd_vbus_detect_gpio_enable(int gpio_num)
{
    gpio_config_t io_conf = {
        .intr_type = GPIO_INTR_DISABLE,
        .pin_bit_mask = (1ULL<<gpio_num),
        //set as input mode
        .mode = GPIO_MODE_INPUT,
        .pull_up_en = 0,
        .pull_down_en = 0,
    };
    gpio_config(&io_conf);
    esp_rom_gpio_connect_in_signal(gpio_num, USB_OTG_VBUSVALID_IN_IDX, 0);
    esp_rom_gpio_connect_in_signal(gpio_num, USB_SRP_BVALID_IN_IDX, 0);
    esp_rom_gpio_connect_in_signal(gpio_num, USB_SRP_SESSEND_IN_IDX, 1);
    return;
}
```

For ESP-IDF 5.0 and later versions:

1. Same as above, the hardware needs to occupy an additional IO (arbitrarily specified, except for special pins), which is connected to ESP32S2/S3 through two resistor voltage dividers (for example, two 100KΩ);
2. Initialize the IO used to detect VBUS to GPIO input mode;
3. Configure IO directly into `tinyusb_config_t` (For details, please refer to):

```
#define VBUS_MONITORING_GPIO_NUM GPIO_NUM_4
const tinyusb_config_t tusb_cfg = {
    .device_descriptor = &descriptor_config,
    .string_descriptor = string_desc_arr,
    .string_descriptor_count = sizeof(string_desc_arr) / sizeof(string_desc_
↪arr[0]),
    .external_phy = false,
    .configuration_descriptor = desc_configuration,
    .self_powered = true,
    .vbus_monitor_io = VBUS_MONITORING_GPIO_NUM,
};
ESP_ERROR_CHECK(tinyusb_driver_install(&tusb_cfg));
```

If you are using native TinyUSB development, you need to configure this during the PHY initialization stage:

```
const usb_phy_otg_io_conf_t otg_io_conf = USB_PHY_SELF_POWERED_DEVICE(VBUS_
↪MONITORING_GPIO_NUM);
phy_conf.otg_io_conf = &otg_io_conf;
usb_new_phy(&phy_conf, &s_phy_hdl);
```

5.1.9 Prevent Windows from incrementing COM numbers based on USB device serial number

Due to the fact that any device connected to a Windows PC is identified by its VID (Vendor ID), PID (Product ID), and Serial number. If any of these three parameters undergo a change, the PC will detect new hardware and assigns it to a different COM port. For more details, please refer to the Windows USB device registry entries.

In the ESP ROM Code, the configuration of USB descriptors is as follows:

	ESP32S2	ESP32S3	ESP32C3	ESP32C5	ESP32C6	ESP32C61	ESP32H2	ESP32P4
Interface	usb-otg	usb-serial-jtag	usb-serial-jtag	usb-serial-jtag	usb-serial-jtag	usb-serial-jtag	usb-serial-jtag	usb-serial-jtag
VID	0x303a	0x303a	0x303a	0x303a	0x303a	0x303a	0x303a	0x303a
PID	0x0002	0x1001	0x1001	0x1001	0x1001	0x1001	0x1001	0x1001
Revision (bcdDevice)	N/A	v1.01	v1.01	v1.02	v1.02	v1.02	v1.02	v1.02
Serial	0	MAC address	MAC address	MAC address	MAC address	MAC address	MAC address	MAC address

- For ESP32S2 with USB On-The-Go (usb-otg), the Serial is a constant 0, and it remains the same for every device, resulting in consistent COM port numbers.
- For ESP32S3 (usb-serial-jtag) or other chips with usb-serial-jtag interface, the Serial is set to the device MAC address. Each device has a unique MAC address, leading to different COM port numbers by default with incremental assignment.

Incrementing COM numbers pose additional challenges for mass production programming. For customers requiring firmware downloads via USB, it is recommended to modify the Windows rules for incrementing the COM number to prevent incrementing the number based on the Serial number.

Solution

Open Command Prompt in Administrator Mode `on Windows`, and execute the following command. This will add a registry entry to prevent incremental numbering based on the Serial number. **After completing the setup, please restart your computer to apply the changes**.

```
REG ADD HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\usbflags\303A10010101 /
↪V IgnoreHWSerNum /t REG_BINARY /d 01
REG ADD HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\usbflags\303A10010102 /
↪V IgnoreHWSerNum /t REG_BINARY /d 01
```

Users have the option to download the script `ignore_hwserial_usb_serial_jtag.bat` , then run it with administrative privileges by right-clicking and selecting **Run as Administrator**.

Note: The above command is for Windows 10/11. For Windows 7 or other versions, the registry path is different. Please download the script `ignore_hwserial_usb_serial_jtag_win7.bat` and run it with Administrative privileges.

5.1.10 USB Signal Quality Test

Signal quality testing of the USB 2.0 interface is a critical step to ensure that USB 2.0 devices comply with the specification and qualify for the official USB 2.0 certification logo. The scope of signal quality testing primarily includes: eye diagram measurement, signaling rate, End-of-Packet (EOP) bit width, signal crossover voltage range, JK jitter, KJ jitter, consecutive jitter, as well as rise and fall times. Among these, the eye diagram test is one of the most fundamental and essential measurements in serial data applications.

Eye Diagram Test

For USB 2.0 High-Speed signaling, the measured eye diagram must meet the compliance template requirements specified by the USB-IF.

The eye diagram test results of USB 2.0 High-Speed signals shall comply with the eye diagram template specified in the USB 2.0 specification.

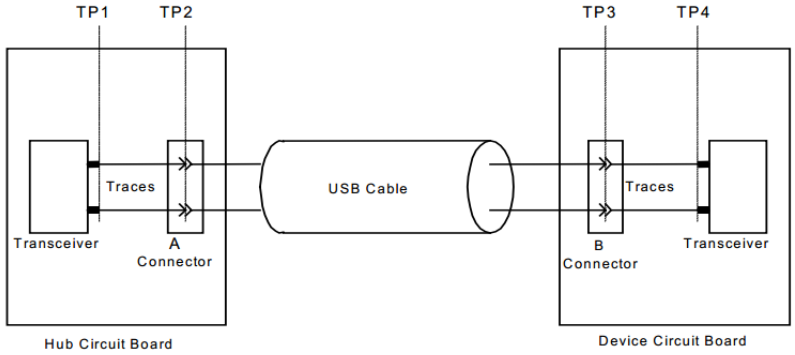
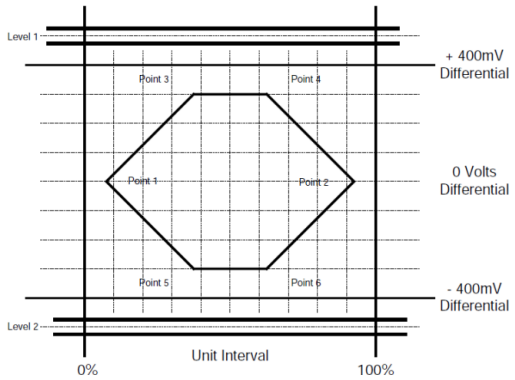
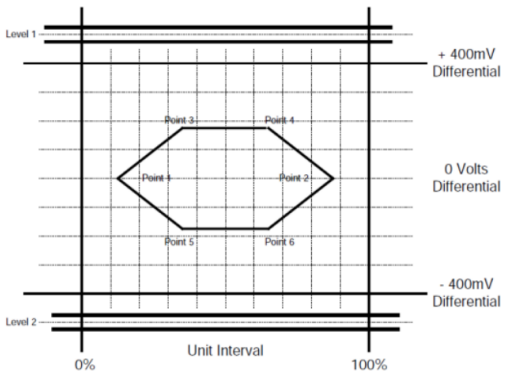


Fig. 1: USB 2.0 High-Speed Signal Measurement Plane



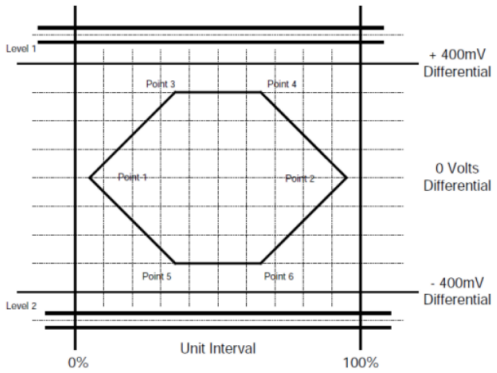
	Voltage Level (D+ - D-)	Time (% of Unit Interval)
Level 1	525 mV in UI following a transition, 475 mV in all others	N/A
Level 2	-525 mV in UI following a transition, -475 in all others	N/A
Point 1	0 V	7.5% UI
Point 2	0 V	92.5% UI
Point 3	300 mV	37.5% UI
Point 4	300 mV	62.5% UI
Point 5	-300 mV	37.5% UI
Point 6	-300 mV	62.5% UI

Fig. 2: The transmit waveform requirements for hubs measured at TP2 and for devices without captive cables measured at TP3



	Voltage Level (D+ - D-)	Time (% of Unit Interval)
Level 1	525 mV in UI following a transition, 475 mV in all others	N/A
Level 2	-525 mV in UI following a transition, -475 in all others	N/A
Point 1	0 V	12.5% UI
Point 2	0 V	87.5% UI
Point 3	175 mV	35% UI
Point 4	175 mV	65% UI
Point 5	-175 mV	35% UI
Point 6	-175 mV	65% UI

Fig. 3: The transmit waveform requirements for devices with captive cables measured at TP2



	Voltage Level (D+ - D-)	Time (% of Unit Interval)
Level 1	525 mV in UI following a transition, 475 mV in all others	N/A
Level 2	-525 mV in UI following a transition, -475 in all others	N/A
Point 1	0 V	5% UI
Point 2	0 V	95% UI
Point 3	300 mV	35% UI
Point 4	300 mV	65% UI
Point 5	-300 mV	35% UI
Point 6	-300 mV	65% UI

Fig. 4: The transmit waveform requirements for hub transceivers measured at TP1 and for device transceivers measured at TP4

Required Materials

- Windows PC
- USB XHSETT: USB-IF Compliance Tools, used to control USB 2.0 devices and hosts, as well as USB 3.1 hosts and hubs, and to place them into compliance test modes
- USB Test Fixtures: Provided by instrument vendors or purchased from USB-IF recommended test labs
- Oscilloscope (> 2 GHz), compliance test software, probes, etc

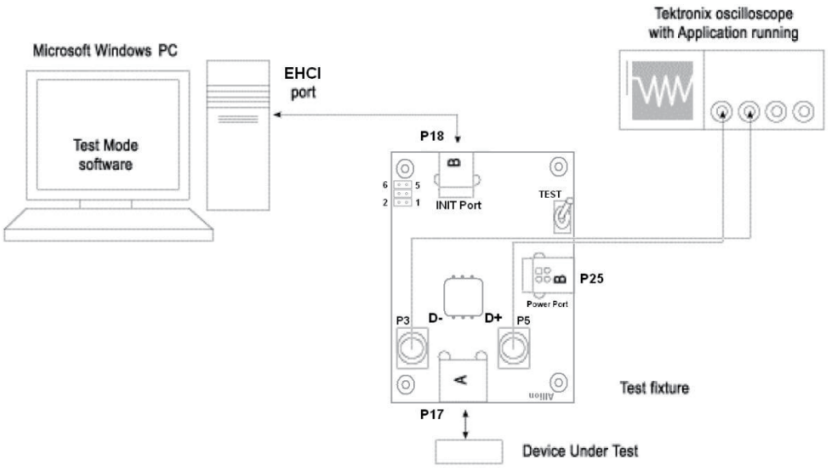


Fig. 5: USB 2.0 Signal Quality Test System

Test Procedure

1. ESP32-S2/S3/P4 USB Test Firmware Download the ESP32-S2/S3/P4 USB test firmware from the Launchpad platform. ESP32-P4 is used for High-Speed testing, and ESP32-S2/S3 is used for Full-Speed testing.

2. Hardware Connection Connect the PC, oscilloscope, and USB test fixtures. Note that the High-Speed test fixtures differ from those used for Full/Low-Speed testing. Please refer to the connection diagrams provided in the oscilloscope compliance test software.

3. XHSETT Installation and Configuration Check the USB port information in the PC Device Manager:

- If the PC has a USB 3.x host controller, use the XHSETT version. The xHCI host controller supports USB 1.x/2.0/3.x.

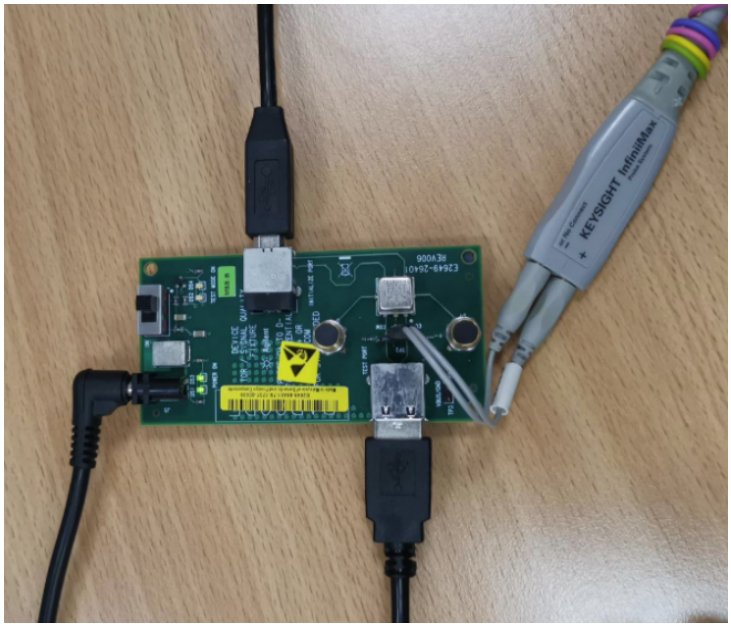


Fig. 6: High-Speed Device Signal Quality Test Connection Diagram

- If the PC has a USB 2.0 host controller, use the HSETT version. HSETT supports USB 1.x/2.0.

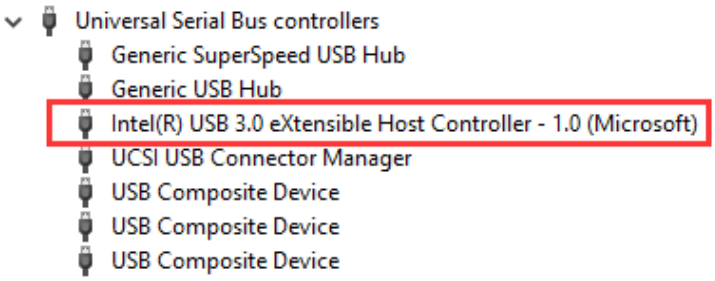


Fig. 7: USB 3.x Host Controller

Note: After the HSETT/XHSETT software is installed and launched, it will take control of all USB ports on the PC, causing them to become temporarily unavailable. Therefore, before running the HSETT/XHSETT software and restarting the test host, make sure to connect a PS/2 keyboard and mouse in advance to avoid interruption of operation. If a PS/2 mouse is not available, consider using remote desktop or other remote control tools as an alternative.

After completing the installation of the HSETT/XHSETT software, configure the settings as follows:

Launch the XHSETT software and select Device mode:

Click the Enumerate Bus button to scan. After the device is detected, select the device under test and choose Device Command:

- For High-Speed devices, select TEST_PACKET under Device Command
- For Full-Speed devices, select (LOOP) DEVICE DESCRIPTOR under Device Command

4. Oscilloscope Eye Diagram Test The USB 2.0 compliance analysis software generates an eye diagram by analyzing the standard test signals issued through XHSETT/HSETT. The operation of compliance test software may vary depending on the oscilloscope vendor; please refer to the oscilloscope vendor' s user documentation for detailed instructions on performing the eye diagram test.

In USB 2.0 eye diagram testing, a more open eye indicates better signal quality. The test results must remain entirely

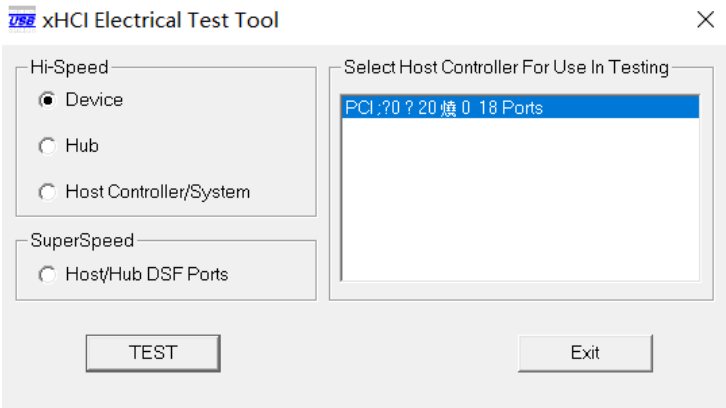


Fig. 8: XHSETT Mode Selection

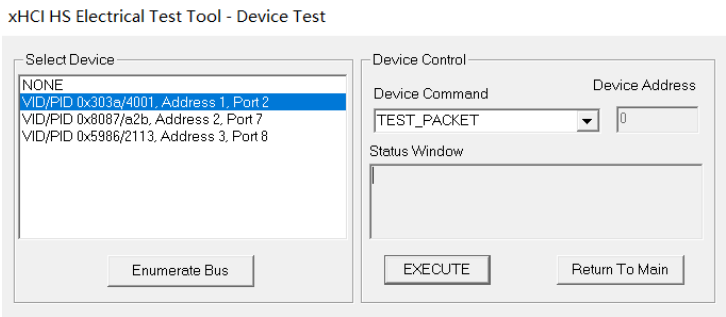


Fig. 9: XHSETT Device Selection

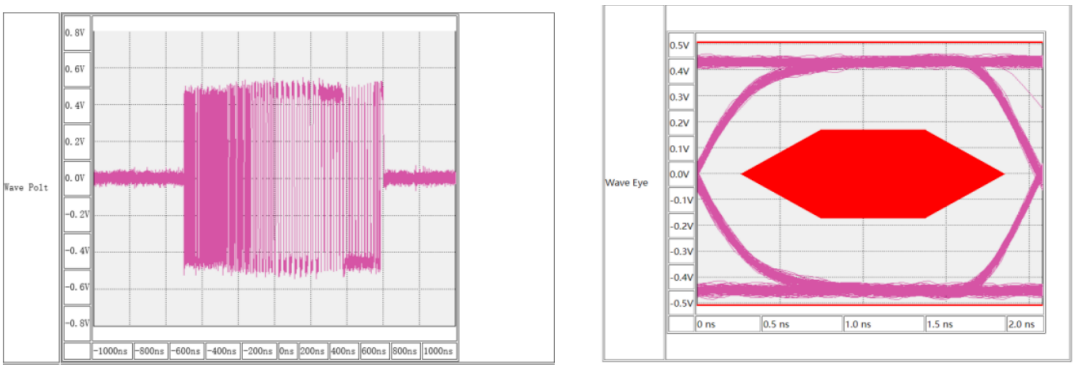


Fig. 10: Oscilloscope Eye Diagram Test

within the compliance region of the eye diagram template. Any trace touching or crossing the template boundary is considered a signal quality failure.

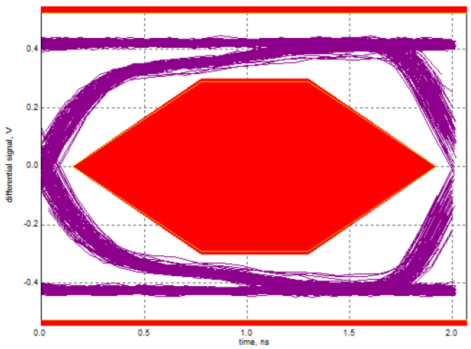


Fig. 11: Passing Eye Diagram

4.1 USB 2.0 High-Speed For eye diagrams that fail compliance testing, analysis can be performed from the following aspects:

- Verify that the oscilloscope probes are properly calibrated
- Ensure that standardized USB test fixtures are used
- Replace cables/connectors to rule out poor-quality accessories
- Check whether the PCB design complies with USB 2.0 specifications. For example, if ESD protection devices with high parasitic capacitance are used on the USB High-Speed D+/D- lines, the eye diagram may fail the test. It is recommended to select ESD diodes with parasitic capacitance lower than 1 pF

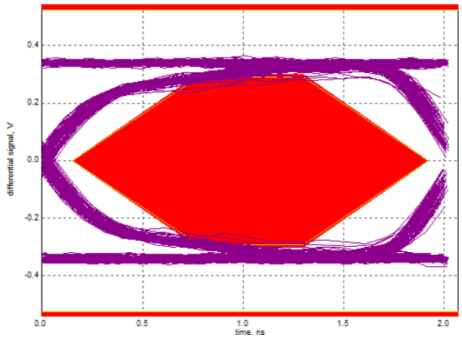


Fig. 12: Failing Eye Diagram

4.2 USB 2.0 Full-Speed For Full-Speed devices, please refer to the USB compliance test software provided by the oscilloscope vendor to perform the eye diagram test:

For the above Full-Speed eye diagram test failures, series resistors should be added on D+/D- to mitigate overshoot. Test results show that using 22 Ω, 33 Ω, or 44 Ω resistors can all improve overshoot, with 33 Ω providing the best effect.

5.1.11 TinyUSB Application Guide

This guide contains the following content:

Table of Contents

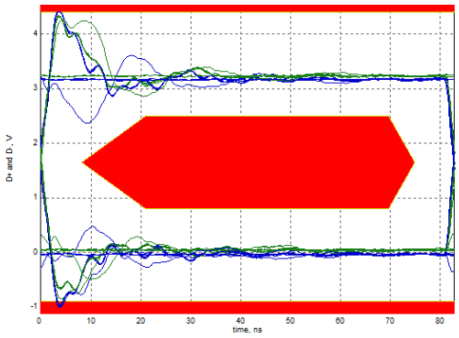


Fig. 13: Eye diagram that failed the test (overshoot)

- *Introduction to TinyUSB*
  - *Chip Selection*
- *TinyUSB Components*
  - *esp_tinyusb Component*
  - *espressif/tinyusb Component*
- *Introduction to USB Device*
  - *USB Audio (UAC)*
  - *USB Video (UVC)*

Introduction to TinyUSB

TinyUSB is an open-source embedded USB Host/Device stack library primarily designed to support USB functionalities on small microcontrollers. Developed and maintained by Adafruit, it aims to provide a lightweight, cross-platform, and easy-to-integrate USB protocol stack. TinyUSB supports various USB device types, including HID (Human Interface Device), MSC (Mass Storage Class), CDC (Communication Device Class), MIDI (Musical Instrument Digital Interface), and more, making it suitable for various embedded systems and IoT devices.Based on the native TinyUSB, the following components have been packaged.

Chip Selection

SoC	USB1.1 Full Speed	USB2.0 High Speed
ESP32-S2	Supported	
ESP32-S3	Supported	
ESP32-P4	Supported	Supported

TinyUSB Components

esp_tinyusb Component The esp_tinyusb component wraps a series of TinyUSB APIs, making it easy to integrate USB CDC-ACM, MSC, MIDI, HID, DFU, ECM/NCM/RNDIS classes into your project.

How to Use

- Run `idf.py add-dependency esp_tinyusb~1.0.0` in the project directory to add a dependency on the `esp_tinyusb` library.
- Configure the desired USB class in `menuconfig`.
- Use the esp_tinyusb API in your project.

The esp_tinyusb application examples provide examples for USB drives, serial ports, HID devices, composite devices, and more.

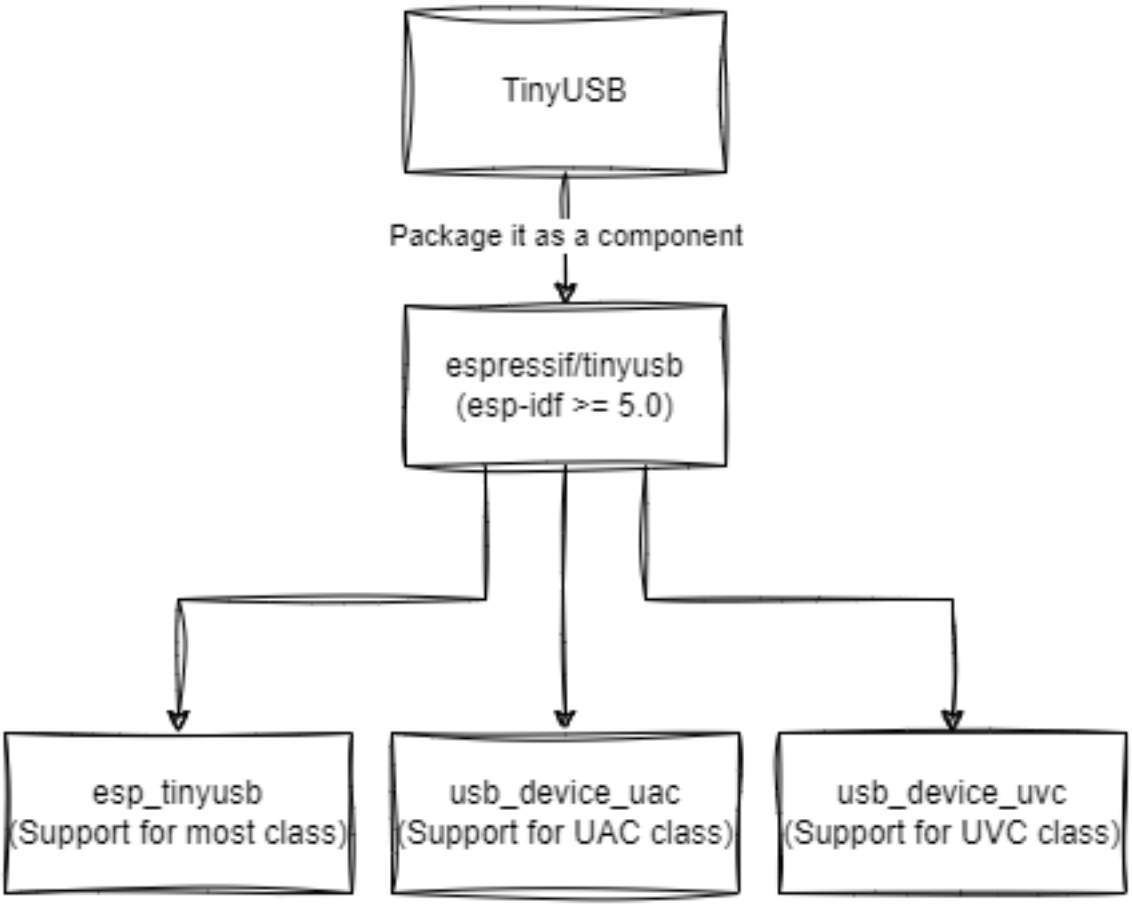


Fig. 14: TinyUSB Components

- USB Composite Device Example
- USB Serial Device Example
- USB Log Output Example
- USB HID Device Example
- USB MIDI Music Device Example
- USB MSC Device Example
- USB Network Device Example

Note: The esp_tinyusb library encapsulates many USB classes, making it easy to develop USB classes supported by esp_tinyusb. It is worth noting that this also makes certain modifications difficult. It is suitable for simple USB applications.

espressif/tinyusb Component The espressif/tinyusb component is based on the original tinyusb repository, encapsulating the tinyusb repository code into a component for easier use in your projects.

Note: This component requires ESP-IDF release/v5.0 or higher.

How to Use:

- Run `idf.py add-dependency "tinyusb~0.15.10"` in the project directory to add a dependency on the `espressif/tinyusb` library.
- Create a `tusb_config.h` file that defines a series of macros to configure TinyUSB and provide it to the espressif/tinyusb component. Also, add the following code in the main component' s CMakeLists.txt:

```
idf_component_get_property(tusb_lib espressif__tinyusb COMPONENT_LIB)
target_include_directories(${tusb_lib} PRIVATE path_to_your_tusb_
↪config)
```

- Create a `usb_descriptor.h` file that defines the USB device descriptors, including device descriptors, configuration descriptors, interface descriptors, etc.
- Create a `usb_descriptors.c` file that provides USB descriptor callbacks for tinyusb.

espressif/tinyusb application examples:

- USB Keyboard and Mouse Device Example
- USB Wireless Drive Example
- Windows Surface Dial HID Example
- Serial to USB Example

Note: The espressif/tinyusb library offers more flexibility, allowing for easier customization of USB devices. It is suitable for complex USB applications. On IDF release/v4.4, you can use the leeebo/tinyusb_src <https://components.espressif.com/components/leeebo/tinyusb_src>_ component, which serves the same purpose as espressif/tinyusb. It mainly supplements the support for ESP-IDF release/v4.4 in espressif/tinyusb.

Introduction to USB Device

USB Audio (UAC) TinyUSB supports the USB UAC2.0 standard for transmitting audio data over USB. It has the following features:

- Supports up to 32-bit/384kHz audio streams
- Compatible with USB1.1 Full Speed and USB2.0 High Speed
- Lower latency

Transmission Methods: UAC only supports synchronous transmission in USB transfers, so UAC audio devices use synchronous endpoints. Because synchronous transmission does not involve retransmission, it has low latency. However, because the transmission between the host and the device is asynchronous, it may cause brief silences or pops. This results in three synchronization methods:

- **SYNC Synchronization** Synchronize the output clock with the SOF packet of each frame.
- **Adaptive** Adjust the output sampling rate according to the host' s data transfer rate.
- **ASYNC Asynchronous** Unlike the other two methods, it adds a feedback endpoint. The device informs the host of the subsequent transmission rate based on the current rate, thereby completing data retransmission or under-transmission, without needing to adapt to the host' s transmission frequency.

About the Feedback Endpoint for ASYNC Asynchronous Transmission Enable the macro `CFG_TUD_AUDIO_ENABLE_FEEDBACK_EP` to implement feedback rate calculation. TinyUSB provides multiple feedback data calculation methods, with the FIFO-based feedback calculation (`AUDIO_FEEDBACK_METHOD_FIFO_COUNT`) being the simplest and most practical. The following virtual function needs to be implemented to complete the setup.

```
void tud_audio_feedback_params_cb(uint8_t func_id, uint8_t alt_itf, audio_feedback_
↪params_t* feedback_param)
{
    (void)func_id;
    (void)alt_itf;
    // Set feedback method to FIFO counting
    feedback_param->method = AUDIO_FEEDBACK_METHOD_FIFO_COUNT;
    feedback_param->sample_freq = s_uac_device->current_sample_rate;

    ESP_LOGD(TAG, "Feedback method: %d, sample freq: %d", feedback_param->method,␣
↪feedback_param->sample_freq);
}
```

The working principle is that the UAC Class maintains a software FIFO with a size of `CFG_TUD_AUDIO_FUNC_1_EP_OUT_SW_BUF_SZ`. By setting this memory size to 10ms of data, the UAC driver has a buffer area. The driver will maintain the FIFO' s water level at half its size through the feedback endpoint. When data is missing, the host will send more data in one packet, and when data is lacking, the host will send less data.

In practical applications, it is recommended to buffer half the FIFO size of data in the UAC internally before starting playback at the beginning of each new audio transmission (e.g., if no data arrives for more than 100ms, it is considered a new audio transmission). This ensures that the I2S always has data to fetch, avoiding pops and noise. The software FIFO' s size will remain stable based on the feedback endpoint.

Refer to *USB Device UAC* for more information.

USB Video (UVC) TinyUSB supports the USB UVC1.5 standard for transmitting video data over USB, capable of transmitting various video formats, including uncompressed YUV formats, compressed formats like MJPEG, H264, and H265.

Transmission Methods:

- When the video streaming interface (USB Video streaming) transmits video, its transmission endpoints are isochronous or bulk endpoints.
- When the video streaming interface transmits still images, its transmission type is bulk endpoints.

Transmitting Images: UVC can transmit various video formats, which are defined by the video descriptors' Format and Frame.

Image Type	Format	Frame
MJPEG	FORMAT_MJPEG: 0x06	FRAME_MJPEG: 0x07
YUV2/NV12/M420/I420	FORMAT_UNCOMPRESSED: 0x04	FRAME_UNCOMPRESSED: 0x05
H264	FORMAT_H264: 0x013	FRAME_H264: 0x014
H265	FORMAT_FRAME_BASED: 0x10	FRAME_FRAME_BASED: 0x11

The Frame-based format is special in that it can store any image format as long as the image is stored frame by frame, such as MJPG, H264, H265, etc. The specific image format is indicated by the GUID field.

Dual Camera In UVC devices, a single physical camera has a VC (video control) descriptor, and a VC descriptor can have multiple VS (video streaming) descriptors, indicating that this camera can transmit multiple image formats. However, in some special hardware, there are two hardware cameras, in which case two VC descriptors are needed.

```
USB Descriptor
    |
    |-- Video Control
    |    |-- Video Streaming
    |
    |-- Video Control
         |-- Video Streaming
```

Refer to *USB Device UVC* for more information.

5.1.12 Developing with Native tinyusb

This guide contains the following content:

Table of Contents

- *Project Directory*
- *tusb_config.h File*
- *usb_descriptors.h File (Optional)*
- *usb_descriptors.c File*
- *Initializing USB Phy*
- *Initializing the TinyUSB Stack*
- *Device-Level Weak Functions*
- *Implementing USB Class-Specific Callback Functions.*

This section will introduce how to use tinyusb components for development.

Project Directory

First, create the following directory structure:

```
project_name
|
|-- main
    |-- CMakeLists.txt
    |-- idf_component.yml
    |-- main.c
    |-- tusb
         |-- tusb_config.h
         |-- usb_descriptors.c
         |-- usb_descriptors.h
```

Add component dependencies in the main component of the project, referencing *espressif/tinyusb*.

The tusb folder primarily contains files to be provided to tinyusb. We place them in a separate folder to ensure simplicity in dependency relationships. Then, add the following statements in the `main/CMakeLists.txt` file (after idf_component_register):

```
# espressif__tinyusb should match the current tinyusb dependency name
idf_component_get_property(tusb_lib espressif__tinyusb COMPONENT_LIB)

target_include_directories(${tusb_lib} PUBLIC "${COMPONENT_DIR}/tusb")
target_sources(${tusb_lib} PUBLIC "${COMPONENT_DIR}/tusb/usb_descriptors.c")
```

Note: Regarding reverse dependency issues, since the project needs to depend on tinyusb and provide header files to tinyusb, it inevitably leads to reverse dependency problems. Currently, this can be resolved by directly compiling all key files of tinyusb into the main component.

tusb_config.h File

Most of tinyusb' s functionalities are controlled through macros, so we need to declare the required functionalities in the `tusb_config.h` file. Here are some key macros:

System Settings Macros:

- **CFG_TUSB_RHPORT0_MODE**: Defines the connection method and speed to the USB Phy. The following method indicates a USB device with USB full speed.

```
#define CFG_TUSB_RHPORT0_MODE     (OPT_MODE_DEVICE | OPT_MODE_FULL_
↪SPEED)
```

- **CFG_TUSB_RHPORT1_MODE**: Defines the connection method and speed to the USB Phy. The following method indicates a USB device with USB high speed.

```
#define CFG_TUSB_RHPORT1_MODE     (OPT_MODE_DEVICE | OPT_MODE_HIGH_
↪SPEED)
```

- **ESP_PLATFORM**: This macro needs to be enabled when using the esp-idf platform for compilation.
- **CFG_TUSB_OS**: Defines the operating system for tinyusb. If using FreeRTOS, this macro needs to be enabled. It can also be disabled if not using an OS.

```
#define CFG_TUSB_OS            OPT_OS_FREERTOS
```

- **CFG_TUSB_OS_INC_PATH**: In ESP-IDF, it requires adding the "freertos/" prefix in the include path.

```
#define CFG_TUSB_OS_INC_PATH   freertos/
```

- **CFG_TUSB_DEBUG**: Enables the LOG print level of tinyusb. There are three levels in total.

```
#define CFG_TUSB_DEBUG         0
```

- **CFG_TUSB_DEBUG_PRINTF**: Defines the log printing function for tinyusb.

```
#define CFG_TUSB_DEBUG_PRINTF      esp_rom_printf
```

- **CFG_TUD_ENABLED**: Set to 1 to enable tinyusb device functionality.
- **CFG_TUSB_MEM_SECTION**: This macro can be enabled to allocate tinyusb memory to a specific memory section.
- **CFG_TUSB_MEM_ALIGN**: Defines the memory alignment method.

```
#define CFG_TUSB_MEM_ALIGN      __attribute__ ((aligned(4)))
```

USB Device Macros:

- **CFG_TUD_ENDPOINT0_SIZE**: Defines the maximum packet size for endpoint 0.

USB Class Macros:

Here, using the UVC Class as an example, each USB Class has its own macros:

- **CFG_TUD_VIDEO**: Configures the number of video control interfaces.
- **CFG_TUD_VIDEO_STREAMING**: Configures the number of video streaming interfaces.

Refer to the following file examples:

- ../components/usb/usb_device_uac/tusb/tusb_config.h
- ../components/usb/usb_device_uvc/tusb/tusb_config.h
- /usb/device/usb_hid_device/hid_device/tusb_config.h

usb_descriptors.h File (Optional)

This file is mainly used to place custom USB descriptors. Tinyusb provides many descriptor templates, but if they do not meet your needs, you need to define your own set of USB descriptors. Note that it is best to use the predefined descriptors in tinyusb, as it makes descriptor assembly and length calculation more convenient.

Refer to the following file examples:

- ../components/usb/usb_device_uac/tusb_uac/uac_descriptors.h
- ../components/usb/usb_device_uvc/tusb/usb_descriptors.h
- /usb/device/usb_hid_device/hid_device/usb_descriptors.h

usb_descriptors.c File

This file primarily implements several weak functions for obtaining descriptors, such as getting the device descriptor, configuration descriptor, and string descriptor.

```
uint8_t const *tud_descriptor_device_cb(void);

uint8_t const *tud_descriptor_configuration_cb(uint8_t index);

uint16_t const *tud_descriptor_string_cb(uint8_t index, uint16_t langid);
```

Points to Note:

- The length of the configuration descriptor must equal the actual length.
- The endpoint numbers used in the configuration descriptor’s endpoint descriptors must not overlap.

Refer to the following file examples:

- ../components/usb/usb_device_uvc/tusb/usb_descriptors.c
- ../components/usb/usb_device_uac/tusb/usb_descriptors.c
- /usb/device/usb_hid_device/hid_device/usb_descriptors.c

Initializing USB Phy

To initialize the internal USB Phy:

```
static void usb_phy_init(void)
{
    // Configure USB PHY
    usb_phy_config_t phy_conf = {
        .controller = USB_PHY_CTRL_OTG,
        .otg_mode = USB_OTG_MODE_DEVICE,
        .target = USB_PHY_TARGET_INT,
    };
    usb_new_phy(&phy_conf, &s_uvc_device.phy_hdl);
}
```

If using an external USB Phy, refer to *Use an external PHY (ESP32-S2/S3)*.

Initializing the TinyUSB Stack

Use the following code:

```
static void tusb_device_task(void *arg)
{
    while (1) {
        tud_task();
    }
}

int main(void) {
    usb_phy_init();
    bool usb_init = tusb_init();
    if (!usb_init) {
        ESP_LOGE(TAG, "USB Device Stack Init Fail");
        return ESP_FAIL;
    }
    xTaskCreatePinnedToCore(tusb_device_task, "TinyUSB", 4096, NULL, 5, NULL, 0);
}
```

Device-Level Weak Functions

These functions allow you to handle events such as device insertion, removal, suspension, and resumption.

```
// Invoked when the device is mounted
void tud_mount_cb(void)
{
}

// Invoked when the device is unmounted
void tud_umount_cb(void)
{
}

// Invoked when the device is suspended
void tud_suspend_cb(bool remote_wakeup_en)
{
}

// Invoked when the USB bus is resumed
void tud_resume_cb(void)
{
}
```

Implementing USB Class-Specific Callback Functions.

USB classes provide some weak functions to complete basic functions. Taking the UVC driver as an example, the source file is *video_device <https://github.com/hathach/tinyusb/blob/master/src/class/video/video_device.h>*.

By observing the API, it can be seen that the UVC Class provides two functions and one callback function:

```
bool tud_video_n_streaming(uint_fast8_t ctl_idx, uint_fast8_t stm_idx);

bool tud_video_n_frame_xfer(uint_fast8_t ctl_idx, uint_fast8_t stm_idx, void␣
↪*buffer, size_t bufsize);
```

(continues on next page)

(continued from previous page)

```
TU_ATTR_WEAK void tud_video_frame_xfer_complete_cb(uint_fast8_t ctl_idx, uint_
↪fast8_t stm_idx);
```

The `tud_video_n_frame_xfer` function is used to transfer a frame of image, and the `tud_video_frame_xfer_complete_cb` callback is used to check if the transfer is complete.

Additionally, different USB classes have special macro definitions to define software FIFO sizes or enable certain features. For example, the macro `CFG_TUD_VIDEO_STREAMING_EP_BUFSIZE` in the UVC Class is used to define the buffer size of the video streaming interface endpoint.

5.1.13 USB Type-C Hardware Design Guide

USB Type-C Overview

Since its official release in 2014, USB Type-C has rapidly gained widespread adoption in various electronic devices due to its compact size, reversible connector design, and support for high-power transmission capabilities. Compared to traditional USB Type-A interfaces, USB-C not only offers greater convenience in use but also better suits the design requirements of modern portable devices, demonstrating strong compatibility and practicality in mobile devices, laptops, and even certain high-power applications.

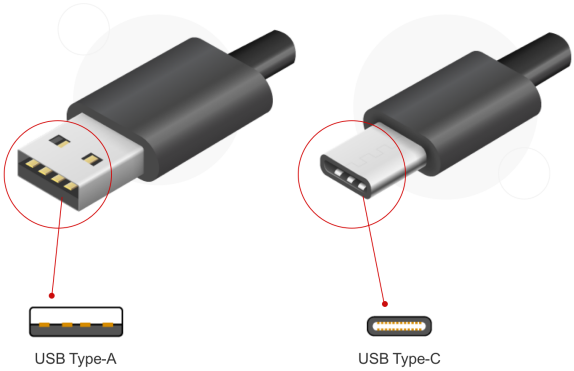


Fig. 15: USB Type-A and USB Type-C connectors

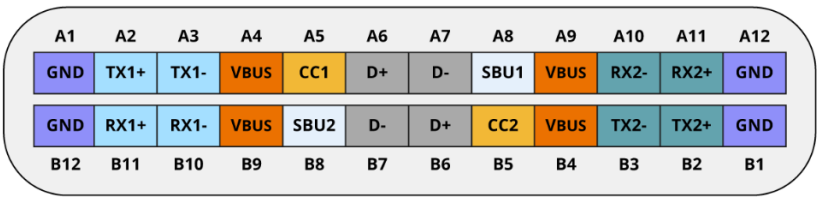


Fig. 16: USB Type-C Connector Pins

Table 1: USB Type-C Connector Pin Functions

Function	Pins	Description
USB 3.x	A2, A3, B2, B3, A10, A11, B10, B11	USB 3.x data transfer
USB 2.0	A6/B6 (D+), A7/B7 (D−)	USB 2.0 data transfer
Config	CC1, CC2	Configuration interface, plug detection, power delivery protocol info transfer, Vconn function
Sideband Use	SBU1, SBU2	Low speed signals are allocated exclusively for use in alternate modes.
Power	VBUS, GND	Power supply

In addition, USB Type-C connectors are available in various pin configurations to suit different application scenarios, such as 24-pin, 16-pin, 12-pin, and 6-pin versions. The 6-pin configuration includes only power pins and does not support data transmission; the 12-pin and 16-pin configurations add USB 2.0 data support on top of power delivery; the 24-pin configuration provides full-featured Type-C functionality. Users can select the appropriate pin configuration based on their specific application needs to achieve a balance between performance and cost.

USB Type-C Role Identification and Power Detection

Compared to previous USB specifications, USB Type-C defines a broader range of roles:

- **Power roles:** A USB-C port can act as a **Source** (power provider), a **Sink** (power consumer), or a **Dual-Role Power (DRP)** port.
- **Data roles:** A USB-C port can operate as a **Downstream-Facing Port (DFP)**, an **Upstream-Facing Port (UFP)**, or a **Dual-Role Data (DRD)** port.

The identification and configuration of these roles are accomplished through the CC lines within the Type-C connector.

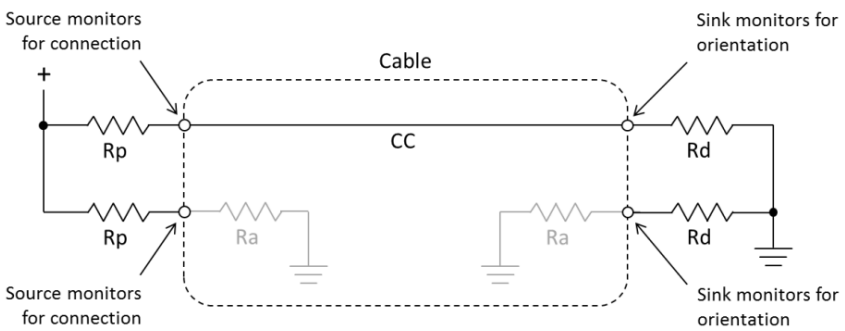


Fig. 17: Type-C CC Pull-up and Pull-down Model

- As a Source, the USB-C port asserts one pull-up resistor (Rp) on each CC pin.
- As a Sink, the USB-C port asserts a pull-down resistor (Rd) on each CC pin.
- When a Source-side USB-C port is connected to a Sink-side USB-C port, the Source detects the Sink' s Rd on CC and interprets this as a valid attachment, confirming the partner is a Sink. If both sides are Sources or both are Sinks, no valid connection (attachment) is established, and neither side will operate normally.

Note: For dual-role ports, the CC line will continuously alternate between pull-up and pull-down states.

The pull-down resistor Rd is typically 5.1 KΩ, while the pull-up resistor Rp determines the current-carrying capability. Without USB Power Delivery (PD), USB Type-C supports up to 5V and a maximum current of 3A:

Table 2: USB Type-C Power Capabilities

Power Profile	Rp (5V drive)	Rp (3.3V drive)
Default Power	56 KΩ ±20%	36 KΩ ±20%
1.5A @ 5V	22 KΩ ±20%	12 KΩ ±20%
3A @ 5V	10 KΩ ±20%	4.7 KΩ ±20%

Note: If your application requires power negotiation above 15W, support for alternate functions such as DisplayPort, or if the power and data roles do not match, a USB PD controller is required.

Introduction to USB PD

By default, the USB Type-C interface supports up to 5V and 3A (i.e., 15W), which is suitable for powering common low-power devices. To meet the needs of higher-power devices, the USB Power Delivery (PD) specification was introduced. The latest USB PD 3.2 version significantly increases power delivery capability, allowing up to 240W over a single USB Type-C cable, greatly expanding the application range of USB-C. Note that for voltages of 20V and power above 60W, the current is limited by cable specifications.

USB Version	BC 1.2	PD 1.0	PD 2.0	PD 3.0	PD 3.0 PPS	PD 3.1
Release date	2010	2012	2014	2015	2017	2021
USB type	Type-A	Type-A, Type-B	Type-C	Type-C	Type-C	Type-C
Output support	5V1, 5A		5V 3A, 9V 3A, 15V 3A, 20V 2.25A, 20V 3A, 20V 5A	5V 3A, 9V 3A, 15V 3A, 20V 2.25A, 20V 3A, 20V 5A	5V 3A, 9V 3A, 15V 3A, 20V 2.25A, 20V 3A, 20V 5A	5V 3A, 9V 3A, 15V 3A, 20V 3A, 20V 5A
					PPS: 3.3V-5.9V 3A, 3.3-11V 3A, 3.3-16V 3A, 3.3-21V 3A, 3.3-21V 5A	PPS: 3.3V-5.9V 3A, 3.3-11V 3A, 3.3-16V 3A, 3.3-21V 3A, 3.3-21V 5A
						AVS: 15-28V 5A, 15-36V 5A, 15-48V 5A

Fig. 18: Evolution of USB PD Versions

USB PD Power Negotiation Process USB PD operates by using the CC line of the USB Type-C interface as a data channel to negotiate voltage, current, and power direction. Once a power source and sink are connected via USB Type-C, USB PD messages are transmitted over the CC line using a 300kbps ±10% bi-phase mark code (BMC) signal. The negotiation process includes the following steps:

- The power source sends a Source_Capabilities message containing its available power profiles (e.g., 5V/3A, 9V/3A, etc.).
- The sink receives the Source_Capabilities message, selects the desired power profile, and sends a Request message.
- The source receives the Request message, checks if it can supply the requested power, and if so, sends an Accept message. The source then sends a PS_Rdy message to indicate that the requested power is being supplied. If the request cannot be met, a Reject message is sent.

The current ESP32 series chips are unable to parse the USB PD protocol, so an external PD controller is required to implement specific voltage requests.

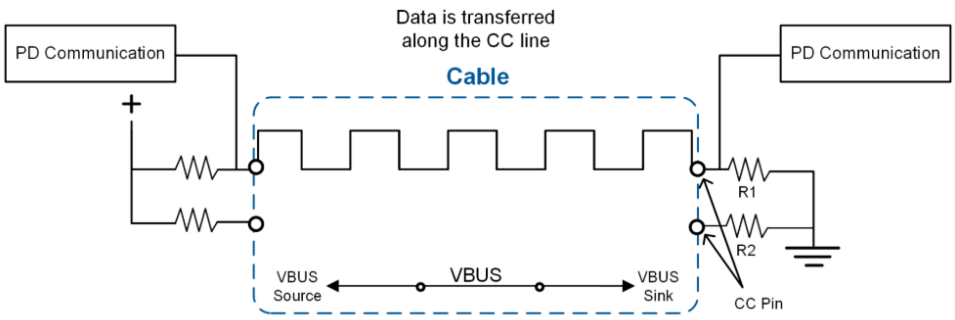


Fig. 19: USB PD Communication

USB Type-C Data Transmission

USB Type-C is compatible with multiple USB specifications, including USB 1.0, 1.1, 2.0, 3.2 Gen 1, 3.2 Gen 2, 3.2 Gen 2×2, USB4 20 Gbps, and USB4 40 Gbps. Given that current ESP32-series chips support up to USB 2.0 High-Speed, this document focuses on USB 2.0 and earlier specifications.

USB Type-C provides two sets of D+/D− pins **A6/A7** and **B6/B7**. To guarantee proper differential signaling regardless of plug orientation, **A6 is tied to B6** as **D+**, and **A7 is tied to B7** as **D−**. This differential pair carries the USB data, ensuring the link operates correctly in either insertion direction.

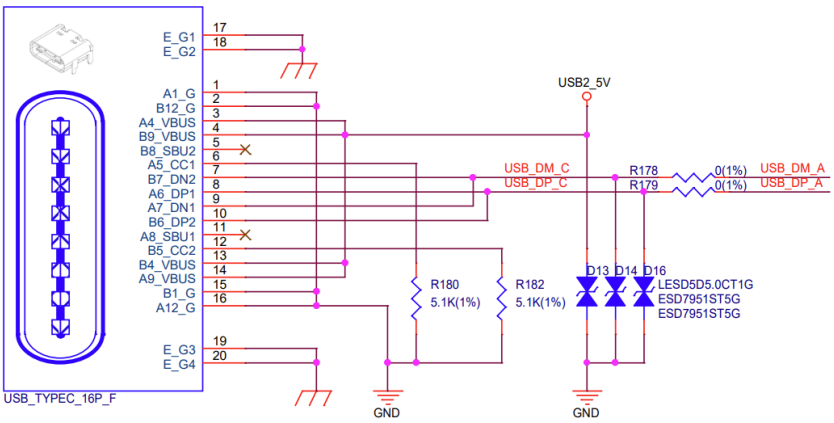


Fig. 20: D+/D− pins on USB Type-C

For the **SuperSpeed differential pairs (TX/RX)**, since current ESP32 devices do **not** support USB 3.x data rates, these pins should be left **unconnected (NC)**. No routing or terminations are required.

USB Type-C Hardware Reference Designs

Based on the requested voltage, the USB Type-C hardware design can be divided into two schemes: with or without an external PD controller. For the hardware solution with an external PD controller, this section will focus solely on the Device hardware design.

Device Hardware Design with PD Controller Since the USB-PD protocol is relatively complex, the device side can use a PD controller chip to simplify the design, quickly complete voltage negotiation and output the required voltage:

The IP2721 can automatically detect device connection status through the CC pin and use hardware PD protocol to parse power source capability information, thereby automatically requesting matching voltage output. Users can also select other models of PD controllers according to actual requirements to implement voltage request functionality.

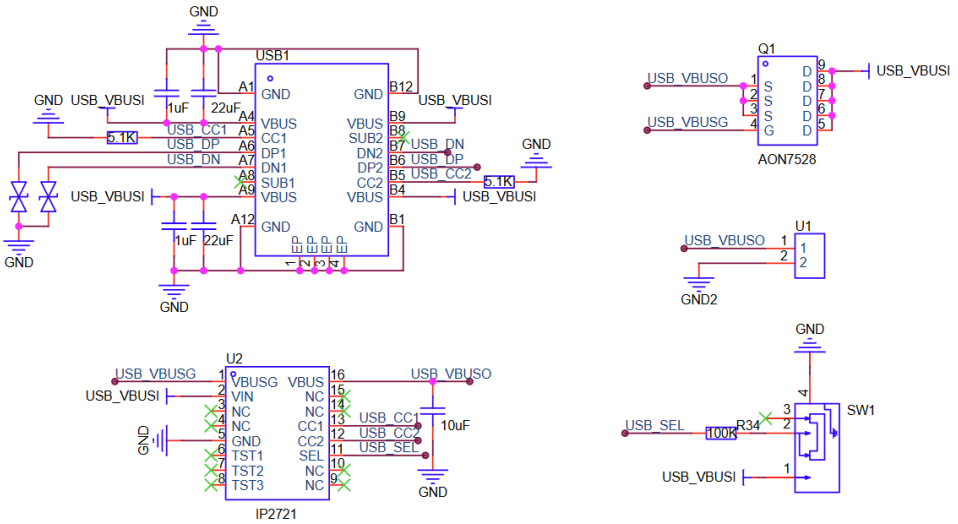


Fig. 21: IP2721 PD Controller

Hardware Design for Device, Host, and DRP without PD Controller

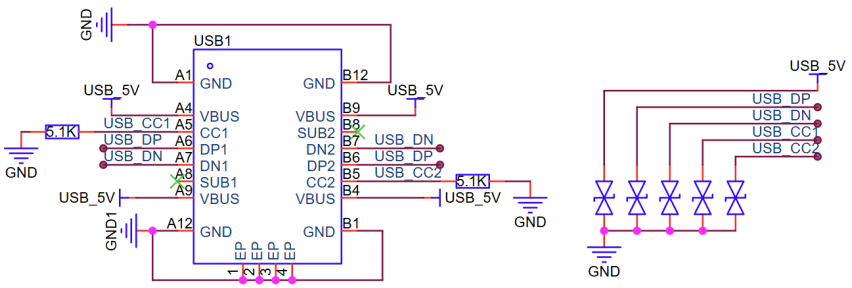


Fig. 22: USB Type-C Device Design

USB Type-C Device Design In the device design, CC1 and CC2 need to be pulled down with 5.1 KΩ resistors.

Important: When a device needs more than the USB default current (500 mA for USB 2.0; 900 mA after USB 3.x enumeration), it is necessary to read the source' s advertised current level on the Type-C CC line (using CC logic and a Type-C port controller) and set the device' s input current limit no higher than that level; otherwise, the source may be overloaded.

USB Type-C Host and DRP Design The Host and DRP design use the TI TUSB320LA Type-C CC logic and port controller, which configures the TUSB320 to switch between DFP, UFP, and DRP roles through the I2C interface. Users can also select other models of CC logic and port controllers according to actual requirements.

Important: In mass-produced products, USB ports must also account for conditions such as **overcurrent and overtemperature**. This ensures that in abnormal situations, current output can be limited to protect the power supply system. You can opt for USB port power switch chips with adjustable current limit thresholds, such as the CH217, to simplify the Host VBUS design.

USB Type-C Reference Documents

- Universal Serial Bus Type-C Cable and Connector Specification

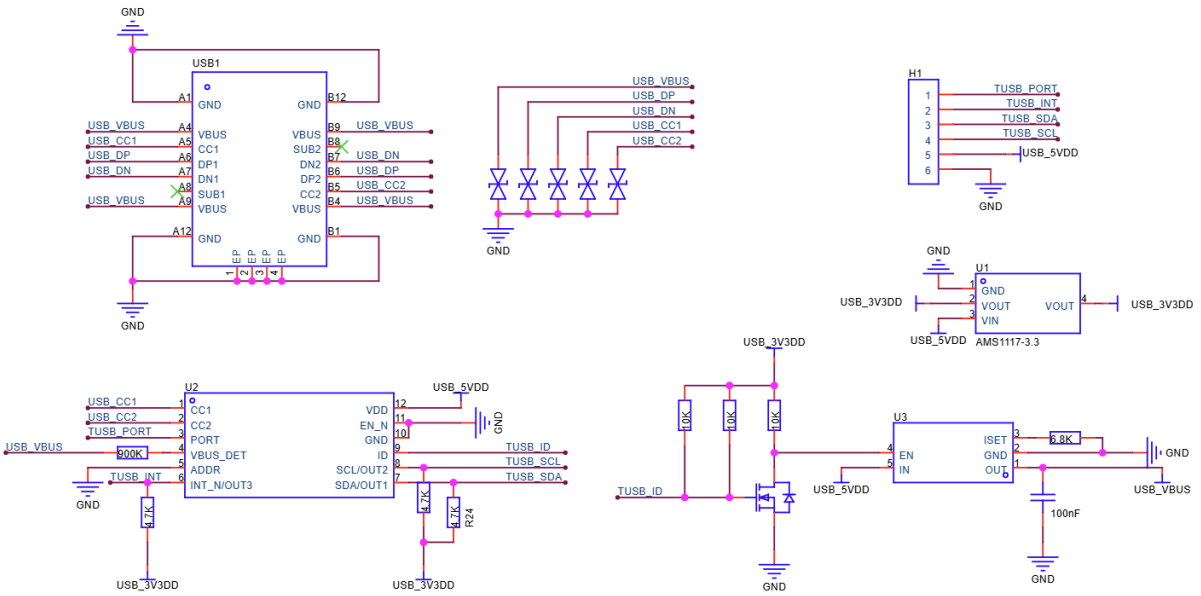


Fig. 23: USB Type-C Host and DRP Design

- USB Power Delivery: The Technology 2 - USB Type-C and Role Swap
- USB Type-C Charging Connectors: Design, Optimization, and Interoperability
- TUSB320 Datasheet

5.2 USB Host Drivers

5.2.1 USB Stream Component

`usb_stream` is an USB `UVC` + `UAC` host driver for ESP32-S2/ESP32-S3, which supports read/write/control multimedia streaming from usb device. For example, at most one UVC + one Microphone + one Speaker streaming can be supported at the same time.

Features:

1. Support video stream through UVC Stream interface, support both isochronous and bulk mode
2. Support microphone stream and speaker stream through the UAC Stream interface
3. Support volume, mute and other features control through the UAC Control interface
4. Supports automatic parse of device configuration descriptors
5. Support stream separately suspend and resume

USB Stream User Guide

- Development board
  1. Any `ESP32-S2` / `ESP32-S3` board with USB Host port can be used. Note that the port must be able to output voltage.
- USB UVC function
  1. Camera must be compatible with `USB1.1` full-speed mode
  2. Camera must support `MJPEG` output
  3. Users can manually specify the camera interface, transmission mode, and image frame params through *`uvc_streaming_config()`*

    4. For isochronous mode camera, interface `max packet size` should no more than `512` bytes
    5. For isochronous mode camera, frame stream bandwidth should be less than `4 Mbps` (500 KB/s)
    6. For bulk mode camera, frame stream bandwidth should be less than `8.8 Mbps` (1100 KB/s)
    7. Please refer example readme for special camera requirements
- USB UAC function
    1. Audio function must be compatible with `UAC1.0` protocol
    2. Users need to manually specify the spk/mic sampling rate, bit width params through *`uac_streaming_config()`* function
- USB UVC + UAC
    1. The UVC and UAC functions can be enabled separately. For example, only the UAC be configured to drive a usb headset, or only the UVC be configured to drive a USB camera
    2. If you need to enable UVC and UAC at the same time, note that the driver currently only supports `Composite devices` with both camera and audio interfaces, rather than two separate devices.

USB Stream Config Reference

1. Using *`uvc_config_t`* to configure camera resolution and frame rate parameters. Using *`uac_config_t`* to configure the audio sampling rate, bit width and other parameters. The parameters are described as follows:

```
uvc_config_t uvc_config = {
    .frame_width = 320, // mjpeg width pixel, for example 320
    .frame_height = 240, // mjpeg height pixel, for example 240
    .frame_interval = FPS2INTERVAL(15), // frame interval (100µs units), such as␣
↪15fps
    .xfer_buffer_size = 32 * 1024, // single frame image size, need to be␣
↪determined according to actual testing, 320 * 240 generally less than 35KB
    .xfer_buffer_a = pointer_buffer_a, // the internal transfer buffer
    .xfer_buffer_b = pointer_buffer_b, // the internal transfer buffer
    .frame_buffer_size = 32 * 1024, // single frame image size, need to determine␣
↪according to actual test
    .frame_buffer = pointer_frame_buffer, // the image frame buffer
    .frame_cb = &camera_frame_cb, //camera callback, can block in here
    .frame_cb_arg = NULL, // camera callback args
}

uac_config_t uac_config = {
    .mic_bit_resolution = 16, //microphone resolution, bits
    .mic_samples_frequence = 16000, //microphone frequency, Hz
    .spk_bit_resolution = 16, //speaker resolution, bits
    .spk_samples_frequence = 16000, //speaker frequency, Hz
    .spk_buf_size = 16000, //size of speaker send buffer, should be a multiple of␣
↪spk_ep_mps
    .mic_buf_size = 0, //mic receive buffer size, 0 if not use, else should be a␣
↪multiple of mic_min_bytes
    .mic_cb = &mic_frame_cb, //mic callback, can not block in here
    .mic_cb_arg = NULL, //mic callback args
};
```

2. Use the *`uvc_streaming_config()`* to config the UVC driver, or use the *`uac_streaming_config()`* to config the UAC driver
3. Use the *`usb_streaming_start()`* to turn on the stream, then the driver will handle USB connection and negotiation.
4. The host will matches the descriptors of the connected devices according to the user parameters. If the device fails to meet the configuration requirements, the driver prompt warning message
5. If the device meets user configuration requirements, the Host will continue to receive the IN stream (UVC and UAC mic), and will call the user' s callbacks when new frames ready.
    1. The camera callback will be triggered after a new MJPEG image ready. The callback can block during processing, because which works in an independent task context.
    2. The mic callback will be triggered after `mic_min_bytes()` bytes data received. But the callback here must not block in any way, otherwise it will affect the reception of the next frame. If the block

operations for mic is necessary, you can use the polling mode instead of the callback mode through `uac_mic_streaming_read()` api.

6. User can send speaker OUT stream using `uac_spk_streaming_write()` through a ringbuffer, the Host will fetch the data when USB is free to send.
7. Use the `usb_streaming_control()` to control the stream suspend/resume, uac volume/mute control can also be support if the USB device has such feature unit;
8. Use the `usb_streaming_stop()` to stop the usb stream, USB resource will be completely released.

Bug report

ESP32-S2 ECO0 Chip SPI screen jitter when work with usb camera

1. In earlier versions of the ESP32-S2 chip, USB transfers can cause SPI data contamination (esp32s2>=ECO1 and esp32s3 do not have this bug)
2. Software workaround

- `spi_ll.h` add below function

```
//components/hal/esp32s2/include/hal/spi_ll.h
static inline uint32_t spi_ll_tx_get_fifo_cnt(spi_dev_t *hw)
{
    return hw->dma_out_status.out_fifo_cnt;
}
```

- modify `spi_new_trans` implement as below

```
// The function is called to send a new transaction, in ISR or in the task.
// Setup the transaction-specified registers and linked-list used by the DMA (or␣
↪FIFO if DMA is not used)
static void SPI_MASTER_ISR_ATTR spi_new_trans(spi_device_t *dev, spi_trans_priv_t␣
↪*trans_buf)
{
    //...................
    spi_hal_setup_trans(hal, hal_dev, &hal_trans);
    spi_hal_prepare_data(hal, hal_dev, &hal_trans);

    //Call pre-transmission callback, if any
    if (dev->cfg.pre_cb) dev->cfg.pre_cb(trans);
#if 1
    //USB Bug workaround
    //while (!((spi_ll_tx_get_fifo_cnt(SPI_LL_GET_HW(host->id)) == 12) || (spi_ll_
↪tx_get_fifo_cnt(SPI_LL_GET_HW(host->id)) == trans->length / 8))) {
    while (trans->length && spi_ll_tx_get_fifo_cnt(SPI_LL_GET_HW(host->id)) == 0) {
        __asm__ __volatile__("nop");
        __asm__ __volatile__("nop");
        __asm__ __volatile__("nop");
    }
#endif
    //Kick off transfer
    spi_hal_user_start(hal);
}
```

Examples

1. usb/host/usb_camera_mic_spk
2. usb/host/usb_camera_lcd_display
3. usb/host/usb_audio_player

API Reference

Header File

- components/usb/usb_stream/include/usb_stream.h

Functions

esp_err_t **uvc_streaming_config**(const *uvc_config_t* *config)

Config UVC streaming with user defined parameters.For normal use, user only need to specify no-optional parameters, and set optional parameters to 0 (the driver will find the correct value from the device descriptors). For quick start mode, user should specify all parameters manually to skip get and process descriptors steps.

Parameters **config** –parameters defined in uvc_config_t

Returns esp_err_t ESP_ERR_INVALID_STATE USB streaming is running, user need to stop streaming first ESP_ERR_INVALID_ARG Invalid argument ESP_OK Success

esp_err_t **uac_streaming_config**(const *uac_config_t* *config)

Config UAC streaming with user defined parameters.For normal use, user only need to specify no-optional parameters, and set optional parameters to 0 (the driver will find the correct value from the device descriptors). For quick start mode, user should specify all parameters manually to skip get and process descriptors steps.

Parameters **config** –parameters defined in uvc_config_t

Returns esp_err_t ESP_ERR_INVALID_STATE USB streaming is running, user need to stop streaming first ESP_ERR_INVALID_ARG Invalid argument ESP_OK Success

esp_err_t **usb_streaming_start**(void)

Start usb streaming with pre-configs, usb driver will create internal tasks to handle usb data from stream pipe, and run user’s callback after new frame ready.

Returns ESP_ERR_INVALID_STATE streaming not configured, or streaming running already ESP_FAIL start failed ESP_OK start succeed

esp_err_t **usb_streaming_stop**(void)

Stop current usb streaming, internal tasks will be delete, related resource will be free.

Returns ESP_ERR_INVALID_STATE streaming not started ESP_ERR_TIMEOUT stop wait timeout ESP_OK stop succeed

esp_err_t **usb_streaming_connect_wait**(size_t timeout_ms)

Wait for USB device connection.

Parameters **timeout_ms** –timeout in ms

Returns esp_err_t ESP_ERR_INVALID_STATE: usb streaming not started ESP_ERR_TIMEOUT: timeout ESP_OK: device connected

esp_err_t **usb_streaming_state_register**(*state_callback_t* cb, void *user_ptr)

This function registers a callback for USB streaming, please note that only one callback can be registered, the later registered callback will overwrite the previous one.

Parameters

- **cb** –A pointer to a function that will be called when the USB streaming state changes.
- **user_ptr** –user_ptr is a void pointer.

Returns esp_err_t

- ESP_OK Success
- ESP_ERR_INVALID_STATE USB streaming is running, callback need register before start

esp_err_t **usb_streaming_control**(*usb_stream_t* stream, *stream_ctrl_t* ctrl_type, void *ctrl_value)

Control USB streaming with specific stream and control type.

Parameters

- **stream** –stream type defined in usb_stream_t
- **ctrl_type** –control type defined in stream_ctrl_t

- **ctrl_value** –control value

Returns ESP_ERR_INVALID_ARG invalid arg ESP_ERR_INVALID_STATE driver not configured or not started ESP_ERR_NOT_SUPPORTED current device not support this control type ESP_FAIL control failed ESP_OK succeed

esp_err_t **uac_spk_streaming_write** (void *data, size_t data_bytes, size_t timeout_ms)

Write data to the speaker buffer, will be send out when USB device is ready.

Parameters

- **data** –The data to be written.
- **data_bytes** –The size of the data to be written.
- **timeout_ms** –The timeout value for writing data to the buffer.

Returns ESP_ERR_INVALID_STATE spk stream not config ESP_ERR_NOT_FOUND spk interface not found ESP_ERR_TIMEOUT spk ringbuf full ESP_OK succeed

esp_err_t **uac_mic_streaming_read** (void *buf, size_t buf_size, size_t *data_bytes, size_t timeout_ms)

Read data from internal mic buffer, the actual size will be returned.

Parameters

- **buf** –pointer to the buffer to store the received data
- **buf_size** –The size of the data buffer.
- **data_bytes** –The actual size read from buffer
- **timeout_ms** –The timeout value for the read operation.

Returns ESP_ERR_INVALID_ARG parameter error ESP_ERR_INVALID_STATE mic stream not config ESP_ERR_NOT_FOUND mic interface not found ESP_TIMEOUT timeout ESP_OK succeed

esp_err_t **uac_frame_size_list_get** (*usb_stream_t* stream, *uac_frame_size_t* *frame_list, size_t *list_size, size_t *cur_index)

Get the audio frame size list of current stream, the list contains audio channel number, bit resolution and samples frequency. IF list_size equals 1 and the samples_frequence equals 0, which means the frequency can be set to any value between samples_frequence_min and samples_frequence_max.

Parameters

- **stream** –the stream type
- **frame_list** –the output frame list, NULL to only get the list size
- **list_size** –frame list size
- **cur_index** –current frame index

Returns esp_err_t

- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE USB device not active
- ESP_OK Success

esp_err_t **uac_frame_size_reset** (*usb_stream_t* stream, uint8_t ch_num, uint16_t bit_resolution, uint32_t samples_frequence)

Reset audio channel number, bit resolution and samples frequency, please reset when the streaming in suspend state. The new configs will be effective after streaming resume.

Parameters

- **stream** –stream type
- **ch_num** –audio channel numbers
- **bit_resolution** –audio bit resolution
- **samples_frequence** –audio samples frequency

Returns esp_err_t

- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE USB device not active
- ESP_ERR_NOT_FOUND frequency not found
- ESP_OK Success
- ESP_FAIL Reset failed

esp_err_t **uvc_frame_size_list_get** (*uvc_frame_size_t* *frame_list, size_t *list_size, size_t *cur_index)

Get the frame size list of current connected camera.

Parameters

- **frame_list** –the frame size list, can be NULL if only want to get list size
- **list_size** –the list size
- **cur_index** –current frame index

Returns esp_err_t ESP_ERR_INVALID_ARG parameter error ESP_ERR_INVALID_STATE uvc stream not config or not active ESP_OK succeed

esp_err_t **uvc_frame_size_reset** (uint16_t frame_width, uint16_t frame_height, uint32_t frame_interval)

Reset the expected frame size and frame interval, please reset when uvc streaming in suspend state.The new configs will be effective after streaming resume.

Note: frame_width and frame_height can be set to 0 at the same time, which means no change on frame size.

Parameters

- **frame_width** –frame width, FRAME_RESOLUTION_ANY means any width
- **frame_height** –frame height, FRAME_RESOLUTION_ANY means any height
- **frame_interval** –frame interval, 0 means no change

Returns esp_err_t

Structures

struct **uvc_config**

UVC configurations, for params with (optional) label, users do not need to specify manually, unless there is a problem with descriptors, or users want to skip the get and process descriptors steps.

Public Members

uint16_t **frame_width**

Picture width, set FRAME_RESOLUTION_ANY for any resolution

uint16_t **frame_height**

Picture height, set FRAME_RESOLUTION_ANY for any resolution

uint32_t **frame_interval**

Frame interval in 100-ns units, 666666 ~ 15 Fps

uint32_t **xfer_buffer_size**

Transfer buffer size, using double buffer here, must larger than one frame size

uint8_t ***xfer_buffer_a**

Buffer a for usb payload

uint8_t ***xfer_buffer_b**

Buffer b for usb payload

uint32_t **frame_buffer_size**

Frame buffer size, must larger than one frame size

uint8_t ***frame_buffer**

Buffer for one frame

uvc_frame_callback_t **frame_cb**

callback function to handle incoming frame

void ***frame_cb_arg**

callback function arg

uvc_format_t **format**

(optional) UVC stream format, default using MJPEG Optional configs, Users need to specify parameters manually when they want to skip the get and process descriptors steps (used to speed up startup)

uvc_xfer_t **xfer_type**

(optional) UVC stream transfer type, UVC_XFER_ISOC or UVC_XFER_BULK

uint8_t **format_index**

(optional) Format index

uint8_t **frame_index**

(optional) Frame index, to choose resolution

uint16_t **interface**

(optional) UVC stream interface number

uint16_t **interface_alt**

(optional) UVC stream alternate interface, to choose MPS (Max Packet Size), bulk fix to 0

uint8_t **ep_addr**

(optional) endpoint address of selected alternate interface

uint32_t **ep_mps**

(optional) MPS of selected interface_alt

uint32_t **flags**

(optional) flags to control the driver behavers

struct **mic_frame_t**

mic frame type

Public Members

void ***data**

mic data

uint32_t **data_bytes**

mic data size

uint16_t **bit_resolution**

bit resolution in buffer

uint32_t **samples_frequence**

mic sample frequency

struct **uvc_frame_size_t**

uvc frame type

Public Members

uint16_t **width**

frame width

uint16_t **height**

frame height

uint32_t **interval**

frame interval

uint32_t **interval_min**

frame min interval

uint32_t **interval_max**

frame max interval

uint32_t **interval_step**

frame interval step

struct **uac_frame_size_t**

uac frame type

Public Members

uint8_t **ch_num**

channel numbers

uint16_t **bit_resolution**

bit resolution in buffer

uint32_t **samples_frequence**

sample frequency

uint32_t **samples_frequence_min**

min sample frequency

uint32_t **samples_frequence_max**

max sample frequency

struct **uac_config_t**

UAC configurations, for params with (optional) label, users do not need to specify manually, unless there is a problem with descriptor parse, or a problem with the device descriptor.

Public Members

uint8_t **spk_ch_num**

speaker channel numbers, UAC_CH_ANY for any channel number

uint8_t **mic_ch_num**

microphone channel numbers, UAC_CH_ANY for any channel number

uint16_t **mic_bit_resolution**

microphone resolution(bits), UAC_BITS_ANY for any bit resolution

uint32_t **mic_samples_frequence**

microphone frequency(Hz), UAC_FREQUENCY_ANY for any frequency

uint16_t **spk_bit_resolution**

speaker resolution(bits), UAC_BITS_ANY for any

uint32_t **spk_samples_frequence**

speaker frequency(Hz), UAC_FREQUENCY_ANY for any frequency

uint32_t **spk_buf_size**

size of speaker send buffer, should be a multiple of spk_ep_mps

uint32_t **mic_buf_size**

mic receive buffer size, 0 if not use

mic_callback_t **mic_cb**

mic callback, can not block in here!, NULL if not use

void ***mic_cb_arg**

mic callback args, NULL if not use Optional configs, Users need to specify parameters manually when they want to skip the get and process descriptors steps (used to speed up startup)

uint16_t **mic_interface**

(optional) microphone stream interface number, set 0 if not use

uint8_t **mic_ep_addr**

(optional) microphone interface endpoint address

uint32_t **mic_ep_mps**

(optional) microphone interface endpoint mps

uint16_t **spk_interface**

(optional) speaker stream interface number, set 0 if not use

uint8_t **spk_ep_addr**

(optional) speaker interface endpoint address

uint32_t **spk_ep_mps**

(optional) speaker interface endpoint mps

uint16_t **ac_interface**

(optional) audio control interface number, set 0 if not use

uint8_t **mic_fu_id**

(optional) microphone feature unit id, set 0 if not use

uint8_t **spk_fu_id**

(optional) speaker feature unit id, set 0 if not use

uint32_t **flags**

(optional) flags to control the driver behavers

Macros

FRAME_RESOLUTION_ANY

any uvc frame resolution

UAC_FREQUENCY_ANY

any uac sample frequency

UAC_BITS_ANY

any uac bit resolution

UAC_CH_ANY

any uac channel number

FPS2INTERVAL(fps)

convert fps to uvc frame interval

FRAME_INTERVAL_FPS_5

5 fps

FRAME_INTERVAL_FPS_10

10 fps

FRAME_INTERVAL_FPS_15

15 fps

FRAME_INTERVAL_FPS_20

20 fps

FRAME_INTERVAL_FPS_30

25 fps

FLAG_UVC_SUSPEND_AFTER_START

suspend uvc after usb_streaming_start

FLAG_UAC_SPK_SUSPEND_AFTER_START

suspend uac speaker after usb_streaming_start

FLAG_UAC_MIC_SUSPEND_AFTER_START

suspend uac microphone after usb_streaming_start

Type Definitions

typedef struct *uvc_config* **uvc_config_t**

UVC configurations, for params with (optional) label, users do not need to specify manually, unless there is a problem with descriptors, or users want to skip the get and process descriptors steps.

typedef void (***mic_callback_t**)(*mic_frame_t* *frame, void *user_ptr)

user callback function to handle incoming mic frames

typedef void (***state_callback_t**)(*usb_stream_state_t* state, void *user_ptr)

user callback function to handle usb device connection status

Enumerations

enum **uvc_xfer_t**

UVC stream usb transfer type, most camera using isochronous mode, bulk mode can also be support for higher bandwidth.

Values:

enumerator **UVC_XFER_ISOC**

Isochronous Transfer Mode

enumerator **UVC_XFER_BULK**

Bulk Transfer Mode

enumerator **UVC_XFER_UNKNOWN**

Unknown Mode

enum **usb_stream_t**

UVC stream format type, default using MJPEG format,.

Stream id, used for control

Values:

enumerator **STREAM_UVC**

usb video stream

enumerator **STREAM_UAC_SPK**

usb audio speaker stream

enumerator **STREAM_UAC_MIC**

usb audio microphone stream

enumerator **STREAM_MAX**

max stream id

enum **usb_stream_state_t**

USB device connection status.

Values:

enumerator **STREAM_CONNECTED**

enumerator **STREAM_DISCONNECTED**

enum **stream_ctrl_t**

Stream control type, which also depends on if device support.

Values:

enumerator **CTRL_NONE**

None

enumerator **CTRL_SUSPEND**

streaming suspend control. ctrl_data NULL

enumerator **CTRL_RESUME**

streaming resume control. ctrl_data NULL

enumerator **CTRL_UAC_MUTE**

mute control. ctrl_data (false/true)

enumerator **CTRL_UAC_VOLUME**

volume control. ctrl_data (0~100)

enumerator **CTRL_MAX**

max type value

5.2.2 ESP MSC OTA

esp_msc_ota` is an OTA (Over-The-Air) driver based on USB MSC (USB Mass Storage Class). It supports reading programs from a USB flash drive and burning them into a designated OTA partition, thereby enabling OTA upgrades via USB.

Features:

1. Supports OTA updates by retrieving programs from a USB flash drive via USB interface.
2. Supports hot-plugging of the USB flash drive.

User Guide

Hardware requirements:

- Any ESP32-S2/ESP32-S3/ESP32-P4 development board with a USB interface capable of providing external power.
- A USB flash drive using the BOT (Bulk-Only Transport) protocol and Transparent SCSI command set.

Partition Table:

- Includes an OTA partition.

Code examples

1. Call *esp_msc_host_install* to initialize the MSC host driver.

```
esp_msc_host_config_t msc_host_config = {
    .base_path = "/usb",
    .host_driver_config = DEFAULT_MSC_HOST_DRIVER_CONFIG(),
    .vfs_fat_mount_config = DEFAULT_ESP_VFS_FAT_MOUNT_CONFIG(),
    .host_config = DEFAULT_USB_HOST_CONFIG()
};
esp_msc_host_handle_t host_handle = NULL;
esp_msc_host_install(&msc_host_config, &host_handle);
```

2. Call *esp_msc_ota* to complete OTA updates. Use `ota_bin_path` to specify the OTA file path and `wait_msc_connect` to specify the waiting time for USB drive insertion in FreeRTOS ticks.

```
esp_msc_ota_config_t config = {
    .ota_bin_path = "/usb/ota_test.bin",
    .wait_msc_connect = pdMS_TO_TICKS(5000),
};
esp_msc_ota(&config);
```

3. Call *esp_event_handler_register* to register the event handler for obtaining OTA process details.

```
esp_event_loop_create_default();
esp_event_handler_register(ESP_MSC_OTA_EVENT, ESP_EVENT_ANY_ID, &event_handler,␣
↪NULL);
```

API Reference

Header File

- components/usb/esp_msc_ota/include/esp_msc_ota.h

Functions

esp_err_t **`esp_msc_ota_begin`**(*esp_msc_ota_config_t* *config, *esp_msc_ota_handle_t* *handle)

Start MSC OTA Firmware upgrade.

If this function succeeds, then call `esp_msc_ota_perform`to continue with the OTA process otherwise call `esp_msc_ota_end`.

Parameters

- **`config`** –**[in]** pointer to *esp_msc_ota_config_t* structure
- **`handle`** –**[out]** pointer to an allocated data of type `esp_msc_ota_handle_t` which will be initialised in this function

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG: Invalid argument (missing/incorrect config, handle, etc.)

- ESP_ERR_NO_MEM: Failed to allocate memory for msc_ota handle
- ESP_FAIL: For generic failure.

esp_err_t **esp_msc_ota_perform** (*esp_msc_ota_handle_t* handle)

Read data from the firmware on the USB flash drive and start the upgrade,.

It is necessary to call this function several times and ensure that the value returned each time is ESP_OK. and call `esp_msc_ota_is_complete_data_received` to monitor whether the firmware upgrade is complete or not. Make sure that the VFS file system is not unmounted during the `fread` process. If you manually unplug the USB flash drive or log out of the USB HOST, stop calling `esp_msc_ota_perform` before and call `esp_msc_ota_abort` afterwards.

Parameters **handle** –**[in]** Handle for the MSC ota

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_INVALID_STATE: Invalid state (handle not initialized, etc.)
- ESP_ERR_INVALID_SIZE: Fread failed
- ESP_FAIL: For generic failure.
- For other errors, please check the API for the specific error.

esp_err_t **esp_msc_ota_end** (*esp_msc_ota_handle_t* handle)

Clean-up MSC OTA Firmware upgrade.

Note: If this API returns successfully, esp_restart() must be called to boot from the new firmware image esp_https_ota_finish should not be called after calling esp_msc_ota_abort

Parameters **handle** –**[in]** Handle for the MSC ota

Returns

- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_INVALID_STATE: Incorrect status
- ESP_OK: Success
- For other errors, please check the API for the specific error.

esp_err_t **esp_msc_ota_abort** (*esp_msc_ota_handle_t* handle)

Clean-up MSC OTA Firmware upgrade and call `esp_ota_abort`

Note: esp_msc_ota_abort should not be called after calling esp_msc_ota_finish

Parameters **handle** –**[in]** Handle for the MSC ota

Returns

- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_INVALID_STATE: Incorrect status
- ESP_OK: Success
- For other errors, please check the API for the specific error.

esp_err_t **esp_msc_ota** (*esp_msc_ota_config_t* *config)

MSC OTA Firmware upgrade.

This function provides a complete set of MSC_OTA upgrade procedures. When the USB flash disk is inserted, it will be upgraded automatically. After the upgrade is completed, please call `esp_restart()`

Parameters **config** –**[in]** pointer to *esp_msc_ota_config_t* structure

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_OK: Success
- For other errors, please check the API for the specific error.

esp_err_t **esp_msc_ota_get_img_desc**(*esp_msc_ota_handle_t* 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.

Parameters
- **handle** –**[in]** pointer to *esp_msc_ota_config_t* structure
- **new_app_info** –**[out]** pointer to an allocated esp_app_desc_t structure

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_INVALID_STATE: Incorrect status
- ESP_FAIL: Fail to read image header

esp_err_t **esp_msc_ota_set_msc_connect_state**(*esp_msc_ota_handle_t* handle, bool if_connect)

When you manage the MSC HOST function, call this api to notify msc_ota that the u-disk has been inserted and the VFS filesystem has been mounted.

Parameters
- **handle** –**[in]** Handle for the MSC ota
- **if_connect** –**[in]**

Returns alaways return ESP_OK

esp_msc_ota_status_t **esp_msc_ota_get_status**(*esp_msc_ota_handle_t* handle)

Get the status of the MSC ota.

Parameters **handle** –**[in]** Handle for the MSC ota
Returns esp_msc_ota_status_t

bool **esp_msc_ota_is_complete_data_received**(*esp_msc_ota_handle_t* handle)

Checks if complete data was received or not.

This API can be called just before esp_msc_ota_end() to validate if the complete image was indeed received.

Parameters **handle** –**[in]** Handle for the MSC ota
Returns true
Returns false

Structures

struct **esp_msc_ota_config_t**

esp msc ota config

Public Members

const char ***ota_bin_path**

OTA binary name, must be an exact match. Note: By default file names cannot exceed 11 bytes e.g. "/usb/ota.bin"

bool **bulk_flash_erase**

Erase entire flash partition during initialization. By default flash partition is erased during write operation and in chunk of 4K sector size

bool **skip_msc_connect_wait**

Skip waiting for MSC device to connect, if true means MSC device is already connected

TickType_t **wait_msc_connect**

Wait time for MSC device to connect

size_t **buffer_size**

Buffer size for OTA write operation, must larger than 1024

Type Definitions

typedef void ***esp_msc_ota_handle_t**

Handle for the MSC ota.

Enumerations

enum **esp_msc_ota_event_t**

Declare Event Base for ESP MSC OTA.

Values:

enumerator **ESP_MSC_OTA_START**

Start update

enumerator **ESP_MSC_OTA_READY_UPDATE**

Ready to update

enumerator **ESP_MSC_OTA_WRITE_FLASH**

Flash write operation

enumerator **ESP_MSC_OTA_FAILED**

Update failed

enumerator **ESP_MSC_OTA_GET_IMG_DESC**

Get image description

enumerator **ESP_MSC_OTA_VERIFY_CHIP_ID**

Verify chip id

enumerator **ESP_MSC_OTA_UPDATE_BOOT_PARTITION**

Boot partition update after successful ota update

enumerator **ESP_MSC_OTA_FINISH**

OTA finished

enumerator **ESP_MSC_OTA_ABORT**

OTA aborted

enum **esp_msc_ota_status_t**

Values:

enumerator **ESP_MSC_OTA_INIT**

enumerator **ESP_MSC_OTA_BEGIN**

enumerator **ESP_MSC_OTA_IN_PROGRESS**

enumerator **ESP_MSC_OTA_SUCCESS**

5.2.3 USB Host CDC

The `iot_usbh_cdc` component implements a simplified USB host CDC driver. With simple configuration, users can use USB CDC devices. The component supports hot-plug and optionally enables an internal ring buffer to simplify data sending and receiving.

Usage Guide

1. Use `usbh_cdc_driver_install` to configure the USB CDC driver. By setting the `skip_init_usb_host_driver` option, the component can initialize the USB Host Driver stack internally. When other USB drivers also need to use the USB Host Driver stack, set this option to true to avoid repeated initialization.

```
/* Install the USB CDC driver and initialize the USB Host Driver stack inside the␣
↪driver */
usbh_cdc_driver_config_t config = {
    .task_stack_size = 1024 * 4,
    .task_priority = 5,
    .task_coreid = 0,
    .skip_init_usb_host_driver = false,
};
usbh_cdc_driver_install(&config);
```

2. Use `usbh_cdc_register_dev_event_cb` to register a CDC device event, and configure the device match list `dev_match_id_list`. When a new USB device is connected, the driver will match it against this list. If any item matches, the driver will invoke the event callback. In this callback, the user can call `usbh_cdc_port_open` to open one or more ports of the device for communication as needed by the application.

```
static void device_event_cb(usbh_cdc_device_event_t event, usbh_cdc_device_event_
↪data_t *event_data, void *user_ctx)
{
    switch (event) {
    case CDC_HOST_DEVICE_EVENT_CONNECTED: {
        ESP_LOGI(TAG, "Device connected: dev_addr=%d, matched_intf_num=%d",
                 event_data->new_dev.dev_addr, event_data->new_dev.matched_intf_
↪num);

        usbh_cdc_port_handle_t cdc_port = NULL;
        usbh_cdc_port_config_t cdc_port_config = {
            .dev_addr = event_data->new_dev.dev_addr,
            .itf_num = 0,
            .in_transfer_buffer_size = 512,
            .out_transfer_buffer_size = 512,
            .cbs = {
                .notif_cb = NULL,
                .recv_data = NULL,
                .closed = NULL,
                .user_data = NULL,
            },
        };
        usbh_cdc_port_open(&cdc_port_config, &cdc_port);
```

(continues on next page)

(continued from previous page)

```
        } break;

    case CDC_HOST_DEVICE_EVENT_DISCONNECTED:
        ESP_LOGI(TAG, "Device disconnected: dev_addr=%d, dev_hdl=%p",
                 event_data->dev_gone.dev_addr, event_data->dev_gone.dev_hdl);
        break;

    default:
        break;
    }
}

const static usb_device_match_id_t match_id_list[] = {
    {
        .match_flags = USB_DEVICE_ID_MATCH_VENDOR | USB_DEVICE_ID_MATCH_PRODUCT,
        .idVendor = USB_DEVICE_VENDOR_ANY,
        .idProduct = USB_DEVICE_PRODUCT_ANY,
    },
    { 0 }
};
usbh_cdc_register_dev_event_cb(match_id_list, device_event_cb, NULL);
```

The parameter `dev_match_id_list` pointer requires an array that is not on the stack and must be null-terminated. An array entry can specify conditions to match one or more devices. For example, the above example uses `USB_DEVICE_VENDOR_ANY` and `USB_DEVICE_PRODUCT_ANY` to match all USB devices. Users can also specify specific `idVendor` and `idProduct` to match specific devices. You can further specify conditions using `match_flags`, such as `USB_DEVICE_ID_MATCH_DEV_CLASS`, `USB_DEVICE_ID_MATCH_DEV_SUBCLASS`, `USB_DEVICE_ID_MATCH_DEV_PROTOCOL`, etc. For more matching conditions, see *usb_dev_match_flags_t*.

Do not call communication functions such as `usbh_cdc_write_bytes` in the `device_event_cb` callback, because this callback runs in the USB task context and calling these functions can cause a deadlock. It is recommended to create a new task in the `device_event_cb` callback or send an event to the application task, and perform device communication in the new task or the application task.

3. After a port is opened, the host will automatically receive USB data from the CDC device into the USB buffer. The user can poll `usbh_cdc_get_rx_buffer_size` to read the buffered data length, or register a receive callback to be notified when data is ready. Then `usbh_cdc_read_bytes` can be used to read the buffered data. When `in_ringbuf_size` is greater than 0, the driver enables an internal ring buffer to cache received data.
4. `usbh_cdc_write_bytes` can be used to send data to the USB device. When `out_ringbuf_size` is greater than 0, data is first written to the internal transmission `ringbuffer` instead of directly to the USB buffer.
5. `usbh_cdc_port_close` can delete the USB CDC device and release its resources.
6. `usbh_cdc_driver_uninstall` can fully uninstall the USB driver to release all resources; if a device is still connected, the driver cannot be uninstalled directly.

Note: Since USB device hot-plug is asynchronous, a USB device may be unplugged at any time during runtime. Users must ensure that when calling functions such as `usbh_cdc_write_bytes` or `usbh_cdc_read_bytes`, the device is still connected and the port is open; otherwise these functions will return an error.

Supported CDC Device Descriptors

The driver supports the following standard CDC device descriptors:

```
        ------------------- IAD Descriptor --------------------
bLength                 : 0x08 (8 bytes)
```

(continues on next page)

(continued from previous page)

```
bDescriptorType          : 0x0B (Interface Association Descriptor)
bFirstInterface          : 0x00 (Interface 0)
bInterfaceCount          : 0x02 (2 Interfaces)
bFunctionClass           : 0x02 (Communications and CDC Control)
bFunctionSubClass        : 0x06
bFunctionProtocol        : 0x00
iFunction                : 0x07 (String Descriptor 7)
Language 0x0409         : ""

        ---------------- Interface Descriptor -----------------
bLength                  : 0x09 (9 bytes)
bDescriptorType          : 0x04 (Interface Descriptor)
bInterfaceNumber         : 0x00 (Interface 0)
bAlternateSetting        : 0x00
bNumEndpoints            : 0x01 (1 Endpoint)
bInterfaceClass          : 0x02 (Communications and CDC Control)
bInterfaceSubClass       : 0x06 (Ethernet Networking Control Model)
bInterfaceProtocol       : 0x00 (No class specific protocol required)
iInterface               : 0x07 (String Descriptor 7)
Language 0x0409         : ""

        ----------------- Endpoint Descriptor -----------------
bLength                  : 0x07 (7 bytes)
bDescriptorType          : 0x05 (Endpoint Descriptor)
bEndpointAddress         : 0x87 (Direction=IN EndpointID=7)
bmAttributes             : 0x03 (TransferType=Interrupt)
wMaxPacketSize           : 0x0020
Bits 15..13             : 0x00 (reserved, must be zero)
Bits 12..11             : 0x00 (0 additional transactions per microframe -> allows␣
↪1..1024 bytes per packet)
Bits 10..0              : 0x20 (32 bytes per packet)
bInterval                : 0x09 (256 microframes -> 32 ms)

        ---------------- Interface Descriptor -----------------
bLength                  : 0x09 (9 bytes)
bDescriptorType          : 0x04 (Interface Descriptor)
bInterfaceNumber         : 0x01 (Interface 1)
bAlternateSetting        : 0x00
bNumEndpoints            : 0x00 (Default Control Pipe only)
bInterfaceClass          : 0x0A (CDC-Data)
bInterfaceSubClass       : 0x00
bInterfaceProtocol       : 0x00
iInterface               : 0x00 (No String Descriptor)

        ---------------- Interface Descriptor -----------------
bLength                  : 0x09 (9 bytes)
bDescriptorType          : 0x04 (Interface Descriptor)
bInterfaceNumber         : 0x01 (Interface 1)
bAlternateSetting        : 0x01
bNumEndpoints            : 0x02 (2 Endpoints)
bInterfaceClass          : 0x0A (CDC-Data)
bInterfaceSubClass       : 0x00
bInterfaceProtocol       : 0x00
iInterface               : 0x00 (No String Descriptor)

        ----------------- Endpoint Descriptor -----------------
bLength                  : 0x07 (7 bytes)
bDescriptorType          : 0x05 (Endpoint Descriptor)
bEndpointAddress         : 0x0C (Direction=OUT EndpointID=12)
bmAttributes             : 0x02 (TransferType=Bulk)
wMaxPacketSize           : 0x0200 (max 512 bytes)
```

(continues on next page)

(continued from previous page)

```
bInterval                : 0x00 (never NAKs)

        ----------------- Endpoint Descriptor -----------------
bLength                  : 0x07 (7 bytes)
bDescriptorType          : 0x05 (Endpoint Descriptor)
bEndpointAddress         : 0x83 (Direction=IN EndpointID=3)
bmAttributes             : 0x02 (TransferType=Bulk)
wMaxPacketSize           : 0x0200 (max 512 bytes)
bInterval                : 0x00 (never NAKs)
```

The driver also supports some vendor-specific CDC device descriptors, for example:

```
        ---------------- Interface Descriptor -----------------
bLength                  : 0x09 (9 bytes)
bDescriptorType          : 0x04 (Interface Descriptor)
bInterfaceNumber         : 0x03 (Interface 3)
bAlternateSetting        : 0x00
bNumEndpoints            : 0x03 (3 Endpoints)
bInterfaceClass          : 0xFF (Vendor Specific)
bInterfaceSubClass       : 0x00
bInterfaceProtocol       : 0x00
iInterface               : 0x09 (String Descriptor 9)
Language 0x0409         : "Mobile AT Interface"

        ----------------- Endpoint Descriptor -----------------
bLength                  : 0x07 (7 bytes)
bDescriptorType          : 0x05 (Endpoint Descriptor)
bEndpointAddress         : 0x85 (Direction=IN EndpointID=5)
bmAttributes             : 0x03 (TransferType=Interrupt)
wMaxPacketSize           : 0x0010
Bits 15..13             : 0x00 (reserved, must be zero)
Bits 12..11             : 0x00 (0 additional transactions per microframe -> allows␣
↪1..1024 bytes per packet)
Bits 10..0              : 0x10 (16 bytes per packet)
bInterval                : 0x09 (256 microframes -> 32 ms)

        ----------------- Endpoint Descriptor -----------------
bLength                  : 0x07 (7 bytes)
bDescriptorType          : 0x05 (Endpoint Descriptor)
bEndpointAddress         : 0x82 (Direction=IN EndpointID=2)
bmAttributes             : 0x02 (TransferType=Bulk)
wMaxPacketSize           : 0x0200 (max 512 bytes)
bInterval                : 0x00 (never NAKs)

        ----------------- Endpoint Descriptor -----------------
bLength                  : 0x07 (7 bytes)
bDescriptorType          : 0x05 (Endpoint Descriptor)
bEndpointAddress         : 0x0B (Direction=OUT EndpointID=11)
bmAttributes             : 0x02 (TransferType=Bulk)
wMaxPacketSize           : 0x0200 (max 512 bytes)
bInterval                : 0x00 (never NAKs)
```

Note: When connecting multiple devices using a USB HUB, if the HOST' s channels are insufficient, you can try enabling the `USBH_CDC_FLAGS_DISABLE_NOTIFICATION` flag in `usbh_cdc_port_config_t` to forcibly disable interrupt endpoint and reduce the use of one channel. This will not work if the device itself does not have interrupt endpoint.

Example Code

usb/host/usb_cdc_basic

API Reference

Header File

- components/usb/iot_usbh_cdc/include/iot_usbh_cdc.h

Functions

esp_err_t **usbh_cdc_driver_install** (const *usbh_cdc_driver_config_t* *config)

Install the USB CDC driver.

This function installs and initializes the USB CDC driver. It sets up internal data structures, creates necessary tasks, and registers the USB host client. If the USB host driver is not already initialized, it will create and initialize it.

The function performs the following steps:

a. Verifies input arguments and state.
b. Optionally initializes the USB host driver.
c. Allocates memory for the USB CDC object.
d. Creates necessary synchronization primitives (event group, mutex).
e. Creates the driver task.
f. Registers the USB host client with the provided configuration.
g. Initializes the CDC driver structure and resumes the driver task.

In case of an error, the function will clean up any resources that were allocated before the failure occurred.

Note: If the `skip_init_usb_host_driver` flag in the config is set to true, the function will skip the USB host driver initialization.

Parameters **config** –**[in]** Pointer to a configuration structure (*usbh_cdc_driver_config_t*)

Returns

- ESP_OK: Driver installed successfully
- ESP_ERR_INVALID_ARG: Null configuration parameter or invalid arguments
- ESP_ERR_INVALID_STATE: USB CDC driver already installed
- ESP_ERR_NO_MEM: Memory allocation failed
- ESP_FAIL: Failed to create necessary tasks or USB host installation failed

esp_err_t **usbh_cdc_driver_uninstall** (void)

Uninstall the USB CDC driver.

This function uninstalls the USB CDC driver, releasing any allocated resources and stopping all driver-related tasks. It ensures that all CDC devices are closed before proceeding with the uninstallation.

Returns

- ESP_OK: Driver uninstalled successfully
- ESP_ERR_INVALID_STATE: Driver is not installed or devices are still active
- ESP_ERR_NOT_FINISHED: Timeout occurred while waiting for the CDC task to finish

esp_err_t **usbh_cdc_register_dev_event_cb** (const *usb_device_match_id_t* *dev_match_id_list, *usbh_cdc_device_event_callback_t* event_cb, void *user_data)

Register an additional callback function for new device detection.

This function registers a callback that will be invoked when a new USB CDC device is connected. The callback will be automatically unregistered when usbh_cdc_driver_uninstall() is called.

Parameters
- **dev_match_id_list** –**[in]** List of device match IDs to filter devices that the callback will be called for The list should be terminated with a NULL entry.
- **event_cb** –**[in]** Callback function to be called when a new device is connected
- **user_data** –**[in]** User data pointer that will be passed to the callback function

Returns
- ESP_OK: Callback registered successfully
- ESP_ERR_INVALID_ARG: Invalid CDC handle or callback function
- ESP_ERR_NO_MEM: Failed to allocate memory for callback registration

esp_err_t **usbh_cdc_unregister_dev_event_cb**(*usbh_cdc_device_event_callback_t* event_cb)

Unregister the new device callback which same as the one registered by usbh_cdc_register_dev_event_cb.

Parameters **event_cb** –**[in]** The new device callback function to unregister

Returns
- ESP_OK: Callback unregistered successfully
- ESP_ERR_INVALID_ARG: Invalid callback function

esp_err_t **usbh_cdc_port_open**(const *usbh_cdc_port_config_t* *port_config, *usbh_cdc_port_handle_t* *cdc_port_out)

Open a USB CDC port on a USB CDC device.

- This function opens a USB CDC port on the specified USB CDC device, allowing communication through the port.

Parameters
- **port_config** –**[in]** Pointer to a `usb_cdc_port_config_t` structure
- **cdc_port_out** –**[out]** Pointer to a variable that will hold the opened CDC port handle upon successful opening

Returns
- ESP_OK: Port opened successfully
- ESP_ERR_INVALID_ARG: Invalid arguments (NULL handle or port_config)
- ESP_ERR_INVALID_STATE: Device is not connected

esp_err_t **usbh_cdc_port_close**(*usbh_cdc_port_handle_t* cdc_port_handle)

Close a USB CDC port.

Parameters **cdc_port_handle** –**[in]** The CDC port handle to close

Returns
- ESP_OK: Port closed successfully
- ESP_ERR_INVALID_ARG: Invalid CDC port handle provided
- ESP_ERR_INVALID_STATE: USB CDC driver is not installed or port is not open

esp_err_t **usbh_cdc_send_custom_request**(*usbh_cdc_port_handle_t* cdc_port_handle, uint8_t bmRequestType, uint8_t bRequest, uint16_t wValue, uint16_t wIndex, uint16_t wLength, uint8_t *data)

Send a custom control request to the USB CDC port.

This function sends a custom control request to the USB CDC device with the specified parameters. It can be used for both IN and OUT transfers.

Parameters
- **cdc_port_handle** –**[in]** The CDC port handle to send the request to
- **bmRequestType** –**[in]** The request type bitmask

- **bRequest** –**[in]** The request ID
- **wValue** –**[in]** The request value
- **wIndex** –**[in]** The request index
- **wLength** –**[in]** The length of data to transfer
- **data** –**[inout]** Pointer to the data buffer:
  - For OUT transfers: data to be sent to device
  - For IN transfers: buffer to store received data

Returns

- ESP_OK: Request sent successfully
- ESP_ERR_INVALID_ARG: Invalid arguments (NULL handle/data buffer)
- ESP_ERR_INVALID_STATE: Device is not connected
- ESP_ERR_TIMEOUT: Control transfer timed out
- ESP_ERR_INVALID_RESPONSE: Control transfer failed

esp_err_t **usbh_cdc_write_bytes**(*usbh_cdc_port_handle_t* cdc_port_handle, const uint8_t *buf, size_t length, TickType_t ticks_to_wait)

Write data to the USB CDC port.

With internal ring buffer, this function writes data to the specified USB CDC device by pushing the data into the output ring buffer. If the buffer is full or the device is not connected, the write will fail.

Without internal ring buffer, this function writes data to the specified USB CDC device by sending data directly to the device. Please set the ticks_to_wait for blocking write.

Parameters

- **cdc_port_handle** –**[in]** The CDC port handle
- **buf** –**[in]** Pointer to the data buffer to write
- **length** –**[in]** Pointer to the length of data to write. On success, it remains unchanged, otherwise set to 0.
- **ticks_to_wait** –**[in]** The maximum amount of time to wait for the write operation to complete

Returns

- ESP_OK: Data written successfully
- ESP_ERR_INVALID_ARG: Invalid argument (NULL handle, buffer, or length)
- ESP_ERR_INVALID_STATE: Device is not connected

esp_err_t **usbh_cdc_read_bytes**(*usbh_cdc_port_handle_t* cdc_port_handle, uint8_t *buf, size_t *length, TickType_t ticks_to_wait)

Read data from the USB CDC port.

With internal ring buffer, this function reads data from the specified USB CDC device by popping data from the input ring buffer. If no data is available or the device is not connected, the read will fail.

Without internal ring buffer, this function reads data from the specified USB CDC device by receiving data directly from the device. You can call this function in the recv_data callback function. Please set ticks_to_wait to zero for non-blocking read.

Note: If the port is not opened with ring buffer, the length must be equal to the internal transfer buffer size, otherwise the function will return ESP_ERR_INVALID_ARG.

Parameters

- **cdc_port_handle** –**[in]** The CDC port handle
- **buf** –**[out]** Pointer to the buffer where the read data will be stored
- **length** –**[inout]** Pointer to the length of data to read. On success, it is updated with the actual bytes read.
- **ticks_to_wait** –**[in]** The maximum amount of time to wait for the read operation to complete

Returns

- ESP_OK: Data read successfully

- ESP_FAIL: Failed to read data
- ESP_ERR_INVALID_ARG: Invalid argument (NULL handle, buffer, or length)
- ESP_ERR_INVALID_STATE: Device is not connected

esp_err_t **usbh_cdc_flush_rx_buffer**(*usbh_cdc_port_handle_t* cdc_port_handle)

Flush the receive buffer of the USB CDC port.

This function clears the receive buffer, discarding any data currently in the buffer.

Not supported for no internal ring buffer mode.

Parameters **cdc_port_handle** –**[in]** The CDC port handle

Returns

- ESP_OK: Receive buffer flushed successfully
- ESP_ERR_INVALID_ARG: Invalid CDC handle provided
- ESP_ERR_NOT_SUPPORTED: Not supported for no internal ring buffer mode

esp_err_t **usbh_cdc_flush_tx_buffer**(*usbh_cdc_port_handle_t* cdc_port_handle)

Flush the transmit buffer of the USB CDC port.

This function clears the transmit buffer, discarding any data currently in the buffer.

Not supported for no internal ring buffer mode.

Parameters **cdc_port_handle** –**[in]** The CDC port handle

Returns

- ESP_OK: Transmit buffer flushed successfully
- ESP_ERR_INVALID_ARG: Invalid CDC handle provided
- ESP_ERR_NOT_SUPPORTED: Not supported for no internal ring buffer mode

esp_err_t **usbh_cdc_get_rx_buffer_size**(*usbh_cdc_port_handle_t* cdc_port_handle, size_t *size)

Get the size of the receive buffer of the USB CDC port.

This function retrieves the current size of the receive buffer in bytes.

For no internal ring buffer, you can call this function in the recv_data callback function.

Parameters

- **cdc_port_handle** –**[in]** The CDC port handle
- **size** –**[out]** Pointer to store the size of the receive buffer

Returns

- ESP_OK: Size retrieved successfully
- ESP_ERR_INVALID_ARG: Invalid CDC handle provided

esp_err_t **usbh_cdc_desc_print**(*usbh_cdc_port_handle_t* cdc_port_handle)

Print the USB CDC device descriptors.

This function retrieves and prints the USB device and configuration descriptors for the specified USB CDC device. It checks that the device is properly connected and open before retrieving and printing the descriptors.

Parameters **cdc_port_handle** –**[in]** The CDC port handle

Returns

- ESP_OK: Descriptors printed successfully
- ESP_ERR_INVALID_ARG: Invalid CDC handle provided
- ESP_ERR_INVALID_STATE: Device is not connected or not yet open

esp_err_t **usbh_cdc_get_dev_handle**(*usbh_cdc_port_handle_t* cdc_port_handle, usb_device_handle_t *dev_handle)

Get the device handle of the USB CDC device.

This function retrieves the device handle associated with the specified USB CDC device.

Parameters

- **cdc_port_handle** –**[in]** The CDC port handle
- **dev_handle** –**[out]** Pointer to store the device handle

Returns
- ESP_OK: Device handle retrieved successfully
- ESP_ERR_INVALID_ARG: Invalid CDC handle provided
- ESP_ERR_INVALID_STATE: Device is not connected or not yet open

esp_err_t **usbh_cdc_port_get_intf_desc**(*usbh_cdc_port_handle_t* port_handle, const usb_intf_desc_t **notif_intf, const usb_intf_desc_t **data_intf)

Get the parsed information of the USB CDC device.

Parameters
- **port_handle** –**[in]** The CDC port handle
- **notif_intf** –**[out]** Pointer to store the notification interface descriptor
- **data_intf** –**[out]** Pointer to store the data interface descriptor

Returns
- ESP_OK: Parsed information retrieved successfully
- ESP_ERR_INVALID_ARG: Invalid CDC handle provided
- ESP_ERR_INVALID_STATE: Device is not connected or not yet open

Unions

union **usbh_cdc_device_event_data_t**

#include <iot_usbh_cdc.h> USB CDC HOST device event data.

Public Members

uint8_t **dev_addr**

Device address

int **matched_intf_num**

Number of matched interface, -1 is invalid

const usb_device_desc_t ***device_desc**

Device descriptor

const usb_config_desc_t ***active_config_desc**

Active configuration descriptor

struct *usbh_cdc_device_event_data_t*::[anonymous] **new_dev**

New device connected event data

usb_device_handle_t **dev_hdl**

Device handle for the event

struct *usbh_cdc_device_event_data_t*::[anonymous] **dev_gone**

Device gone event data

Structures

struct **usbh_cdc_driver_config_t**

CDC driver configuration.

Public Members

size_t **task_stack_size**

Stack size of the driver' s task

unsigned **task_priority**

Priority of the driver' s task

int **task_coreid**

Core of the driver' s task, Set it to -1 to not specify the core.

bool **skip_init_usb_host_driver**

Skip initialization of USB host driver

struct **usbh_cdc_port_event_callbacks_t**

Callback structure for CDC port events.

Public Members

void (***closed**)(*usbh_cdc_port_handle_t* port_handle, void *user_data)

Callback function for when the CDC port is opened, set NULL if not use.

Param port_handle [in] Handle to the CDC port that was opened
Param user_data [in] User data pointer that was passed during callback registration

void (***recv_data**)(*usbh_cdc_port_handle_t* port_handle, void *user_data)

Callback function for when data is received on the CDC port, set NULL if not use.

Param port_handle [in] Handle to the CDC port that received data
Param user_data [in] User data pointer that was passed during callback registration

void (***notif_cb**)(*usbh_cdc_port_handle_t* port_handle, iot_cdc_notification_t *notif, void *user_data)

Callback function for when a notification is received on the CDC port, set NULL if not use.

Param port_handle [in] Handle to the CDC port that received a notification
Param notif [in] Pointer to the notification data structure containing notification details
Param user_data [in] User data pointer that was passed during callback registration

void ***user_data**

Pointer to user data that will be passed to the callbacks

struct **usbh_cdc_port_config_t**

Configuration structure for initializing a USB CDC port.

Public Members

uint8_t **dev_addr**

Device address of the CDC device

uint8_t **itf_num**

Interface number for the CDC port

size_t **in_ringbuf_size**

Size of the receive ringbuffer size, if set to 0, not use ringbuffer

size_t **out_ringbuf_size**

Size of the transmit ringbuffer size, if set to 0, not use ringbuffer

size_t **in_transfer_buffer_size**

Size of the IN transfer buffer size

size_t **out_transfer_buffer_size**

Size of the OUT transfer buffer size

usbh_cdc_port_event_callbacks_t **cbs**

Event callbacks for the CDC port

usbh_cdc_port_flags_t **flags**

Flags for the CDC port configuration

Type Definitions

typedef struct usbh_cdc_port_t ***usbh_cdc_port_handle_t**

typedef void (***usbh_cdc_device_event_callback_t**)(*usbh_cdc_device_event_t* event, *usbh_cdc_device_event_data_t* *event_data, void *user_ctx)

USB Host CDC device event callback function.

Param event [in] Event structure
Param event_data [in] Event data structure
Param user_ctx [in] User's argument passed to callback function

Enumerations

enum **usbh_cdc_device_event_t**

USB CDC HOST Driver event id.

Values:

enumerator **CDC_HOST_DEVICE_EVENT_CONNECTED**

Device connected event

enumerator **CDC_HOST_DEVICE_EVENT_DISCONNECTED**

Device disconnected event

enum **usbh_cdc_port_flags_t**

Flags for CDC port configuration.

These flags can be used to modify the behavior of the CDC port.

Values:

enumerator **USBH_CDC_FLAGS_DISABLE_NOTIFICATION**

Force disable notification functionality on the CDC port.

Header File

- components/usb/iot_usbh_cdc/include/usbh_helper.h

Functions

int **usbh_match_device** (const usb_device_desc_t *dev_desc, const *usb_device_match_id_t* *id)

Match a USB device against a set of criteria.

This function checks if the provided USB device descriptor matches the specified criteria in the match ID. It compares vendor ID, product ID, device class, subclass, protocol, and other fields based on the match flags.

Parameters
- **dev_desc** –**[in]** Pointer to the USB device descriptor to match against
- **id** –**[in]** Pointer to the USB device match ID containing the criteria for matching

Returns
- 1 if the device matches the criteria
- 0 if it does not match

int **usbh_match_interface** (const usb_intf_desc_t *intf_desc, const *usb_device_match_id_t* *id)

Match a USB interface against a set of criteria.

This function checks if the provided USB interface descriptor matches the specified criteria in the match ID. It compares interface class, subclass, protocol, and other fields based on the match flags.

Parameters
- **intf_desc** –**[in]** Pointer to the USB interface descriptor to match against
- **id** –**[in]** Pointer to the USB device match ID containing the criteria for matching

Returns
- 1 if the interface matches the criteria
- 0 if it does not match

int **usbh_match_device_from_id_list** (const usb_device_desc_t *dev_desc, const *usb_device_match_id_t* *id_list)

Match a USB device against a set of criteria.

This function checks if the provided USB device descriptor and configuration descriptor match the specified criteria in the match ID. It compares vendor ID, product ID, device class, subclass, protocol, details based on the match flags.

Parameters
- **dev_desc** –**[in]** Pointer to the USB device descriptor to match against
- **id_list** –**[in]** Pointer to list of USB device match ID containing the criteria for matching

Returns
- 1 if both device match the criteria
- 0 if they do not match

int **usbh_match_interface_in_configuration** (const usb_config_desc_t *config_desc, const *usb_device_match_id_t* *id, int *matched_intf_num)

Match a USB interface in a configuration against a set of criteria.

This function checks if any interface in the provided USB configuration descriptor matches the specified criteria in the match ID. It iterates through all interfaces in the configuration and compares their details based on the match flags.

Parameters
- **config_desc** –**[in]** Pointer to the USB configuration descriptor to search for matching interfaces
- **id** –**[in]** Pointer to the USB device match ID containing the criteria for matching
- **matched_intf_num** –**[out]** Pointer to an integer to store the number of matched interfaces, set NULL if not needed

Returns
- 1 if any interface matches the criteria

- 0 if no interface matches

int **usbh_match_id_from_list** (const usb_device_desc_t *dev_desc, const usb_config_desc_t *config_desc, const *usb_device_match_id_t* *id_list, int *matched_intf_num)

Match a USB device and interface against a set of criteria.

This function checks if the provided USB device descriptor and configuration descriptor match the specified criteria in the match ID. It compares vendor ID, product ID, device class, subclass, protocol, and interface details based on the match flags.

Parameters

- **dev_desc** –**[in]** Pointer to the USB device descriptor to match against
- **config_desc** –**[in]** Pointer to the USB configuration descriptor to match against
- **id_list** –**[in]** Pointer to list of USB device match ID containing the criteria for matching
- **matched_intf_num** –**[out]** Pointer to an integer to store the number of matched interfaces, set NULL if not needed

Returns

- 1 if both device and interface match the criteria
- 0 if they do not match

Structures

struct **usb_device_match_id_t**

Match a USB device against a set of criteria.

Public Members

usb_dev_match_flags_t **match_flags**

which fields to match against?

uint16_t **idVendor**

Vendor ID

uint16_t **idProduct**

Product ID

uint16_t **bcdDevice**

Device release number in binary-coded decimal (BCD)

uint8_t **bDeviceClass**

Device class

uint8_t **bDeviceSubClass**

Device subclass

uint8_t **bDeviceProtocol**

Device protocol

uint8_t **bInterfaceClass**

Interface class

uint8_t **bInterfaceSubClass**

Interface subclass

uint8_t **bInterfaceProtocol**

Interface protocol

uint8_t **bInterfaceNumber**

Interface number

Macros

USB_DEVICE_VENDOR_ANY

USB_DEVICE_PRODUCT_ANY

ESP_USB_DEVICE_MATCH_ID_ANY

Enumerations

enum **usb_dev_match_flags_t**

Match a USB device against a set of fields.

Values:

enumerator **USB_DEVICE_ID_MATCH_VENDOR**

enumerator **USB_DEVICE_ID_MATCH_PRODUCT**

enumerator **USB_DEVICE_ID_MATCH_DEV_BCD**

enumerator **USB_DEVICE_ID_MATCH_DEV_CLASS**

enumerator **USB_DEVICE_ID_MATCH_DEV_SUBCLASS**

enumerator **USB_DEVICE_ID_MATCH_DEV_PROTOCOL**

enumerator **USB_DEVICE_ID_MATCH_INT_CLASS**

enumerator **USB_DEVICE_ID_MATCH_INT_SUBCLASS**

enumerator **USB_DEVICE_ID_MATCH_INT_PROTOCOL**

enumerator **USB_DEVICE_ID_MATCH_INT_NUMBER**

enumerator **USB_DEVICE_ID_MATCH_ALL**

enumerator **USB_DEVICE_ID_MATCH_DEV**

Match device descriptor

enumerator **USB_DEVICE_ID_MATCH_INTF**

Match interface descriptor

enumerator **USB_DEVICE_ID_MATCH_VID_PID**

Match vendor and product ID

5.2.4 USB RNDIS Host Driver

iot_usbh_rndis is a host driver based on the RNDIS protocol over the USB interface.

RNDIS Protocol

RNDIS (Remote NDIS) achieves network communication by encapsulating TCP/IP in USB packets.

USB Descriptors

By default, an RNDIS device is the only function in a USB configuration. For composite devices, RNDIS expects to be the first function in the USB configuration, i.e., the interface with `bInterfaceNumber` equal to 0.

An RNDIS interface typically consists of one IAD descriptor and two interface descriptors. The current driver supports the following two types of IAD descriptors:

1. `bFunctionClass` = 0xe0, `bFunctionSubClass` = 0x01, `bFunctionProtocol` = 0x03
2. `bFunctionClass` = 0xef, `bFunctionSubClass` = 0x04, `bFunctionProtocol` = 0x01

Example:

```
*** Interface Association Descriptor ***
bLength 8
bDescriptorType 11
bFirstInterface 0
bInterfaceCount 2
bFunctionClass 0xe0
bFunctionSubClass 0x1
bFunctionProtocol 0x3
iFunction 5
    *** Interface descriptor ***
    bLength 9
    bDescriptorType 4
    bInterfaceNumber 0
    bAlternateSetting 0
    bNumEndpoints 1
    bInterfaceClass 0xe0
    bInterfaceSubClass 0x1
    bInterfaceProtocol 0x3
    iInterface 5
            *** Endpoint descriptor ***
            bLength 7
            bDescriptorType 5
            bEndpointAddress 0x85   EP 5 IN
            bmAttributes 0x3        INT
            wMaxPacketSize 16
            bInterval 16
    *** Interface descriptor ***
    bLength 9
    bDescriptorType 4
```

(continues on next page)

(continued from previous page)

```
    bInterfaceNumber 1
    bAlternateSetting 0
    bNumEndpoints 2
    bInterfaceClass 0xa
    bInterfaceSubClass 0x0
    bInterfaceProtocol 0x0
    iInterface 5
            *** Endpoint descriptor ***
            bLength 7
            bDescriptorType 5
            bEndpointAddress 0x87   EP 7 IN
            bmAttributes 0x2        BULK
            wMaxPacketSize 64
            bInterval 0
            *** Endpoint descriptor ***
            bLength 7
            bDescriptorType 5
            bEndpointAddress 0x6    EP 6 OUT
            bmAttributes 0x2        BULK
            wMaxPacketSize 64
            bInterval 0
```

Communication Flow

1. Send the initialization message `REMOTE_NDIS_INITIALIZE_MSG` via the control endpoint; the device will respond with `REMOTE_NDIS_INITIALIZE_CMPLT_MSG`.
2. Then the host needs to send `REMOTE_NDIS_QUERY_MSG` to query Object Identifiers (OIDs), and use OIDs to query and support the device, such as obtaining the device MAC address, link status, link speed, etc.
3. Next, the host can initiate a DHCP request to obtain an IP address.

Frame Format

The frame format of the RNDIS protocol is as follows:

Transmit: An RNDIS packet needs the RNDIS data header `rndis_data_packet_t` prepended to the Ethernet frame, followed by the Ethernet frame payload.

Receive: An RNDIS packet has the RNDIS data header `rndis_data_packet_t` before the Ethernet frame header. Parse the RNDIS data header first, then obtain the Ethernet frame.

Tested 4G Modules

Model	Firmware Version	ESP32-S2	ESP32-S3	ESP32-P4	USB function switch AT commands (for reference only; refer to the module manual)
ML307R	ML307R-DC-MBRH0S00	✓	✓	✓	• AT+MDIALUPCFG=”mode”,0 • 0:RNDIS 1:ECM
ML302	ML302-CNLM_MBRH0S00	✓	✓	✓	• AT+SYSNV=1,”usb-mode”,3 • 3:RNDIS+serials 5:ECM+serials
AIR720 SL	AirM2M_720SL_V545_LTE_AT	✓	✓	✓	• AT+SETUSB=1 • 1:RNDIS 2:ECM
EC800E-CN	EC800ECNLCR01A21M04	✓	✓	✓	• AT+QCFG=”usbnet”,1 • 1:ECM 3:RNDIS
EC801E-CN	EC801ECNCGR03A01M04_BETA_0521A	✓	✓	✓	• AT+QCFG=”usbnet”,1 • 1:ECM 3:RNDIS
YM310	YM310.X09S_AT.A60_R2.1.3.241121	✓	✓	✓	• AT+ECPCFG=”usbNet”,0 • 0:RNDIS 1:ECM
MC610-EU	16000.1000.00.97.20.10	✓	✓	✓	• AT+GTUSBMODE=33 • 33:RNDIS 32:ECM • Requires AT+GTRNDIS=1,1 to acti-

Note: Some 4G modules need to use AT commands to preconfigure auto dialing in RNDIS mode in order to connect to the network. Please refer to the corresponding module' s manual.

Example Code

usb/host/usb_rndis_4g_module

Throughput Test

ESP32-S3 connects to a 4G dongle and starts softAP; a phone connects to the softAP for network speed testing:

Chip	Uplink(Mbps)	Downlink(Mbps)
ESP32-S3	5.8	7.9

API Reference

Header File

- components/usb/iot_usbh_rndis/include/iot_usbh_rndis.h

Functions

esp_err_t **iot_eth_new_usb_rndis** (const *iot_usbh_rndis_config_t* *config, iot_eth_driver_t **ret_handle)

Create a new USB RNDIS Ethernet driver.

This function initializes a new USB RNDIS Ethernet driver with the specified configuration. It allocates memory for the driver, sets up the driver functions, and returns a handle to the driver.

Parameters
- **config** –Pointer to the RNDIS configuration structure.
- **ret_handle** –Pointer to a location where the handle to the new Ethernet driver will be stored.

Returns
- ESP_OK: Driver created successfully
- ESP_ERR_INVALID_ARG: Invalid argument (NULL config or ret_handle)
- ESP_ERR_NO_MEM: Memory allocation failed

usbh_cdc_port_handle_t **usb_rndis_get_cdc_port_handle** (const iot_eth_driver_t *rndis_drv)

Get the CDC port handle of the RNDIS driver.

Parameters **rndis_drv** –**[in]** RNDIS Ethernet driver handle
Returns usbh_cdc_port_handle_t CDC port handle of the RNDIS driver

Structures

struct **iot_usbh_rndis_config_t**

USB Host Ethernet RNDIS Configuration.

Public Members

const *usb_device_match_id_t* ***match_id_list**

USB device match ID for RNDIS

5.2.5 USB PPP

PPP is one of the most fundamental protocols in networking. The Point-to-Point Protocol (PPP) is a data link layer protocol designed for simple links that transmit packets between peers. Such links provide full-duplex operation and deliver packets in order. PPP provides a common solution for simple connections among hosts, bridges, and routers.

iot_usbh_modem implements the complete process of PPP dialing on the USB host. It supports connecting 4G Cat.1/4 modules via the USB interface to access the Internet using PPP.

Features:

- Fast startup
- Hot-plug support
- Supports Modem+AT dual interfaces (requires module support)
- Supports the PPP standard (most 4G modules support it)
- Supports NAPT network address translation

Supported module models:

The table below lists the 4G modules already supported. You can add support for your own module following *Example Code*.

Note: The same module may have multiple firmware variants; please consult the vendor to confirm whether PPP dialing is supported.

Model	Firmware Version	ESP32-S2	ESP32-S3	ESP32-P4
ML302	ML302-CNLM_MBRH0S00	✓	✓	✓
Air780E	AirM2M_780EVT_V2010_LTE_AT	✓	✓	✓
EC600N-CN	EC600NCNLDR03A03M16_OCP	✓	✓	✓
EC20	EC20CEFHLGR06A07M1G	✓	✓	✓
YM310	YM310.X09S_AT.A60_R2.1.3.241121	✓	✓	✓
A7600C1	Model: A7600C1-LNASRevision: A7600M6_V8.18.1	✓	✓	✓
A7670E	Model: A7670E-FASERevision: A7670M7_V1.11.1	✓	✓	✓
SIM7080G	Revision: 1951B17SIM7080	✓	✓	✓
LE270-CN	12007.6005.00.02.03.04	✓	✓	✓
MC610-EU	16000.1000.00.97.20.10	✓	✓	✓

Add support for a new 4G module

A 4G module's USB descriptor typically has multiple interfaces, such as a Modem interface, an AT command interface, and a Debug interface. The host can communicate with the cellular communication module via the virtual Modem interface, for example, by sending AT commands for control operations or sending and receiving network data. Normally, the Modem interface switches to data mode to transmit PPP data after a successful dial-up connection, and can return to command mode by sending "+++". Some modules also have a separate AT command interface, allowing data communication and AT command interaction to occur simultaneously. Below are the steps to add support for a new 4G module:

1. Confirm the 4G module **supports USB PPP dialing**;
2. Confirm the **4G SIM card is activated** and network access is enabled;
3. Identify the module's USB descriptor information: `Vendor ID`, `Product ID`, the Modem interface number, and the optional AT command interface number. You can use usbtreeview to view them;

4. Fill the above descriptor information into the `usbh_modem_config_t` structure;

Note: The basic AT commands supported by different 4G modules are largely similar, but some special commands may need to be implemented by the user.

Note: Some modules expose the PPP interface not as a standard CDC interface but as a Vendor Specific Class interface; the driver also supports some of these modules.

Note: Some 4G modules can automatically switch from data mode to command mode via PPP frames without sending "+++". In this case, disable the `MODEM_EXIT_PPP_WITH_AT_CMD` configuration option.

Example Code

usb/host/usb_cdc_4g_module

API Reference

Header File

- components/usb/iot_usbh_modem/include/iot_usbh_modem.h

Functions

esp_err_t **usbh_modem_install**(const *usbh_modem_config_t* *config)

Install and initialize the USB modem subsystem.

Sets up the USB modem manager, creates internal resources and prepares the modem for operation according to the provided configuration.

Parameters **config** –**[in]** Pointer to modem configuration

Returns

- ESP_OK: Installed successfully
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_INVALID_STATE: Already installed
- ESP_ERR_NO_MEM: Memory allocation failed

esp_err_t **usbh_modem_uninstall**(void)

Uninstall the USB modem subsystem.

Releases resources allocated during installation and stops related tasks.

Returns

- ESP_OK: Uninstalled successfully
- ESP_ERR_INVALID_STATE: Not installed

esp_netif_t ***usbh_modem_get_netif**()

Get the ESP-NETIF of the modem.

Returns the network interface created for PPP networking.

Returns Pointer to `esp_netif_t` if available, otherwise NULL

esp_err_t **usbh_modem_ppp_auto_connect**(bool enable)

Enable or disable PPP auto-connect.

When enabled, the modem will attempt to establish PPP automatically after device connection or recoveries.

Note: PPP auto-connect is enabled by default.

Parameters `enable` –**[in]** true to enable, false to disable
Returns ESP_OK on success, error otherwise

esp_err_t **usbh_modem_ppp_start**(TickType_t timeout)

Start PPP connection and wait for link up.

Parameters `timeout` –**[in]** Maximum time to wait for PPP to become connected
Returns
- ESP_OK: PPP connected within timeout
- ESP_ERR_TIMEOUT: Timed out waiting for connection
- ESP_ERR_INVALID_STATE: Modem not ready
- ESP_FAIL: Failed to start PPP

esp_err_t **usbh_modem_ppp_stop**()

Stop PPP connection.

Returns
- ESP_OK: Stopped successfully (or already stopped)
- ESP_ERR_INVALID_STATE: Modem not ready
- ESP_FAIL: Operation failed

at_handle_t **usbh_modem_get_atparser**()

Get the AT command parser handle associated with the modem.

Returns AT parser handle (`at_handle_t`), or NULL if unavailable

Structures

struct **usbh_modem_config_t**

USB modem configuration structure.

Public Members

const usb_modem_id_t ***modem_id_list**

USB modem ID list

size_t **at_tx_buffer_size**

AT command tx buffer size

size_t **at_rx_buffer_size**

AT command rx buffer size

5.2.6 USB ECM Host Driver

iot_usbh_ecm is a host driver based on the ECM protocol over the USB interface.

ECM Protocol

ECM (Ethernet Control Model) is used to exchange Ethernet frame data between a USB device and a host. It uses the CDC class interface to configure and manage various Ethernet features, and achieves network communication by encapsulating Ethernet frames in USB packets.

USB Descriptors

According to the USB specification, an ECM interface generally consists of two interface descriptors and class-specific descriptors. In the first interface descriptor, `bInterfaceClass` is `0x02`, and `bInterfaceSubClass` is `0x06`. In the ECM functional descriptor `usb_ecm_function_desc_t`, `iMACAddress` is the string index of the MAC address, and `wMaxSegmentSize` is the maximum Ethernet frame length (typically 1514).

After the ECM driver is installed, it automatically scans the interface descriptors of newly attached USB devices, finds the interface that conforms to the ECM protocol, and performs parsing and initialization. After insertion, devices typically send network link status changes through the notify endpoint, and the driver handles the status accordingly. When a device does not send notifications, and the macro `ECM_AUTO_LINK_UP_AFTER_NO_NOTIFICATION` is enabled, the driver will automatically set the network link status to connected after `ECM_AUTO_LINK_UP_WAIT_TIME_MS` milliseconds to be compatible with some ECM devices that do not send notifications. This option is enabled by default.

Note: Some ECM devices do not use the default configuration descriptor config 1, so you need to enable the macro `CONFIG_USB_HOST_ENABLE_ENUM_FILTER_CALLBACK` and implement the enumeration filter callback function.

Example descriptor:

```
*** Interface descriptor ***
bLength 9
bDescriptorType 4
bInterfaceNumber 0
bAlternateSetting 0
bNumEndpoints 1
bInterfaceClass 0x2
bInterfaceSubClass 0x6
bInterfaceProtocol 0x0
iInterface 5
        *** Endpoint descriptor ***
        bLength 7
        bDescriptorType 5
        bEndpointAddress 0x81   EP 1 IN
        bmAttributes 0x3        INT
        wMaxPacketSize 64
        bInterval 8
*** Interface descriptor ***
bLength 9
bDescriptorType 4
bInterfaceNumber 1
bAlternateSetting 0
bNumEndpoints 0
bInterfaceClass 0xa
bInterfaceSubClass 0x0
bInterfaceProtocol 0x0
iInterface 0
*** Interface descriptor ***
bLength 9
bDescriptorType 4
bInterfaceNumber 1
bAlternateSetting 1
```

(continues on next page)

(continued from previous page)

```
bNumEndpoints 2
bInterfaceClass 0xa
bInterfaceSubClass 0x0
bInterfaceProtocol 0x0
iInterface 0
        *** Endpoint descriptor ***
        bLength 7
        bDescriptorType 5
        bEndpointAddress 0x82   EP 2 IN
        bmAttributes 0x2        BULK
        wMaxPacketSize 64
        bInterval 0
        *** Endpoint descriptor ***
        bLength 7
        bDescriptorType 5
        bEndpointAddress 0x3    EP 3 OUT
        bmAttributes 0x2        BULK
        wMaxPacketSize 64
        bInterval 0
```

Tested ECM Devices

USB-to-Ethernet Adapters

Chip	ESP32-S2	ESP32-S3	ESP32-P4
CH397A	✅	✅	✅
RTL8152B	✅	✅	✅
NX7202D	✅	✅	✅

4G Modules

Model	Firmware Version	ESP32-S2	ESP32-S3	ESP32-P4	USB function switch AT cmds (ref only; see module manual)
ML307R	ML307R-DC-MBRH0S00	✗	✗	✓	• AT+MDIALUPCFG=”mode”,1 • 0:RNDIS 1:ECM
ML302	ML302-CNLM_MBRH0S00	✓	✓	✓	• AT+SYSNV=1,”usb-mode”,5 • 3:RNDIS 5:ECM
AIR720 SL	AirM2M_720SL_V5.45_LTE_AT	✗	✗	✓	• AT+SetUSB=2 • 1:RNDIS 2:ECM
EC800E-CN	EC800ECNLCR01A21M04	✓	✓	✓	• AT+QCFG=”usbnet”,1 • 3:RNDIS 1:ECM
EC801E-CN	EC801ECNCGR03A01M04_BETA_0521A	✓	✓	✓	• AT+QCFG=”usbnet”,1 • 3:RNDIS 1:ECM
YM310	YM310.X09S_AT.A60_R2.1.3.241121	✓	✓	✓	• AT+ECPCFG=”usbNet”,1 • 0:RNDIS 1:ECM
MC610-EU	16000.1000.00.97.20.10	✓	✓	✓	• AT+GTUSBMODE=32 • 33:RNDIS 32:ECM • Requires AT+GTRNDIS=1,1 to activate dialing
Lierda NT26	NT26FCNB30WNA-Q01010950	✓	✓	✓	• Release master • AT+ECPCFG=”usbNet”,1

Note: ✓ indicates supported; ✗ indicates endpoint size not supported (because these 4G modules have endpoint MPS exceeding the standard 64 bytes at Full Speed).

Note: Some 4G modules need to use AT commands to preconfigure auto dialing in ECM mode in order to connect to the network. Please refer to the corresponding module' s manual.

FAQ

1. Ping times out after the 4G module obtains an IP address. How to resolve this?

   The network is not connected. Check if automatic dialing in ECM mode is activated. Please refer to the corresponding module' s documentation for details. Check if the SIM card is inserted correctly and registered with the network.
2. The 4G module prints "No ECM interface found for device VID: xxxx, PID: xxxx, ignore it" after connecting to USB.

   The ECM interface was not detected. Please check if the module has been correctly set to ECM mode. For details, please refer to the documentation manual for the corresponding module.
3. After the 4G module is connected to the USB, it prints "E (887) HCD DWC: EP MPS (512) exceeds supported limit (408)" .

   The USB endpoint MPS of this module exceed the size allowed by the ESP chip hardware. Please try using the *RNDIS example* or replace it with another 4G module.

Example Code

usb/host/usb_ecm_4g_module

Downloads

CDC 1.2 specification

API Reference

Header File

- components/usb/iot_usbh_ecm/include/iot_usbh_ecm.h

Functions

esp_err_t **iot_eth_new_usb_ecm**(const *iot_usbh_ecm_config_t* *config, iot_eth_driver_t **ret_handle)

Create a new USB ECM Ethernet driver.

This function initializes a new USB ECM Ethernet driver with the specified configuration. It allocates memory for the driver, sets up the driver functions, and returns a handle to the driver.

Parameters

- **config** –**[in]** Pointer to the ECM configuration structure.
- **ret_handle** –**[out]** Pointer to a location where the handle to the new Ethernet driver will be stored.

Returns

- ESP_OK: Driver created successfully
- ESP_ERR_INVALID_ARG: Invalid argument (NULL config or ret_handle)
- ESP_ERR_NO_MEM: Memory allocation failed

usbh_cdc_port_handle_t **usb_ecm_get_cdc_port_handle** (const iot_eth_driver_t *ecm_drv)

Get the CDC port handle of the ECM driver.

Parameters **ecm_drv** –**[in]** ECM Ethernet driver handle
Returns usbh_cdc_port_handle_t CDC port handle of the ECM driver

Structures

struct **iot_usbh_ecm_config_t**

USB Host Ethernet ECM Configuration.

Public Members

const *usb_device_match_id_t* ***match_id_list**

USB device match IDs for ECM

5.3 USB Device Drivers

5.3.1 USB Device UVC

`usb_device_uvc` is a USB `UVC` device driver for ESP32-S2/ESP32-S3/ESP32-P4, which supports streaming JPEG frames to the USB Host. User can wrapper the Camera or any devices as a UVC standard device through the callback functions.

Features:

1. Support video stream through the UVC Stream interface
2. Support both `Isochronous` and `Bulk` mode
3. Support multiple resolutions and frame rates

Add component to your project

Please use the component manager command `add-dependency` to add the `usb_device_uvc` to your project's dependency, during the `CMake` step the component will be downloaded automatically

```
idf.py add-dependency "espressif/usb_device_uvc=*"
```

User Reference

The component provides only one API to configure the UVC device. As the driver based on the `TinyUSB` stack, the deinit API is not provided.

```
#include "usb_device_uvc.h"

static esp_err_t camera_start_cb(uvc_format_t format, int width, int height, int␣
↪rate, void *cb_ctx)
{
    // user can initialize the camera here
    // camera should be initialized with the given format, width, height and frame␣
↪rate
```

(continues on next page)

(continued from previous page)

```
    return ESP_OK;
}

static void camera_stop_cb(void *cb_ctx)
{
    //user code
    return;
}

static uvc_fb_t* camera_fb_get_cb(void *cb_ctx)
{
    // user code to return a image frame buffer
    // camera should prepare next frame, and return the frame buffer
    return uvc_fb;
}

static void camera_fb_return_cb(uvc_fb_t *fb, void *cb_ctx)
{
    // the frame buffer is returned after it is copied to the transfer buffer
    // user code to recycle the frame buffer
    return;
}

//the buffer is used to store the data to be sent to the host
const size_t buff_size = 30 * 1024;
uint8_t *uvc_buffer = (uint8_t *)heap_caps_malloc(buff_size, MALLOC_CAP_DEFAULT);
assert(uvc_buffer != NULL);

uvc_device_config_t config = {
    .uvc_buffer = uvc_buffer,
    .uvc_buffer_size = 40 * 1024,
    .start_cb = camera_start_cb,
    .fb_get_cb = camera_fb_get_cb,
    .fb_return_cb = camera_fb_return_cb,
    .stop_cb = camera_stop_cb,
    .cb_ctx = NULL,
};

ESP_ERROR_CHECK(uvc_device_config(0, &config));
ESP_ERROR_CHECK(uvc_device_init());
```

Examples

usb/device/usb_webcam usb/device/usb_dual_uvc_device

API Reference

Header File

- components/usb/usb_device_uvc/include/usb_device_uvc.h

Functions

esp_err_t **uvc_device_config**(int index, *uvc_device_config_t* *config)

Configure the UVC device by uvc device number.

Parameters

- **index** –UVC device index number [0,1]
- **config** –Configuration for the UVC device

Returns ESP_OK on success ESP_ERR_INVALID_ARG if the configuration is invalid ESP_FAIL if the UVC device could not be initialized

esp_err_t **uvc_device_init**(void)

Initialize the UVC device, after this function is called, the UVC device will be visible to the host and the host can open the UVC device with the specific format and resolution.

Returns ESP_OK on success ESP_FAIL if the UVC device could not be initialized

esp_err_t **uvc_device_deinit**(void)

Deinitialize the UVC device.

Returns ESP_OK on success ESP_ERR_INVALID_STATE if the UVC device is not initialized, or event group is NULL

Structures

struct **uvc_fb_t**

Frame buffer structure.

Public Members

uint8_t ***buf**

Pointer to the frame data

size_t **len**

Length of the buffer in bytes

size_t **width**

Width of the image frame in pixels

size_t **height**

Height of the image frame in pixels

uvc_format_t **format**

Format of the frame data

struct timeval **timestamp**

Timestamp since boot of the frame

struct **uvc_device_config_t**

Configuration for the UVC device.

Public Members

uint8_t ***uvc_buffer**

UVC transfer buffer

uint32_t **uvc_buffer_size**

UVC transfer buffer size, should bigger than one frame size

uvc_input_start_cb_t **start_cb**

callback function of host open the UVC device with the specific format and resolution

uvc_input_fb_get_cb_t **fb_get_cb**

callback function of host request a new frame buffer

uvc_input_fb_return_cb_t **fb_return_cb**

callback function of the frame buffer is no longer used

uvc_input_stop_cb_t **stop_cb**

callback function of host close the UVC device

void ***cb_ctx**

callback context, for user specific usage

Type Definitions

typedef esp_err_t (***uvc_input_start_cb_t**)(*uvc_format_t* format, int width, int height, int rate, void *cb_ctx)

type of callback function when host open the UVC device

typedef *uvc_fb_t* *(***uvc_input_fb_get_cb_t**)(void *cb_ctx)

type of callback function when host request a new frame buffer

typedef void (***uvc_input_fb_return_cb_t**)(*uvc_fb_t* *fb, void *cb_ctx)

type of callback function when the frame buffer is no longer used

typedef void (***uvc_input_stop_cb_t**)(void *cb_ctx)

type of callback function when host close the UVC device

Enumerations

enum **uvc_format_t**

UVC format.

Values:

enumerator **UVC_FORMAT_JPEG**

JPEG format

enumerator **UVC_FORMAT_H264**

H264 format

5.3.2 USB Device UAC

`esp_device_uac` is a USB Audio Class driver library based on TinyUSB. It supports simulating an ESP chip as an audio device, with customizable audio sampling rates, microphone channels, and speaker channels.

Features:

1. Supports ISO FeedBack communication interface by default. It automatically syncs with the host based on the remaining size of the UAC FIFO memory. Reference.
2. Allows custom audio sampling rates, microphone channels, and speaker channels.
3. Buffers a segment of data before transmission when speaker data arrives, which helps reduce the frequency of audio data transmission interruptions.

USB Device UAC User Guide

- Development Board
  1. Any ESP32-S2/ESP32-S3/ESP32-P4 development board with a USB interface can be used.
- USB MIC Callback Function
  1. The `uac_input_cb_t` callback function is used to transfer audio data to the USB host. Users should transfer audio data according to the timeline or block the callback function while waiting for audio data.
  2. **Set the length of the audio data read by the callback function by defining the `CONFIG_UAC_MIC_INTERVAL_MS`**
     - Setting `CONFIG_UAC_MIC_INTERVAL_MS=10` with a sampling rate of 48000Hz, 16-bit precision, and single channel results in a read data length of 10ms * 48000Hz / 1000 * 2bytes = 960bytes.
  3. Set the length of the first audio write by the callback function using the `UAC_SPK_INTERVAL_MS` macro. To prevent high audio data transmission interruption frequency, the default is 10ms. Subsequent audio writes will be transmitted in approximately 1ms data lengths.
  4. The `UAC_SPK_NEW_PLAY_INTERVAL` macro determines whether the incoming audio is new. If it is new, it will buffer a segment of data before transmission, which helps reduce the frequency of audio data transmission interruptions.
  5. The `UAC_SUPPORT_MACOS` macro supports MacOS. Note that enabling this macro may make the device unrecognizable by Windows systems.

USB Device UAC API Reference

1. Users can configure four callback functions for audio input/output, mute, and volume by calling the *`uac_device_config_t`* function.

```
uac_device_config_t config = {
    .output_cb = uac_device_output_cb,          // Speaker output callback
    .input_cb = uac_device_input_cb,            // Microphone input callback
    .set_mute_cb = uac_device_set_mute_cb,      // Set mute callback
    .set_volume_cb = uac_device_set_volume_cb,  // Set volume callback
    .cb_ctx = NULL,
};
uac_device_init(&config);
```

Example

- usb/device/usb_uac

API Reference

Header File

- components/usb/usb_device_uac/include/usb_device_uac.h

Functions

esp_err_t **`uac_device_init`** (*uac_device_config_t* *config)

Initialize the USB Audio Class (UAC) device.

Parameters **config** –Pointer to the UAC device configuration structure.

Returns

- ESP_OK on success
- ESP_FAIL on failure

Structures

struct **uac_device_config_t**

USB UAC Device Config.

Public Members

bool **skip_tinyusb_init**

if true, the Tinyusb and usb phy will not be initialized

uac_output_cb_t **output_cb**

callback function for UAC data output, if NULL, output will be disabled

uac_input_cb_t **input_cb**

callback function for UAC data input, if NULL, input will be disabled

uac_set_mute_cb_t **set_mute_cb**

callback function for set mute, if NULL, the set mute request will be ignored

uac_set_volume_cb_t **set_volume_cb**

callback function for set volume, if NULL, the set volume request will be ignored

void ***cb_ctx**

callback context, for user specific usage

Type Definitions

typedef esp_err_t (***uac_output_cb_t**)(uint8_t *buf, size_t len, void *cb_ctx)

typedef esp_err_t (***uac_input_cb_t**)(uint8_t *buf, size_t len, size_t *bytes_read, void *cb_ctx)

typedef void (***uac_set_mute_cb_t**)(uint32_t mute, void *cb_ctx)

typedef void (***uac_set_volume_cb_t**)(uint32_t volume, void *cb_ctx)

5.3.3 ESP TinyUF2

`esp_tinyuf2` is an enhanced version of TinyUF2 for ESP chips with USB support. Which features:

- support over-the-air (OTA) updates through the virtual USB drive
- support dumping NVS key-value pairs to ini file in the virtual USB drive
- support modify ini file and write back to NVS

UF2 is a file format developed by Microsoft for PXT, that is particularly suitable for flashing microcontrollers over MSC (Mass Storage Class). For a more friendly explanation, check out the blog post.

Support UF2 OTA/NVS in Your Project

1. Add the component to your project using `idf.py add_dependency` command.

```
idf.py add-dependency "esp_tinyuf2"
```

2. Customer your partition table. Like other OTA solutions, you need to reserve at least two OTA app partitions. Please refer to Partition Tables and usb_uf2_ota example for details.

```
# Partition Table Example
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     ,        0x4000,
otadata,  data, ota,     ,        0x2000,
phy_init, data, phy,     ,        0x1000,
ota_0,    app,  ota_0,   ,        1500K,
ota_1,    app,  ota_1,   ,        1500K,
```

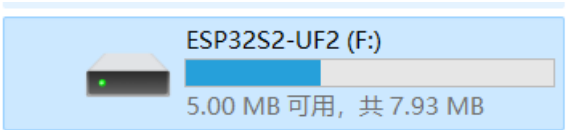
3. Using `idf.py menuconfig` to config the component' s behavior in `(Top) → Component config → TinyUF2 Config`

- `USB Virtual Disk size(MB)`: The size of the virtual U-disk shows in File Explorer, 8MB by default
- `Max APP size(MB)`: Maximum APP size, 4MB by default
- `Flash cache size(KB)`: Cache size used for writing Flash efficiently, 32KB by default
- `USB Device VID`: Espressif VID (0x303A) by default
- `USB Device PID`: Espressif test PID (0x8000) by default, refer esp-usb-pid to apply new.
- `USB Disk Name`: The name of the virtual U-disk shows in File Explorer, `ESP32Sx-UF2` by default
- `USB Device Manufacture`: `Espressif` by default
- `Product Name`: `ESP TinyUF2` by default
- `Product ID`: `12345678` by default
- `Product URL`: A `index` file will be added to the U-disk, users can click to goto the webpage, `https://products.espressif.com/` by default
- `UF2 NVS ini file size`: The `ini` file size prepares for NVS function

4. Install tinyuf2 function like below, for more details, please refer example usb_uf2_nvs and usb_uf2_ota

```
/* install UF2 OTA */
tinyuf2_ota_config_t ota_config = DEFAULT_TINYUF2_OTA_CONFIG();
ota_config.complete_cb = uf2_update_complete_cb;
/* disable auto restart, if false manual restart later */
ota_config.if_restart = false;
/* install UF2 NVS */
tinyuf2_nvs_config_t nvs_config = DEFAULT_TINYUF2_NVS_CONFIG();
nvs_config.part_name = "nvs";
nvs_config.namespace_name = "myuf2";
nvs_config.modified_cb = uf2_nvs_modified_cb;
esp_tinyuf2_install(&ota_config, &nvs_config);
```

5. Run `idf.py build flash` for the initial download, later `idf.py uf2-ota` can be used to generate new `uf2` app bin
6. Drag and drop UF2 format file to the disk, to upgrade to new `uf2` app bin



UF2 NVS Hidden

By enabling **UF2_INI_NVS_VALUE_HIDDEN** via menuconfig (*(Top) → Component config → TinyUF2 Config → Enable UF2 ini file hide NVS value*), the values in NVS will be encrypted and replaced with ****.

```
esp_tinyuf2_add_key_hidden("password")
```

Enable UF2 USB Console

Through menuconfig `(Top) → Component config → TinyUF2 Config → Enable USB Console For log`, the log will be output to the USB Serial port (Output to UART by default).

Build APP to UF2 format

The new command `idf.py uf2-ota` is added by this component, which can be used to build the APP to UF2 format. After the build is complete, the UF2 file (`${PROJECT_NAME}.uf2`) will be generated in the current `project` directory.

```
idf.py uf2-ota
```

Convert Existing APP to UF2 Format

To convert your existing APP binary to UF2 format, simply use the uf2conv.py on a `.bin` file, specifying the family id as `ESP32S2`, `ESP32S3` or their magic number as follows. And you must specify the address of 0x00 with the `-b` switch, the tinyuf2 will use it as offset to write to the OTA partition.

1. convert as follows
   using:

   ```
   uf2conv.py your_firmware.bin -c -b 0x00 -f ESP32S3
   ```

   or:

   ```
   uf2conv.py your_firmware.bin -c -b 0x00 -f 0xc47e5767
   ```

Note

- To use the UF2 OTA function continuously, the TinyUF2 function must be enabled in the updated APP.

Using UF2 in Bootloader

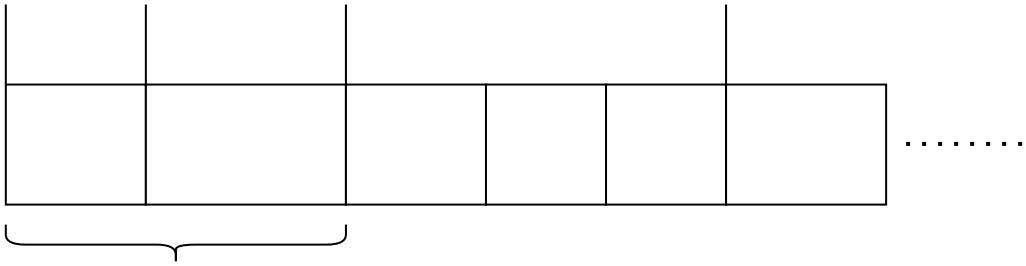
By embedding a specific APP bin with UF2 functionality into the Bootloader, the following features can be achieved: 1. Automatically enter UF2 download mode when the `factory/test/ota` partitions do not contain firmware. 2. Manually enter UF2 download mode by pulling the `BOOT_UF2` pin low. 3. Manually enter UF2 download mode by calling *esp_restart_from_tinyuf2()* function in user app.

Example usb/device/bootloader_uf2

Instructions:

1. By default, only `nvs/phy_init/factory` partitions are supported. To support `test/ota/spiffs` partitions, modify the partition table manually and recompile.
2. Drag-and-drop upgrades default to the `factory` partition. Ensure the `factory` partition exists or modify the code manually.
3. The default `CONFIG_PARTITION_TABLE_OFFSET` is set to 0x60000. If the firmware is too large, modify this value.
4. The default `nvs` partition name is `CONFIG_BOOTLOADER_UF2_NVS_PART_NAME` ("nvs"), and the default NVS namespace is `CONFIG_BOOTLOADER_UF2_NVS_NAMESPACE_NAME` ("uf2_nvs"). Ensure the user firmware uses the same NVS partition name and namespace as the bootloader for proper functionality.

5. A working indicator LED is supported by default. The default LED GPIO is `CONFIG_BOOTLOADER_UF2_LED_INDICATOR_GPIO_NUM` (2). Ensure this matches the hardware connection.



Flash Partition Reference:

Notes

- The `Bootloader UF2` bin must be flashed at an address aligned to a multiple of `CONFIG_MMU_PAGE_SIZE`; otherwise, it will not work.

Note: The `CONFIG_MMU_PAGE_SIZE` for ESP32-S2/ESP32-S3/ESP32-P4 defaults to 64KB (0x10000), so flashing must align to a 64KB (0x10000) boundary.

- Configure `CONFIG_PARTITION_TABLE_OFFSET` to accommodate the combined size of `bootloader.bin + bootloader_uf2.bin`; otherwise, `bootloader_uf2.bin` cannot be flashed.
- Enable the `CONFIG_ENABLE_BOOTLOADER_UF2` macro.
- Enable the `CONFIG_SPI_FLASH_DANGEROUS_WRITE_ALLOWED` macro: Since the position for bootloader_uf2.bin is not explicitly shown in the partition table, the checks must be disabled.
- Flash the Bin files to the firmware according to the following addresses:
    1. CONFIG_BOOTLOADER_OFFSET_IN_FLASH(The starting addresses vary for different chips.) - bootloader.bin
    2. 0x10000 - bootloader_uf2.bin
    3. CONFIG_PARTITION_TABLE_OFFSET (0x6000) - partition-table.bin

  Add the following snippet to your project' s *CMakeLists.txt* to automatically generate a merged Bin file after compilation:

```
add_custom_command(
   TARGET app
   POST_BUILD
   COMMAND ${CMAKE_COMMAND} -E echo "Flash merged bin merge_uf2.bin to address ${CONFIG_BOOTLOADER_OFFSET_IN_FLASH}"
   COMMAND ${ESPTOOLPY} --chip ${IDF_TARGET} merge_bin -o merge_uf2.bin ${CONFIG_BOOTLOADER_OFFSET_IN_FLASH} ${BUILD_DIR}/bootloader/bootloader.bin 0x10000 ${BUILD_DIR}/${PROJECT_BIN} ${CONFIG_PARTITION_TABLE_OFFSET} ${BUILD_DIR}/partition_table/partition-table.bin
)
```

API Reference

Header File

- components/usb/esp_tinyuf2/esp_tinyuf2.h

Functions

esp_err_t **esp_tinyuf2_install** (*tinyuf2_ota_config_t* *ota_config, *tinyuf2_nvs_config_t* *nvs_config)

Flashing app to specified partition through USB UF2 (Virtual USB Disk), and support operate NVS partition through USB UF2 CONFIG.ini file.

Parameters
- **ota_config** –tinyuf2 configs described in *tinyuf2_ota_config_t*
- **nvs_config** –tinyuf2 nvs configs described in *tinyuf2_nvs_config_t*

Returns
- ESP_ERR_INVALID_ARG invalid parameter, please check partitions
- ESP_ERR_INVALID_STATE tinyuf2 already installed
- ESP_OK Success

esp_err_t **esp_tinyuf2_uninstall** (void)

Uninstall tinyuf2, only reset USB to default state.

Note: not release memory due to tinyusb not support teardown

Returns esp_err_t
- ESP_ERR_INVALID_STATE tinyuf2 not installed
- ESP_OK Success

tinyuf2_state_t **esp_tinyuf2_current_state** (void)

Get tinyuf2 current state.

Returns tinyuf2_state_t

void **esp_restart_from_tinyuf2** (void)

Restart system and set reset reason to UF2_RESET_REASON_VALUE.

Structures

struct **tinyuf2_ota_config_t**

tinyuf2 configurations

Public Members

esp_partition_subtype_t **subtype**

Partition subtype. if ESP_PARTITION_SUBTYPE_ANY will use the next_update_partition by default.

const char ***label**

Partition label. Set this value if looking for partition with a specific name. if subtype==ESP_PARTITION_SUBTYPE_ANY, label default to NULL.

bool **if_restart**

if restart system to new app partition after UF2 flashing done

update_complete_cb_t **complete_cb**

user callback called after uf2 update complete

struct **tinyuf2_nvs_config_t**

tinyuf2 nvs configurations

Public Members

const char ***part_name**

Partition name.

const char ***namespace_name**

Namespace name.

nvs_modified_cb_t **modified_cb**

user callback called after uf2 update complete

Macros

DEFAULT_TINYUF2_OTA_CONFIG()

DEFAULT_TINYUF2_NVS_CONFIG()

UF2_RESET_REASON_VALUE

Type Definitions

typedef void (***update_complete_cb_t**)(void)

user callback called after uf2 update complete

typedef void (***nvs_modified_cb_t**)(void)

user callback called after nvs modified

Enumerations

enum **tinyuf2_state_t**

tinyuf2 current state

Values:

enumerator **TINYUF2_STATE_NOT_INSTALLED**

tinyuf2 driver not installed

enumerator **TINYUF2_STATE_INSTALLED**

tinyuf2 driver installed

enumerator **TINYUF2_STATE_MOUNTED**

USB mounted

Chapter 6

Audio

6.1 PWM Audio

The PWM audio function uses the internal LEDC peripheral in the ESP32 series chips to generate PWM audio, which does not need an external audio Codec chip. This is mainly used for cost-sensitive applications with low audio quality requirements.

6.1.1 Features

- Allows any GIPO with output capability as an audio output pin
- Supports 8-bit ~ 10-bit PWM resolution
- Supports stereo
- Supports 8 ~ 48 KHz sampling rate

6.1.2 Structure

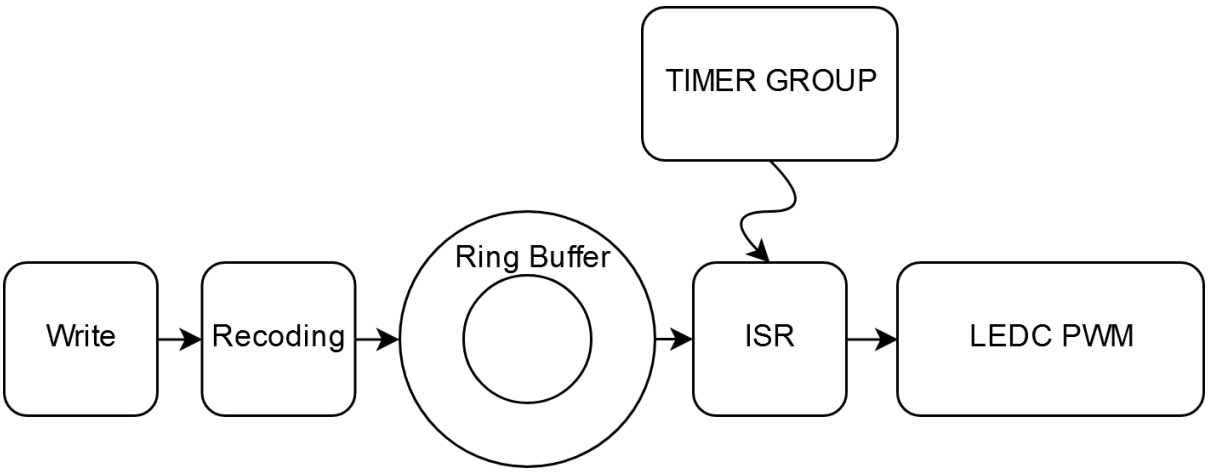


Fig. 1: Structure

1. First, the data is recoded to meet the PWM input requirements, including the shift and offset of the data;
2. Send data to the ISR (Interrupt Service Routines) function of the Timer Group via Ring Buffer;

3. The Timer Group reads data from the Ring Buffer according to the pre-defined sampling rate, and write the data into LEDC.

Note: Since the output is a PWM signal, it needs to be low-pass filtered to get the audio waveform.

6.1.3 PWM Frequency

The frequency of the output PWM cannot be configured directly, but needs to be calculated by configuring the number of PWM resolution bits, as shown below:

$$f_{pwm} = \frac{f_{APB_CLK}}{2^{res_bits}} - \left(\frac{f_{APB_CLK}}{2^{res_bits}} MOD 1000\right)$$

The f_{APB_CLK} here is 80 MHz, and res_bits is the number of PWM resolution bits. When the resolution is `LEDC_TIMER_10_BIT`, the PWM frequency is 78 KHz. As we all know, a higher PWM frequency and resolution can better reproduce the audio signal. However, this formula shows that increasing PWM frequency means lower resolution and increasing resolution means lower PWM frequency. Thus, please adjust these parameters according to the actual application scenario to achieve a good balance.

6.1.4 Application Example

```
pwm_audio_config_t pac;
pac.duty_resolution    = LEDC_TIMER_10_BIT;
pac.gpio_num_left      = LEFT_CHANNEL_GPIO;
pac.ledc_channel_left  = LEDC_CHANNEL_0;
pac.gpio_num_right     = RIGHT_CHANNEL_GPIO;
pac.ledc_channel_right = LEDC_CHANNEL_1;
pac.ledc_timer_sel     = LEDC_TIMER_0;
pac.tg_num             = TIMER_GROUP_0;
pac.timer_num          = TIMER_0;
pac.ringbuf_len        = 1024 * 8;

pwm_audio_init(&pac));              /**< Initialize pwm audio */
pwm_audio_set_param(48000, 8, 2);   /**< Set sample rate, bits and channel␣
↪number */
pwm_audio_start();                  /**< Start to run */

while(1) {

    //< Prepare audio data, such as decode mp3/wav file

    /**< Write data to play */
    pwm_audio_write(audio_data, length, &written, 1000 / portTICK_PERIOD_
↪MS);
}
```

6.1.5 API Reference

Header File

- components/audio/pwm_audio/include/pwm_audio.h

Functions

esp_err_t **pwm_audio_init** (const *pwm_audio_config_t* *cfg)

Initializes and configure the pwm audio.

Parameters **cfg** –configurations - see *pwm_audio_config_t* struct

Returns

- ESP_OK Success
- ESP_FAIL timer_group or ledc initialize failed
- ESP_ERR_INVALID_ARG if argument wrong
- ESP_ERR_INVALID_STATE The pwm audio already configure
- ESP_ERR_NO_MEM Memory allocate failed

esp_err_t **pwm_audio_deinit** (void)

Deinitialize LEDC timer_group and output gpio.

Returns

- ESP_OK Success
- ESP_FAIL pwm_audio not initialized

esp_err_t **pwm_audio_start** (void)

Start to run pwm audio.

Returns

- ESP_OK Success
- ESP_ERR_INVALID_STATE pwm_audio not initialized or it already running

esp_err_t **pwm_audio_stop** (void)

Stop pwm audio.

Attention Only stop timer, and the pwm will keep to output. If you want to stop pwm output, call pwm_audio_deinit function

Returns

- ESP_OK Success
- ESP_ERR_INVALID_STATE pwm_audio not initialized or it already idle

esp_err_t **pwm_audio_write** (uint8_t *inbuf, size_t len, size_t *bytes_written, TickType_t ticks_to_wait)

Write data to play.

Parameters

- **inbuf** –Pointer source data to write
- **len** –length of data in bytes
- **bytes_written** –**[out]** Number of bytes written, if timeout, the result will be less than the size passed in.
- **ticks_to_wait** –TX buffer wait timeout in RTOS ticks. If this many ticks pass without space becoming available in the DMA transmit buffer, then the function will return (note that if the data is written to the DMA buffer in pieces, the overall operation may still take longer than this timeout.) Pass portMAX_DELAY for no timeout.

Returns

- ESP_OK Success
- ESP_FAIL Write encounter error
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **pwm_audio_set_param** (int rate, ledc_timer_bit_t bits, int ch)

Set audio parameter, Similar to pwm_audio_set_sample_rate(), but also sets bit width.

Attention After start pwm audio, it can't modify parameters by call function pwm_audio_set_param. If you want to change the parameters, must stop pwm audio by call function pwm_audio_stop.

Parameters

- **rate** –sample rate (ex: 8000, 44100···)
- **bits** –bit width
- **ch** –channel number

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **pwm_audio_set_sample_rate**(int rate)

Set sample rate.

Parameters **rate** –sample rate (ex: 8000, 44100···)

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **pwm_audio_set_volume**(int8_t volume)

Set volume for pwm audio.

Attention when the volume is too small, it will produce serious distortion

Parameters **volume** –Volume to set (-16 ~ 16). Set to 0 for original output; Set to less then 0 for attenuation, and -16 is mute; Set to more than 0 for enlarge, and 16 is double output

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **pwm_audio_get_volume**(int8_t *volume)

Get current volume.

Parameters **volume** –Pointer to volume

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **pwm_audio_get_param**(int *rate, int *bits, int *ch)

Get parameter for pwm audio.

Parameters

- **rate** –sample rate, if you don't care about this parameter, set it to NULL
- **bits** –bit width, if you don't care about this parameter, set it to NULL
- **ch** –channel number, if you don't care about this parameter, set it to NULL

Returns

- Always return ESP_OK

esp_err_t **pwm_audio_get_status**(*pwm_audio_status_t* *status)

get pwm audio status

Parameters **status** –current pwm_audio status

Returns

- ESP_OK Success

Structures

struct **pwm_audio_config_t**

Configuration parameters for pwm_audio_init function.

Public Members

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 **ringbuf_len**

ringbuffer size

Enumerations

enum **pwm_audio_status_t**

pwm audio status

Values:

enumerator **PWM_AUDIO_STATUS_UN_INIT**

pwm audio uninitialized

enumerator **PWM_AUDIO_STATUS_IDLE**

pwm audio idle

enumerator **PWM_AUDIO_STATUS_BUSY**

pwm audio busy

enum **pwm_audio_channel_t**

pwm audio channel define

Values:

enumerator **PWM_AUDIO_CH_MONO**

1 channel (mono)

enumerator **PWM_AUDIO_CH_STEREO**

2 channel (stereo)

enumerator **PWM_AUDIO_CH_MAX**

6.2 DAC Audio

ESP32 has two independent DAC channels and can play audio using I2S directly via DMA. The APIs in this document have been simplified on the basis of ESP-IDF, and the related data has been recoded to support more types of sampling bit width.

6.2.1 API Reference

Header File

- components/audio/dac_audio/include/dac_audio.h

Functions

esp_err_t **dac_audio_init** (*dac_audio_config_t* *cfg)

initialize i2s built-in dac to play audio with

Attention only support ESP32, because i2s of ESP32S2 not have a built-in dac

Parameters **cfg** –configurations - see *dac_audio_config_t* struct

Returns

- ESP_OK Success
- ESP_FAIL Encounter error
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM Out of memory

esp_err_t **dac_audio_deinit** (void)

deinitialize dac

Returns

- ESP_OK Success
- ESP_FAIL Encounter error

esp_err_t **dac_audio_start** (void)

Start dac to play.

Returns

- ESP_OK Success
- ESP_FAIL Encounter error

esp_err_t **dac_audio_stop** (void)

Stop play.

Returns

- ESP_OK Success
- ESP_FAIL Encounter error

esp_err_t **dac_audio_set_param** (int rate, int bits, int ch)

Configuration dac parameter.

Parameters

- **rate** –sample rate (ex: 8000, 44100···)
- **bits** –bit width
- **ch** –channel number

Returns

- ESP_OK Success
- ESP_FAIL Encounter error
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **dac_audio_set_volume**(int8_t volume)

Set volume.

Attention Using volume greater than 0 may cause variable overflow and distortion Usually you should enter a volume less than or equal to 0

Parameters **volume** –Volume to set (-16 ~ 16), see Macro VOLUME_0DB Set to 0 for original output; Set to less then 0 for attenuation, and -16 is mute; Set to more than 0 for enlarge, and 16 is double output

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **dac_audio_write**(uint8_t *inbuf, size_t len, size_t *bytes_written, TickType_t ticks_to_wait)

Write data to play.

Parameters

- **inbuf** –Pointer source data to write
- **len** –length of data in bytes
- **bytes_written** –**[out]** Number of bytes written, if timeout, the result will be less than the size passed in.
- **ticks_to_wait** –TX buffer wait timeout in RTOS ticks. If this many ticks pass without space becoming available in the DMA transmit buffer, then the function will return (note that if the data is written to the DMA buffer in pieces, the overall operation may still take longer than this timeout.) Pass portMAX_DELAY for no timeout.

Returns

- ESP_OK Success
- ESP_FAIL Write encounter error
- ESP_ERR_INVALID_ARG Parameter error

Structures

struct **dac_audio_config_t**

Configuration parameters for dac_audio_init function.

Public Members

i2s_port_t **i2s_num**

I2S_NUM_0, I2S_NUM_1

int **sample_rate**

I2S sample rate

i2s_bits_per_sample_t **bits_per_sample**

I2S bits per sample

i2s_dac_mode_t **dac_mode**

DAC mode configurations - see i2s_dac_mode_t

int **dma_buf_count**

DMA buffer count, number of buffer

int **dma_buf_len**

DMA buffer length, length of each buffer

uint32_t **max_data_size**

one time max write data size

6.3 ADC Microphone

Using ADC to capture analog microphone data eliminates the need for an external audio codec chip, making it suitable for applications that do not require high sampling rates and are cost-sensitive. The code structure follows the engineering model of esp_codec_dev.

6.3.1 Features

- Supports multi-channel ADC audio sampling.
- Supports up to 12-bit ADC sampling resolution.

Note: The maximum resolution for multi-channel sampling is determined by the ADC' s maximum sampling rate (83.3K for ESP32-S3) divided by the number of sampling channels.

6.3.2 Reference Circuit

The external reference circuit is shown below. You can choose between transimpedance amplifier or inverting/non-inverting amplifier circuits to amplify the MIC input signal. In most cases, using a transimpedance amplifier can achieve better signal-to-noise ratio. In this example, R29 is used to add DC bias for the MIC, using the general-purpose op-amp LMV321 as the operational amplifier, with R26 and C19 forming a feedback circuit to make the op-amp work in transimpedance amplification mode, where the resistance and capacitance are adjusted according to the MIC sensitivity. R31 and R30 are used to set the midpoint voltage to avoid crossover distortion, and R27 and R32 are used to provide DC bias to avoid distortion during ADC sampling. For more design reference information about front-end MIC amplifiers, please refer to Single-Supply, Electret Microphone Pre-Amplifier Reference Design.

Note:

- Bias: ADC midpoint bias is necessary (such as Vref/2) to avoid ADC sampling distortion. By default, ESP32-C3' s Vref is approximately 0.9V.
- Power supply: It is recommended to use LDO for MIC power supply regulation to avoid power supply fluctuations affecting ADC sampling. If external LDO is not used, please add separate RC filtering on the power supply side (in this example, R28 and C23 are used for filtering). It is recommended that MIC and op-amp have independent power supplies.
- It is recommended to select MIC with sensitivity greater than -46dB to avoid low signal-to-noise ratio after amplification. Adjust the amplification factor according to the microphone sensitivity.

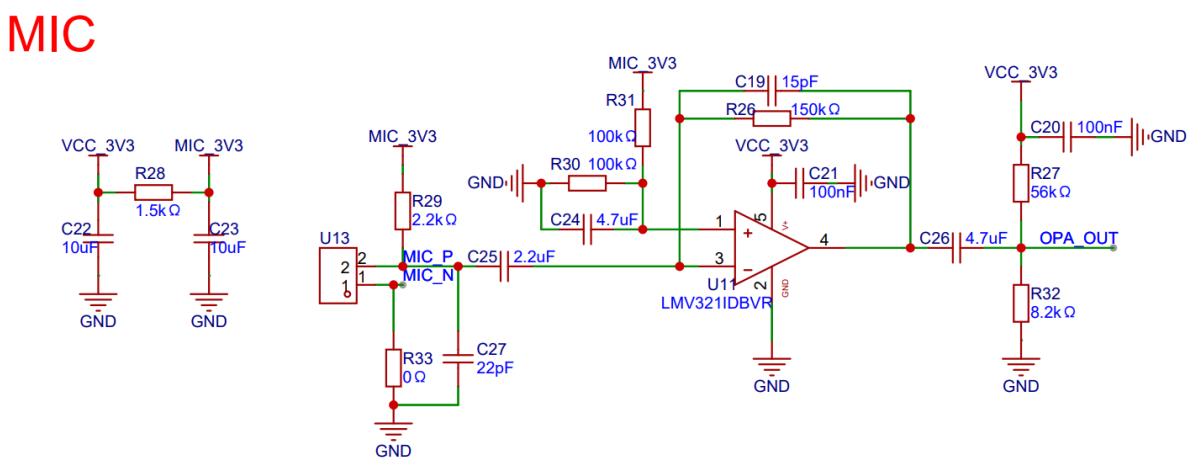


Fig. 2: ADC mic hardware reference design

6.3.3 External Configuration for ADC Continuous

You can initialize the ADC continuous mode externally and pass in an `adc_continuous_handle_t`.

```
adc_continuous_handle_t handle;
adc_continuous_handle_cfg_t adc_config = {
    .max_store_buf_size = adc_cfg->max_store_buf_size,
    .conv_frame_size = adc_cfg->conv_frame_size,
    .flags.flush_pool = true,
};

adc_continuous_new_handle(&adc_config, &handle);
```

6.3.4 Using with ADC Oneshot Mode

- If ADC oneshot and ADC continuous use different ADC units, there will be no conflicts.
- If they use the same ADC unit, make sure to call `esp_codec_dev_close()` to close `adc_mic` before calling `adc_oneshot_read`.

```
adc_continuous_start(handle);
adc_continuous_read();
adc_continuous_stop();
adc_oneshot_read();
```

6.3.5 Example Code

```
audio_codec_adc_cfg_t cfg = DEFAULT_AUDIO_CODEC_ADC_MONO_CFG(ADC_CHANNEL_0, 16000);
const audio_codec_data_if_t *adc_if = audio_codec_new_adc_data(&cfg);

esp_codec_dev_cfg_t codec_dev_cfg = {
    .dev_type = ESP_CODEC_DEV_TYPE_IN,
    .data_if = adc_if,
};
esp_codec_dev_handle_t dev = esp_codec_dev_new(&codec_dev_cfg);
esp_codec_dev_sample_info_t fs = {
    .sample_rate = 16000,
    .channel = 1,
    .bits_per_sample = 16,
```

(continues on next page)

(continued from previous page)

```
};
esp_codec_dev_open(dev, &fs);
uint16_t *audio_buffer = malloc(16000 * sizeof(uint16_t));
assert(audio_buffer);
while (1) {
    int ret = esp_codec_dev_read(dev, audio_buffer, sizeof(uint16_t) * 16000);
    ESP_LOGI(TAG, "esp_codec_dev_read ret: %d\n", ret);
}
```

6.3.6 API Reference

Header File

- components/audio/adc_mic/adc_mic.h

Functions

const audio_codec_data_if_t ***audio_codec_new_adc_data**(*audio_codec_adc_cfg_t* *adc_cfg)

Initialize codec adc.

Parameters **adc_cfg** –pointer of configuration struct
Returns const audio_codec_data_if_t* adc data interface.

Structures

struct **audio_codec_adc_cfg_t**

codec adc configuration

Public Members

adc_continuous_handle_t ***handle**

Type of adc continuous mode driver handle, if NULL will create new one internal, else will use the handle

size_t **max_store_buf_size**

Max length of the conversion results that driver can store, in bytes.

size_t **conv_frame_size**

Conversion frame size, in bytes. This should be in multiples of `SOC_ADC_DIGI_DATA_BYTES_PER_CONV`

adc_unit_t **unit_id**

ADC unit

uint8_t ***adc_channel_list**

Channel of ADC

uint8_t **adc_channel_num**

Number of channels

adc_atten_t **atten**

It should correspond to the actual range, with 0dB attenuation having the least ripple.

uint32_t **sample_rate_hz**

Sample rate of ADC

Macros

DEFAULT_AUDIO_CODEC_ADC_MONO_CFG(_channel, _sample)

DEFAULT_AUDIO_CODEC_ADC_STEREO_CFG(_channel_l, _channel_r, _sample)

Chapter 7

Audio/Video Codec

7.1 AVI Player

This component supports parsing AVI files and processes them at the FPS rate, providing audio and video data to the user through callbacks.

AVI (Audio Video Interleave) files are a multimedia container format used to store audio and video data. They can contain video and audio streams encoded in various formats and support different compression methods.

This component currently supports the following audio and video formats:

Video: - MJPEG - H264

Audio: - PCM

7.1.1 API Reference

Header File

- components/avi_player/include/avi_player.h

Functions

esp_err_t **`avi_player_play_from_memory`**(*avi_player_handle_t* handle, uint8_t *avi_data, size_t avi_size)

Plays an AVI file from memory. The buffer of the AVI will be passed through the set callback function.

This function initializes and plays an AVI file from a memory buffer.

Parameters

- **`handle`** –**[in]** AVI player handle
- **`avi_data`** –**[in]** Pointer to the AVI file data in memory.
- **`avi_size`** –**[in]** Size of the AVI file data in bytes.

Returns esp_err_t ESP_OK if successful, otherwise an error code.

esp_err_t **`avi_player_play_from_file`**(*avi_player_handle_t* handle, const char *filename)

Plays an AVI file from the filesystem. The buffer of the AVI will be passed through the set callback function.

This function initializes and plays an AVI file from the filesystem using its filename.

Parameters
- **handle** –**[in]** AVI player handle
- **filename** –**[in]** Path to the AVI file on the filesystem.

Returns esp_err_t ESP_OK if successful, otherwise an error code.

esp_err_t **avi_player_get_video_buffer**(*avi_player_handle_t* handle, void **buffer, size_t *buffer_size, *video_frame_info_t* *info, TickType_t ticks_to_wait)

Get one video frame from AVI stream.

Parameters
- **handle** –**[in]** AVI player handle
- **buffer** –**[out]** Pointer to external buffer to hold one frame
- **buffer_size** –**[inout]** Size of external buffer
- **info** –**[out]** Information of the video frame
- **ticks_to_wait** –**[in]** Maximum blocking time

Returns
- ESP_OK Success
- ESP_ERR_TIMEOUT Timeout
- ESP_ERR_INVALID_ARG NULL arguments
- ESP_ERR_NO_MEM External buffer not enough

esp_err_t **avi_player_get_audio_buffer**(*avi_player_handle_t* handle, void **buffer, size_t *buffer_size, *audio_frame_info_t* *info, TickType_t ticks_to_wait)

Get the audio buffer from AVI file.

Parameters
- **handle** –**[in]** AVI player handle
- **buffer** –**[out]** pointer to the audio buffer
- **buffer_size** –**[in]** size of the audio buffer
- **info** –**[out]** audio frame information
- **ticks_to_wait** –**[in]** maximum blocking time in ticks

Returns
- ESP_OK on success
- ESP_ERR_TIMEOUT if semaphore is not acquired before timeout
- ESP_ERR_INVALID_ARG if buffer or info is NULL or buffer_size is zero
- ESP_ERR_NO_MEM if buffer size is not enough

esp_err_t **avi_player_play_stop**(*avi_player_handle_t* handle)

Stop AVI player.

Parameters **handle** –**[in]** AVI player handle

Returns
- ESP_OK: Stop AVI player successfully
- ESP_ERR_INVALID_STATE: AVI player not playing

esp_err_t **avi_player_init**(*avi_player_config_t* config, *avi_player_handle_t* *handle)

Initialize the AVI player.

Parameters
- **config** –**[in]** Configuration of AVI player
- **handle** –**[out]** Pointer to store the AVI player handle

Returns
- ESP_OK: succeed
- ESP_ERR_NO_MEM: Cannot allocate memory for AVI player
- ESP_ERR_INVALID_STATE: AVI player has already been initialized

esp_err_t **avi_player_deinit**(*avi_player_handle_t* handle)

Deinitializes the AVI player.

This function deinitializes and cleans up resources used by the AVI player.

Parameters **handle** –**[in]** AVI player handle
Returns esp_err_t ESP_OK if successful, otherwise an error code.

Structures

struct **video_frame_info_t**

video frame info

Public Members

uint32_t **width**

Width of image in pixels

uint32_t **height**

Height of image in pixels

video_frame_format **frame_format**

Pixel data format

struct **audio_frame_info_t**

audio frame info

Public Members

uint8_t **channel**

Audio output channel

uint8_t **bits_per_sample**

Audio bits per sample

uint32_t **sample_rate**

Audio sample rate

audio_frame_format **format**

Audio format

struct **frame_data_t**

frame data

Public Members

uint8_t ***data**

Image data for this frame

size_t **data_bytes**

Size of image data buffer

frame_type_t **type**

Frame type: video or audio

video_frame_info_t **video_info**

Video frame info

audio_frame_info_t **audio_info**

Audio frame info

union *frame_data_t*::[anonymous] **[anonymous]**

frame info

struct **avi_player_config_t**

avi player config

Public Members

size_t **buffer_size**

Internal buffer size

video_write_cb **video_cb**

Video frame callback

audio_write_cb **audio_cb**

Audio frame callback

audio_set_clock_cb **audio_set_clock_cb**

Audio set clock callback

avi_play_end_cb **avi_play_end_cb**

AVI play end callback

UBaseType_t **priority**

FreeRTOS task priority

BaseType_t **coreID**

ESP32 core ID

void ***user_data**

User data

int **stack_size**

Stack size for the player task

bool **stack_in_psram**

If you read file/data from flash, do not set true

Type Definitions

typedef void (*`video_write_cb`)(*frame_data_t* *data, void *arg)

typedef void (*`audio_write_cb`)(*frame_data_t* *data, void *arg)

typedef void (*`audio_set_clock_cb`)(uint32_t rate, uint32_t bits_cfg, uint32_t ch, void *arg)

typedef void (*`avi_play_end_cb`)(void *arg)

typedef void *`avi_player_handle_t`

Enumerations

enum `video_frame_format`

video frame format

Values:

enumerator `FORMAT_MJEPG`

enumerator `FORMAT_H264`

enum `audio_frame_format`

audio frame format

Values:

enumerator `FORMAT_PCM`

enum `frame_type_t`

frame type: video or audio

Values:

enumerator `FRAME_TYPE_VIDEO`

enumerator `FRAME_TYPE_AUDIO`

Chapter 8

AI

8.1 OpenAI

The *openai* component is a porting version of OpenAI-ESP32 arduino library, which simplifies the process of invoking the OpenAI API on *ESP-IDF*.

This library supports the following OpenAI API capabilities:

- [ ] Responses[1]
- [x] Chat Completions: Supports text, image, and audio input; outputs text.[2] Streaming output is not currently supported.
- [x] Audio: Speech & Transcription. Speech supports streaming output.
- [x] Images: Create image, Edits & Variations
- [x] Embeddings
- [ ] Evals
- [ ] Fine-tuning
- [ ] Batch
- [ ] Files
- [x] Moderations

Note: Real-time capabilities can be implemented using the following libraries:

- **WebSocket solution**: - https://github.com/openai/openai-realtime-embedded
- **WebRTC solution**: - https://github.com/espressif/esp-webrtc-solution/tree/main/solutions/openai_demo

For more details, please refer to OpenAI API REFERENCE.

8.1.1 FAQ

Error *Failed to allocate request buffer!* when using *audioTranscription->file*

This error indicates that dynamic memory allocation failed, usually due to insufficient available memory. The required memory size depends on the current available RAM or PSRAM size.

Solutions

[1] Currently, please use the Chat Completions API instead.
[2] Multimodal input requires using `multiModalMessage`.

1. **Enable PSRAM**: If the device supports PSRAM, enable PSRAM to increase available memory and enable the *CONFIG_SPIRAM_USE_MALLOC* macro to use PSRAM.
2. **Reduce the audio data size**: For example, reduce the audio sampling rate or the length of the audio.

8.1.2 API Reference

Header File

- components/openai/include/OpenAI.h

Functions

OpenAI_t ***OpenAICreate** (const char *api_key)

Create an *OpenAI* object.

Parameters **api_key** –The key of openai
Returns OpenAI_t* The *OpenAI* object

void **OpenAIDelete** (*OpenAI_t* *oai)

Clear the *OpenAI* object and release resources.

Parameters **oai** –The *OpenAI* object

void **OpenAIChangeBaseURL** (*OpenAI_t* *oai, const char *baseURL)

Modify the Base URL of the *OpenAI* object.

Structures

struct **OpenAI_EmbeddingData_t**

Struct for Embedding data.

Public Members

uint32_t **len**

Length of the data

double ***data**

Pointer to the data

struct **OpenAI_EmbeddingResponse**

To get an embedding, send your text string to the embeddings API endpoint along with a choice of embedding model ID (e.g., text-embedding-ada-002). The response will contain an embedding, which you can extract, save, and use.

Public Members

uint32_t (***getUsage**)(struct *OpenAI_EmbeddingResponse* *stringResponse)

Get the usage of *OpenAI_EmbeddingResponse*.

Param stringResponse[in] The pointer to *OpenAI_EmbeddingResponse*
Return uint32_t

uint32_t (*`getLen`)(struct *OpenAI_EmbeddingResponse* *embeddingData)

Get the length of *OpenAI_EmbeddingResponse*.

Param embeddingData[in] The pointer to *OpenAI_EmbeddingResponse*
Return uint32_t

OpenAI_EmbeddingData_t *(*`getData`)(struct *OpenAI_EmbeddingResponse* *embeddingData, uint32_t index)

Get the data of *OpenAI_EmbeddingResponse*.

Param embeddingData[in] The pointer to *OpenAI_EmbeddingResponse*
Param index[in] The index of the data
Return OpenAI_EmbeddingData_t*

char *(*`getError`)(struct *OpenAI_EmbeddingResponse* *embeddingData)

Get the error of *OpenAI_EmbeddingResponse*.

Param embeddingData[in] The pointer to *OpenAI_EmbeddingResponse*
Return char*

void (*`deleteResponse`)(struct *OpenAI_EmbeddingResponse* *embeddingData)

delete the embedding response, should free it after use.

Param embeddingData[in] the point of *OpenAI_EmbeddingResponse*

struct `OpenAI_ModerationResponse`

The moderations endpoint is a tool you can use to check whether content complies with *OpenAI*' s usage policies. Developers can thus identify content that our usage policies prohibits and take action, for instance by filtering it.

Public Members

uint32_t (*`getLen`)(struct *OpenAI_ModerationResponse* *moderationResponse)

Get the length of *OpenAI_ModerationResponse*.

Param moderationResponse[in] The pointer to *OpenAI_ModerationResponse*
Return uint32_t

bool (*`getData`)(struct *OpenAI_ModerationResponse* *moderationResponse, uint32_t index)

Get the moderation result of *OpenAI_ModerationResponse*.

Param moderationResponse[in] The pointer to *OpenAI_ModerationResponse*
Param index[in] The index of the moderation result
Return bool

char *(*`getError`)(struct *OpenAI_ModerationResponse* *moderationResponse)

Get the error message of *OpenAI_ModerationResponse*.

Param moderationResponse[in] The pointer to *OpenAI_ModerationResponse*
Return char*

void (*`deleteResponse`)(struct *OpenAI_ModerationResponse* *moderationResponse)

delete the moderation response, should free it after use.

Param moderationResponse[in] the point of } OpenAI_ModerationResponse_t

struct **OpenAI_ImageResponse**

Save the image which is generated by *OpenAI*.

Public Members

uint32_t (***getLen**)(struct *OpenAI_ImageResponse* *imageResponse)

Get the length of *OpenAI_ImageResponse*.

Param imageResponse[in] The pointer to *OpenAI_ImageResponse*
Return uint32_t

char *(***getData**)(struct *OpenAI_ImageResponse* *imageResponse, uint32_t index)

Get the data of *OpenAI_ImageResponse*.

Param imageResponse[in] The pointer to *OpenAI_ImageResponse*
Param index[in] The index of the image data
Return char*

char *(***getError**)(struct *OpenAI_ImageResponse* *imageResponse)

Get the error message of *OpenAI_ImageResponse*.

Param imageResponse[in] The pointer to *OpenAI_ImageResponse*
Return char*

void (***deleteResponse**)(struct *OpenAI_ImageResponse* *imageResponse)

delete the image response

Param imageResponse[in] the point of } OpenAI_ImageResponse_t

struct **OpenAI_StringResponse**

Parse the returned json data into OpenAI_StringResponse_t.

Public Members

uint32_t (***getUsage**)(struct *OpenAI_StringResponse* *stringResponse)

get the usage of openai response

Param stringResponse[in] the point of OpenAI_StringResponse_t
Return uint32_t

uint32_t (***getLen**)(struct *OpenAI_StringResponse* *stringResponse)

get the len of openai response

Param stringResponse[in] the point of OpenAI_StringResponse_t
Return uint32_t

char *(***getData**)(struct *OpenAI_StringResponse* *stringResponse, uint32_t index)

get the data of openai response

Param stringResponse[in] the point of OpenAI_StringResponse_t
Param index[in] the index of data
Return char*

char *(*`getError`)(struct *OpenAI_StringResponse* *stringResponse)

get the error of openai response

Param stringResponse[in] the point of OpenAI_StringResponse_t
Return char*

void (*`deleteResponse`)(struct *OpenAI_StringResponse* *stringResponse)

delete the openai response

Param stringResponse[in] the point of OpenAI_StringResponse_t

struct `OpenAI_SpeechResponse`

Store the returned data into a OpenAI_SpeechResponse_t structure.

Public Members

uint32_t (*`getLen`)(struct *OpenAI_SpeechResponse* *SpeechResponse)

get the len of openai speech response

Param speechResponse[in] the point of OpenAI_SpeechResponse_t
Return uint32_t

char *(*`getData`)(struct *OpenAI_SpeechResponse* *SpeechResponse)

get the data of openai response

Param SpeechResponse[in] the point of OpenAI_SpeechResponse_t
Return char*

void (*`deleteResponse`)(struct *OpenAI_SpeechResponse* *SpeechResponse)

delete the openai response

Param SpeechResponse[in] the point of OpenAI_SpeechResponse_t

struct `OpenAI_Completion`

Given a prompt, the model will return one or more predicted completions, and can also return the probabilities of alternative tokens at each position.

Public Members

void (*`setModel`)(struct *OpenAI_Completion* *completion, const char *m)

Set the model to use for completion.

Param completion[in] the point of OpenAI_Completion_t
Param m[in] the name of the model to use for completion

void (*`setMaxTokens`)(struct *OpenAI_Completion* *completion, uint32_t mt)

Set the maximum number of tokens to generate in the completion.

Param completion[in] the point of OpenAI_Completion_t
Param mt[in] the maximum number of tokens to generate in the completion

void (*`setTemperature`)(struct *OpenAI_Completion* *completion, float t)

Set the temperature of the completion.

Param completion[in] the point of OpenAI_Completion_t
Param t[in] float between 0 and 2. Higher value gives more random results.

void (*`setTopP`)(struct *OpenAI_Completion* *completion, float tp)

Set the value of top_p for the completion.

Param completion[in] the point of OpenAI_Completion_t
Param tp[in] float between 0 and 1. recommended to alter this or temperature but not both.

void (*`setN`)(struct *OpenAI_Completion* *completion, uint32_t n)

Set the number of completions to generate for each prompt.

Param completion[in] the point of OpenAI_Completion_t
Param n[in] the number of completions to generate for each prompt

void (*`setEcho`)(struct *OpenAI_Completion* *completion, bool e)

Echo back the prompt in addition to the completion.

Param completion[in] the point of OpenAI_Completion_t
Param e[in] true if the prompt should be echoed back, false otherwise

void (*`setStop`)(struct *OpenAI_Completion* *completion, const char *s)

Set up to 4 sequences where the API will stop generating further tokens.

Param completion[in] the point of OpenAI_Completion_t
Param s[in] the sequences where the API will stop generating further tokens

void (*`setPresencePenalty`)(struct *OpenAI_Completion* *completion, float pp)

Set the presence penalty for the completion.

Param completion[in] the point of OpenAI_Completion_t
Param pp[in] float between -2.0 and 2.0. Positive values increase the model's likelihood to talk about new topics.

void (*`setFrequencyPenalty`)(struct *OpenAI_Completion* *completion, float fp)

Set the frequency penalty for the completion.

Param completion[in] the point of OpenAI_Completion_t
Param fp[in] float between -2.0 and 2.0. Positive values decrease the model's likelihood to repeat the same line verbatim.

void (*`setBestOf`)(struct *OpenAI_Completion* *completion, uint32_t bo)

Generates best_of completions server-side and returns the "best". "best_of" must be greater than "n".

Param completion[in] the point of OpenAI_Completion_t
Param bo[in] the number of best_of completions to generate server-side and return the "best"

void (*`setUser`)(struct *OpenAI_Completion* *completion, const char *u)

A unique identifier representing your end-user, which can help *OpenAI* to monitor and detect abuse.

Param completion[in] the point of OpenAI_Completion_t
Param u[in] the unique identifier representing your end-user

OpenAI_StringResponse_t *(***prompt**)(struct *OpenAI_Completion* *completion, char *p)

Send the prompt for completion.

Param completion[in] the point of OpenAI_Completion_t
Param p[in] the prompt for completion
Return OpenAI_StringResponse_t*

struct **OpenAI_ChatCompletion**

Given a list of messages comprising a conversation, the model will return a response.

Public Members

void (***setModel**)(struct *OpenAI_ChatCompletion* *chatCompletion, const char *m)

Set the model to use for completion.

Param chatCompletion[in] the point of *OpenAI_ChatCompletion*
Param m[in] the name of the model to use for chatCompletion

void (***setSystem**)(struct *OpenAI_ChatCompletion* *chatCompletion, const char *s)

Set the system to use for completion.

Param chatCompletion[in] the point of *OpenAI_ChatCompletion*
Param s[in] description of the required assistant

void (***setMaxTokens**)(struct *OpenAI_ChatCompletion* *chatCompletion, uint32_t mt)

Set the maximum number of tokens to generate in the completion.

Param chatCompletion[in] the point of *OpenAI_ChatCompletion*
Param mt[in] the maximum number of tokens to generate in the completion

void (***setTemperature**)(struct *OpenAI_ChatCompletion* *chatCompletion, float t)

Set the temperature for the completion.

Param chatCompletion[in] the point of *OpenAI_ChatCompletion*
Param t[in] float between 0 and 2. Higher value gives more random results.

void (***setTopP**)(struct *OpenAI_ChatCompletion* *chatCompletion, float tp)

Set the top_p for the completion.

Param chatCompletion[in] the point of *OpenAI_ChatCompletion*
Param tp[in] float between 0 and 1. recommended to alter this or temperature but not both.

void (***setStop**)(struct *OpenAI_ChatCompletion* *chatCompletion, const char *s)

Set up to 4 sequences where the API will stop generating further tokens.

Param chatCompletion[in] the point of *OpenAI_ChatCompletion*
Param s[in] the sequences where the API will stop generating further tokens

void (***setPresencePenalty**)(struct *OpenAI_ChatCompletion* *chatCompletion, float pp)

Set the presence penalty for the completion.

Param chatCompletion[in] the point of *OpenAI_ChatCompletion*
Param pp[in] float between -2.0 and 2.0. Positive values increase the model' s likelihood to talk about new topics.

void (*`setFrequencyPenalty`)(struct *OpenAI_ChatCompletion* *chatCompletion, float fp)

Set the frequency penalty for the completion.

Param chatCompletion[in] the point of *OpenAI_ChatCompletion*
Param fp[in] float between -2.0 and 2.0. Positive values decrease the model's likelihood to repeat the same line verbatim.

void (*`setUser`)(struct *OpenAI_ChatCompletion* *chatCompletion, const char *u)

A unique identifier representing your end-user, which can help *OpenAI* to monitor and detect abuse.

Param chatCompletion[in] the point of *OpenAI_ChatCompletion*
Param u[in] the unique identifier representing your end-user

void (*`clearConversation`)(struct *OpenAI_ChatCompletion* *chatCompletion)

Clears the accumulated conversation.

Param chatCompletion[in] the point of *OpenAI_ChatCompletion*

OpenAI_StringResponse_t *(*`message`)(struct *OpenAI_ChatCompletion* *chatCompletion, const char *p, bool save)

Send the message for completion. Save it with the first response if selected.

Param chatCompletion[in] the point of *OpenAI_ChatCompletion*
Param p[in] the message for completion
Param save[in] save it with the first response if selected
Return OpenAI_StringResponse_t*

OpenAI_StringResponse_t *(*`multiModalMessage`)(struct *OpenAI_ChatCompletion* *chatCompletion, const char *type, const char *p, bool save)

Send a multi-modal message (text / image_url / input_audio) for completion. This is an extended API supporting richer content types.

Param chatCompletion[in] The chat completion object
Param type[in] The content type string ("text", "image_url", "input_audio", etc.)
Param p[in] The content value corresponding to the type
Param save[in] Save conversation with the first assistant response if true
Return OpenAI_StringResponse_t* The response from *OpenAI*

struct `OpenAI_Edit`

Given a prompt and an instruction, the model will return an edited version of the prompt.

Public Members

void (*`setModel`)(struct *OpenAI_Edit* *edit, const char *m)

Set the model to use for edit.

Param edit[in] the point of OpenAI_Edit_t
Param m[in] the name of the model to use for edit

void (*`setTemperature`)(struct *OpenAI_Edit* *edit, float t)

Set the temperature for the edit.

Param edit[in] the point of OpenAI_Edit_t
Param t[in] float between 0 and 2. Higher value gives more random results.

void (*`setTopP`)(struct *OpenAI_Edit* *edit, float tp)

Set the top_p for the edit.

Param edit[in] the point of OpenAI_Edit_t
Param tp[in] float between 0 and 1. recommended to alter this or temperature but not both.

void (*`setN`)(struct *OpenAI_Edit* *edit, uint32_t n)

Set the number of edits to generate for the input and instruction.

Param edit[in] the point of OpenAI_Edit_t
Param n[in] the number of edits to generate for the input and instruction

OpenAI_StringResponse_t *(*`process`)(struct *OpenAI_Edit* *edit, char *instruction, char *input)

Creates a new edit for the provided input, instruction, and parameters.

Param edit[in] the point of OpenAI_Edit_t
Param instruction[in] the instruction for the edit
Param input[in] the input text to be edited
Return OpenAI_StringResponse_t* the edited text

struct `OpenAI_ImageGeneration`

Creates an image given a prompt.

Public Members

void (*`setSize`)(struct *OpenAI_ImageGeneration* *imageGeneration, *OpenAI_Image_Size* s)

Set the size of the generated images.

Param imageGeneration[in] the point of *OpenAI_ImageGeneration*
Param s[in] the size of the generated images

void (*`setResponseFormat`)(struct *OpenAI_ImageGeneration* *imageGeneration, *OpenAI_Image_Response_Format* rf)

Set the format in which the generated images are returned.

Param imageGeneration[in] the point of *OpenAI_ImageGeneration*
Param rf[in] the format in which the generated images are returned

void (*`setN`)(struct *OpenAI_ImageGeneration* *imageGeneration, uint32_t n)

Set the number of images to generate. Must be between 1 and 10.

Param imageGeneration[in] the point of *OpenAI_ImageGeneration*
Param n[in] the number of images to generate

void (*`setUser`)(struct *OpenAI_ImageGeneration* *imageGeneration, const char *u)

Set a unique identifier representing your end-user, which can help *OpenAI* to monitor and detect abuse.

Param imageGeneration[in] the point of *OpenAI_ImageGeneration*
Param u[in] the unique identifier representing your end-user

OpenAI_ImageResponse_t *(*`prompt`)(struct *OpenAI_ImageGeneration* *imageGeneration, char *p)

Creates image/images from given a prompt.

Param imageGeneration[in] the point of *OpenAI_ImageGeneration*
Param p[in] the prompt for image generation
Return OpenAI_ImageResponse_t* the generated image/images

struct **OpenAI_ImageVariation**

Creates a variation of a given image.

Public Members

void (***setSize**)(struct *OpenAI_ImageVariation* *imageVariation, *OpenAI_Image_Size* s)

Set the size of the generated images.

Param imageVariation[in] the point of *OpenAI_ImageVariation*
Param s[in] the size of the generated images

void (***setResponseFormat**)(struct *OpenAI_ImageVariation* *imageVariation, *OpenAI_Image_Response_Format* rf)

Set the format in which the generated images are returned.

Param imageVariation[in] the point of *OpenAI_ImageVariation*
Param rf[in] the format in which the generated images are returned

void (***setN**)(struct *OpenAI_ImageVariation* *imageVariation, uint32_t n)

Set the number of images to generate. Must be between 1 and 10.

Param imageVariation[in] the point of *OpenAI_ImageVariation*
Param n[in] the number of images to generate

void (***setUser**)(struct *OpenAI_ImageVariation* *imageVariation, const char *u)

Set a unique identifier representing your end-user, which can help *OpenAI* to monitor and detect abuse.

Param imageVariation[in] the point of *OpenAI_ImageVariation*
Param u[in] the unique identifier representing your end-user

OpenAI_ImageResponse_t *(***image**)(struct *OpenAI_ImageVariation* *imageVariation, uint8_t *data, size_t len)

Creates an image variation from given image data.

Param imageVariation[in] the point of *OpenAI_ImageVariation*
Param data[in] the input image data
Param len[in] the length of the input image data
Return OpenAI_ImageResponse_t* the generated image variation

struct **OpenAI_ImageEdit**

Creates an edited or extended image given an original image and a prompt.

Public Members

void (***setPrompt**)(struct *OpenAI_ImageEdit* *imageEdit, const char *p)

Set the prompt for the image edit.

Param imageEdit[in] the point of OpenAI_ImageEdit_t
Param p[in] the prompt for the image edit

void (***setSize**)(struct *OpenAI_ImageEdit* *imageEdit, *OpenAI_Image_Size* s)

Set the size of the generated images.

Param imageEdit[in] the point of OpenAI_ImageEdit_t

Param s[in] the size of the generated images

void (*`setResponseFormat`)(struct *OpenAI_ImageEdit* *imageEdit, *OpenAI_Image_Response_Format* rf)

Set the format in which the generated images are returned.

Param imageEdit[in] the point of OpenAI_ImageEdit_t
Param rf[in] the format in which the generated images are returned

void (*`setN`)(struct *OpenAI_ImageEdit* *imageEdit, uint32_t n)

Set the number of images to generate. Must be between 1 and 10.

Param imageEdit[in] the point of OpenAI_ImageEdit_t
Param n[in] the number of images to generate

void (*`setUser`)(struct *OpenAI_ImageEdit* *imageEdit, const char *u)

Set a unique identifier representing your end-user, which can help *OpenAI* to monitor and detect abuse.

Param imageEdit[in] the point of OpenAI_ImageEdit_t
Param u[in] the unique identifier representing your end-user

OpenAI_ImageResponse_t *(*`image`)(struct *OpenAI_ImageEdit* *imageEdit, uint8_t *data, size_t len, uint8_t *mask_data, size_t mask_len)

Creates an edited or extended image given an original image, a mask, and a prompt.

Param imageEdit[in] the point of OpenAI_ImageEdit_t
Param data[in] the input image data
Param len[in] the length of the input image data
Param mask_data[in] the input mask data
Param mask_len[in] the length of the input mask data
Return OpenAI_ImageResponse_t* the edited or extended image

struct **`OpenAI_AudioTranscription`**

Transcribes audio into the input language.

Public Members

void (*`setPrompt`)(struct *OpenAI_AudioTranscription* *audioTranscription, const char *p)

Set the prompt for the audio transcription.

Param audioTranscription[in] the point of OpenAI_AudioTranscription_t
Param p[in] the prompt for the audio transcription

void (*`setResponseFormat`)(struct *OpenAI_AudioTranscription* *audioTranscription, *OpenAI_Audio_Response_Format* rf)

Set the format of the transcript output.

Param audioTranscription[in] the point of OpenAI_AudioTranscription_t
Param rf[in] the format of the transcript output

void (*`setTemperature`)(struct *OpenAI_AudioTranscription* *audioTranscription, float t)

Set the temperature for the audio transcription.

Param audioTranscription[in] the point of OpenAI_AudioTranscription_t
Param t[in] float between 0 and 1

void (*`setLanguage`)(struct *OpenAI_AudioTranscription* *audioTranscription, const char *l)

Set the language of the input audio.

Param audioTranscription[in] the point of OpenAI_AudioTranscription_t
Param l[in] the language in ISO-639-1 format of the input audio. NULL for Auto.

char *(*`file`)(struct *OpenAI_AudioTranscription* *audioTranscription, uint8_t *data, size_t len, *OpenAI_Audio_Input_Format* f)

Transcribe an audio file.

Param audioTranscription[in] the point of OpenAI_AudioTranscription_t
Param data[in] the input audio data
Param len[in] the length of the input audio data
Param f[in] the format of the input audio data
Return char* the transcribed text, you should free it after use.

struct `OpenAI_AudioSpeech`

Given a list of messages comprising a conversation, the model will return a response.

Public Members

void (*`setModel`)(struct *OpenAI_AudioSpeech* *createSpeech, const char *m)

Set the model to use for completion.

Param createSpeech[in] the point of OpenAI_SpeechResponse_t
Param m[in] the name of the model to use for audio response

void (*`setVoice`)(struct *OpenAI_AudioSpeech* *createSpeech, const char *m)

Set the voice to use for completion.

Param createSpeech[in] the point of OpenAI_SpeechResponse_t
Param m[in] the name of the model to use for audio response

void (*`setSpeed`)(struct *OpenAI_AudioSpeech* *createSpeech, float t)

Set the speed of the output audio.

Param createSpeech[in] the point of OpenAI_SpeechResponse_t
Param t[in] float between 0.25 to 4.0

void (*`setResponseFormat`)(struct *OpenAI_AudioSpeech* *createSpeech, *OpenAI_Audio_Output_Format* rf)

Set the format of the output.

Param createSpeech[in] the point of OpenAI_SpeechResponse_t
Param rf[in] the format of the output audio

OpenAI_SpeechResponse_t *(*`speech`)(struct *OpenAI_AudioSpeech* *createSpeech, char *p)

Send the message for completion.

Param createSpeech[in] the point of OpenAI_SpeechResponse_t
Param p[in] the message for audio generation
Return *

void (*`speechStream`)(struct *OpenAI_AudioSpeech* *createSpeech, char *p, *OpenAI_StreamCallback* stream_callback)

Send the message for completion.

Param createSpeech[in] the point of OpenAI_SpeechResponse_t
Param p[in] the message for audio generation
Param stream_callback[in] the callback function for audio stream

struct **`OpenAI_AudioTranslation`**

Translates audio into English.

Public Members

void (*`setPrompt`)(struct *OpenAI_AudioTranslation* *audioTranslation, const char *p)

Set the prompt for the audio translation.

Param audioTranslation[in] the point of OpenAI_AudioTranslation_t
Param p[in] the prompt for the audio translation

void (*`setResponseFormat`)(struct *OpenAI_AudioTranslation* *audioTranslation, *OpenAI_Audio_Response_Format* rf)

Set the format of the transcript output.

Param audioTranslation[in] the point of OpenAI_AudioTranslation_t
Param rf[in] the format of the transcript output

void (*`setTemperature`)(struct *OpenAI_AudioTranslation* *audioTranslation, float t)

Set the temperature for the audio translation.

Param audioTranslation[in] the point of OpenAI_AudioTranslation_t
Param t[in] float between 0 and 2. Higher value gives more random results.

char *(*`file`)(struct *OpenAI_AudioTranslation* *audioTranslation, uint8_t *data, size_t len, *OpenAI_Audio_Input_Format* f)

Transcribe and translate an audio file into English.

Param audioTranslation[in] the point of OpenAI_AudioTranslation_t
Param data[in] the input audio data
Param len[in] the length of the input audio data
Param f[in] the format of the input audio data
Return char* the translated text in English, you should free it after use.

struct **`OpenAI`**

The entry point for calling the Openai api.

Type Definitions

typedef void (*`OpenAI_StreamCallback`)(const uint8_t *data, size_t length)

typedef struct *OpenAI_EmbeddingResponse* **`OpenAI_EmbeddingResponse_t`**

To get an embedding, send your text string to the embeddings API endpoint along with a choice of embedding model ID (e.g., text-embedding-ada-002). The response will contain an embedding, which you can extract, save, and use.

typedef struct *OpenAI_ModerationResponse* **`OpenAI_ModerationResponse_t`**

The moderations endpoint is a tool you can use to check whether content complies with *OpenAI*' s usage policies. Developers can thus identify content that our usage policies prohibits and take action, for instance by filtering it.

typedef struct *OpenAI_ImageResponse* **OpenAI_ImageResponse_t**

Save the image which is generated by *OpenAI*.

typedef struct *OpenAI_StringResponse* **OpenAI_StringResponse_t**

Parse the returned json data into OpenAI_StringResponse_t.

typedef struct *OpenAI_SpeechResponse* **OpenAI_SpeechResponse_t**

Store the returned data into a OpenAI_SpeechResponse_t structure.

typedef struct *OpenAI_Completion* **OpenAI_Completion_t**

Given a prompt, the model will return one or more predicted completions, and can also return the probabilities of alternative tokens at each position.

typedef struct *OpenAI_ChatCompletion* **OpenAI_ChatCompletion_t**

Given a list of messages comprising a conversation, the model will return a response.

typedef struct *OpenAI_Edit* **OpenAI_Edit_t**

Given a prompt and an instruction, the model will return an edited version of the prompt.

typedef struct *OpenAI_ImageGeneration* **OpenAI_ImageGeneration_t**

Creates an image given a prompt.

typedef struct *OpenAI_ImageVariation* **OpenAI_ImageVariation_t**

Creates a variation of a given image.

typedef struct *OpenAI_ImageEdit* **OpenAI_ImageEdit_t**

Creates an edited or extended image given an original image and a prompt.

typedef struct *OpenAI_AudioTranscription* **OpenAI_AudioTranscription_t**

Transcribes audio into the input language.

typedef struct *OpenAI_AudioSpeech* **OpenAI_AudioSpeech_t**

Given a list of messages comprising a conversation, the model will return a response.

typedef struct *OpenAI_AudioTranslation* **OpenAI_AudioTranslation_t**

Translates audio into English.

typedef struct *OpenAI* **OpenAI_t**

The entry point for calling the Openai api.

Enumerations

enum **OpenAI_Image_Size**

Values:

enumerator **OPENAI_IMAGE_SIZE_1024x1024**

enumerator **OPENAI_IMAGE_SIZE_512x512**

enumerator **OPENAI_IMAGE_SIZE_256x256**

enum **OpenAI_Image_Response_Format**

Values:

enumerator **OPENAI_IMAGE_RESPONSE_FORMAT_URL**

enumerator **OPENAI_IMAGE_RESPONSE_FORMAT_B64_JSON**

enum **OpenAI_Audio_Response_Format**

Values:

enumerator **OPENAI_AUDIO_RESPONSE_FORMAT_JSON**

enumerator **OPENAI_AUDIO_RESPONSE_FORMAT_TEXT**

enumerator **OPENAI_AUDIO_RESPONSE_FORMAT_SRT**

enumerator **OPENAI_AUDIO_RESPONSE_FORMAT_VERBOSE_JSON**

enumerator **OPENAI_AUDIO_RESPONSE_FORMAT_VTT**

enum **OpenAI_Audio_Input_Format**

Values:

enumerator **OPENAI_AUDIO_INPUT_FORMAT_MP3**

enumerator **OPENAI_AUDIO_INPUT_FORMAT_MP4**

enumerator **OPENAI_AUDIO_INPUT_FORMAT_MPEG**

enumerator **OPENAI_AUDIO_INPUT_FORMAT_MPGA**

enumerator **OPENAI_AUDIO_INPUT_FORMAT_M4A**

enumerator **OPENAI_AUDIO_INPUT_FORMAT_WAV**

enumerator **OPENAI_AUDIO_INPUT_FORMAT_WEBM**

enum **OpenAI_Audio_Output_Format**

Values:

enumerator **OPENAI_AUDIO_OUTPUT_FORMAT_MP3**

enumerator **OPENAI_AUDIO_OUTPUT_FORMAT_OPUS**

enumerator **OPENAI_AUDIO_OUTPUT_FORMAT_AAC**

enumerator **OPENAI_AUDIO_OUTPUT_FORMAT_FLAC**

enumerator **OPENAI_AUDIO_OUTPUT_FORMAT_WAV**

enumerator **OPENAI_AUDIO_OUTPUT_FORMAT_PCM**

enumerator **OPENAI_AUDIO_OUTPUT_FORMAT_MAX**

8.2 Model Context Protocol (MCP)

Supported chips	ESP32	ESP32-C2	ESP32-C3	ESP32-C6	ESP32-S3

MCP component provides a simplified API interface for accessing common MCP functions. It supports HTTP and other common scenarios.

8.2.1 Examples

1. MCP Server Example: mcp/mcp_server. Runs an MCP server on ESP32 over HTTP, exposing JSON-RPC tools for simple device controls (volume, brightness, theme, HSV/RGB color).
2. MCP Client Example: mcp/mcp_client. Runs an MCP client on ESP32 over HTTP, accessing a remote MCP server.

8.2.2 API Reference

Header File

- components/mcp-c-sdk/include/esp_mcp_mgr.h

Functions

esp_err_t **esp_mcp_mgr_init** (*esp_mcp_mgr_config_t* config, *esp_mcp_mgr_handle_t* *handle)

Initialize MCP transport manager.

Initializes an MCP (Model Context Protocol) transport manager with the specified transport (HTTP, BLE, etc.). The transport manager handles communication between ESP32 devices and AI applications using the MCP protocol.

Parameters

- **config** –**[in]** MCP transport manager configuration (transport, config, server instance)
- **handle** –**[out]** Pointer to store the created MCP transport manager handle

Returns

- ESP_OK: MCP transport manager initialized successfully
- ESP_ERR_INVALID_ARG: Invalid arguments (NULL handle or invalid transport functions)
- ESP_ERR_NO_MEM: Out of memory

esp_err_t **esp_mcp_mgr_deinit** (*esp_mcp_mgr_handle_t* handle)

Deinitialize MCP transport manager.

Deinitializes the MCP transport manager and releases all associated resources, including transport instances and configurations. The transport must be stopped (via esp_mcp_mgr_stop()) before deinitialization.

Parameters **handle** –**[in]** MCP transport manager handle

Returns

- ESP_OK: MCP transport manager deinitialized successfully
- ESP_ERR_INVALID_ARG: Invalid handle

esp_err_t **esp_mcp_mgr_start** (*esp_mcp_mgr_handle_t* handle)

Start MCP transport layer.

Starts the underlying transport layer (e.g., HTTP server, BLE service) to begin accepting MCP protocol messages from AI applications. The transport manager must be initialized before calling this function.

Parameters **handle** –**[in]** MCP transport manager handle

Returns

- ESP_OK: Transport layer started successfully
- ESP_ERR_INVALID_ARG: Invalid handle
- ESP_ERR_FAIL: Failed to start transport layer (e.g., port already in use)

esp_err_t **esp_mcp_mgr_stop** (*esp_mcp_mgr_handle_t* handle)

Stop MCP transport layer.

Stops the underlying transport layer (e.g., HTTP server, BLE service) and stops accepting new MCP protocol messages. Existing connections may be gracefully closed.

Parameters **handle** –**[in]** MCP transport manager handle

Returns

- ESP_OK: Transport layer stopped successfully
- ESP_ERR_INVALID_ARG: Invalid handle

esp_err_t **esp_mcp_mgr_register_endpoint** (*esp_mcp_mgr_handle_t* handle, const char *ep_name, void *priv_data)

Register an MCP endpoint.

Registers an endpoint (e.g., "tools/list", "tools/call") for routing MCP protocol messages. Endpoints are used to organize different types of MCP operations. The endpoint will be accessible via the transport layer (e.g., HTTP path).

Parameters

- **handle** –**[in]** MCP transport manager handle
- **ep_name** –**[in]** Endpoint name (e.g., "mcp_server", "tools/list")
- **priv_data** –**[in]** Private data passed to the endpoint handler (typically NULL or MCP engine instance)

Returns

- ESP_OK: Endpoint registered successfully
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_INVALID_STATE: Endpoint already exists
- ESP_ERR_NO_MEM: Memory allocation failed

esp_err_t **esp_mcp_mgr_unregister_endpoint** (*esp_mcp_mgr_handle_t* handle, const char *ep_name)

Unregister an MCP endpoint.

Removes an endpoint from the MCP transport manager. The endpoint will no longer accept MCP protocol messages. All associated resources are freed.

Parameters

- **handle** –**[in]** MCP transport manager handle
- **ep_name** –**[in]** Endpoint name to remove

Returns

- ESP_OK: Endpoint removed successfully

- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_NOT_FOUND: Endpoint not found

esp_err_t **esp_mcp_mgr_req_handle** (*esp_mcp_mgr_handle_t* handle, const char *ep_name, const uint8_t *inbuf, uint16_t inlen, uint8_t **outbuf, uint16_t *outlen)

Handle an MCP protocol request.

Processes an incoming MCP protocol message (JSON-RPC 2.0 format) and routes it to the appropriate endpoint handler. The handler parses the message, executes the requested MCP operation (e.g., initialize, tools/list, tools/call), and returns the response. This function is typically called by transport layer implementations (e.g., HTTP handler) when receiving MCP protocol messages from AI applications.

Note: The caller is responsible for freeing the outbuf using cJSON_free() after sending the response.

Note: If the function returns an error, outbuf may be NULL or contain an error response.

Parameters

- **handle** –**[in]** MCP transport manager handle
- **ep_name** –**[in]** Endpoint name to route the request to (e.g., "mcp_server")
- **inbuf** –**[in]** Input buffer containing the MCP protocol message (JSON-RPC 2.0 format, null-terminated)
- **inlen** –**[in]** Length of the input buffer in bytes
- **outbuf** –**[out]** Pointer to store the output buffer containing the MCP response JSON (caller must free with cJSON_free after use)
- **outlen** –**[out]** Pointer to store the output buffer length in bytes

Returns

- ESP_OK: Request handled successfully, response available in outbuf
- ESP_ERR_INVALID_ARG: Invalid arguments (NULL pointer or empty endpoint name)
- ESP_ERR_NOT_FOUND: Endpoint not found
- ESP_FAIL: Failed to process MCP request (parse error, tool execution error, etc.)

esp_err_t **esp_mcp_mgr_perform_handle** (*esp_mcp_mgr_handle_t* handle, const uint8_t *inbuf, uint16_t inlen)

Perform an outbound MCP request via the configured transport (client-side)

Sends a JSON-RPC request to a remote MCP server endpoint over the configured transport. The transport must implement the optional `transport.request` callback.

Parameters

- **handle** –**[in]** MCP transport manager handle
- **inbuf** –**[in]** Input buffer containing the MCP protocol message (JSON-RPC request)
- **inlen** –**[in]** Length of the input buffer in bytes

Returns

- ESP_OK: Request performed successfully (fire-and-forget)
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_NOT_SUPPORTED: Transport does not support outbound requests
- Other: Transport-specific failure

esp_err_t **esp_mcp_mgr_post_info_init** (*esp_mcp_mgr_handle_t* handle, const *esp_mcp_mgr_req_t* *req)

Build and send initialize request to remote MCP server.

Constructs and sends an MCP initialize request to establish a session with a remote MCP server. The response will be delivered via the callback specified in the request structure.

Parameters

- **handle** –**[in]** MCP transport manager handle

- **req** –**[in]** Request structure containing endpoint name, callback, and initialization parameters (protocol_version, name, version)

Returns

- ESP_OK: Request sent successfully
- ESP_ERR_INVALID_ARG: Invalid arguments (NULL handle or request)
- ESP_ERR_NOT_SUPPORTED: Transport does not support outbound requests
- Other: Transport-specific failure

esp_err_t **esp_mcp_mgr_post_tools_list**(*esp_mcp_mgr_handle_t* handle, const *esp_mcp_mgr_req_t* *req)

Build and send tools/list request to remote MCP server.

Constructs and sends an MCP tools/list request to retrieve available tools from a remote MCP server. Supports cursor-based pagination. The response will be delivered via the callback specified in the request structure.

Parameters

- **handle** –**[in]** MCP transport manager handle
- **req** –**[in]** Request structure containing endpoint name, callback, and optional cursor for pagination

Returns

- ESP_OK: Request sent successfully
- ESP_ERR_INVALID_ARG: Invalid arguments (NULL handle or request)
- ESP_ERR_NOT_SUPPORTED: Transport does not support outbound requests
- Other: Transport-specific failure

esp_err_t **esp_mcp_mgr_post_tools_call**(*esp_mcp_mgr_handle_t* handle, const *esp_mcp_mgr_req_t* *req)

Build and send tools/call request to remote MCP server.

Constructs and sends an MCP tools/call request to execute a tool on a remote MCP server. The tool name and arguments (as JSON string) must be provided. The response will be delivered via the callback specified in the request structure.

Parameters

- **handle** –**[in]** MCP transport manager handle
- **req** –**[in]** Request structure containing endpoint name, callback, tool name, and arguments JSON

Returns

- ESP_OK: Request sent successfully
- ESP_ERR_INVALID_ARG: Invalid arguments (NULL handle, request, or missing tool_name)
- ESP_ERR_NOT_SUPPORTED: Transport does not support outbound requests
- Other: Transport-specific failure

esp_err_t **esp_mcp_mgr_req_destroy_response**(*esp_mcp_mgr_handle_t* handle, uint8_t *response_buffer)

Destroy a MCP response buffer.

Destroys a MCP response buffer and frees the memory. This function uses the message ID from the last request processed by the handler.

Note: This function uses the msg_id from the last request. For more precise control, use esp_mcp_mgr_req_destroy_response_by_id() instead.

Parameters

- **handle** –**[in]** MCP transport manager handle
- **response_buffer** –**[in]** MCP response buffer to destroy (currently unused, kept for API compatibility)

Returns

- ESP_OK: Response buffer destroyed successfully

- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_NOT_FOUND: Message ID not found

esp_err_t **esp_mcp_mgr_req_destroy_response_by_id**(*esp_mcp_mgr_handle_t* handle, uint16_t id)

Destroy a MCP response buffer by message ID.

Destroys a MCP response buffer for the specified message ID and frees the memory. This is the recommended way to free response buffers when you know the message ID.

Note: This function should be called after sending the response to prevent memory leaks. Each handled request may allocate a response buffer that must be freed using this function or esp_mcp_mgr_req_destroy_response().

Parameters

- **handle** –**[in]** MCP transport manager handle
- **id** –**[in]** Message ID of the response to destroy

Returns

- ESP_OK: Response buffer destroyed successfully
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_NOT_FOUND: Message ID not found

Structures

struct **esp_mcp_mgr_req_t**

MCP request structure.

Structure for transmitting MCP requests.

Public Members

const char ***ep_name**

Endpoint name

esp_mcp_mgr_resp_cb_t **cb**

Response callback

void ***user_ctx**

User context

const char ***protocol_version**

Protocol version string, default "2024-11-05" if NULL

const char ***name**

Client name

const char ***version**

Client version

struct *esp_mcp_mgr_req_t*::[anonymous]::[anonymous] **init**

Parameters for initialize request

const char ***cursor**

Cursor

struct *esp_mcp_mgr_req_t*::[anonymous]::[anonymous] **list**

Parameters for tools/list request

const char ***tool_name**

Tool name

const char ***args_json**

Tool arguments JSON

struct *esp_mcp_mgr_req_t*::[anonymous]::[anonymous] **call**

Parameters for tools/call request

union *esp_mcp_mgr_req_t*::[anonymous] **u**

Union containing request-specific parameters

struct **esp_mcp_transport_s**

MCP transport function table.

Structure defining the function callbacks for a specific MCP transport implementation. Each transport (HTTP, BLE, etc.) provides its own implementation of these functions to handle protocol-specific operations like starting/stopping the transport layer and managing endpoints for MCP message routing.

Public Members

esp_err_t (***init**)(*esp_mcp_mgr_handle_t* handle, *esp_mcp_transport_handle_t* *transport_handle)

Create a new transport instance

esp_err_t (***deinit**)(*esp_mcp_transport_handle_t* handle)

Destroy a transport instance

esp_err_t (***start**)(*esp_mcp_transport_handle_t* handle, void *config)

Start the transport layer (e.g., start HTTP server)

esp_err_t (***stop**)(*esp_mcp_transport_handle_t* handle)

Stop the transport layer (e.g., stop HTTP server)

esp_err_t (***create_config**)(const void *config, void **config_out)

Create transport-specific configuration

esp_err_t (***delete_config**)(void *config)

Destroy transport-specific configuration

esp_err_t (***register_endpoint**)(*esp_mcp_transport_handle_t* handle, const char *ep_name, void *priv_data)

Register an MCP endpoint for message routing

esp_err_t (*`unregister_endpoint`)(*esp_mcp_transport_handle_t* handle, const char *ep_name)

Unregister an MCP endpoint

esp_err_t (*`request`)(*esp_mcp_transport_handle_t* handle, const uint8_t *inbuf, uint16_t inlen)

Perform an outbound MCP request (client-side), optional, fire-and-forget

struct **`esp_mcp_mgr_config_t`**

MCP transport manager configuration structure.

Configuration structure for initializing an MCP transport manager instance. Specifies the transport to use (HTTP, BLE, etc.) and associated configuration.

Public Members

esp_mcp_transport_t **`transport`**

Transport function table (e.g., esp_mcp_transport_http)

void ***`config`**

Transport-specific configuration (e.g., HTTP server port, BLE service UUID)

void ***`instance`**

MCP instance context (esp_mcp_t *) for handling MCP protocol messages

Type Definitions

typedef uint32_t **`esp_mcp_mgr_handle_t`**

MCP transport manager handle type.

Opaque handle representing an initialized MCP (Model Context Protocol) transport manager instance. The transport manager handles communication between ESP32 devices and AI applications using the MCP protocol over various transport layers (HTTP, BLE, etc.). Created by esp_mcp_mgr_init() and used for all subsequent MCP transport operations.

typedef uint32_t **`esp_mcp_transport_handle_t`**

MCP transport handle type.

Opaque handle representing an initialized MCP transport instance. A transport defines how MCP protocol messages are transmitted (e.g., HTTP, BLE). Created by the transport ' s new() function and used for transport-specific operations.

typedef esp_err_t (*`esp_mcp_mgr_resp_cb_t`)(int error_code, const char *ep_name, const char *resp_json, void *user_ctx)

Response callback type for outbound requests.

This callback is invoked when a response is received for an outbound MCP request. The meaning of parameters depends on the response type:

Note: The callback is called synchronously. The ep_name and resp_json pointers are valid only during the callback execution. The framework will free these resources after the callback returns. The callback should NOT attempt to free these strings.

Param error_code Error indicator or error code, meaning depends on resp_json:
- If resp_json is not NULL and contains application result:
  - 0: Application-level success (result.isError = false or not present)
  - 1: Application-level error (result.isError = true)
- If resp_json is not NULL and contains protocol error:
  - JSON-RPC error code (negative integer, e.g., -32601 for "Method not found")
  - See MCP_ERROR_CODE_* constants in esp_mcp_error.h
- If resp_json is NULL (parse failure):
  - esp_err_t error code (e.g., ESP_ERR_INVALID_ARG, ESP_ERR_NO_MEM)

Param ep_name Endpoint name or tool name (may be NULL); valid only during callback execution, do NOT free

Param resp_json Response JSON body, meaning depends on error_code:
- If error_code is 0 or 1: Contains result object JSON (application-level response)
- If error_code is negative (JSON-RPC error code): Contains error message string (protocol-level error)
- If error_code is positive esp_err_t: NULL (parse failure) Valid only during callback execution, do NOT free

Param user_ctx User context provided when sending the request

Return ESP_OK on success, or error code if callback processing failed

typedef struct *esp_mcp_transport_s* **`esp_mcp_transport_t`**

MCP transport function table.

Structure defining the function callbacks for a specific MCP transport implementation. Each transport (HTTP, BLE, etc.) provides its own implementation of these functions to handle protocol-specific operations like starting/stopping the transport layer and managing endpoints for MCP message routing.

8.3 Touch Digit Recognition

8.3.1 Touch Principle and Data Acquisition

Touch Principle

The ESP_Touch_Kit_Touchpad is used as the touch pad. This touchpad consists of a 6*7 array of touch channels.

When a finger moves across the touch pad, it changes the capacitance values of the touch channels, allowing us to detect the finger position by monitoring these capacitance changes.

Detection Algorithm

1. Due to hardware variations, each channel has different maximum and minimum trigger values. Therefore, the capacitance values need to be normalized. This is done by recording the maximum and minimum values as the finger slides across the pad.
2. During finger movement, one channel will show the highest rate of capacitance change. By identifying this channel and its adjacent channel with the next highest change rate, we can locate two neighboring channels.
3. Using a ratio-based calculation between these two channels' values, we can determine the relative coordinate of the finger along that direction (the offset from the center point between the two channels).

$$x = \frac{Fa - Fb}{Fa + Fb}$$

4. By applying the above steps, we can obtain relative coordinates in both directions, thus determining the finger's position.

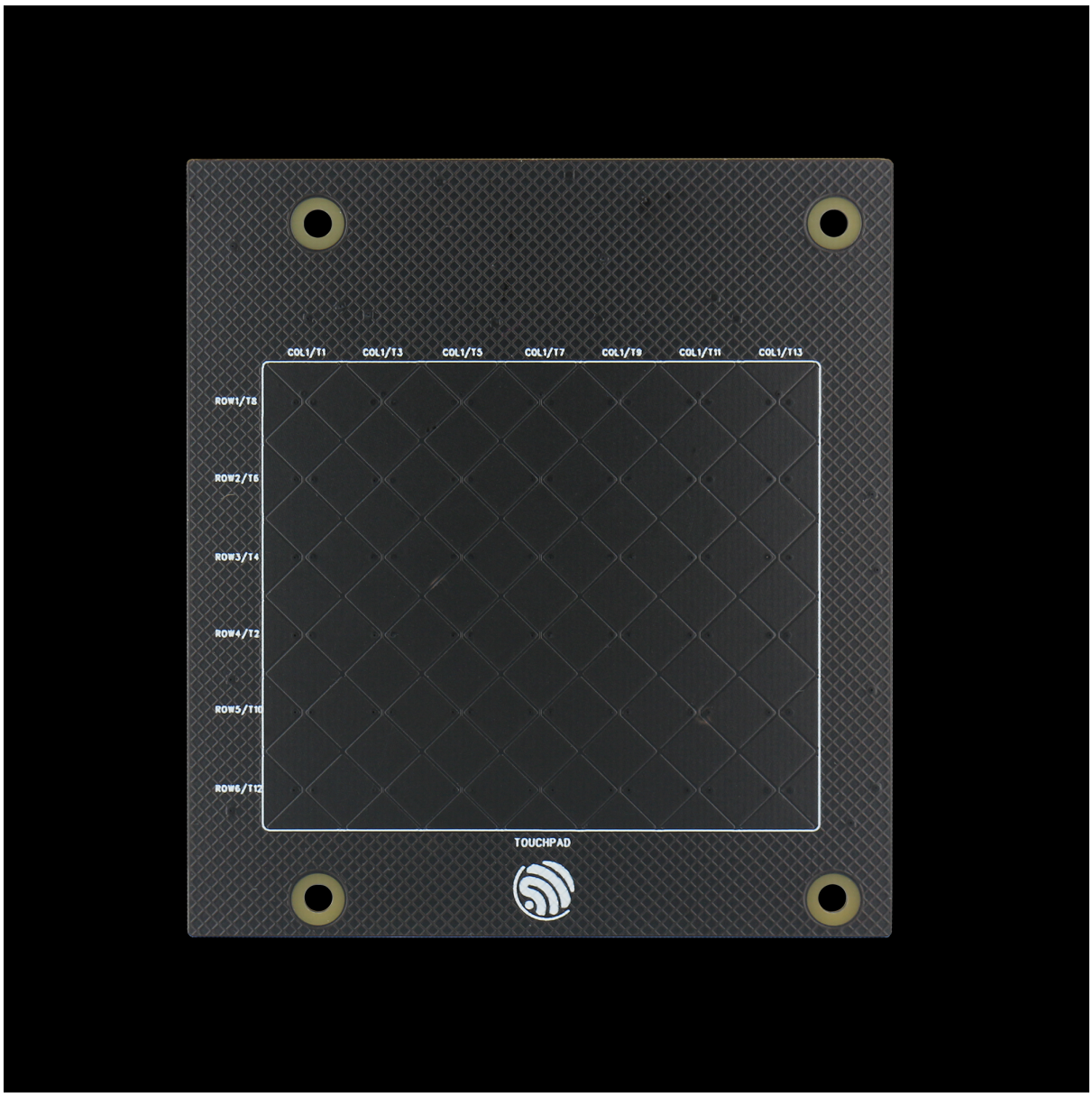


Fig. 1: Physical Touch Pad

Note: An appropriate trigger threshold needs to be set to determine whether the finger is moving on the touch pad.

5. When finger lift-off is detected, the drawn data can be saved.

Data Collection

In real-world scenarios, the digits drawn on a touchpad are often visually different from the handwritten digits in the MNIST dataset. As a result, models trained directly on MNIST perform poorly when applied to actual touch input. Therefore, it is necessary to collect a custom dataset of digits drawn on a touchpad and use it for training.

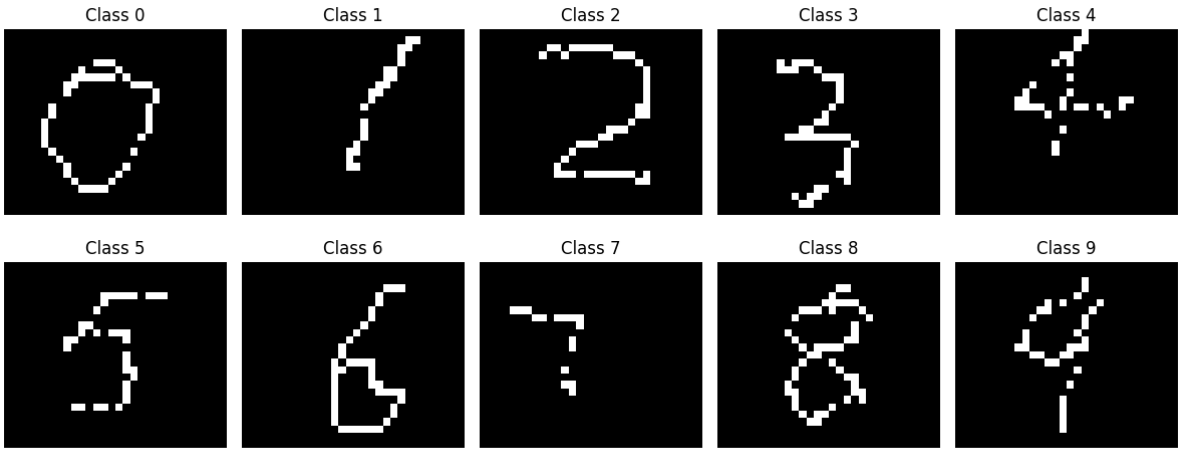


Fig. 2: Real Dataset Based on Touchpad Drawn Input

Note: The size of the handwritten data image after interpolation is 30×25.

Click here to download the dataset used in this example: `touch_dataset.zip`

8.3.2 Model Training and Deployment

Model Construction

Based on the PyTorch framework, a neural network model suitable for touch-based handwritten digit recognition is constructed. The architecture is as follows:

```
class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.model = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3,␣
↪stride=1, padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2),

            torch.nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3,␣
↪stride=1, padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2),
```

(continues on next page)

(continued from previous page)

```
            torch.nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,␣
↪stride=1, padding=1),
            torch.nn.ReLU(),

            torch.nn.Flatten(),
            torch.nn.Linear(in_features=7 * 6 * 64, out_features=256),
            torch.nn.ReLU(),
            torch.nn.Dropout(p=0.5),
            torch.nn.Linear(in_features=256, out_features=10),
            torch.nn.Softmax(dim=1)
        )

    def forward(self, x):
        output = self.model(x)
        return output
```

Model Training

The training process of the model includes dataset loading and preprocessing, configuration of training parameters, monitoring of the training progress, and saving of the trained model.

Data Loading and Preprocessing The images corresponding to different digits are organized under the `dataset/extra` directory, with each digit stored in a separate subfolder named after the digit. Image preprocessing is performed using `transforms.Compose`, including grayscale conversion, random rotation and translation, and normalization. The entire dataset is then loaded using `ImageFolder` and split into training and test sets in an 8:2 ratio. Finally, DataLoader is used to construct batch loaders for subsequent model training and evaluation.

```
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, random_split
from torchvision import datasets, transforms

transform = transforms.Compose([
    transforms.Grayscale(num_output_channels=1),
    transforms.RandomAffine(degrees=10, translate=(0.1, 0.1)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,)),
])

dataset = datasets.ImageFolder(root='./dataset/extra', transform=transform)

train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = random_split(dataset, [train_size, test_size])

train_loader = DataLoader(dataset=train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=32, shuffle=False)
```

Model Training Parameter Configuration Model training parameters include learning rate, optimizer, loss function, and others. In the actual training process, cross-entropy is used as the loss function, and the Adam optimizer is employed to update the model parameters.

```
device = "cuda:0" if torch.cuda.is_available() else "cpu"
model = Net().to(device)
```

(continues on next page)

(continued from previous page)

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Model Training and Saving The number of training epochs is set to 100. During training, the model parameters are updated using the training set, while the test set is used to evaluate the model' s performance after each epoch. Once training is complete, the model parameters are saved to the file `./models/final_model.pth`.

```
def train_epoch(model, train_loader, criterion, optimizer, device):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    epoch_loss = running_loss / len(train_loader)
    epoch_acc = 100 * correct / total
    return epoch_loss, epoch_acc


def test_epoch(model, test_loader, criterion, device):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            outputs = model(inputs)
            loss = criterion(outputs, labels)

            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    epoch_loss = running_loss / len(test_loader)
    epoch_acc = 100 * correct / total
    return epoch_loss, epoch_acc

num_epochs = 100
train_acc_array = []
test_acc_array = []
for epoch in range(num_epochs):
    train_loss, train_acc = train_epoch(model, train_loader, criterion, optimizer,␣
↪device)
```

(continues on next page)

(continued from previous page)

```
    test_loss, test_acc = test_epoch(model, test_loader, criterion, device)

    print(f'Epoch [{epoch + 1}/{num_epochs}], '
        f'Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.2f}%, '
        f'Test Loss: {test_loss:.4f}, Test Acc: {test_acc:.2f}%')
    train_acc_array.append(train_acc)
    test_acc_array.append(test_acc)

torch.save(model.state_dict(), './models/final_model.pth')
```

During the training process, the accuracy curves of the training and test sets evolve as follows:

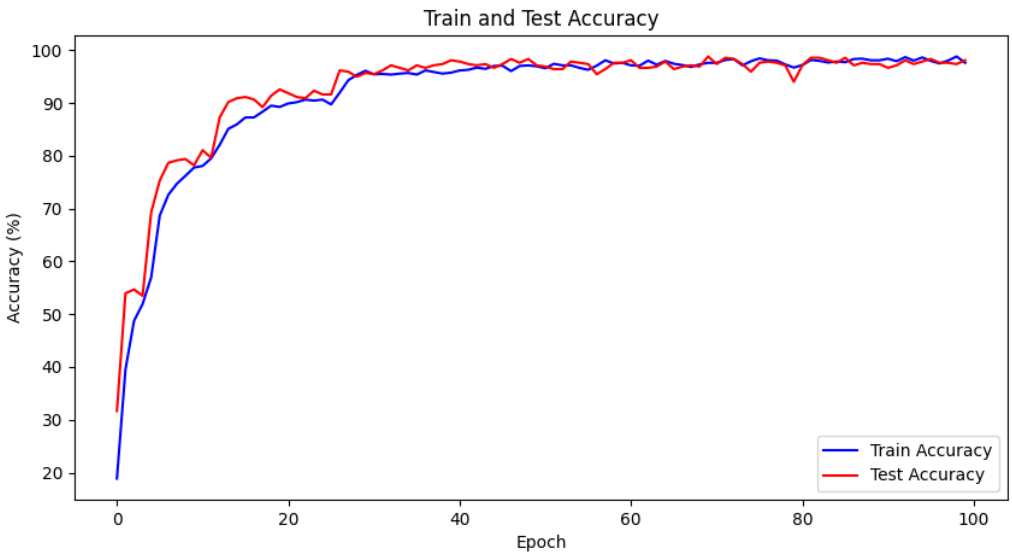


Fig. 3: Accuracy Curves of the Training and Test Sets

Model Deployment

ESP-PPQ Environment Configuration `ESP-PPQ` is a quantization tool based on `ppq`. Please use the following command to install `ESP-PPQ`:

```
pip uninstall ppq
pip install git+https://github.com/espressif/esp-ppq.git
```

Model Quantization and Deployment Refer to How to quantize model for model quantization and export. If you need to export a model for ESP32P4, set `TARGET` to `esp32p4`.

```
import torch
from PIL import Image
from ppq.api import espdl_quantize_torch
from torch.utils.data import Dataset
from torch.utils.data import random_split
from torchvision import transforms, datasets

DEVICE = "cpu"

class FeatureOnlyDataset(Dataset):
    def __init__(self, original_dataset):
```

(continues on next page)

(continued from previous page)

```
        self.features = []
        for item in original_dataset:
            self.features.append(item[0])

    def __len__(self):
        return len(self.features)

    def __getitem__(self, idx):
        return self.features[idx]


def collate_fn2(batch):
    features = torch.stack(batch)
    return features.to(DEVICE)


if __name__ == '__main__':
    BATCH_SIZE = 32
    INPUT_SHAPE = [1, 25, 30]
    TARGET = "esp32s3"
    NUM_OF_BITS = 8
    ESPDL_MODEL_PATH = "./s3/touch_recognition.espdl"

    transform = transforms.Compose([
        transforms.Grayscale(num_output_channels=1),
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (0.5,)),
    ])

    dataset = datasets.ImageFolder(root="../dataset/extra", transform=transform)
    train_size = int(0.8 * len(dataset))
    test_size = len(dataset) - train_size
    train_dataset, test_dataset = random_split(dataset, [train_size, test_size])

    image = Image.open("../dataset/extra/9/20250225_140331.png").convert('L')
    input_tensor = transform(image).unsqueeze(0)
    print(input_tensor)

    feature_only_test_data = FeatureOnlyDataset(test_dataset)

    testDataLoader = torch.utils.data.DataLoader(dataset=feature_only_test_data,␣
↪batch_size=BATCH_SIZE, shuffle=False,
                                                 collate_fn=collate_fn2)

    model = Net().to(DEVICE)
    model.load_state_dict(torch.load("./final_model.pth", map_location=DEVICE))
    model.eval()

    quant_ppq_graph = espdl_quantize_torch(
        model=model,
        espdl_export_file=ESPDL_MODEL_PATH,
        calib_dataloader=testDataLoader,
        calib_steps=8,
        input_shape=[1] + INPUT_SHAPE,
        inputs=[input_tensor],
        target=TARGET,
        num_of_bits=NUM_OF_BITS,
        device=DEVICE,
        error_report=True,
        skip_export=False,
        export_test_values=True,
```

(continues on next page)

(continued from previous page)

```
        verbose=1,
        dispatching_override=None
    )
```

To facilitate model debugging, ESP-DL provides the functionality to add test data during quantization and view inference results on the PC side. In the above process, `image` is loaded into `espdl_quantize_torch` for testing. After model conversion is complete, the inference results of the test data will be saved in a file with the `*.info` extension:

```
test outputs value:
%23, shape: [1, 10], exponents: [0],
value: array([9.85415445e-34, 1.92874989e-22, 7.46892081e-43, 1.60381094e-28,
    3.22134028e-27, 1.05306175e-20, 4.07960022e-41, 1.42516404e-21,
    2.38026637e-26, 1.00000000e+00, 0.00000000e+00, 0.00000000e+00],
    dtype=float32)
```

Important: During model quantization and deployment, please set the `shuffle` parameter in `torch.utils.data.DataLoader` to `False`.

8.3.3 On-device Inference

Refer to How to load test profile model and How to run model for implementing model loading and inference.

It' s important to note that in this example, the Touch driver reports pressed and unpressed states as 1 and 0, while the model input is normalized image data. Therefore, preprocessing of the data reported by the Touch driver is necessary:

```
for (size_t i = 0; i < m_feature_size; i++) {
    int8_t value = (input_data[i] == 0 ? -1 : 1);
    quant_buffer[i] = dl::quantize<int8_t>((float)value, m_input_scale);
}
```

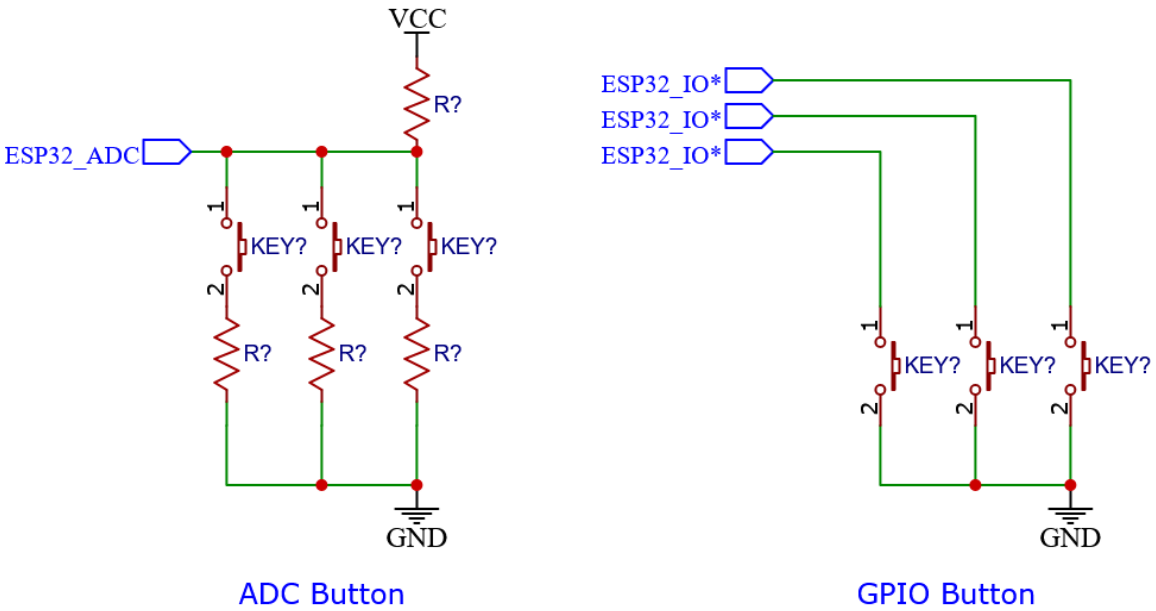
For the complete project, please refer to: ai/esp_dl/touchpad_digit_recognition

Chapter 9

Input Device

9.1 Button

The button component supports GPIO and ADC mode, and it allows the creation of two different kinds of the button at the same time. The following figure shows the hardware design of the button:



- GPIO button: The advantage of the GPIO button is that each button occupies an independent IO and therefore does not affect each other, and has high stability however when the number of buttons increases, it may take too many IO resources.
- ADC button: The advantage of using the ADC button is that one ADC channel can share multiple buttons and occupy fewer IO resources. The disadvantages include that you cannot press multiple buttons at the same time, and instability increases due to increase in the closing resistance of the button due to oxidation and other factors.

Note:

- The GPIO button needs to pay attention to the problem of pull-up and pull-down resistor inside the chip, which will be enabled by default. But there is no such resistor inside the IO that only supports input, **external connection requires**.
- The voltage of the ADC button should not exceed the ADC range.

9.1.1 Button event

Triggering conditions for each button event are enlisted in the table below:

Event	Trigger Condition
BUTTON_PRESS_DOWN	Pressed
BUTTON_PRESS_UP	Released
BUTTON_PRESS_REPEAT	Pressed and released >= 2 times
BUTTON_PRESS_REPEAT_DONE	Repeated press completed
BUTTON_SINGLE_CLICK	Pressed and released once
BUTTON_DOUBLE_CLICK	Pressed and released twice
BUTTON_MULTIPLE_CLICK	Pressed and released N times specified, triggers when achieved
BUTTON_LONG_PRESS_START	Instant when held for a threshold duration of time
BUTTON_LONG_PRESS_HOLD	Triggered continuously during long press
BUTTON_LONG_PRESS_UP	Released after a long press
BUTTON_PRESS_REPEAT_DONE	Repeated press and release ended
BUTTON_PRESS_END	Indicate that the button has completed its current detection.

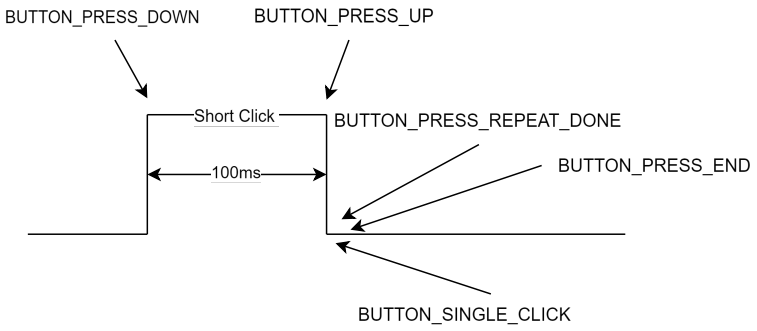
Each button supports **call-back** and **pooling** mode.

- Call-back: Each event of a button can register a call-back function for it, and the call-back function will be called when an event is generated. This method has high efficiency and real-time performance, and no events will be lost.
- Polling: Periodically call `iot_button_get_event()` in the program to query the current event of the button. This method is easy to use and is suitable for occasions with simple tasks. Not all key press events are captured in time, leading to a risk of missing events.

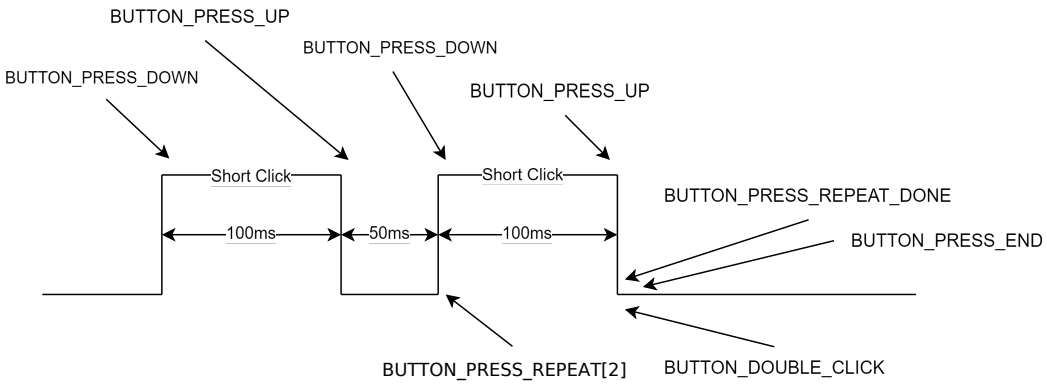
Note: you can also combine the above two methods.

Attention: No blocking operations such as **TaskDelay** are allowed in the call-back function

Single Click

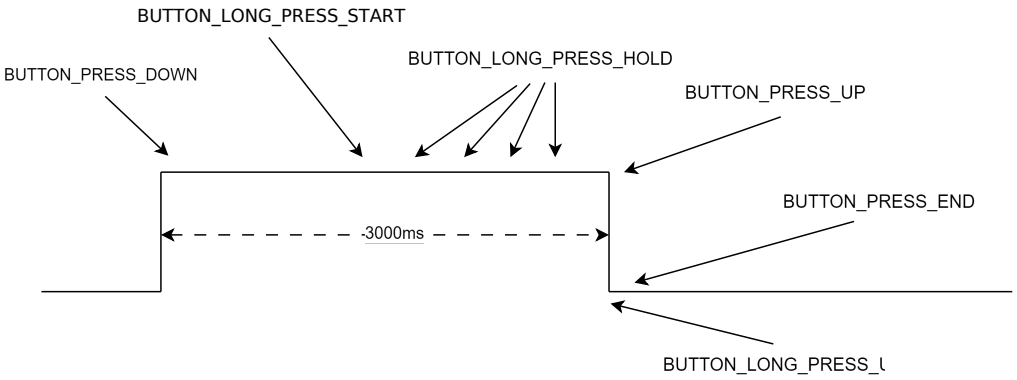


Double Click



LongPress

BUTTON_LONG_PRESS_TIME_MS: 1500ms



9.1.2 Configuration

- BUTTON_PERIOD_TIME_MS : scan cycle
- BUTTON_DEBOUNCE_TICKS : debounce time
- BUTTON_SHORT_PRESS_TIME_MS : short press down effective time
- BUTTON_LONG_PRESS_TIME_MS : long press down effective time
- ADC_BUTTON_MAX_CHANNEL : maximum number of channel for ADC

- ADC_BUTTON_MAX_BUTTON_PER_CHANNEL : maximum number of ADC buttons per channel
- ADC_BUTTON_SAMPLE_TIMES: The number of samples per ADC scan.
- BUTTON_LONG_PRESS_HOLD_SERIAL_TIME_MS: The interval time for triggering the callback during a long press.

9.1.3 Demonstration

Create a button

```
// create gpio button
const button_config_t btn_cfg = {0};
const button_gpio_config_t btn_gpio_cfg = {
    .gpio_num = 0,
    .active_level = 0,
};
button_handle_t gpio_btn = NULL;
esp_err_t ret = iot_button_new_gpio_device(&btn_cfg, &btn_gpio_cfg, &gpio_btn);
if(NULL == gpio_btn) {
    ESP_LOGE(TAG, "Button create failed");
}

// create adc button
const button_config_t btn_cfg = {0};
button_adc_config_t btn_adc_cfg = {
    .unit_id = ADC_UNIT_1,
    .adc_channel = 0,
    .button_index = 0,
    .min = 100,
    .max = 400,
};

button_handle_t adc_btn = NULL;
esp_err_t ret = iot_button_new_adc_device(&btn_cfg, &btn_adc_cfg, &adc_btn);
if(NULL == adc_btn) {
    ESP_LOGE(TAG, "Button create failed");
}

// create matrix keypad button
const button_config_t btn_cfg = {0};
const button_matrix_config_t matrix_cfg = {
    .row_gpios = (int32_t[]){4, 5, 6, 7},
    .col_gpios = (int32_t[]){3, 8, 16, 15},
    .row_gpio_num = 4,
    .col_gpio_num = 4,
};
button_handle_t matrix_button = NULL;
esp_err_t ret = iot_button_new_matrix_device(&btn_cfg, &matrix_cfg, btns, &matrix_
↪button);
if(NULL == matrix_button) {
    ESP_LOGE(TAG, "Button create failed");
}
```

Note: When using ADC1 for ADC buttons and ADC1 is also used elsewhere in the project, please pass in adc_handle and adc_channel to configure the ADC button.

Register callback function

The Button component supports registering callback functions for multiple events, with each event capable of having its own callback function. When an event occurs, the callback function will be invoked.

In this context:

- The *BUTTON_LONG_PRESS_START* and *BUTTON_LONG_PRESS_UP* enumerations support setting specific long press times.
- The *BUTTON_MULTIPLE_CLICK* enumeration supports setting the number of consecutive button presses.
- Here's a simple example:

```
static void button_single_click_cb(void *arg,void *usr_data)
{
    ESP_LOGI(TAG, "BUTTON_SINGLE_CLICK");
}

iot_button_register_cb(gpio_btn, BUTTON_SINGLE_CLICK, NULL, button_
↪single_click_cb,NULL);
```

- And here's an example involving multiple callback functions:

```
static void button_long_press_1_cb(void *arg,void *usr_data)
{
    ESP_LOGI(TAG, "BUTTON_LONG_PRESS_START_1");
}

static void button_long_press_2_cb(void *arg,void *usr_data)
{
    ESP_LOGI(TAG, "BUTTON_LONG_PRESS_START_2");
}

button_event_args_t args = {
    .long_press.press_time = 2000,
};

iot_button_register_cb(gpio_btn, BUTTON_LONG_PRESS_START, &args,␣
↪button_auto_check_cb_1, NULL);

args.long_press.press_time = 5000;
iot_button_register_cb(gpio_btn, BUTTON_LONG_PRESS_START, &args,␣
↪button_long_press_2_cb, NULL);
```

Dynamically Modifying Default Button Values

```
iot_button_set_param(btn, BUTTON_LONG_PRESS_TIME_MS, 5000);
```

Find an event

```
button_event_t event;
event = iot_button_get_event(button_handle);
```

Low power

In light_sleep mode, the *esp_timer* triggers periodically, resulting in sustained high overall CPU power consumption. To address this issue, the button component offers a low-power mode.

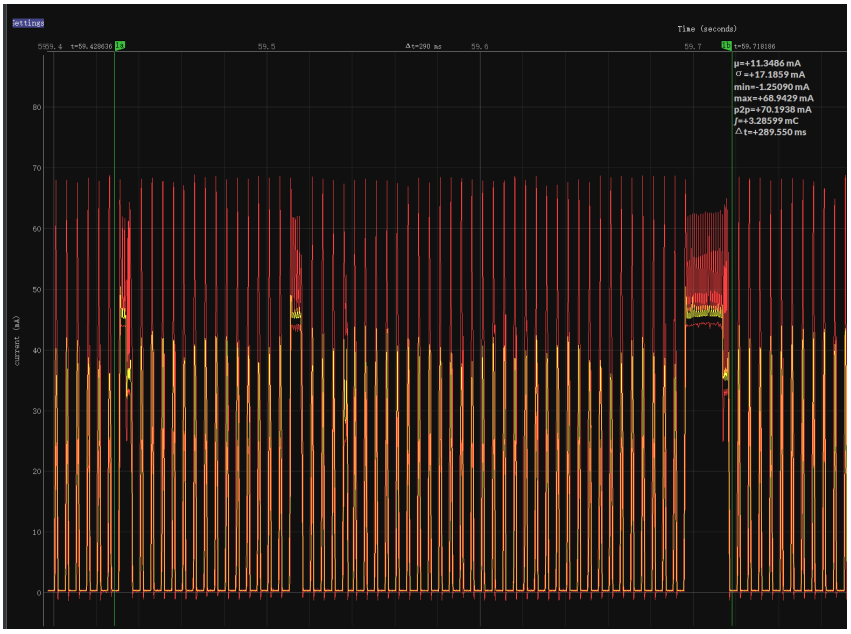
Configuration Required:

- Ensure all created buttons type are GPIO type and have *enable_power_save* activated. The presence of other buttons may render the low-power mode ineffective.

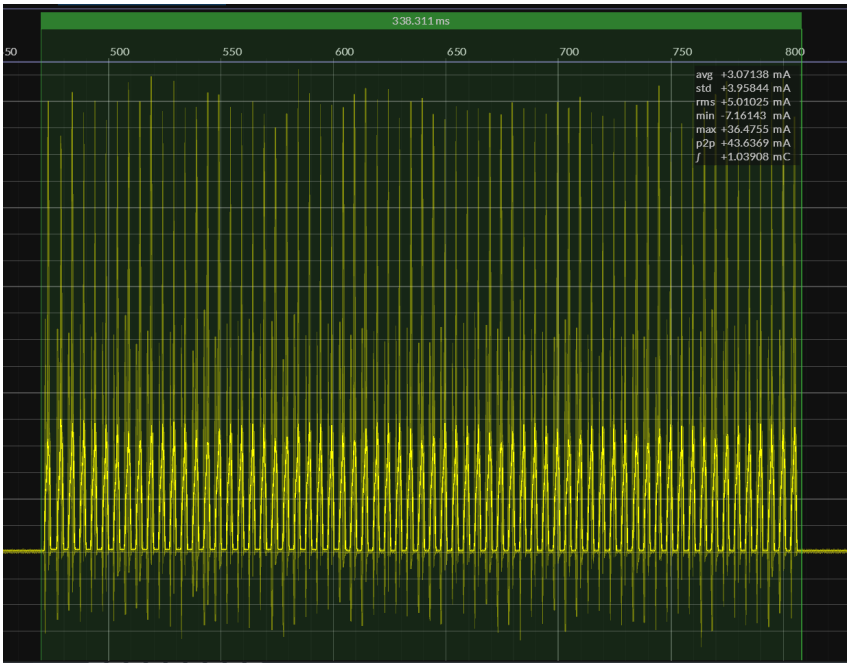
Note: This feature ensures that the Button component only wakes up the CPU when in use, but does not guarantee the CPU will always enter low-power mode.

Power Consumption Comparison:

- Without enabling low-power mode, pressing the button once:



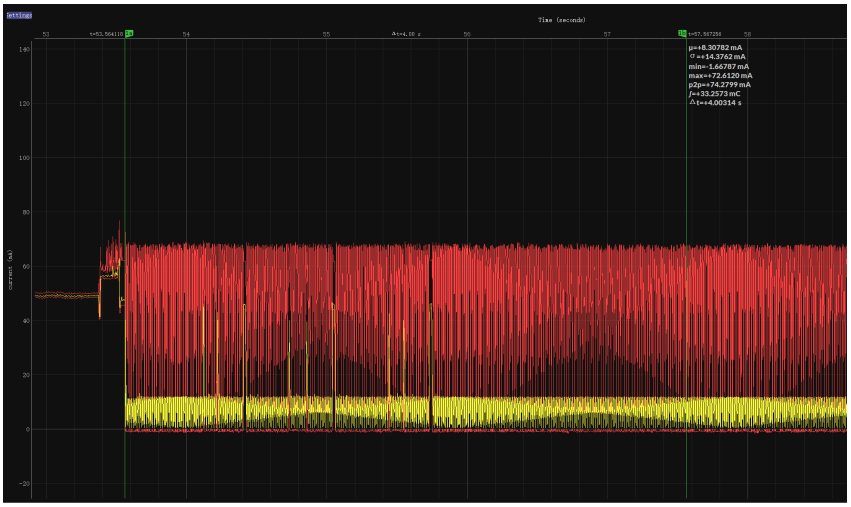
- With low-power mode enabled, pressing the button once:



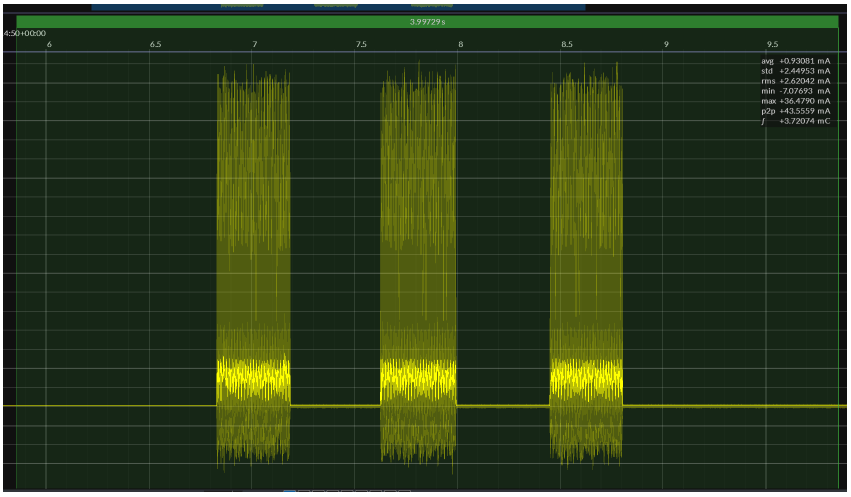
Because GPIO wakes up the CPU, supporting only level triggering, the CPU is awakened only when the button is

at its operating level. Therefore, in low-power mode, the average current during a single press is higher than when low-power mode is not enabled, depending on the duration of the button press. However, over larger operational periods, it saves more power than when low-power mode is not enabled.

- Without enabling low-power mode, pressing the button three times within 4 seconds:



- With low-power mode enabled, pressing the button three times within 4 seconds:



As shown, low-power mode results in more power savings.

```
button_config_t btn_cfg = {0};
button_gpio_config_t gpio_cfg = {
    .gpio_num = button_num,
    .active_level = BUTTON_ACTIVE_LEVEL,
    .enable_power_save = true,
};

button_handle_t btn;
iot_button_new_gpio_device(&btn_cfg, &gpio_cfg, &btn);
```

When to Enter Light Sleep

- Using Auto Light Sleep: The device will enter Light Sleep automatically after the button closes the esp_timer.
- User-Controlled Light Sleep: The device can enter Light Sleep when `enter_power_save_cb` is called.

```
void btn_enter_power_save(void *usr_data)
{
    ESP_LOGI(TAG, "Can enter power save now");
}

button_power_save_config_t config = {
    .enter_power_save_cb = btn_enter_power_save,
};

iot_button_register_power_save_cb(&config);
```

How to Use Buttons Normally After Enabling the CONFIG_PM_POWER_DOWN_PERIPHERAL_IN_LIGHT_SLEEP Option?

- When this macro is enabled, the GPIO module will be powered down. To use the button functionality, you must use RTC/LP GPIO and change the wake-up source to EXT 1.

GPIO Type	CONFIG_PM_POWER_DOWN_PERIPHERAL_IN_LIGHT_SLEEP Enabled?	Wake-Up Source
Digital Pin	N	GPIO Level Trigger
Digital Pin	Y	None
RTC/LP Pin	N	GPIO Level Trigger / EXT 1
RTC/LP Pin	Y	EXT 1

Note: The LP GPIOs of ESP32-C5 and ESP32-C6 support both GPIO level wake-up and EXT 1 wake-up, and you also need to enable `gpio_hold_en`.

Stop and resume

The component supports being turned on and off at any given moment.

```
// stop button
iot_button_stop();
// resume button
iot_button_resume();
```

9.1.4 API Reference

Header File

- components/button/include/iot_button.h

Functions

esp_err_t **`iot_button_delete`**(button_handle_t btn_handle)

Delete a button.

Parameters **`btn_handle`** –A button handle to delete

Returns

- ESP_OK Success
- ESP_FAIL Failure

esp_err_t **iot_button_register_cb**(button_handle_t btn_handle, *button_event_t* event, *button_event_args_t* *event_args, *button_cb_t* cb, void *usr_data)

Register the button event callback function.

Parameters
- **btn_handle** –A button handle to register
- **event** –Button event
- **event_args** –Button event arguments
- **cb** –Callback function.
- **usr_data** –user data

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG Arguments is invalid.
- ESP_ERR_INVALID_STATE The Callback is already registered. No free Space for another Callback.
- ESP_ERR_NO_MEM No more memory allocation for the event

esp_err_t **iot_button_unregister_cb**(button_handle_t btn_handle, *button_event_t* event, *button_event_args_t* *event_args)

Unregister all the callbacks associated with the event.

Parameters
- **btn_handle** –A button handle to unregister
- **event** –Button event
- **event_args** –Used for unregistering a specific callback.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG Arguments is invalid.
- ESP_ERR_INVALID_STATE No callbacks registered for the event

size_t **iot_button_count_cb**(button_handle_t btn_handle)

counts total callbacks registered

Parameters **btn_handle** –A button handle to the button

Returns
- 0 if no callbacks registered, or 1 .. (BUTTON_EVENT_MAX-1) for the number of Registered Buttons.
- ESP_ERR_INVALID_ARG if btn_handle is invalid

size_t **iot_button_count_event_cb**(button_handle_t btn_handle, *button_event_t* event)

how many callbacks are registered for the event

Parameters
- **btn_handle** –A button handle to the button
- **event** –Button event

Returns
- 0 if no callbacks registered, or 1 .. (BUTTON_EVENT_MAX-1) for the number of Registered Buttons.
- ESP_ERR_INVALID_ARG if btn_handle is invalid

button_event_t **iot_button_get_event**(button_handle_t btn_handle)

Get button event.

Parameters **btn_handle** –Button handle

Returns Current button event. See button_event_t

const char ***iot_button_get_event_str**(*button_event_t* event)

Get the string representation of a button event.

This function returns the corresponding string for a given button event. If the event value is outside the valid range, the function returns error string "event value is invalid".

Parameters **event** –**[in]** The button event to be converted to a string.

Returns
- Pointer to the event string if the event is valid.
- ” invalid event” if the event value is invalid.

esp_err_t **iot_button_print_event**(button_handle_t btn_handle)

Log the current button event as a string.

This function prints the string representation of the current event associated with the button.

Parameters **btn_handle** –**[in]** Handle to the button object.
Returns
- ESP_OK: Successfully logged the event string.
- ESP_FAIL: Invalid button handle.

uint8_t **iot_button_get_repeat**(button_handle_t btn_handle)

Get button repeat times.

Parameters **btn_handle** –Button handle
Returns button pressed times. For example, double-click return 2, triple-click return 3, etc.

uint32_t **iot_button_get_pressed_time**(button_handle_t btn_handle)

Get button pressed time.

Parameters **btn_handle** –Button handle
Returns Actual time from press down to up (ms).

uint32_t **iot_button_get_ticks_time**(button_handle_t btn_handle)

Get button ticks time.

Deprecated:

This function is deprecated and will be removed in a future release. Please use iot_button_get_pressed_time() instead.

Parameters **btn_handle** –Button handle
Returns Actual time from press down to up (ms).

uint16_t **iot_button_get_long_press_hold_cnt**(button_handle_t btn_handle)

Get button long press hold count.

Parameters **btn_handle** –Button handle
Returns Count of trigger cb(BUTTON_LONG_PRESS_HOLD)

esp_err_t **iot_button_set_param**(button_handle_t btn_handle, *button_param_t* param, void *value)

Dynamically change the parameters of the iot button.

Parameters
- **btn_handle** –Button handle
- **param** –Button parameter
- **value** –new value

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG Arguments is invalid.

uint8_t **iot_button_get_key_level**(button_handle_t btn_handle)

Get button key level.

Parameters **btn_handle** –Button handle
Returns
- 1 if key is pressed
- 0 if key is released or invalid button handle

esp_err_t **iot_button_resume** (void)

resume button timer, if button timer is stopped. Make sure iot_button_create() is called before calling this API.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE timer state is invalid.

esp_err_t **iot_button_stop** (void)

stop button timer, if button timer is running. Make sure iot_button_create() is called before calling this API.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE timer state is invalid

esp_err_t **iot_button_register_power_save_cb** (const *button_power_save_config_t* *config)

Register a callback function for power saving. The config->enter_power_save_cb function will be called when all keys stop working.

Parameters **config** –Button power save config

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE No button registered
- ESP_ERR_INVALID_ARG Arguments is invalid
- ESP_ERR_NO_MEM Not enough memory

Unions

union **button_event_args_t**

#include <iot_button.h> Button events arg.

Public Members

struct *button_event_args_t*::*long_press_t* **long_press**

long press struct, for event BUTTON_LONG_PRESS_START and BUTTON_LONG_PRESS_UP

struct *button_event_args_t*::*multiple_clicks_t* **multiple_clicks**

multiple clicks struct, for event BUTTON_MULTIPLE_CLICK

struct **long_press_t**

#include <iot_button.h> Long press time event data.

Public Members

uint16_t **press_time**

press time(ms) for the corresponding callback to trigger

struct **multiple_clicks_t**

#include <iot_button.h> Multiple clicks event data.

Public Members

uint16_t **clicks**

number of clicks, to trigger the callback

Structures

struct **button_power_save_config_t**

Structs to store power save callback info.

Public Members

button_power_save_cb_t **enter_power_save_cb**

Callback function when entering power save mode

void ***usr_data**

User data for the callback

Type Definitions

typedef void (***button_cb_t**)(void *button_handle, void *usr_data)

typedef void (***button_power_save_cb_t**)(void *usr_data)

Enumerations

enum **button_event_t**

Button events.

Values:

enumerator **BUTTON_PRESS_DOWN**

enumerator **BUTTON_PRESS_UP**

enumerator **BUTTON_PRESS_REPEAT**

enumerator **BUTTON_PRESS_REPEAT_DONE**

enumerator **BUTTON_SINGLE_CLICK**

enumerator **BUTTON_DOUBLE_CLICK**

enumerator **BUTTON_MULTIPLE_CLICK**

enumerator **BUTTON_LONG_PRESS_START**

enumerator **BUTTON_LONG_PRESS_HOLD**

enumerator **BUTTON_LONG_PRESS_UP**

enumerator **BUTTON_PRESS_END**

enumerator **BUTTON_EVENT_MAX**

enumerator **BUTTON_NONE_PRESS**

enum **button_param_t**

Button parameter.

Values:

enumerator **BUTTON_LONG_PRESS_TIME_MS**

enumerator **BUTTON_SHORT_PRESS_TIME_MS**

enumerator **BUTTON_PARAM_MAX**

9.2 Keyboard Scanning

The Keyboard Scanning Component implements fast and efficient keyboard scanning, supporting key debouncing, key release and press event reporting, as well as combination keys.

This component uses matrix key row-column scanning and, through special circuit design, achieves full-key rollover circuit detection.

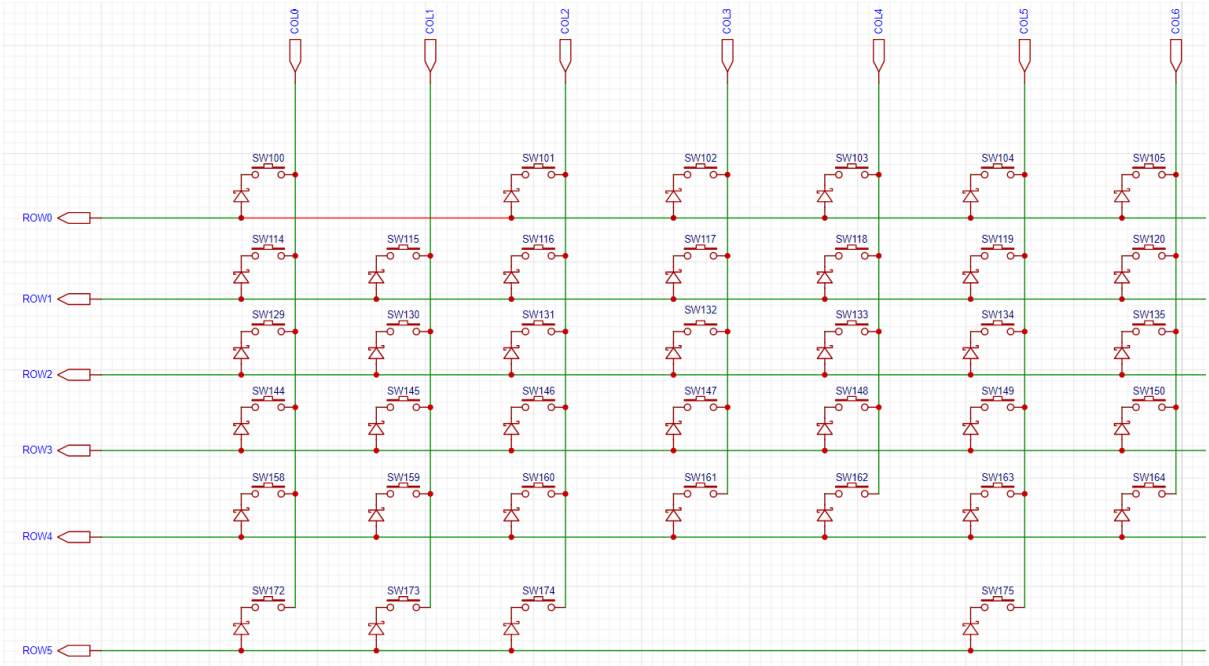
- In this circuit, rows output high-level signals sequentially, detecting whether columns have high-level signals. If they do, it indicates the key is pressed.

Note:

- Since the logic of this component does not involve swapping row-column scanning, it is not suitable for traditional row-column scanning circuits and is only applicable to full-key rollover circuits for keyboards.

9.2.1 Component Events

- *KBD_EVENT_PRESSED*: Reports data when there is a change in key states.
  - *key_pressed_num*: Number of keys pressed.
  - *key_release_num*: Number of keys released.
  - *key_change_num*: Number of keys with changed states compared to the previous state. **>0** indicates an increase in pressed keys, **<0** indicates a decrease.
  - *key_data*: Information of the currently pressed keys, with positions (x, y). Indexed in the order they were pressed, with smaller indexes being pressed earlier.
  - *key_release_data*: Information of keys released compared to the previous state, with positions (x, y).



- *KBD_EVENT_COMBINATION*: Combination key event. Triggered when a combination key is pressed.
  - *key_num*: Number of keys in the combination.
  - *key_data*: Position information of the combination keys. For setting a combination key (1,1) and (2,2), (1,1) must be pressed first followed by (2,2) to trigger the combination key event. Combination keys only trigger with increasing combinations.

9.2.2 Application Example

Initializing Keyboard Scanning

```
keyboard_btn_config_t cfg = {
    .output_gpios = (int[])
    {
        40, 39, 38, 45, 48, 47
    },
    .output_gpio_num = 6,
    .input_gpios = (int[])
    {
        21, 14, 13, 12, 11, 10, 9, 4, 5, 6, 7, 15, 16, 17, 18
    },
    .input_gpio_num = 15,
    .active_level = 1,
    .debounce_ticks = 2,
    .ticks_interval = 500,       // us
    .enable_power_save = false, // enable power save
};
keyboard_btn_handle_t kbd_handle = NULL;
keyboard_button_create(&cfg, &kbd_handle);
```

Registering Callback Functions

- Registration of *KBD_EVENT_PRESSED* event is as follows:

```
keyboard_btn_cb_config_t cb_cfg = {
    .event = KBD_EVENT_PRESSED,
    .callback = keyboard_cb,
};
keyboard_button_register_cb(kbd_handle, cb_cfg, NULL);
```

- Registration of *KBD_EVENT_COMBINATION* event requires passing combination key information through the *combination* member:

```
keyboard_btn_cb_config_t cb_cfg = {
    .event = KBD_EVENT_COMBINATION,
    .callback = keyboard_combination_cb1,
    .event_data.combination.key_num = 2,
    .event_data.combination.key_data = (keyboard_btn_data_t[]) {
        {5, 1},
        {1, 1},
    },
};

keyboard_button_register_cb(kbd_handle, cb_cfg, NULL);
```

Note: Additionally, multiple callbacks can be registered for each event. When registering multiple callbacks, it's recommended to save **keyboard_btn_cb_handle_t *rtn_cb_hdl** for later unbinding of specific callbacks.

Key Scanning Efficiency

- Testing with *ESP32S3* chip scanning a *5*16* matrix keyboard, the maximum scanning rate can reach 20K.

Low Power Support

- Set enable_power_save to true during initialization to activate the low-power mode. In this mode, key scanning is suspended when no key changes occur, allowing the CPU to enter a sleep state. The CPU wakes up when a key is pressed.

Note: This feature only ensures that it does not occupy the CPU; it does not guarantee that the CPU will necessarily enter low-power mode. Currently, only Light Sleep mode is supported.

9.2.3 API Reference

Header File

- components/keyboard_button/include/keyboard_button.h

Functions

esp_err_t **keyboard_button_create**(*keyboard_btn_config_t* *kbd_cfg, *keyboard_btn_handle_t* *kbd_handle)

Create a keyboard instance.

Parameters

- **kbd_cfg** –keyboard configuration
- **kbd_handle** –keyboard handle

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG Arguments is invalid.
- ESP_ERR_NO_MEM No more memory allocation.

esp_err_t **keyboard_button_delete** (*keyboard_btn_handle_t* kbd_handle)

Delete the keyboard instance.

Parameters **kbd_handle** –keyboard handle

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG Arguments is invalid.

esp_err_t **keyboard_button_register_cb** (*keyboard_btn_handle_t* kbd_handle, *keyboard_btn_cb_config_t* cb_cfg, *keyboard_btn_cb_handle_t* *rtn_cb_hdl)

Register the button callback function.

Parameters
- **kbd_handle** –keyboard handle
- **cb_cfg** –callback configuration
- **rtn_cb_hdl** –callback handle for unregister

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG Arguments is invalid.
- ESP_ERR_NO_MEM No more memory allocation for the event

esp_err_t **keyboard_button_unregister_cb** (*keyboard_btn_handle_t* kbd_handle, *keyboard_btn_event_t* event, *keyboard_btn_cb_handle_t* rtn_cb_hdl)

Unregister the button callback function.

Note: If only the event is provided, all callbacks associated with this event will be canceled. If rtn_cb_hdl is provided, only the specified callback will be unregistered.

Parameters
- **kbd_handle** –keyboard handle
- **event** –event type
- **rtn_cb_hdl** –callback handle for unregister

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG Arguments is invalid.
- ESP_ERR_NO_MEM No more memory allocation for the event

esp_err_t **keyboard_button_get_index_by_gpio** (*keyboard_btn_handle_t* kbd_handle, uint32_t gpio_num, kbd_gpio_mode_t gpio_mode, uint32_t *index)

Get index by gpio number.

Parameters
- **kbd_handle** –keyboard handle
- **gpio_num** –gpio number
- **gpio_mode** –gpio mode, input or output
- **index** –return index

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG Arguments is invalid.
- ESP_ERR_NOT_FOUND The gpio number is not found.

esp_err_t **keyboard_button_get_gpio_by_index**(*keyboard_btn_handle_t* kbd_handle, uint32_t index, kbd_gpio_mode_t gpio_mode, uint32_t *gpio_num)

Get gpio number by index.

Parameters

- **kbd_handle** –keyboard handle
- **index** –index
- **gpio_mode** –gpio mode, input or output
- **gpio_num** –return gpio number

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG Arguments is invalid.
- ESP_ERR_NOT_FOUND The index is not found.

Unions

union **keyboard_btn_event_data_t**

#include <keyboard_button.h> keyboard button event data

Public Members

struct *keyboard_btn_event_data_t*::*combination_t* **combination**

combination event

struct **combination_t**

#include <keyboard_button.h> combination event data eg: Set key_data = {(1,1), (2,2)} means that the button sequence is (1,1) -> (2,2)

Public Members

uint32_t **key_num**

Number of keys

keyboard_btn_data_t ***key_data**

Array, contains key codes by index. The button sequence is also provided through this

Structures

struct **keyboard_btn_data_t**

keyboard button data

Public Members

uint8_t **output_index**

key position’s output gpio number

uint8_t **input_index**

key position' s input gpio number

struct **keyboard_btn_report_t**

keyboard button report data

Public Members

int **key_change_num**

Number of key changes

uint32_t **key_pressed_num**

Number of keys pressed

uint32_t **key_release_num**

Number of keys released

keyboard_btn_data_t ***key_data**

Array, contains key codes

keyboard_btn_data_t ***key_release_data**

Array, contains key codes

struct **keyboard_btn_cb_config_t**

keyboard button callback config

Public Members

keyboard_btn_event_t **event**

Event type

keyboard_btn_event_data_t **event_data**

Event data

keyboard_btn_callback_t **callback**

Callback function

void ***user_data**

Callback user data

struct **keyboard_btn_config_t**

keyboard button config

Public Members

const int ***output_gpios***

Array, contains output GPIO numbers used by rom/col line

const int ***input_gpios***

Array, contains input GPIO numbers used by rom/col line

uint32_t **output_gpio_num**

output_gpios array size

uint32_t **input_gpio_num**

input_gpios array size

uint32_t **active_level**

active level for the input gpios

uint32_t **debounce_ticks**

debounce time in ticks

uint32_t **ticks_interval**

interval time in us

bool **enable_power_save**

enable power save mode

UBaseType_t **priority**

FreeRTOS task priority

BaseType_t **core_id**

ESP32 core ID

Type Definitions

typedef struct keyboard_btn_t ***keyboard_btn_handle_t***

keyboard handle type

typedef void ***keyboard_btn_cb_handle_t***

keyboard callback handle for unregister

typedef void (***keyboard_btn_callback_t***)(*keyboard_btn_handle_t* kbd_handle, *keyboard_btn_report_t* kbd_report, void *user_data)

Enumerations

enum **keyboard_btn_event_t**

Keyboard button event.

Values:

enumerator **KBD_EVENT_PRESSED**

Report all currently pressed keys when a key is either pressed or released.

enumerator **KBD_EVENT_COMBINATION**

When the component buttons are pressed in sequence, report.

enumerator **KBD_EVENT_MAX**

9.3 Knob

Knob is the component that provides the software PCNT, it can be used on chips(esp32c2, esp32c3) that do not have PCNT hardware capabilities. By using knob you can quickly use a physical encoder, such as the EC11 encoder.

9.3.1 Applicable Scenarios

This is suitable for low-speed rotary knob counting scenarios where the pulse rate is less than 30 pulses per second, such as the EC11 encoder. It is suitable for scenarios where 100% accuracy of pulse counting is not required.

Note: For precise or fast pulse counting, please use the hardware PCNT function. The hardware PCNT is supported by ESP32, ESP32-C6, ESP32-H2, ESP32-S2, ESP32-S3 chips.

9.3.2 Hardware Design

The reference design for the rotary encoder is shown below:

9.3.3 Knob Event

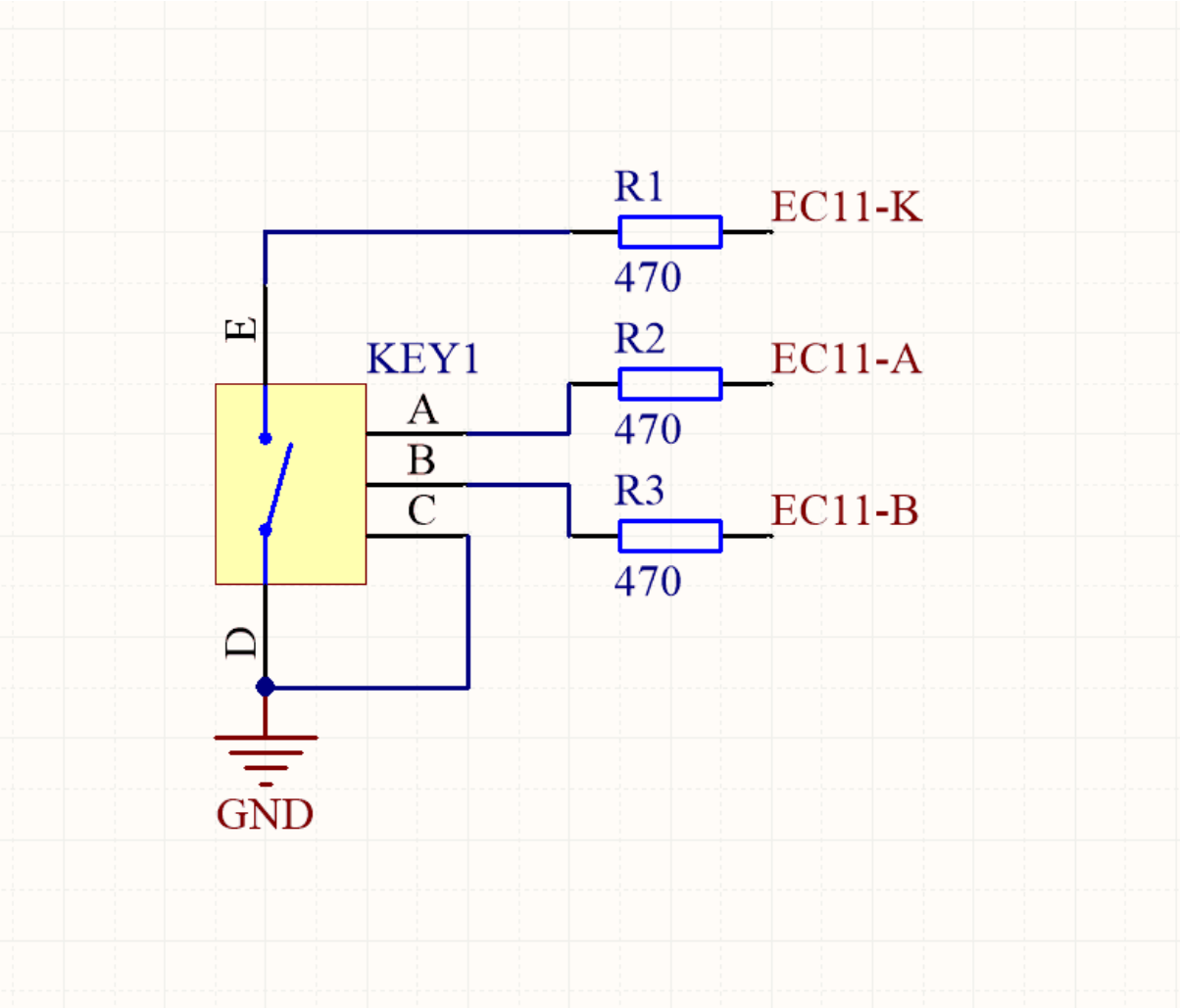
Each knob has the 5 events in the following table.

Events	Trigger Conditions
KNOB_LEFT	Left
KNOB_RIGHT	Right
KNOB_H_LIM	Count reaches maximum limit
KNOB_L_LIM	Count reaches the minimum limit
KNOB_ZERO	Count back to 0

Each knob can have **Callback** usage.

- Callbacks: Each event of a knob can have a callback function registered for it, and the callback function will be called when the event is generated. This approach is efficient and real-time, and no events are lost.

Attention: No blocking operations such as TaskDelay in the callback function


R1
EC11-K
470
E
KEY1
R2
EC11-A
A
470
B
R3
C
EC11-B
470
D
GND

9.3.4 Configuration items

- KNOB_PERIOD_TIME_MS : Scan cycle
- KNOB_DEBOUNCE_TICKS : Number of de-shaking
- KNOB_HIGH_LIMIT : The highest number that can be counted by the knob
- KNOB_LOW_LIMIT : The lowest number that can be counted by the knob

9.3.5 Application Examples

Create Knob

```
// create knob
knob_config_t cfg = {
    .default_direction =0,
    .gpio_encoder_a = GPIO_KNOB_A,
    .gpio_encoder_b = GPIO_KNOB_B,
};
s_knob = iot_knob_create(&cfg);
if(NULL == s_knob) {
    ESP_LOGE(TAG, "knob create failed");
}
```

Register callback function

```
static void _knob_left_cb(void *arg, void *data)
{
    ESP_LOGI(TAG, "KNOB: KNOB_LEFT,count_value:%"PRId32"",iot_knob_get_count_
↪value((button_handle_t)arg));
}
iot_knob_register_cb(s_knob, KNOB_LEFT, _knob_left_cb, NULL);
```

9.3.6 Low Power Support

In light_sleep mode, the *esp_timer* wakes up the CPU, resulting in high power consumption. The Knob component offers a low power solution through GPIO level wake-up.

Required Configuration:

- Enable the *enable_power_save* option in *knob_config_t*.

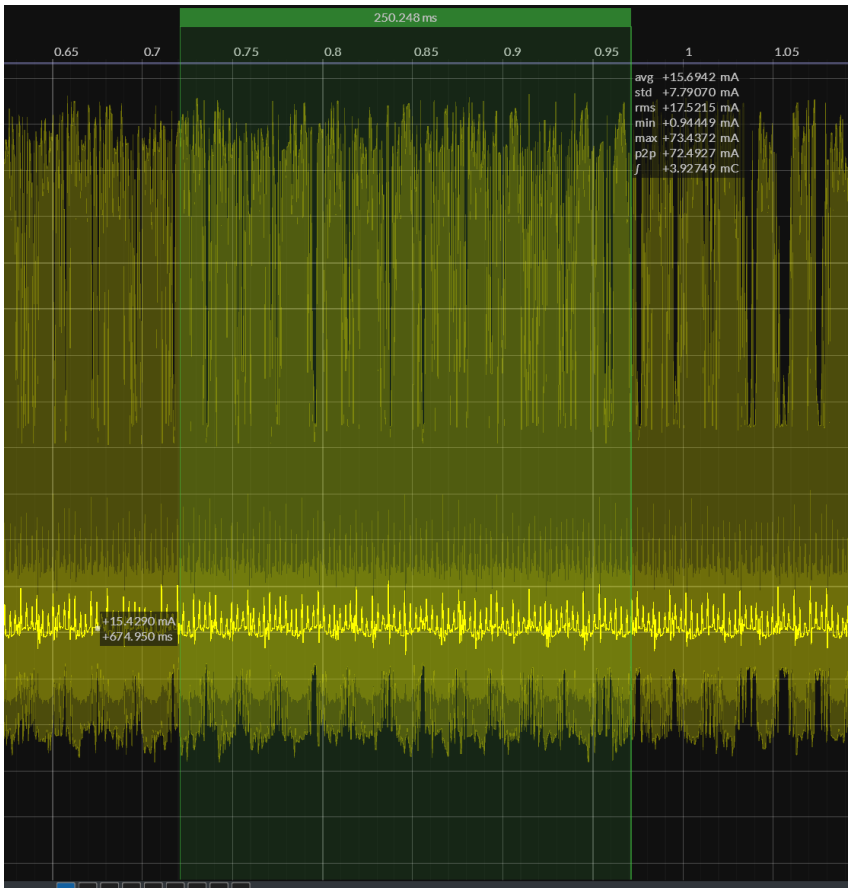
Power Consumption Comparison:

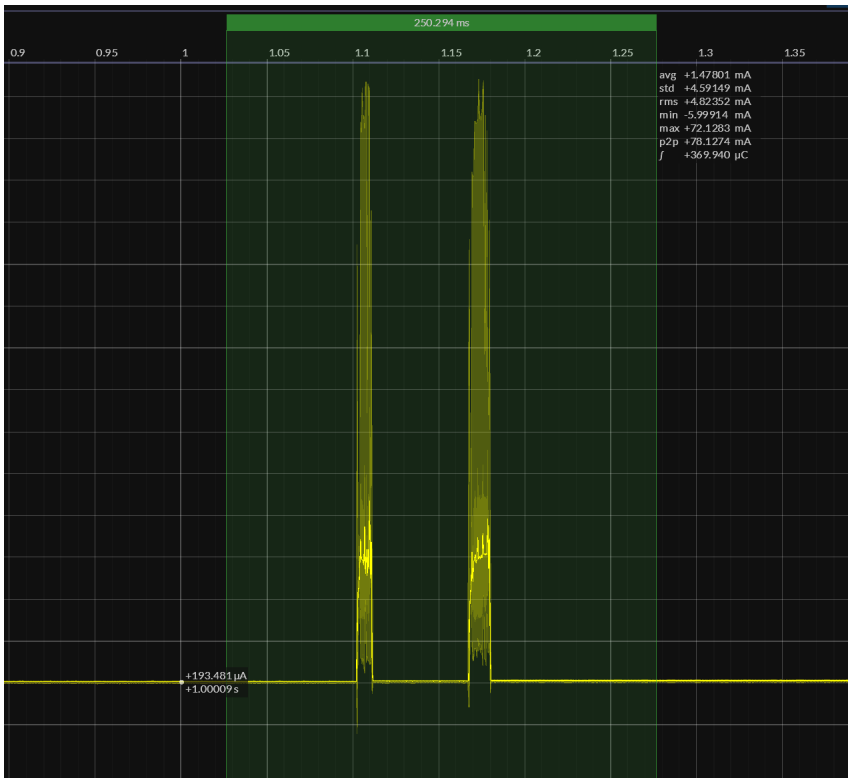
- Without low power mode, one rotation within 250ms
- With low power mode, one rotation within 250ms
- With low power mode, ten rotations within 4.5s

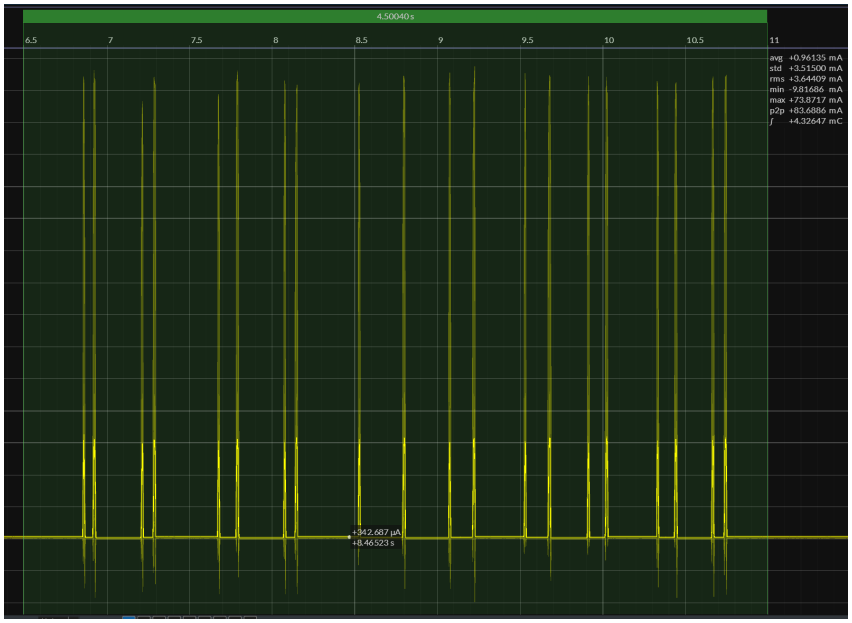
The knob is responsive and consumes less power in low power mode.

9.3.7 Enable and Disable

The component supports enabling and disabling at any time.


250.248 ms
0.65
0.7
0.75
0.8
0.85
0.9
0.95
1
1.05
avg +15.6942 mA
std +7.79070 mA
rms +17.5215 mA
min +0.94449 mA
max +73.4372 mA
p2p +72.4927 mA
∫ +3.92749 mC
+15.4290 mA
+674.950 ms


250.294 ms
0.9
0.95
1
1.05
1.1
1.15
1.2
1.25
1.3
1.35
avg +1.47801 mA
std +4.59149 mA
rms +4.82352 mA
min -5.99914 mA
max +72.1283 mA
p2p +78.1274 mA
∫ +369.940 µC
+193.481 µA
+1.00009 s




```
// Stop knob
iot_knob_stop();
// Resume knob
iot_knob_resume();
```

9.3.8 API Reference

Header File

- components/knob/include/iot_knob.h

Functions

knob_handle_t **iot_knob_create** (const *knob_config_t* *config)

create a knob

Parameters **config** –pointer of knob configuration
Returns A handle to the created knob

esp_err_t **iot_knob_delete** (*knob_handle_t* knob_handle)

Delete a knob.

Parameters **knob_handle** –A knob handle to delete
Returns
- ESP_OK Success
- ESP_FAIL Failure

esp_err_t **iot_knob_register_cb** (*knob_handle_t* knob_handle, *knob_event_t* event, *knob_cb_t* cb, void *usr_data)

Register the knob event callback function.

Parameters
- **knob_handle** –A knob handle to register
- **event** –Knob event
- **cb** –Callback function
- **usr_data** –user data

Returns
- ESP_OK Success
- ESP_FAIL Failure

esp_err_t **iot_knob_unregister_cb**(*knob_handle_t* knob_handle, *knob_event_t* event)

Unregister the knob event callback function.

Parameters
- **knob_handle** –A knob handle to register
- **event** –Knob event

Returns
- ESP_OK Success
- ESP_FAIL Failure

knob_event_t **iot_knob_get_event**(*knob_handle_t* knob_handle)

Get knob event.

Parameters **knob_handle** –A knob handle to register
Returns knob_event_t Knob event

int **iot_knob_get_count_value**(*knob_handle_t* knob_handle)

Get knob count value.

Parameters **knob_handle** –A knob handle to register
Returns int count_value

esp_err_t **iot_knob_clear_count_value**(*knob_handle_t* knob_handle)

Clear knob cout value to zero.

Parameters **knob_handle** –A knob handle to register
Returns
- ESP_OK Success
- ESP_FAIL Failure

esp_err_t **iot_knob_resume**(void)

resume knob timer, if knob timer is stopped. Make sure iot_knob_create() is called before calling this API.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE timer state is invalid.

esp_err_t **iot_knob_stop**(void)

stop knob timer, if knob timer is running. Make sure iot_knob_create() is called before calling this API.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE timer state is invalid

Structures

struct **knob_config_t**

Knob config.

Public Members

uint8_t **default_direction**

0:positive increase 1:negative increase

uint8_t **gpio_encoder_a**
Encoder Pin A

uint8_t **gpio_encoder_b**
Encoder Pin B

bool **enable_power_save**
Enable power save mode

Type Definitions

typedef void (*__knob_cb_t__)(void*, void*)

typedef void *__knob_handle_t__

Enumerations

enum **knob_event_t**
Knob events.

Values:

enumerator **KNOB_LEFT**
EVENT: Rotate to the left

enumerator **KNOB_RIGHT**
EVENT: Rotate to the right

enumerator **KNOB_H_LIM**
EVENT: Count reaches maximum limit

enumerator **KNOB_L_LIM**
EVENT: Count reaches the minimum limit

enumerator **KNOB_ZERO**
EVENT: Count back to 0

enumerator **KNOB_EVENT_MAX**
EVENT: Number of events

enumerator **KNOB_NONE**
EVENT: No event

9.4 Touch Panel

Touch panels are now standard components in display applications. ESP-IoT-Solution provides drivers for common types of touch panel controllers based on the esp_lcd_touch component from ESP-BSP.

9.4.1 Supported Touch Controllers

The following touch panel controllers are currently supported in ESP-IoT-Solution:

Touch Controller	Communication Interface	Component Name
ILI2118	I2C	esp_lcd_touch_ili2118
SPD2010	I2C	esp_lcd_touch_spd2010
ST7123	I2C	esp_lcd_touch_st7123

For additional touch controllers, please refer to the esp_lcd_touch drivers available on the ESP Component Registry.

Common touch controllers supported by ESP-IDF and ESP-BSP include:

- **GT911 / GT1151** - Capacitive touch controllers
- **FT5x06** series (FT5216, FT5316, FT5436, FT6336) - Capacitive touch controllers
- **CST816S** - Capacitive touch controller
- **TT21100** - Capacitive touch controller

9.4.2 Architecture

The `esp_lcd_touch` component provides a unified abstraction layer for different touch panel controllers. This architecture allows:

- **Unified API**: Same interface for all touch controllers
- **Easy Integration**: Seamless integration with LVGL and other GUI libraries
- **Hardware Abstraction**: Switch between touch controllers without changing application code
- **I2C Communication**: Efficient I2C-based communication with touch controllers

9.4.3 Touch Panel Features

Capacitive Touch Panels:

- Multi-touch support (depends on controller)
- No calibration required (calibrated by controller)
- High accuracy and responsiveness
- Gesture support (swipe, zoom, rotate - controller dependent)

Configuration Options:

- Coordinate mirroring (X/Y axis)
- Coordinate swapping (swap X and Y)
- Interrupt pin support
- Reset pin support

9.4.4 Application Example

Basic Touch Panel Initialization

Here's a basic example of initializing a touch panel with I2C communication:

```
#include "esp_lcd_touch.h"
#include "esp_lcd_touch_gt911.h"  // Example with GT911
#include "driver/i2c_master.h"

// Initialize I2C bus
i2c_master_bus_config_t i2c_bus_conf = {
    .clk_source = I2C_CLK_SRC_DEFAULT,
    .sda_io_num = GPIO_NUM_8,
```

(continues on next page)

(continued from previous page)

```
    .scl_io_num = GPIO_NUM_18,
    .i2c_port = I2C_NUM_0,
    .flags.enable_internal_pullup = true,
};
i2c_master_bus_handle_t i2c_bus;
ESP_ERROR_CHECK(i2c_new_master_bus(&i2c_bus_conf, &i2c_bus));

// Initialize touch IO (I2C)
esp_lcd_panel_io_handle_t io_handle = NULL;
esp_lcd_panel_io_i2c_config_t io_config = ESP_LCD_TOUCH_IO_I2C_GT911_CONFIG();
ESP_ERROR_CHECK(esp_lcd_new_panel_io_i2c(i2c_bus, &io_config, &io_handle));

// Configure touch panel
esp_lcd_touch_config_t tp_cfg = {
    .x_max = 800,           // Screen width
    .y_max = 480,           // Screen height
    .rst_gpio_num = GPIO_NUM_NC,  // Reset pin (or GPIO_NUM_NC)
    .int_gpio_num = GPIO_NUM_3,   // Interrupt pin (or GPIO_NUM_NC)
    .levels = {
        .reset = 0,
        .interrupt = 0,
    },
    .flags = {
        .swap_xy = 0,
        .mirror_x = 0,
        .mirror_y = 0,
    },
};

// Initialize touch controller
esp_lcd_touch_handle_t touch_handle = NULL;
ESP_ERROR_CHECK(esp_lcd_touch_new_i2c_gt911(io_handle, &tp_cfg, &touch_handle));
```

Reading Touch Data

To read touch coordinates:

```
// Read touch data (call this periodically)
esp_lcd_touch_read_data(touch_handle);

// Get touch coordinates
uint16_t touch_x[1];
uint16_t touch_y[1];
uint16_t touch_strength[1];
uint8_t touch_cnt = 0;

bool pressed = esp_lcd_touch_get_coordinates(touch_handle,
                                             touch_x,
                                             touch_y,
                                             touch_strength,
                                             &touch_cnt,
                                             1);

if (pressed && touch_cnt > 0) {
    printf("Touch at: X=%d, Y=%d, Strength=%d\n",
           touch_x[0], touch_y[0], touch_strength[0]);
}
```

Integration with LVGL

The ESP LVGL Adapter component provides easy integration with LVGL:

```
#include "esp_lv_adapter.h"

// After creating touch_handle as shown above

esp_lv_adapter_touch_config_t touch_cfg =
    ESP_LV_ADAPTER_TOUCH_DEFAULT_CONFIG(disp, touch_handle);

lv_indev_t *indev = esp_lv_adapter_register_touch(&touch_cfg);
```

9.4.5 Using ESP-IoT-Solution Touch Drivers

Adding Components to Your Project

Method 1: Using Component Manager

Add the touch driver to your project using the IDF Component Manager:

```
# For SPD2010
idf.py add-dependency "espressif/esp_lcd_touch_spd2010"

# For ILI2118
idf.py add-dependency "espressif/esp_lcd_touch_ili2118"

# For ST7123
idf.py add-dependency "espressif/esp_lcd_touch_st7123"
```

Method 2: Using idf_component.yml

Create or edit `idf_component.yml` in your main component:

```
dependencies:
  espressif/esp_lcd_touch_spd2010: "*"
```

Example with SPD2010

```
#include "esp_lcd_touch_spd2010.h"

esp_lcd_panel_io_i2c_config_t io_config = ESP_LCD_TOUCH_IO_I2C_SPD2010_CONFIG();

esp_lcd_touch_config_t tp_cfg = {
    .x_max = 320,
    .y_max = 320,
    .rst_gpio_num = GPIO_NUM_NC,
    .int_gpio_num = GPIO_NUM_NC,
    .levels = {
        .reset = 0,
        .interrupt = 0,
    },
    .flags = {
        .swap_xy = 0,
        .mirror_x = 0,
        .mirror_y = 0,
    },
};
```

(continues on next page)

(continued from previous page)

```
esp_lcd_touch_handle_t tp;
ESP_ERROR_CHECK(esp_lcd_touch_new_i2c_spd2010(io_handle, &tp_cfg, &tp));
```

9.4.6 Touch Rotation and Mirroring

Touch coordinates can be adjusted to match your LCD orientation:

Swap XY Coordinates:

Useful when the touch panel is rotated 90° or 270° relative to the LCD.

```
tp_cfg.flags.swap_xy = 1;  // Swap X and Y coordinates
```

Mirror Coordinates:

Flip the coordinates along X or Y axis:

```
tp_cfg.flags.mirror_x = 1;  // Mirror X axis
tp_cfg.flags.mirror_y = 1;  // Mirror Y axis
```

Setting Resolution:

Always set `x_max` and `y_max` to match your actual LCD resolution:

```
tp_cfg.x_max = 800;  // LCD width
tp_cfg.y_max = 480;  // LCD height
```

9.4.7 Related Examples

- LVGL Common Demo - Shows touch integration with LVGL
- LVGL Dummy Draw - Advanced LVGL example with touch
- RGB LCD with Touch - ESP-IDF official example

9.4.8 Component Links

ESP-IoT-Solution Touch Drivers:

- esp_lcd_touch_ili2118
- esp_lcd_touch_spd2010
- esp_lcd_touch_st7123

More Touch Drivers on Component Registry:

Search for esp_lcd_touch on Component Registry to find additional touch panel drivers.

9.4.9 API Reference

For detailed API documentation, please refer to:

- esp_lcd_touch API Documentation
- Component-specific README files in `components/display/lcd_touch/`

Chapter 10

IR

10.1 IR learn

This is an IR learning component based on the RMT module, capable of receiving infrared signals with a carrier frequency of 38KHz. The received signals are stored and forwarded in raw format, without support for the specific parsing of IR protocols.

10.1.1 Application Examples

Create IR_learn

```
const ir_learn_cfg_t config = {
    .learn_count = 4,               /*!< IR learn count needed */
    .learn_gpio = GPIO_NUM_38,      /*!< IR learn io that consumed by the sensor */
    .clk_src = RMT_CLK_SRC_DEFAULT,/*!< RMT clock source */
    .resolution = 1000000,          /*!< RMT resolution, 1M, 1 tick = 1us*/

    .task_stack = 4096,    /*!< IR learn task stack size */
    .task_priority = 5,    /*!< IR learn task priority */
    .task_affinity = 1,    /*!< IR learn task pinned to core (-1 is no affinity) */
    .callback = cb,        /*!< IR learn result callback for user */
};

ESP_ERROR_CHECK(ir_learn_new(&config, &handle));
```

Callback function

```
 void ir_learn_auto_learn_cb(ir_learn_state_t state, uint8_t sub_step, struct ir_
↪learn_sub_list_head *data)
 {
   switch (state) {
   /**< IR learn ready, after successful initialization */
   case IR_LEARN_STATE_READY:
      ESP_LOGI(TAG, "IR Learn ready");
      break;
   /**< IR learn exit */
```

(continues on next page)

(continued from previous page)

```
    case IR_LEARN_STATE_EXIT:
        ESP_LOGI(TAG, "IR Learn exit");
        break;
    /**< IR learn successfully */
    case IR_LEARN_STATE_END:
        ESP_LOGI(TAG, "IR Learn end");
        ir_learn_save_result(&ir_test_result, data);
        ir_learn_print_raw(data);
        ir_learn_stop(&handle);
        break;
    /**< IR learn failure */
    case IR_LEARN_STATE_FAIL:
        ESP_LOGI(TAG, "IR Learn failed, retry");
        break;
    /**< IR learn step, start from 1 */
    case IR_LEARN_STATE_STEP:
    default:
        ESP_LOGI(TAG, "IR Learn step:[%d][%d]", state, sub_step);
        break;
    }
    return;
}
```

IR_learn supports single and multi-packet learning. Upon successful learning, it notifies the IR_LEARN_STATE_END event. Users can handle the learning result independently, and the format of the learned data packet is described in struct `ir_learn_sub_list_t`.

The learning process automatically verifies the data. If the verification fails, it notifies the IR_LEARN_STATE_FAIL event.

- For each packet, if the quantity of symbols is inconsistent, it is considered a learning failure.
- For each level' s time difference, if it exceeds the threshold, it is considered a learning failure. (Threshold can be adjusted using the RMT_DECODE_MARGIN_US in menuconfig.)

Send out

```
void ir_learn_test_tx_raw(struct ir_learn_sub_list_head *rmt_out)
{
    /**< Create RMT TX channel */
    rmt_tx_channel_config_t tx_channel_cfg = {
        .clk_src = RMT_CLK_SRC_DEFAULT,
        .resolution_hz = IR_RESOLUTION_HZ,
        .mem_block_symbols = 128,  // amount of RMT symbols that the channel can␣
↪store at a time
        .trans_queue_depth = 4,    // number of transactions that allowed to pending␣
↪in the background
        .gpio_num = IR_TX_GPIO_NUM,
    };
    rmt_channel_handle_t tx_channel = NULL;
    ESP_ERROR_CHECK(rmt_new_tx_channel(&tx_channel_cfg, &tx_channel));

    /**< Modulate carrier to TX channel */
    rmt_carrier_config_t carrier_cfg = {
        .duty_cycle = 0.33,
        .frequency_hz = 38000, // 38KHz
    };
    ESP_ERROR_CHECK(rmt_apply_carrier(tx_channel, &carrier_cfg));

    rmt_transmit_config_t transmit_cfg = {
        .loop_count = 0, // no loop
```

(continues on next page)

(continued from previous page)

```
    };

    /**< Install IR encoder, here's raw format */
    ir_encoder_config_t raw_encoder_cfg = {
        .resolution = IR_RESOLUTION_HZ,
    };
    rmt_encoder_handle_t raw_encoder = NULL;
    ESP_ERROR_CHECK(ir_encoder_new(&raw_encoder_cfg, &raw_encoder));

    ESP_ERROR_CHECK(rmt_enable(tx_channel));  // Enable RMT TX channels

    /**< Traverse and send commands */
    struct ir_learn_sub_list_t *sub_it;
    SLIST_FOREACH(sub_it, rmt_out, next) {
        vTaskDelay(pdMS_TO_TICKS(sub_it->timediff / 1000));

        rmt_symbol_word_t *rmt_symbols = sub_it->symbols.received_symbols;
        size_t symbol_num = sub_it->symbols.num_symbols;

        ESP_ERROR_CHECK(rmt_transmit(tx_channel, raw_encoder, rmt_symbols, symbol_
↪num, &transmit_cfg));
        rmt_tx_wait_all_done(tx_channel, -1);  // wait all transactions finished
    }

    /**< Disable and delete RMT TX channels */
    rmt_disable(tx_channel);
    rmt_del_channel(tx_channel);
    raw_encoder->del(raw_encoder);
}
```

10.1.2 API Reference

Header File

- components/ir/ir_learn/include/ir_learn.h

Functions

esp_err_t **ir_learn_new** (const *ir_learn_cfg_t* *cfg, *ir_learn_handle_t* *handle_out)

Create new IR learn handle.

Parameters
- **cfg** –**[in]** Config for IR learn
- **handle_out** –**[out]** New IR learn handle

Returns
- ESP_OK Device handle creation success.
- ESP_ERR_INVALID_ARG Invalid device handle or argument.
- ESP_ERR_NO_MEM Memory allocation failed.

esp_err_t **ir_learn_restart** (*ir_learn_handle_t* ir_learn_hdl)

Restart IR learn process.

Parameters **ir_learn_hdl** –**[in]** IR learn handle

Returns
- ESP_OK Restart process success.
- ESP_ERR_INVALID_ARG Invalid device handle or argument.

esp_err_t **ir_learn_stop** (*ir_learn_handle_t* *ir_learn_hdl)

Stop IR learn process.

Note: Delete all

Parameters **ir_learn_hdl** –**[in]** IR learn handle

Returns

- ESP_OK Stop process success.
- ESP_ERR_INVALID_ARG Invalid device handle or argument.

esp_err_t **ir_learn_add_list_node** (struct ir_learn_list_head *learn_head)

Add IR learn list node, every new learn list will create it.

Parameters **learn_head** –**[in]** IR learn list head

Returns

- ESP_OK Create learn list success.
- ESP_ERR_NO_MEM Memory allocation failed.

esp_err_t **ir_learn_add_sub_list_node** (struct ir_learn_sub_list_head *sub_head, uint32_t timediff, const rmt_rx_done_event_data_t *symbol)

Add IR learn sub step list node, every sub step should be added.

Parameters

- **sub_head** –**[in]** IR learn sub step list head
- **timediff** –**[in]** Time diff between each sub step
- **symbol** –**[in]** symbols of each sub step

Returns

- ESP_OK Create learn list success.
- ESP_ERR_NO_MEM Memory allocation failed.

esp_err_t **ir_learn_clean_data** (struct ir_learn_list_head *learn_head)

Delete IR learn list node, will recursively delete sub steps.

Parameters **learn_head** –**[in]** IR learn list head

- ESP_OK Stop process success.
- ESP_ERR_INVALID_ARG Invalid device handle or argument.

esp_err_t **ir_learn_clean_sub_data** (struct ir_learn_sub_list_head *sub_head)

Delete sub steps.

Parameters **sub_head** –**[in]** IR learn sub list head

- ESP_OK Stop process success.
- ESP_ERR_INVALID_ARG Invalid device handle or argument.

esp_err_t **ir_learn_check_valid** (struct ir_learn_list_head *learn_head, struct ir_learn_sub_list_head *result_out)

Add IR learn list node, every new learn list will create it.

Parameters

- **learn_head** –**[in]** IR learn list head
- **result_out** –**[out]** IR learn result

Returns

- ESP_OK Get learn result process.
- ESP_ERR_INVALID_SIZE Size error.

esp_err_t **ir_learn_print_raw** (struct ir_learn_sub_list_head *cmd_list)

Print the RMT symbols.

Parameters **cmd_list** –**[in]** IR learn list head

- ESP_OK Stop process success.
- ESP_ERR_INVALID_ARG Invalid device handle or argument.

Structures

struct **ir_learn_sub_list_t**

An element in the list of infrared (IR) learn data packets.

Public Functions

SLIST_ENTRY (ir_learn_sub_list_t) next

Pointer to the next packet

Public Members

uint32_t **timediff**

The interval time from the previous packet (ms)

rmt_rx_done_event_data_t **symbols**

Received RMT symbols

struct **ir_learn_list_t**

The head of a list of infrared (IR) learn data packets.

Public Functions

SLIST_ENTRY (ir_learn_list_t) next

Pointer to the next packet

Public Members

struct ir_learn_sub_list_head **cmd_sub_node**

Package head of every cmd

struct **ir_learn_cfg_t**

IR learn configuration.

Public Members

rmt_clock_source_t **clk_src**

RMT clock source

uint32_t **resolution**

RMT resolution, in Hz

int **learn_count**

IR learn count needed

int **learn_gpio**

IR learn io that consumed by the sensor

ir_learn_result_cb **callback**

IR learn result callback for user

int **task_priority**

IR learn task priority

int **task_stack**

IR learn task stack size

int **task_affinity**

IR learn task pinned to core (-1 is no affinity)

Type Definitions

typedef void ***ir_learn_handle_t**

Type of IR learn handle.

typedef void (***ir_learn_result_cb**)(*ir_learn_state_t* state, uint8_t sub_step, struct ir_learn_sub_list_head *data)

IR learn result user callback.

Param state [out] IR learn step
Param sub_step [out] Interval less than 500 ms, we think it' s the same command
Param data [out] Command list of this step

Enumerations

enum **ir_learn_state_t**

Type of IR learn step.

Values:

enumerator **IR_LEARN_STATE_STEP**

IR learn step, start from 1

enumerator **IR_LEARN_STATE_READY**

IR learn ready, after successful initialization

enumerator **IR_LEARN_STATE_END**

IR learn successfully

enumerator **IR_LEARN_STATE_FAIL**

IR learn failure

enumerator **IR_LEARN_STATE_EXIT**

IR learn exit

Chapter 11

Low-Power solution

11.1 ESP32-P4 Battery Backup Solution

The ESP32-P4 integrates RTC (Real-Time Clock) functionality, which includes a battery backup (VBAT), an external 32.768kHz crystal oscillator input, and power switching control, allowing it to maintain accurate time even when the main power is disconnected.

> **Warning:** ESP-IDF does not fully support the VBAT scheme yet, and the current scheme is for testing purposes only. Please follow the process below to modify the ESP-IDF driver on the master branch.

Supports the following features:

- Supports timekeeping functionality during main power loss, with backup power supplied by the battery backup (VBAT) after power failure.
- Supports automatic power source switching by the PMU (Power Management Unit) during sleep/wake states.
- Supports battery backup voltage detection with threshold alarms.
- Supports charging of the battery backup; when VDDA is present, the battery backup (VBAT) can be charged through an internal resistor.

11.1.1 Battery Connection and Circuit Protection

The connection of the ESP32-P4 backup battery is very simple. The VBAT pin has an integrated charging protection circuit, so no external resistors or diodes are required, effectively reducing the harmful risks associated with adding components in the backup battery path.

Selection of Battery Backup

- Low Self-Discharge Rate: The battery should have excellent leakage resistance.
- Charge and Discharge Performance: Batteries with a higher charge/discharge cycle count can extend the product' s lifespan.
- Discharge Current: The VBAT supply consumes very little power (a few microamperes), so most batteries' discharge capabilities will be sufficient.
- Charging Current: VDDA supports charging the backup battery on VBAT. The charging current limit resistor can be configured by software, ranging from 1kΩ to 9kΩ.

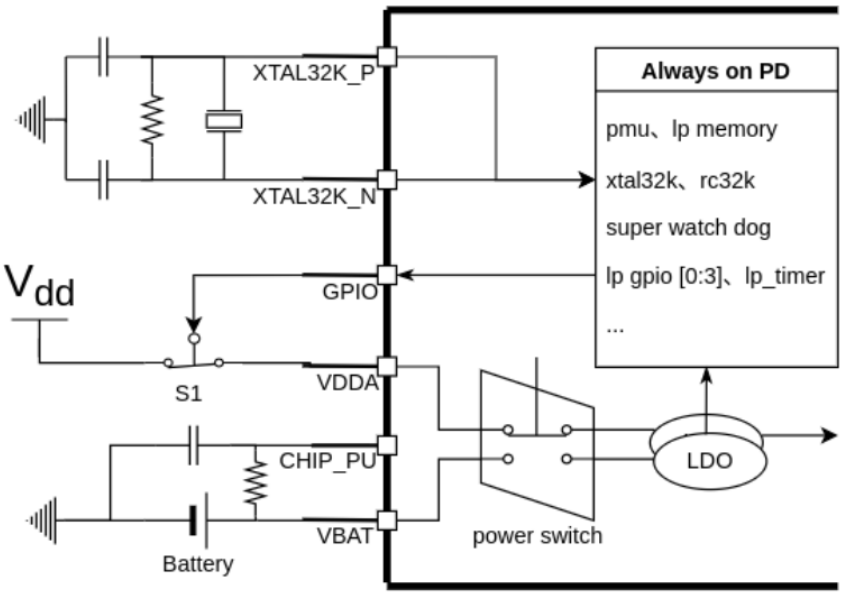


Fig. 1: ESP32-P4 Backup Battery Connection Diagram

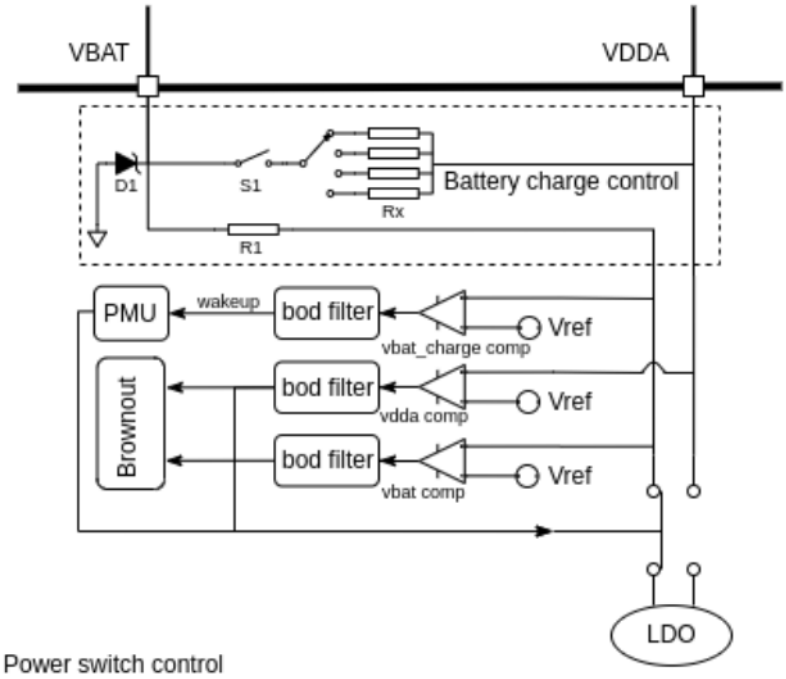


Fig. 2: ESP32-P4 Battery Switch Control Block Diagram

Warning: It is strictly prohibited to connect a 3.6V rechargeable battery, as the voltage of a fully charged 3.6V battery can reach around 4V. Since the input voltage range for the VBAT pin is 2.3V ~ 3.6V, this could potentially damage the VBAT pin.

Decoupling Capacitor on the Battery

In circuit design, users typically add a decoupling capacitor for each power supply pin. However, when VBAT is used as a backup power input, leakage on the capacitor will significantly shorten the battery life. Moreover, since the battery itself has good capacitive characteristics, no external decoupling capacitor is needed.

Note: D1 and R1 are ESD protection components, and Rx is the charging protection resistor, limiting the charging current to below 1 mA. The protection circuit meets UL certification testing requirements.

11.1.2 Power Switching

The power selection between VBAT and VDDA is automatically controlled by the PMU during sleep and wake states, or can be manually selected. Users need to configure the registers in advance. When the chip is in sleep mode, VDDA or VBAT can be configured as the power input.

- PMU_HP_SLEEP_VDDBAT_MODE: Selects the power supply during the HP_SLEEP state. 0: VDDA power supply, 1: VBAT power supply, 2: Auto selection.
- PMU_VDDBAT_SW_UPDATE: Set to 1 to make the configuration in the `PMU_VDDBAT_CFG_REG` register effective.
- PMU_LP_ANA_WAIT_TARGET: Sets the delay time during the wake-up process. The duration should be longer than the power-up stabilization time of the external VDDA power supply.

Note: Currently, ESP-IDF does not fully support the PMU driver. If you need to test the VBAT functionality, please follow the testing procedure below to add the necessary code in ESP-IDF.

The testing procedure is as follows:

1. Add the necessary header files in `components/esp_hw_support/port/esp32p4/pmu_sleep.c`

```
#include "soc/pmu_reg.h"
#include "soc/lp_analog_peri_reg.h"
#include "soc/lp_gpio_struct.h"
#include "esp_private/regi2c_ctrl.h"
#include "soc/lp_analog_peri_reg.h"
#include "soc/lp_analog_peri_struct.h"
#include "esp_private/regi2c_ctrl.h"
#include "soc/regi2c_bias.h"
```

2. Add the following code above `pmu_ll_hp_set_sleep_enable(PMU_instance()->hal->dev);` in the `pmu_sleep_start` function in `components/esp_hw_support/port/esp32p4/pmu_sleep.c`:

Method 1: Automatic Switching Between VDDA and VBAT:

```
#define VBAT_MODE_VDDA 0
#define VBAT_MODE_VBAT 1
#define VBAT_MODE_AUTO 2

REG_SET_FIELD(PMU_HP_SLEEP_LP_DIG_POWER_REG, PMU_HP_SLEEP_VDDBAT_MODE, VBAT_MODE_
↪AUTO);
```

(continues on next page)

(continued from previous page)

```
REG_SET_BIT(PMU_VDDBAT_CFG_REG, PMU_VDDBAT_SW_UPDATE);
while(VBAT_MODE_AUTO != REG_GET_FIELD(PMU_VDDBAT_CFG_REG, PMU_ANA_VDDBAT_MODE));
```

Method 2: Manual Configuration of VDDA and VBAT:

```
#define VBAT_MODE_VDDA 0
#define VBAT_MODE_VBAT 1
#define VBAT_MODE_AUTO 2

REG_SET_FIELD(PMU_HP_SLEEP_LP_DIG_POWER_REG, PMU_HP_SLEEP_VDDBAT_MODE, VBAT_MODE_
↪VDDA);
REG_SET_BIT(PMU_VDDBAT_CFG_REG, PMU_VDDBAT_SW_UPDATE);
while(VBAT_MODE_VDDA != REG_GET_FIELD(PMU_VDDBAT_CFG_REG, PMU_ANA_VDDBAT_MODE));
REG_SET_FIELD(PMU_LP_SLEEP_LP_DIG_POWER_REG, PMU_LP_SLEEP_VDDBAT_MODE, VBAT_MODE_
↪VBAT);
REG_SET_FIELD(PMU_SLP_WAKEUP_CNTL5_REG, PMU_LP_ANA_WAIT_TARGET, 0xFF);
```

Note: After testing, the average VBAT current in deep sleep is 7 µA, and the average VBAT current in light sleep is 21 µA.

Backup Battery Runtime

Taking the CR2032 battery (225 mAh) as an example, the theoretical standby time in deep sleep is 3.669 years.

11.1.3 Brown-out Detector & BOD filter

The Brown-out Detector of the ESP32-P4 supports monitoring the voltage of VDDA and VBAT. It triggers a signal when the voltage rapidly drops below a preset threshold and performs the corresponding actions.

Threshold Configuration and Voltage Relationship:

Threshold Configuration	Voltage
0	2.52v
1	2.57v
2	2.63v
3	2.68v
4	2.74v
5	2.78v
6	2.83v
7	2.89v

Create a new test project and add the following code to check the undervoltage flag:

```
REGI2C_WRITE_MASK(I2C_BIAS, I2C_BIAS_OR_DREFL_VDDA, 6);
REGI2C_WRITE_MASK(I2C_BIAS, I2C_BIAS_OR_DREFH_VDDA, 7);
↪                                          // Vdda comparator voltage threshold
↪set to 2.83 - 2.89
REGI2C_WRITE_MASK(I2C_BIAS, I2C_BIAS_OR_DREFL_VBAT, 6);
REGI2C_WRITE_MASK(I2C_BIAS, I2C_BIAS_OR_DREFH_VBAT, 7);
↪                                          // Vbat comparator voltage threshold
↪set to 2.83 - 2.89

REG_SET_FIELD(LP_ANALOG_PERI_VDDBAT_CHARGE_CNTL_REG, LP_ANALOG_PERI_VDDBAT_CHARGE_
↪UNDERVOLTAGE_TARGET, 20);
REG_SET_FIELD(LP_ANALOG_PERI_VDDBAT_CHARGE_CNTL_REG, LP_ANALOG_PERI_VDDBAT_CHARGE_
↪UPVOLTAGE_TARGET, 10);                    // Set the time threshold for the
↪BOD filter, in 20 MHz cycles.
```

(continues on next page)

(continued from previous page)

```
REG_SET_BIT(LP_ANALOG_PERI_VDDBAT_CHARGE_CNTL_REG, LP_ANALOG_PERI_VDDBAT_CHARGE_
↪CNT_CLR);
esp_rom_delay_us(30);
REG_CLR_BIT(LP_ANALOG_PERI_VDDBAT_CHARGE_CNTL_REG, LP_ANALOG_PERI_VDDBAT_CHARGE_
↪CNT_CLR);

while (1)
{
    printf("value:%lx\n", REG_READ(LP_ANALOG_PERI_VDD_SOURCE_CNTL_REG));
    REG_WRITE(LP_ANALOG_PERI_VDD_SOURCE_CNTL_REG, 0X30020000);                 ␣
↪                                                  // clear VBAT flag
    REG_WRITE(LP_ANALOG_PERI_VDD_SOURCE_CNTL_REG, 0X30040000);                 ␣
↪                                                  // clear VDDA flag
    vTaskDelay(500 / portTICK_PERIOD_MS);
}
```

11.1.4 Battery Charging Circuit

When a rechargeable battery is selected as the VBAT power input, the chip supports detecting the battery voltage and, when it falls below the set threshold, it charges the VBAT-side battery through VDDA in reverse.

Threshold Configuration and Voltage Relationship:

Threshold Configuration	Voltage
0	2.52v
1	2.57v
2	2.63v
3	2.68v
4	2.74v
5	2.78v
6	2.83v
7	2.89v

Create a new test project and add the following code to check the undervoltage flag:

```
REGI2C_WRITE_MASK(I2C_BIAS, I2C_BIAS_OR_DREFL_VDDA, 6);
REGI2C_WRITE_MASK(I2C_BIAS, I2C_BIAS_OR_DREFH_VDDA, 7);                        ␣
↪                                                  // Vdda comparator voltage threshold␣
↪set to 2.83 - 2.89
REGI2C_WRITE_MASK(I2C_BIAS, I2C_BIAS_OR_DREFL_VBAT, 6);
REGI2C_WRITE_MASK(I2C_BIAS, I2C_BIAS_OR_DREFH_VBAT, 7);                        ␣
↪                                                  // Vbat comparator voltage threshold␣
↪set to 2.83 - 2.89

REGI2C_WRITE_MASK(I2C_BIAS, I2C_BIAS_OR_DREFL_VBAT_CHARGER, 6);
REGI2C_WRITE_MASK(I2C_BIAS, I2C_BIAS_OR_DREFH_VBAT_CHARGER, 7);                ␣
↪                                                  // Charging detection comparator␣
↪threshold setting.
REGI2C_WRITE_MASK(I2C_BIAS, I2C_BIAS_OR_FORCE_PU_VBAT_CHARGER, 1);             ␣
↪                                                  // Force power-on of the vbat_
↪charger comparator.

REG_SET_FIELD(LP_ANALOG_PERI_VDDBAT_CHARGE_CNTL_REG, LP_ANALOG_PERI_VDDBAT_CHARGE_
↪UNDERVOLTAGE_TARGET, 20);
REG_SET_FIELD(LP_ANALOG_PERI_VDDBAT_CHARGE_CNTL_REG, LP_ANALOG_PERI_VDDBAT_CHARGE_
↪UPVOLTAGE_TARGET,    10);                      // Set the time threshold for the␣
↪BOD filter, in 20 MHz cycles.
REG_SET_BIT(LP_ANALOG_PERI_VDDBAT_CHARGE_CNTL_REG, LP_ANALOG_PERI_VDDBAT_CHARGE_
↪CNT_CLR);
```

(continues on next page)

(continued from previous page)

```
esp_rom_delay_us(30);
REG_CLR_BIT(LP_ANALOG_PERI_VDDBAT_CHARGE_CNTL_REG, LP_ANALOG_PERI_VDDBAT_CHARGE_
↪CNT_CLR);

while(1)
{
    printf("value:%lx\n", REG_GET_BIT(LP_ANALOG_PERI_VDDBAT_CHARGE_CNTL_REG, LP_
↪ANALOG_PERI_VDDBAT_CHARGE_UNDERVOLTAGE_FLAG)); // Get VBAT pin charge flag
    vTaskDelay(500 / portTICK_PERIOD_MS);
}
```

11.1.5 Peripheral Support in VBAT Power Supply Scenario

RTC TIMER

After switching to VBAT power supply when entering sleep mode, the RTC TIMER continues to count normally after VDDA is powered off. After restarting, the time display is normal. The test code is as follows:

```
time_t now;
char strftime_buf[64];
struct tm timeinfo;
time(&now);
localtime_r(&now, &timeinfo);
strftime(strftime_buf, sizeof(strftime_buf), "%c", &timeinfo);
printf( "The current date/time is: %s\n", strftime_buf);

const gpio_config_t config = {
    .pin_bit_mask = BIT(GPIO_NUM_0),
    .mode = GPIO_MODE_INPUT,
};

ESP_ERROR_CHECK(gpio_config(&config));
esp_deep_sleep_enable_gpio_wakeup(BIT(GPIO_NUM_0), 0);

esp_deep_sleep_start();
```

Note: You need to add the power switching code in ESP-IDF according to the above 'Power Switching' section. Meanwhile, the test process involved waiting for a period after VDDA power-off, then powering it back on, and using GPIO0 to wake up the ESP32-P4, comparing the time difference before and after sleep.

LP GPIO/ADC/UART Peripheral

Please refer to the test driver in `examples/system/ulp/lp_core` in ESP-IDF for testing. The current GPIO/ADC/UART test already support ESP32-P4.

Chapter 12

Sensors

12.1 Sensor Hub

Sensor Hub is a sensor management component that can realize hardware abstraction, device management and data distribution for sensor devices. When developing applications based on Sensor Hub, users do not have to deal with complex sensor implementations, but only need to make simple selections for sensor operation, acquisition interval, range, etc. and then register callback functions to the event messages of your interests. By doing so, users can receive notifications when sensor states are switched or when data is collected.

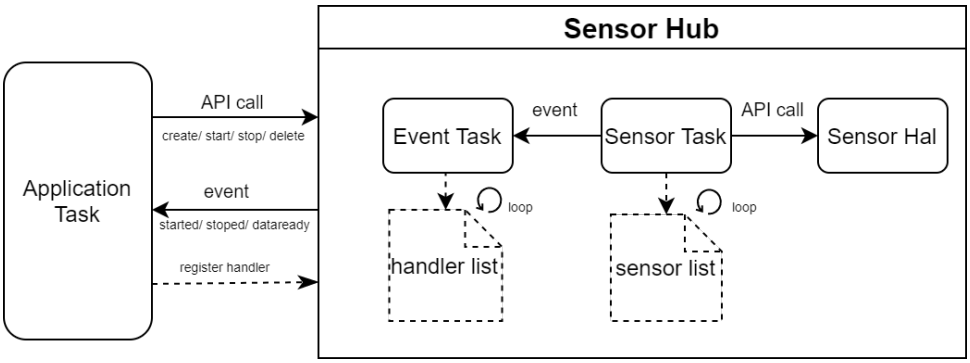


Fig. 1: Sensor Hub Programming Model

Sensor Hub supports loading sensor drivers as components. Users only need to input the name of the sensor registered with the Sensor Hub to directly access the corresponding sensor driver. Additionally, as this component enables centralized management of sensors, it not only simplifies operations but also improves operational efficiency. It can serve as a foundational component for sensor applications and can be applied to various intelligent scenarios such as environmental sensing, motion sensing, health management, and more.

12.1.1 Instructions

`sensor_hub` uses the linker script generation mechanism to register sensor drivers into specific target file sections. For application developers, there is no need to focus on the specific implementation of the sensor drivers; simply adding the corresponding sensor component will load the respective driver.

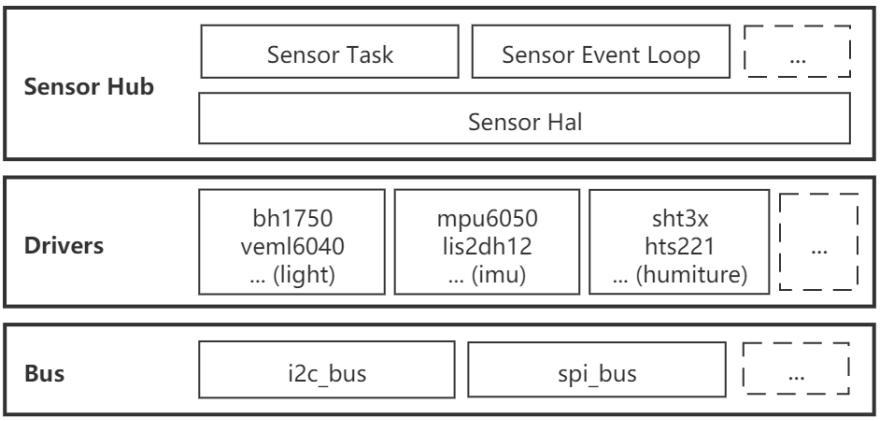


Fig. 2: Sensor Hub Driver

Driver developer:

Taking the `ShT3X` temperature and humidity sensor as an example, driver developers need to populate the `humiture_impl_t` structure with operations related to the sensor and create a sensor detection function. `sensor_hub` provides a sensor registration interface: `SENSOR_HUB_DETECT_FN`, where driver developers can directly insert the corresponding functions into the registration interface.

```
static humiture_impl_t sht3x_impl = {
    .init = humiture_sht3x_init,
    .deinit = humiture_sht3x_deinit,
    .test = humiture_sht3x_test,
    .acquire_humidity = humiture_sht3x_acquire_humidity,
    .acquire_temperature = humiture_sht3x_acquire_temperature,
};

SENSOR_HUB_DETECT_FN(HUMITURE_ID, sht3x, &sht3x_impl);
```

At the same time, add the interface dependency in the component' s `CMakeLists.txt`:

```
target_link_libraries(${COMPONENT_LIB} INTERFACE "-u humiture_sht3x_init")
```

Application developer

1. Add `sensor_hub` and the required sensor components in the project' s `idf_component.yml`.
2. Create a sensor instance: Use *`iot_sensor_create()`* to create a sensor instance. The parameters include the sensor name, sensor configuration options, and a pointer to the sensor handle. The sensor name is used to locate and load the driver registered in the `sensor_hub`. If the sensor supports configurable addresses, multiple instances of the same sensor name can be created. In the configuration options, `bus` specifies the bus to which the sensor is mounted; `addr` specifies the address corresponding to the sensor; `type` specifies the type of the sensor ; `mode` specifies the operating mode of the sensor; and `min_delay` specifies the sensor' s data collection interval. Other options are optional. Upon successful creation, the sensor handle is obtained.
3. Register callback functions for sensor events: when a sensor event occurs, the callback functions will be called in sequence. There are two ways to register a callback function, and the instance handler of the event callback function will be returned after the registration succeed:
   - Use *`iot_sensor_handler_register()`* to register a callback function with the sensor handler
   - Use *`iot_sensor_handler_register_with_type()`* to register a callback function with the sensor type
4. Start a sensor: use *`iot_sensor_start()`* to start a specific sensor. After started, it will trigger a `SENSOR_STARTED` event, then it will collect the sensor data continuously with a set of period and trigger `SEN-`

SOR_XXXX_DATA_READY event. The event callback function can obtain the specific data of each event via the event_data parameter;

5. Stop a sensor: use *iot_sensor_stop()* to stop a specified sensor temporarily. After stopped, the sensor will send out a SENSOR_STOPED event and then stop the data collecting work. If the driver of this sensor supports power management, the sensor will be set to sleep mode in this stage;
6. Unregister callback functions for sensor events: the user program can unregister an event at any time using the instance handler of this event callback function, and this callback function will not be called again when this event occurs afterwards. There are also two ways to do so:
   - Use *iot_sensor_handler_unregister()* to unregister the callback function with the sensor handler
   - Use *iot_sensor_handler_unregister_with_type()* to unregister the callback function with the sensor type
7. Delete sensors: use *iot_sensor_delete()* to delete the corresponding sensor to release the allocated memory and other resources.

12.1.2 Examples

1. Sensor control LED example: sensors/sensor_control_led.
2. Sensor hub monitor example: sensors/sensor_hub_monitor.

12.1.3 API Reference

Header File

- components/sensors/sensor_hub/include/sensor_type.h

Structures

struct **sensor_data_t**

sensor data type

Public Members

int64_t **timestamp**

timestamp

const char ***sensor_name**

sensor name

sensor_type_t **sensor_type**

sensor type

uint8_t **sensor_addr**

sensor addr

int32_t **event_id**

reserved for future use

uint32_t **min_delay**

minimum delay between two events, unit: ms

axis3_t **acce**

Accelerometer. unit: G

axis3_t **gyro**

Gyroscope. unit: dps

axis3_t **mag**

Magnetometer. unit: Gauss

float **temperature**

Temperature. unit: dCelsius

float **humidity**

Relative humidity. unit: percentage

float **baro**

Pressure. unit: pascal (Pa)

float **light**

Light. unit: lux

rgbw_t **rgbw**

Color. unit: lux

uv_t **uv**

ultraviole unit: lux

float **proximity**

Distance. unit: centimeters

float **hr**

Heat rate. unit: HZ

float **tvoc**

TVOC. unit: permillage

float **noise**

Noise Loudness. unit: HZ

float **step**

Step sensor. unit: 1

float **force**

Force sensor. unit: mN

float **current**

Current sensor unit: mA

float **voltage**

Voltage sensor unit: mV

float **data**[4]

for general use

struct **sensor_data_group_t**

sensor data group type

Public Members

uint8_t **number**

effective data number

sensor_data_t **sensor_data**[SENSOR_DATA_GROUP_MAX_NUM]

data buffer

Macros

SENSOR_EVENT_ANY_ID

register handler for any event id

Type Definitions

typedef void ***sensor_device_impl_t**

typedef void ***sensor_driver_handle_t**

hal level sensor driver handle

typedef void ***bus_handle_t**

i2c/spi bus handle

Enumerations

enum **sensor_type_t**

sensor type

Values:

enumerator **NULL_ID**

NULL

enumerator **HUMITURE_ID**

humidity or temperature sensor

enumerator **IMU_ID**

gyro or acc sensor

enumerator **LIGHT_SENSOR_ID**

light illumination or uv or color sensor

enumerator **SENSOR_TYPE_MAX**

max sensor type

enum **sensor_command_t**

sensor operate command

Values:

enumerator **COMMAND_SET_MODE**

set measure mode

enumerator **COMMAND_SET_RANGE**

set measure range

enumerator **COMMAND_SET_ODR**

set output rate

enumerator **COMMAND_SET_POWER**

set power mode

enumerator **COMMAND_SELF_TEST**

sensor self test

enumerator **COMMAND_MAX**

max sensor command

enum **sensor_power_mode_t**

sensor power mode

Values:

enumerator **POWER_MODE_WAKEUP**

wakeup from sleep

enumerator **POWER_MODE_SLEEP**

set to sleep

enumerator **POWER_MAX**

max sensor power mode

enum **sensor_mode_t**

sensor acquire mode

Values:

enumerator **MODE_DEFAULT**

default work mode

enumerator **MODE_POLLING**

polling acquire with a interval time

enumerator **MODE_INTERRUPT**

interrupt mode, acquire data when interrupt comes

enumerator **MODE_MAX**

max sensor mode

enum **sensor_range_t**

sensor acquire range

Values:

enumerator **RANGE_DEFAULT**

default range

enumerator **RANGE_MIN**

minimum range for high-speed or high-precision

enumerator **RANGE_MEDIUM**

medium range for general use

enumerator **RANGE_MAX**

maximum range for full scale

enum **sensor_event_id_t**

sensor general events

Values:

enumerator **SENSOR_STARTED**

sensor started, data acquire will be started

enumerator **SENSOR_STOPED**

sensor stopped, data acquire will be stopped

enumerator **SENSOR_EVENT_COMMON_END**

max common events id

enum **sensor_data_event_id_t**

sensor data ready events

Values:

enumerator **SENSOR_ACCE_DATA_READY**

Accelerometer data ready

enumerator **SENSOR_GYRO_DATA_READY**
Gyroscope data ready

enumerator **SENSOR_MAG_DATA_READY**
Magnetometer data ready

enumerator **SENSOR_TEMP_DATA_READY**
Temperature data ready

enumerator **SENSOR_HUMI_DATA_READY**
Relative humidity data ready

enumerator **SENSOR_BARO_DATA_READY**
Pressure data ready

enumerator **SENSOR_LIGHT_DATA_READY**
Light data ready

enumerator **SENSOR_RGBW_DATA_READY**
Color data ready

enumerator **SENSOR_UV_DATA_READY**
ultraviolet data ready

enumerator **SENSOR_PROXI_DATA_READY**
Distance data ready

enumerator **SENSOR_HR_DATA_READY**
Heat rate data ready

enumerator **SENSOR_TVOC_DATA_READY**
TVOC data ready

enumerator **SENSOR_NOISE_DATA_READY**
Noise Loudness data ready

enumerator **SENSOR_STEP_DATA_READY**
Step data ready

enumerator **SENSOR_FORCE_DATA_READY**
Force data ready

enumerator **SENSOR_CURRENT_DATA_READY**
Current data ready

enumerator **SENSOR_VOLTAGE_DATA_READY**
Voltage data ready

enumerator **SENSOR_EVENT_ID_END**
max common events id

Header File

- components/sensors/sensor_hub/include/iot_sensor_hub.h

Functions

esp_err_t **iot_sensor_create** (const char *sensor_name, const *sensor_config_t* *config, *sensor_handle_t* *p_sensor_handle)

Create a sensor instance with specified name and desired configurations.

Parameters
- **sensor_name** –sensor's name. Sensor information will be registered by ESP_SENSOR_DETECT_FN
- **config** –sensor's configurations detailed in *sensor_config_t*
- **p_sensor_handle** –return sensor handle if succeed, NULL if failed.

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **iot_sensor_start** (*sensor_handle_t* sensor_handle)

start sensor acquisition, post data ready events when data acquired. if start succeed, sensor will start to acquire data with desired mode and post events in min_delay(ms) intervals SENSOR_STARTED event will be posted.

Parameters **sensor_handle** –sensor handle for operation

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **iot_sensor_stop** (*sensor_handle_t* sensor_handle)

stop sensor acquisition, and stop post data events. if stop succeed, SENSOR_STOPED event will be posted.

Parameters **sensor_handle** –sensor handle for operation

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **iot_sensor_delete** (*sensor_handle_t* p_sensor_handle)

delete and release the sensor resource.

Parameters **p_sensor_handle** –sensor handle, will set to NULL if delete succeed.

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail

int **iot_sensor_scan** ()

Scan for valid sensors registered in the system.

Returns int number of valid sensors

esp_err_t **iot_sensor_handler_register** (*sensor_handle_t* sensor_handle, *sensor_event_handler_t* handler, *sensor_event_handler_instance_t* *context)

Register a event handler to a sensor's event with sensor_handle.

Parameters
- **sensor_handle** –sensor handle for operation
- **handler** –the handler function which gets called when the sensor's any event is dispatched
- **context** –An event handler instance object related to the registered event handler and data, can be NULL. This needs to be kept if the specific callback instance should be unregistered before deleting the whole event loop. Registering the same event handler multiple times is possible and yields distinct instance objects. The data can be the same

for all registrations. If no unregistration is needed but the handler should be deleted when the event loop is deleted, instance can be NULL.

Returns esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **iot_sensor_handler_unregister** (*sensor_handle_t* sensor_handle, *sensor_event_handler_instance_t* context)

Unregister a event handler from a sensor's event.

Parameters

- **sensor_handle** –sensor handle for operation
- **context** –the instance object of the registration to be unregistered

Returns esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **iot_sensor_handler_register_with_type** (*sensor_type_t* sensor_type, int32_t event_id, *sensor_event_handler_t* handler, *sensor_event_handler_instance_t* *context)

Register a event handler with sensor_type instead of sensor_handle. the api only care about the event type, don't care who post it.

Parameters

- **sensor_type** –sensor type declared in sensor_type_t.
- **event_id** –sensor event declared in sensor_event_id_t and sensor_data_event_id_t
- **handler** –the handler function which gets called when the event is dispatched
- **context** –An event handler instance object related to the registered event handler and data, can be NULL. This needs to be kept if the specific callback instance should be unregistered before deleting the whole event loop. Registering the same event handler multiple times is possible and yields distinct instance objects. The data can be the same for all registrations. If no unregistration is needed but the handler should be deleted when the event loop is deleted, instance can be NULL.

Returns esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **iot_sensor_handler_unregister_with_type** (*sensor_type_t* sensor_type, int32_t event_id, *sensor_event_handler_instance_t* context)

Unregister a event handler from a event. the api only care about the event type, don't care who post it.

Parameters

- **sensor_type** –sensor type declared in sensor_type_t.
- **event_id** –sensor event declared in sensor_event_id_t and sensor_data_event_id_t
- **context** –the instance object of the registration to be unregistered

Returns esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

Structures

struct **sensor_info_t**

sensor information, written by the driver developer.

Public Members

const char ***name**

sensor name

sensor_type_t **sensor_type**

sensor type

struct **sensor_config_t**

sensor initialization parameter

Public Members

bus_handle_t **bus**

i2c/spi bus handle

uint8_t **addr**

set addr of sensor

sensor_type_t **type**

sensor type

sensor_mode_t **mode**

set acquire mode detiled in sensor_mode_t

sensor_range_t **range**

set measuring range

uint32_t **min_delay**

set minimum acquisition interval

int **intr_pin**

set interrupt pin

int **intr_type**

set interrupt type

struct **sensor_hub_detect_fn_t**

Detection function interface for sensors.

Public Members

void *(***fn**)(*sensor_info_t* *sensor_info)

Function pointer for sensor detection.

This function detects a sensor based on the provided sensor information.

Param sensor_info Pointer to the sensor information structure
Return void* Pointer to the detected sensor instance, or NULL if detection failed

Macros

SENSOR_HUB_DETECT_FN(type_id, name_id, impl)

Defines a sensor detection function to be executed automatically in the application layer.

This macro defines a function and its corresponding metadata to facilitate automatic sensor detection and initialization. The function must return the implementation (`impl`) on success; otherwise, an error is logged, and the automatic detection process in the application layer will be aborted.

To prevent the linker from optimizing out the sensor driver, the driver must include at least one undefined symbol that is explicitly referenced during the linking phase. This ensures that the linker retains the driver, even if no other files directly depend on its symbols.

For example, in the sensor driver's `CMakeLists.txt`, include the following to force the linker to keep the required symbol:

target_link_libraries(${COMPONENT_LIB} INTERFACE "-u <symbol_name>")

Replace `<symbol_name>` with an appropriate symbol, such as `sht3x_init`, defined in the driver.

Parameters

- **type_id** –The sensor type identifier.
- **name_id** –The unique name identifier for the sensor.
- **impl** –The implementation returned on successful detection.

Type Definitions

typedef void ***sensor_handle_t**

sensor handle

typedef void ***sensor_event_handler_instance_t**

sensor event handler handle

typedef void (***sensor_event_handler_t**)(void *event_handler_arg, sensor_event_base_t event_base, int32_t event_id, void *event_data)

function called when an event is posted to the queue

12.2 Humidity and Temperature Sensor

The humidity and temperature sensor can be used as a temperature sensor, a humidity sensor or a sensor with both functions. It is mainly used for environmental temperature and humidity detections in smart home, smart farm and smart factory applications.

12.2.1 Adapted Products

Name	Function	Bus	Vendor	Datasheet	HAL
HDC2010	Temperature, Humidity	I2C	TI	HDC2010 Datasheet	
HTS221	Temperature, Humidity	I2C	ST	HTS221 Datasheet	√
SHT3X	Temperature, Humidity	I2C	Sensirion	SHT3X Datasheet	√
MVH3004D	Temperature, Humidity	I2C	–		

12.2.2 API Reference

The following APIs have implemented hardware abstraction on the humidity and temperature sensor. Users can call the code from this layer directly to write a sensor application, or use the sensor interface in *sensor_hub* for easier development.

Header File

- components/sensors/sensor_hub/include/hal/humiture_hal.h

12.3 Inertial Measurement Unit (IMU)

The Inertial Measurement Unit (IMU) can be used as a gyroscope sensor, an acceleration sensor, a sensor with multiple functions or etc. It is mainly used to measure the acceleration and angular velocity of an object, and then calculate the motion attitude of the object.

12.3.1 Adapted Products

Name	Function	Bus	Vendor	Datasheet	HAL
LIS2DH12	3-axis acceler	I2C	ST	LIS2DH12 Datasheet	√
MPU6050	3-axis acceler + 3-axis gyro	I2C	InvenSense	MPU6050 Datasheet	√

12.3.2 API Reference

The following APIs have implemented hardware abstraction on the IMU. Users can call the code from this layer directly to write a sensor application, or use the sensor interface in *sensor_hub* for easier development.

Header File

- components/sensors/sensor_hub/include/hal/imu_hal.h

12.4 Ambient Light Sensor

The ambient light sensor can be used as a light intensity sensor, a color sensor, a UV sensor or a sensor with multiple functions.

12.4.1 Adapted Products

Name	Function	Bus	Vendor	Datasheet	HAL
BH1750	Light	I2C	rohm	Datasheet	√
VEML6040	Light RGBW	I2C	Vishay	Datasheet	√
VEML6075	Light UVA UVB	I2C	Vishay	Datasheet	√

12.4.2 API Reference

The following APIs have implemented hardware abstraction on the ambient light sensor. Users can call the code from this layer directly to write a sensor application, or use the sensor interface in *sensor_hub* for easier development.

Header File

- components/sensors/sensor_hub/include/hal/light_sensor_hal.h

12.5 Pressure Sensor

The pressure sensor can be used to detect the absolute pressure of gases, calculate altitude and etc. It is mainly used in environmental monitoring, altitude measurement and space positioning equipment.

12.5.1 Adapted Products

Name	Function	Bus	Vendor	Datasheet	HAL
BME280	Pressure	I2C/SPI	BOSCH	Datasheet	

12.6 Gesture Sensor

Gesture sensors are generally sensors that convert measurements of reflected infrared light into relevant physical motions and can be used to achieve non-contact interaction between human and machines, etc.

12.6.1 Adapted Products

Name	Function	Bus	Vendor	Datasheet	HAL
APDS9960	Light, RGB and Gesture Sensor	I2C	Avago	Datasheet	

12.7 NTC_Driver

NTC stands for Negative Temperature Coefficient, which is a type of thermistor with a negative temperature coefficient. Its main characteristic is that the resistance decreases with increasing temperature within the operating temperature range. By utilizing its basic resistance temperature characteristics, voltage and current characteristics, the NTC series thermistors have been widely used in household appliances,To achieve functions such as automatic gain adjustment, overload protection, temperature control, temperature compensation, voltage stabilization, and amplitude stabilization.

The common packaging/forms of NTC are as follows:

1. Probe type NTC thermistor: suitable for various application scenarios, suitable for detecting extremely high or low temperatures
2. SMT type NTC thermistor: one-time mounted on PCB, suitable for temperature compensation circuits, LED lighting temperature monitoring, battery pack temperature monitoring

NTC key parameters:

1. Resistance specifications (resistance values of 1K, 5K, 10K, 50K, 100K, etc. at 25 °C)
2. Resistance accuracy (0.5%, 1%, 2%, 3%, 5%)
3. Temperature range for use
4. B value (material constant, the higher the B value, the higher the rate of resistance change)
5. Probe shape, probe material, wire material, and wire length

NTC Driver Applicable Circuit: Single ADC channel measurement: When the NTC is on top, as the temperature increases, the resistance value of the NTC decreases, causing the voltage at the NTC terminal in the divider circuit to drop, and the voltage at the fixed resistor side rises, and the final output voltage changes accordingly. The opposite is true when the NTC is below, where a drop in the resistance value of the NTC causes the voltage at the NTC side of the divider circuit to rise, and a drop in the voltage at the fixed resistor side, resulting in a corresponding change in the output voltage. As a result, the NTC will get opposite results above and below, which requires special care when designing and using NTC measurement circuits to ensure that the temperature characteristics of the NTC are properly understood and handled.

When using the following circuits: Vcc ——> Rt ——> Rref ——> GND The corresponding circuit mode needs to be selected through the `circuit_mode` field in `ntc_config_t`. When using this circuit mode, `CIRCUIT_MODE_NTC_VCC` should be used.

When using the following circuits: Vcc ——> Rref ——> Rt ——> GND The corresponding circuit mode needs to be selected through the `circuit_mode` field in `ntc_config_t`. When using this circuit mode, `CIRCUIT_MODE_NTC_GND` should be used.

Regarding the value of voltage divider resistor Rref: The resistance value of Rref is generally taken as the resistance value of Rt at 25 degrees Celsius

NTC digital temperature conversion parameter, formula is: Rt = R * EXP (B * (1/T1-1/T2)) -Rt is the resistance of a thermistor at T1 temperature -R is the nominal resistance of a thermistor at room temperature at T2 -B value is an important parameter of thermistors -EXP is the nth power of e -T1 and T2 refer to K degrees, which is the Kelvin temperature. K degrees = 273.15 (absolute temperature) + Celsius

12.7.1 Demonstration

Create an NTC thermistor drive

```
// Create a ntc driver and register call-back
ntc_config_t ntc_config = {
    .b_value = 3950,
    .r25_ohm = 10000,
    .fixed_ohm = 10000,
    .vdd_mv = 3300,
    .circuit_mode = CIRCUIT_MODE_NTC_GND,
    .atten = ADC_ATTEN_DB_11,
    .channel = ADC_CHANNEL_3,
    .unit = ADC_UNIT_1
};

ntc_device_handle_t ntc = NULL;
adc_oneshot_unit_handle_t adc_handle = NULL;
ESP_ERROR_CHECK(ntc_dev_create(&ntc_config, &ntc, &adc_handle));
ESP_ERROR_CHECK(ntc_dev_get_adc_handle(ntc, &adc_handle));

//get ntc temperature
float temp = 0.0;
if (ntc_dev_get_temperature(ntc, &temp) == ESP_OK) {
    ESP_LOGI(TAG, "NTC temperature = %.2f ℃", temp);
}

//delete handle
TEST_ASSERT_EQUAL(ESP_OK, ntc_dev_delete(ntc));
```

12.7.2 API reference

The following API implements hardware abstraction for thermistor sensors. Users can directly call this layer of code to write sensor applications, or use the sensor interface in doc: ‘sensor_hub<sensor_hub>’ to achieve simpler calls.

Header File

- components/sensors/ntc_driver/include/ntc_driver.h

Functions

esp_err_t **ntc_dev_create** (*ntc_config_t* *config, *ntc_device_handle_t* *ntc_handle, adc_oneshot_unit_handle_t *adc_handle)

Initialize the NTC and ADC channel config.

Parameters
- **config** –
- **ntc_handle** –A ntc driver handle
- **adc_handle** –A adc handle

Returns
- ESP_OK Success
- ESP_FAIL Failure

esp_err_t **ntc_dev_get_adc_handle** (*ntc_device_handle_t* ntc_handle, adc_oneshot_unit_handle_t *adc_handle)

Get the adc handle.

Parameters
- **ntc_handle** –A ntc driver handle
- **adc_handle** –A adc handle

Returns
- ESP_OK Success
- ESP_FAIL Failure

esp_err_t **ntc_dev_delete** (*ntc_device_handle_t* ntc_handle)

Delete ntc driver device and ntc detect device.

Parameters **ntc_handle** –A ntc driver handle

Returns
- ESP_OK Success
- ESP_FAIL Failure

esp_err_t **ntc_dev_get_temperature** (*ntc_device_handle_t* ntc_handle, float *temperature)

Get the ntc temperature.

Parameters
- **ntc_handle** –A ntc driver handle
- **temperature** –NTC temperature

Returns
- ESP_OK Success
- ESP_FAIL Failure

Structures

struct **ntc_config_t**

NTC config data type.

Public Members

ntc_circuit_mode_t **circuit_mode**
ntc circuit mode

adc_unit_t **unit**
adc unit

adc_atten_t **atten**
adc atten

adc_channel_t **channel**
adc channel

uint32_t **b_value**
beta value of NTC (K)

uint32_t **r25_ohm**
25°C resistor value of NTC (K)

uint32_t **fixed_ohm**
fixed resistor value (Ω)

uint32_t **vdd_mv**
vdd voltage (mv)

Type Definitions

typedef void ***ntc_device_handle_t**

Enumerations

enum **ntc_circuit_mode_t**
Supported circuit mode.

Values:

enumerator **CIRCUIT_MODE_NTC_VCC**

enumerator **CIRCUIT_MODE_NTC_GND**

12.8 Power Monitor

Power Monitor IC (Integrated Circuit) is an integrated circuit used for monitoring and managing power supply. It can monitor various parameters of the power supply in real time, such as voltage, current, and power, and provide this information for system use. Power Monitor ICs are crucial in various applications, including computers, power management systems, consumer electronics, industrial control systems, and communication devices.

12.8.1 Adapted Products

Name	Function	Bus	Vendor	Datasheet	HAL
INA236	16-Bit Current, Voltage, and Power Monitor IC	I2C	TI	INA236 Datasheet	x

12.8.2 API Reference

The following API implements hardware abstraction for power monitor sensors. Users can directly call this layer of code to write sensor applications.

Header File

- components/sensors/power_monitor/ina236/include/ina236.h

Functions

esp_err_t **ina236_create** (*ina236_handle_t* *handle, *ina236_config_t* *config)

Create and init object and return a handle.

Parameters
- **handle** –Pointer to handle
- **config** –Pointer to configuration

Returns
- ESP_OK Success
- Others Fail

esp_err_t **ina236_delete** (*ina236_handle_t* handle)

Deinit object and free memory.

Parameters **handle** –ina236 handle Handle

Returns
- ESP_OK Success
- Others Fail

esp_err_t **ina236_get_voltage** (*ina236_handle_t* handle, float *volt)

Get the Voltage on the bus.

Parameters
- **handle** –object handle of ina236
- **volt** –Voltage value in volts

Returns
- ESP_OK Success
- Others Fail

esp_err_t **ina236_get_current** (*ina236_handle_t* handle, float *curr)

Get the Current on the bus.

Parameters
- **handle** –object handle of ina236
- **curr** –Current value in A

Returns
- ESP_OK Success
- Others Fail

esp_err_t **ina236_clear_mask** (*ina236_handle_t* handle)

Clear the mask of the alert pin.

Parameters **handle** –object handle of ina236

Returns

- ESP_OK Success
- Others Fail

Structures

struct **ina236_config_t**

ina236 configuration structure

Public Members

i2c_bus_handle_t **bus**

I2C bus object

bool **alert_en**

Enable alert callback

uint8_t **dev_addr**

I2C device address

uint8_t **alert_pin**

Alert pin number

int236_alert_cb_t **alert_cb**

Alert callback function

Macros

INA236_I2C_ADDRESS_DEFAULT

Type Definitions

typedef struct ina236_t ***ina236_handle_t**

INA236 handle.

typedef void (***int236_alert_cb_t**)(void *arg)

ina236 alert callback function

12.9 Power Measure

Power Measure IC (Integrated Circuit) is an integrated circuit used for monitoring and managing power supply. It can monitor various parameters of the power supply in real time, such as voltage, current, and power, and provide this information for system use. Power Measure ICs are crucial in various applications, including computers, power management systems, consumer electronics, industrial control systems, and communication devices.

This example demonstrates how to use the BL0937, INA236, and BL0942 power measurement chips to detect electrical parameters such as voltage, current, active power, and energy consumption (BL0937 and BL0942 only).

12.9.1 Features

- Measures **voltage**, **current**, **active power**, **power factor**, **energy**, and **frequency**.
- Supports **BL0937**, **INA236**, and **BL0942** power measurement chips.
- Configurable chip selection via *BL0937*, *INA236*, and *BL0942* macros.
- Supports overcurrent, overvoltage, and undervoltage protection.
- Energy detection is enabled for accurate readings (BL0937 and BL0942 only).
- Real-time measurements for INA236, cached measurements for BL0937 and BL0942.
- The BL0937 chip employs a hardware pulse counting method with a calibration factor, converting the pulse count into actual electrical parameters via this calibration factor.

12.9.2 BL0937 Calibration Process and Method

The following process can be used to calibrate the voltage (KU), current (KI), and power (KP) factors of the BL0937, improving measurement accuracy:

- Preparation: A traceable multimeter/power meter, a stable AC source/adjustable AC source, and a known resistive load (such as a resistive load or 12W incandescent lamp).
- Terms: - KU: Voltage calibration factor (to make the measured voltage close to the true voltage). - KI: Current calibration factor (to make the measured current close to the true current). - KP: Power calibration factor (to make the measured active power close to the true power).

Steps: 1. Fix the hardware parameters (shunt resistor, voltage divider resistor) and burn the example project, open the serial port log. 2. Voltage calibration (KU): At no load or light load, read *voltage_meas*, use the multimeter to get *voltage_true*, update:

```
KU_new = KU_old * (voltage_true / voltage_meas)
```

3. Current calibration (KI): Connect the known resistive load, read *current_meas*, use the multimeter/power meter to calculate/read *current_true*, update:

```
KI_new = KI_old * (current_true / current_meas)
```

4. Power calibration (KP): At the same load, read *power_meas*, use the multimeter/power meter' s *power_true* as the reference, update:

```
KP_new = KP_old * (power_true / power_meas)
```

5. Repeat steps 2-4 several times until the relative error of voltage/current/power meets the requirements (e.g., ≤1%~2%).

In the code, configure the calibrated KU/KI/KP factors into *power_measure_bl0937_config_t* (the *ki/ku/kp* fields in the example).

More background and overall information can be found in: README.md.

12.9.3 How It Works

The *app_main()* function in *main/power_measure_example.c* demonstrates the handle-based factory API:

1. **(Optional) Initialize I2C bus (INA236 only)**:
   - Use *i2c_bus_create()* with *I2C_MASTER_SCL_IO*/*I2C_MASTER_SDA_IO* to create the I2C bus.
2. **Create a power measurement device**:
   - Prepare common config *power_measure_config_t* (overcurrent/overvoltage/undervoltage thresholds, enable energy detection).
   - For BL0937, prepare *power_measure_bl0937_config_t* (CF/CF1/SEL pins, divider/sampling resistors, KI/KU/KP factors).
   - For INA236, prepare *power_measure_ina236_config_t* (I2C bus, address, alert config).
   - For BL0942, prepare *power_measure_bl0942_config_t* (UART/SPI config, shunt resistor, voltage divider ratio, device address).

- Call *power_measure_new_bl0937_device()*, *power_measure_new_ina236_device()*, or *power_measure_new_bl0942_device()* to get a *power_measure_handle_t*.

3. **Fetch measurements periodically**:
   - Use *power_measure_get_voltage/current/active_power/power_factor()* as needed;
   - *power_measure_get_energy()* returns valid data only on chips that support energy (e.g., BL0937 and BL0942);
   - *power_measure_get_line_freq()* returns valid data only on BL0942 chip;
   - The example logs values every 1 second.
4. **Error handling**:
   - If any API returns a non-*ESP_OK* value, log the error with *ESP_LOGE*.
5. **Resource cleanup**:
   - Release the device with *power_measure_delete(handle)*;
   - For INA236, also release the I2C bus with *i2c_bus_delete(&i2c_bus)*.

12.9.4 Code Snippets

BL0937 (GPIO)

```
#include "power_measure.h"
#include "power_measure_bl0937.h"

power_measure_config_t common = {
    .overcurrent = 2000,
    .overvoltage = 250,
    .undervoltage = 180,
    .enable_energy_detection = true,
};

power_measure_bl0937_config_t bl = {
    .sel_gpio = GPIO_NUM_4,
    .cf1_gpio = GPIO_NUM_7,
    .cf_gpio = GPIO_NUM_3,
    .pin_mode = 0,
    .sampling_resistor = 0.001f,
    .divider_resistor = 1981.0f,
    .ki = 2.75f, .ku = 0.65f, .kp = 15.0f,
};

power_measure_handle_t h;
ESP_ERROR_CHECK(power_measure_new_bl0937_device(&common, &bl, &h));

float u,i,p,e;
power_measure_get_voltage(h, &u);
power_measure_get_current(h, &i);
power_measure_get_active_power(h, &p);
power_measure_get_energy(h, &e);

power_measure_delete(h);
```

INA236 (I2C)

```
#include "power_measure.h"
#include "power_measure_ina236.h"
#include "i2c_bus.h"

i2c_config_t conf = {
    .mode = I2C_MODE_MASTER,
```

(continues on next page)

(continued from previous page)

```
    .sda_io_num = I2C_MASTER_SDA_IO, // GPIO_NUM_20
    .sda_pullup_en = GPIO_PULLUP_ENABLE,
    .scl_io_num = I2C_MASTER_SCL_IO, // GPIO_NUM_13
    .scl_pullup_en = GPIO_PULLUP_ENABLE,
    .master.clk_speed = 100000,
};
i2c_bus_handle_t bus = i2c_bus_create(I2C_NUM_0, &conf);

power_measure_config_t common = {
    .overcurrent = 15,
    .overvoltage = 260,
    .undervoltage = 180,
    .enable_energy_detection = false,
};

power_measure_ina236_config_t ina = {
    .i2c_bus = bus,
    .i2c_addr = 0x41,
    .alert_en = false,
    .alert_pin = -1,
    .alert_cb = NULL,
};

power_measure_handle_t h;
ESP_ERROR_CHECK(power_measure_new_ina236_device(&common, &ina, &h));

float u,i,p;
power_measure_get_voltage(h, &u);
power_measure_get_current(h, &i);
power_measure_get_active_power(h, &p);

power_measure_delete(h);
i2c_bus_delete(&bus);
```

BL0942 (UART/SPI)

```
#include "power_measure.h"
#include "power_measure_bl0942.h"

power_measure_config_t common = {
    .overcurrent = 15,
    .overvoltage = 260,
    .undervoltage = 180,
    .enable_energy_detection = true,
};

power_measure_bl0942_config_t bl0942 = {
    .addr = 0,
    .shunt_resistor = 0.001f,
    .divider_ratio = 3760.0f,
    .use_spi = false,
    .uart = {
        .uart_num = UART_NUM_1,
        .tx_io = GPIO_NUM_6,
        .rx_io = GPIO_NUM_7,
        .sel_io = -1,
        .baud = 4800,
    },
    .spi = {
```

(continues on next page)

(continued from previous page)

```
        .spi_host = SPI2_HOST,
        .mosi_io = -1,
        .miso_io = -1,
        .sclk_io = -1,
        .cs_io = -1,
    },
};

// power_measure_bl0942_config_t bl0942 = {
//     .addr = 0,
//     .shunt_resistor = 0.001f,
//     .divider_ratio = 3760.0f,
//     .use_spi = true,
//     .uart = {
//         .uart_num = UART_NUM_1,
//         .tx_io = -1,
//         .rx_io = -1,
//         .sel_io = -1,
//         .baud = 4800,
//     },
//     .spi = {
//         .spi_host = SPI2_HOST,
//         .mosi_io = GPIO_NUM_11,
//         .miso_io = GPIO_NUM_12,
//         .sclk_io = GPIO_NUM_10,
//         .cs_io = GPIO_NUM_13,
//     },
// };

power_measure_handle_t h;
ESP_ERROR_CHECK(power_measure_new_bl0942_device(&common, &bl0942, &h));

float u, i, p, pf, e, f;
power_measure_get_voltage(h, &u);
power_measure_get_current(h, &i);
power_measure_get_active_power(h, &p);
power_measure_get_power_factor(h, &pf);
power_measure_get_energy(h, &e);
power_measure_get_line_freq(h, &f);

power_measure_delete(h);
```

12.9.5 Troubleshooting

BL0937 Issues:

1. **Failed Initialization** : If the initialization fails, ensure that all GPIO pins are correctly defined and connected to the **BL0937** chip.
2. **Measurement Failures** : If the measurements fail (e.g., voltage, current), verify that the **BL0937** chip is properly powered and communicating with your ESP32 series chips.

INA236 Issues:

1. **I2C Bus Initialization Failed** : Check that the I2C pins (SDA/SCL) are correctly connected and not conflicting with other peripherals.
2. **INA236 Not Detected** : Verify the I2C address (default 0x41) and ensure the chip is properly powered.
3. **Measurement Failures** : Check I2C communication and ensure the INA236 chip is functioning correctly.

4. **No Real-time Data** : INA236 provides real-time measurements, so if you see cached values, check the power_measure component implementation.

BL0942 Issues:

1. **UART Initialization Failed** : Check that UART pins (TX/RX) are correctly connected and verify baud rate settings (default 4800).
2. **SPI Initialization Failed** : Check that SPI pins (MOSI/MISO/SCLK/CS) are correctly connected and not conflicting with other peripherals.
3. **BL0942 Not Detected** : Verify device address (default 0) and ensure the chip is properly powered.
4. **Communication Failures** : Check UART/SPI communication parameters and confirm interface mode configuration.
5. **Abnormal Measurements** : Check shunt resistor and voltage divider ratio configuration to match actual hardware.
6. **Watchdog Reset** : If watchdog reset occurs, check hardware connections and communication configuration.

12.9.6 Adapted Products

Table 1: Power Measurement Chips

Name	Function	Vendor	Datasheet	HAL
BL0937	detect electrical parameters such as voltage, current, active power, and energy consumption	BELLING	BL0937 Datasheet	√
INA236	precision power monitor with I2C interface for voltage, current, and power measurement	TI	INA236 Datasheet	√
BL0942	Single-phase power measurement IC with UART/SPI interface for voltage, current, power, power factor, energy, and frequency measurement	BELLING	BL0942 Datasheet	√

12.9.7 API Reference

The following API implements hardware abstraction for power measure. Users can directly call this layer of code to write sensor applications.

Header File

- components/sensors/power_measure/include/power_measure.h

Functions

esp_err_t **`power_measure_delete`** (*power_measure_handle_t* handle)

Delete a power measurement device.

Parameters **`handle`** –Power measure device handle

Returns

- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid handle

esp_err_t **`power_measure_get_voltage`** (*power_measure_handle_t* handle, float *voltage)

Get current voltage (V)

Parameters

- **`handle`** –Power measure device handle
- **`voltage`** –Pointer to store the voltage value

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_INVALID_STATE: Device not initialized

esp_err_t **power_measure_get_current** (*power_measure_handle_t* handle, float *current)

Get current current (A)

Parameters
- **handle** –Power measure device handle
- **current** –Pointer to store the current value

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_INVALID_STATE: Device not initialized

esp_err_t **power_measure_get_active_power** (*power_measure_handle_t* handle, float *power)

Get current active power (W)

Parameters
- **handle** –Power measure device handle
- **power** –Pointer to store the power value

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_INVALID_STATE: Device not initialized

esp_err_t **power_measure_get_power_factor** (*power_measure_handle_t* handle, float *power_factor)

Get power factor.

Parameters
- **handle** –Power measure device handle
- **power_factor** –Pointer to store the power factor value

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_INVALID_STATE: Device not initialized
- ESP_ERR_NOT_SUPPORTED: Chip doesn’t support this feature

esp_err_t **power_measure_get_energy** (*power_measure_handle_t* handle, float *energy)

Get energy consumption (kWh)

Parameters
- **handle** –Power measure device handle
- **energy** –Pointer to store the energy value

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_INVALID_STATE: Device not initialized
- ESP_ERR_NOT_SUPPORTED: Chip doesn’t support this feature

esp_err_t **power_measure_reset_energy_calculation** (*power_measure_handle_t* handle)

Reset energy calculation.

Parameters **handle** –Power measure device handle

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid handle
- ESP_ERR_INVALID_STATE: Device not initialized
- ESP_ERR_NOT_SUPPORTED: Chip doesn’t support this feature

esp_err_t **power_measure_get_apparent_power** (*power_measure_handle_t* handle, float *apparent_power)

Get apparent power measurement.

Parameters
- **handle** –Power measure device handle
- **apparent_power** –Pointer to store the apparent power value in volt-amperes (VA)

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_INVALID_STATE: Device not initialized
- ESP_ERR_NOT_SUPPORTED: Chip doesn’t support this feature

esp_err_t **power_measure_calibrate_voltage** (*power_measure_handle_t* handle, float expected_voltage)

Dynamic runtime calibration for voltage (BL0937 only)

Parameters
- **handle** –Power measure device handle
- **expected_voltage** –The known reference voltage value in volts

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_INVALID_STATE: Device not initialized
- ESP_ERR_NOT_SUPPORTED: Chip doesn’t support this feature

esp_err_t **power_measure_calibrate_current** (*power_measure_handle_t* handle, float expected_current)

Dynamic runtime calibration for current (BL0937 only)

Parameters
- **handle** –Power measure device handle
- **expected_current** –The known reference current value in amperes

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_INVALID_STATE: Device not initialized
- ESP_ERR_NOT_SUPPORTED: Chip doesn’t support this feature

esp_err_t **power_measure_calibrate_power** (*power_measure_handle_t* handle, float expected_power)

Dynamic runtime calibration for power (BL0937 only)

Parameters
- **handle** –Power measure device handle
- **expected_power** –The known reference power value in watts

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_INVALID_STATE: Device not initialized
- ESP_ERR_NOT_SUPPORTED: Chip doesn’t support this feature

esp_err_t **power_measure_get_voltage_multiplier** (*power_measure_handle_t* handle, float *multiplier)

Get voltage multiplier (BL0937 only, advanced users)

Parameters
- **handle** –Power measure device handle
- **multiplier** –Pointer to store the voltage multiplier value

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid arguments

- ESP_ERR_INVALID_STATE: Device not initialized
- ESP_ERR_NOT_SUPPORTED: Chip doesn't support this feature

esp_err_t **power_measure_get_current_multiplier** (*power_measure_handle_t* handle, float *multiplier)

Get current multiplier (BL0937 only, advanced users)

Parameters
- **handle** –Power measure device handle
- **multiplier** –Pointer to store the current multiplier value

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_INVALID_STATE: Device not initialized
- ESP_ERR_NOT_SUPPORTED: Chip doesn't support this feature

esp_err_t **power_measure_get_power_multiplier** (*power_measure_handle_t* handle, float *multiplier)

Get power multiplier (BL0937 only, advanced users)

Parameters
- **handle** –Power measure device handle
- **multiplier** –Pointer to store the power multiplier value

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_INVALID_STATE: Device not initialized
- ESP_ERR_NOT_SUPPORTED: Chip doesn't support this feature **POWER_MEASURE**

Structures

struct **power_measure_config_t**

Common power measure configuration.

Public Members

uint16_t **overcurrent**

Overcurrent threshold

uint16_t **overvoltage**

Overvoltage threshold

uint16_t **undervoltage**

Undervoltage threshold

bool **enable_energy_detection**

Flag to enable energy detection

Type Definitions

typedef struct power_measure_dev_t ***power_measure_handle_t**

Power measure device handle.

12.10 Battery Fuel Gauge

A battery fuel gauge is a type of sensor used to monitor and measure battery charge. Its basic functions include monitoring voltage, charge/discharge current, and battery temperature, as well as estimating the battery' s SOC and full charge capacity (FCC).

12.10.1 Adapted Products

Name	Function	Bus	Vendor	Datasheet
MAX17048/MAX17049	Provides accurate battery voltage, charge status information, charge/discharge rates, and configurable alarm indicators.	I2C	Analog Devices	Datasheet
BQ27220	A single-cell gauge that uses the Compensated End-ofDischarge Voltage (CEDV) algorithm for fuel gauging	I2C	Texas Instruments	Datasheet

12.11 ADC Sensor

ADC sensor is a device that converts analog voltage signals into digital signals, widely used in data acquisition and signal processing systems for precise monitoring and analysis of voltage variations.

12.11.1 Adapted Products

Name	Function	Bus	Vendor	Datasheet	HAL
MCP3201	ADC	SPI	Microchip	Datasheet	

12.12 Gas Sensor

The gas sensor can be used to detect air quality, specific gas concentration, temperature and humidity, pressure, etc. It is mainly used in environmental monitoring, industrial safety, smart home, etc.

12.12.1 Adapted Products

Name	Function	Bus	Vendor	Datasheet	HAL
BME690	Air Quality, Pressure, Humiture	I2C/SPI	BOSCH	Datasheet	

Chapter 13

Touch Sensor

13.1 Touch Proximity Sensor

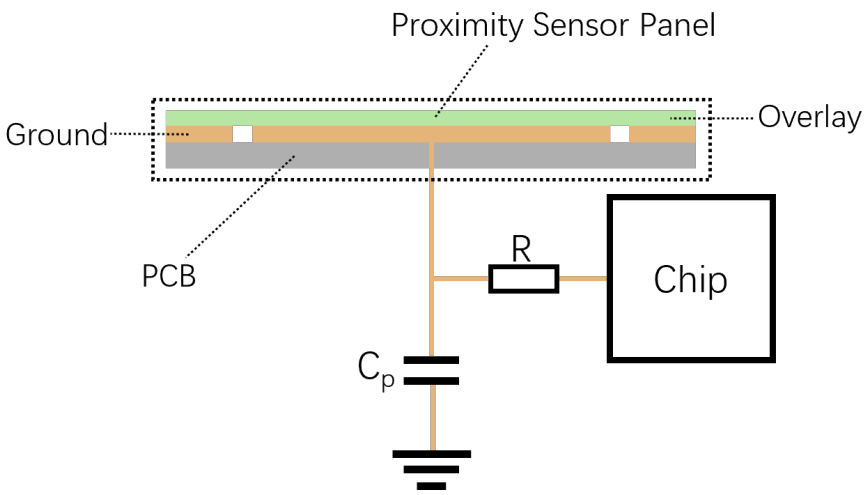
The `touch_proximity_sensor` component is developed based on the built-in touch sensors of ESP32-S3. Using this component makes it easy to implement touch proximity sensing functionality.

Note:

- ESP32/ESP32-S2/ESP32-S3 touch-related components are intended for testing or demo purposes only. Due to the poor anti-interference capability of the touch functionality, it may not pass EMS testing, and therefore, it is not recommended for mass production products.
- This component is currently only applicable to the ESP32-S3 and requires an IDF version greater than or equal to v5.0.

13.1.1 Principle of Operation

The touch proximity sensor is implemented based on the proximity sensing feature of the ESP32-S3 touch sensor. The hardware schematic of the sensor principle is as follows:



When a target object approaches the sensor, its equivalent capacitance changes. The target object can be a human finger, hand, or any conductive object. When the touch sensor is configured in proximity sensing mode, the sensor output is an accumulated value. As the target object approaches the sensor panel, the cumulative value output by the sensor increases. Based on this characteristic, this solution defines the raw data output by the touch sensor (accumulated value) as *raw_value* , and derives two data variables, *baseline* and *smooth_value*, from it. Combined with a reasonable threshold detection algorithm, proximity sensing functionality is achieved.

The specific software implementation involves the following three steps:

1. Determine the validity of new data.
2. Update *smooth_value* and *baseline* based on the update logic of *smooth_value* and *baseline* using *raw_value* as the source data.
3. Determine whether *smooth_value* - *baseline* is greater than 0. If it is greater than 0, then determine whether it is greater than the **trigger threshold** . If it is greater, it is deemed as a valid sensing trigger action; if *smooth_value* - *baseline* is less than 0, first determine whether it is in a triggered state. If it is in a triggered state, then determine whether its absolute value is greater than the **release trigger threshold** . If it is greater, it is deemed as a valid trigger release action.

13.1.2 Hardware Reference for Testing

- Development board
  - Validation testing can be performed using the ESP-S2S3-Touch-DevKit-1 development kit. The mainboard is `MainBoard v1.1`, and the proximity sensing subboard is `Proximity Board V1.0`.

13.1.3 Configuration Reference

Create Proximity Sensing Sensor

Using the `touch_proximity_sensor` component, the proximity sensing sensor can be configured via the *`touch_proxi_config_t`* structure.

```
typedef struct {
    uint32_t channel_num;                          /*!< Number of touch proximity␣
↪sensor channels */
    uint32_t *channel_list;                        /*!< Touch channel list */
    float *channel_threshold;                      /*!< Threshold for touch␣
↪detection for each channel */
    uint32_t debounce_times;                       /*!< Number of consecutive␣
↪readings needed to confirm state change */
    uint32_t *channel_gold_value;                  /*!< Reference values for␣
↪touch channels */
    bool skip_lowlevel_init;                       /*!< Skip low level␣
↪initialization when working with existing touch driver */
} touch_proxi_config_t;
```

The main parameters are:

Parameter	Description
channel_num	Number of touch proximity sensor channels, up to 3
channel_list	Array of touch channel numbers to be used
channel_threshold	Array of threshold values for each channel
debounce_times	Number of consecutive readings needed to confirm state change
channel_gold_value	Optional reference values for touch channels
skip_lowlevel_init	Skip low level init when working with existing touch driver

After configuring the parameters, create the proximity sensor using *`touch_proximity_sensor_create()`*:

```
touch_proxi_config_t config = {
    .channel_num = 1,
    .channel_list = channel_list,
    .channel_threshold = channel_threshold,
    .debounce_times = 2,
};

esp_err_t ret = touch_proximity_sensor_create(&config, &sensor_handle, callback_
↪func, NULL);
```

Event Handling

The proximity sensor requires periodic event handling to update states and trigger callbacks. This can be done in a task:

```
void proximity_task(void *arg)
{
    while (1) {
        touch_proximity_sensor_handle_events(sensor_handle);
        vTaskDelay(pdMS_TO_TICKS(20));
    }
}

// Create task
xTaskCreate(proximity_task, "proximity_task", 2048, NULL, 5, NULL);
```

Delete the Proximity Sensing Sensor

Use `touch_proximity_sensor_delete()` to delete the proximity sensing sensor object and release resources:

```
// Delete the touch proximity sensor
touch_proximity_sensor_delete(sensor_handle);
```

13.1.4 Parameter Adjustment Reference

- The maximum value for *channel_num* is 3.
- The *channel_list* array must be assigned values from the *touch_pad_t* enumeration variable.
- *meas_count* has a default value of 20. Increasing *meas_count* slows down the update rate of new data from the touch sensor.
- *smooth_coef* has a default value of 0.7. It is the coefficient used for data smoothing. The smoothed value *smooth* equals *smooth * (1.0 - smooth_coef) + raw * smooth_coef* . A larger *smooth_coef* gives more weight to *raw* , resulting in poorer smoothing of the waveform, faster response to *raw* , and weaker resistance to interference. Conversely, a smaller *smooth_coef* gives less weight to *raw*, resulting in better smoothing of the waveform, slower response to *raw*, and stronger resistance to interference.
- *baseline_coef* has a default value of 0.05. It is the coefficient for baseline updating. The new baseline value equals *baseline * (1.0 - baseline_coef) + smooth * baseline_coef* . A larger value for *baseline_coef* causes the baseline to follow *smooth* more quickly, resulting in slower response time and stronger resistance to interference.
- *max_p* has a default value of 0.5. When the difference between *raw* and *baseline* exceeds *baseline * max_p* , the *raw* value is considered an outlier and ignored.
- *min_n* has a default value of 0.05. When the difference between *baseline* and *raw* exceeds *baseline * min_n* , the *raw* value is considered an outlier and ignored.
- Increasing *threshold_p* and *threshold_n* values reduces the sensing distance for proximity sensing, but improves resistance to interference.

- *noise_p* has a default value of 0.1, and *noise_n* has a default value of 0.2. Larger values for *noise_p* and *noise_n* make it easier for the baseline to follow *smooth* , resulting in a smaller sensing distance for proximity sensing and better resistance to interference.
- *debounce_p* and *debounce_n* values need to be adjusted based on the value of *meas_count* . Smaller *meas_count* values require larger *debounce_p* and *debounce_n* values to increase resistance to interference.
- *reset_p* has a default value of 0, used for baseline reset positive debounce. Setting to 0 disables this feature.
- *reset_n* has a default value of 50, used for baseline reset negative debounce.

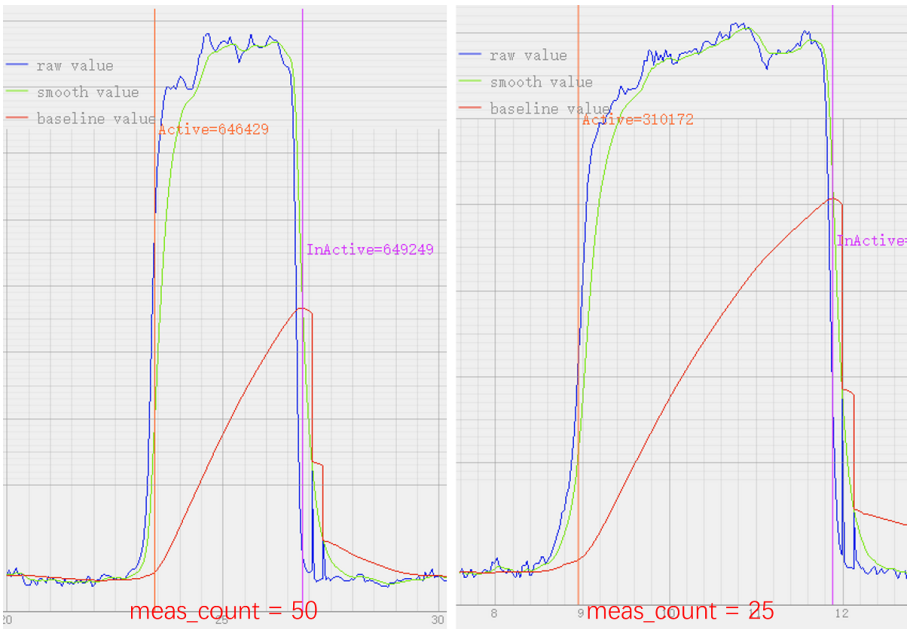
Parameter Adjustment Comparison

The default configuration parameters for the touch proximity sensor are as follows:

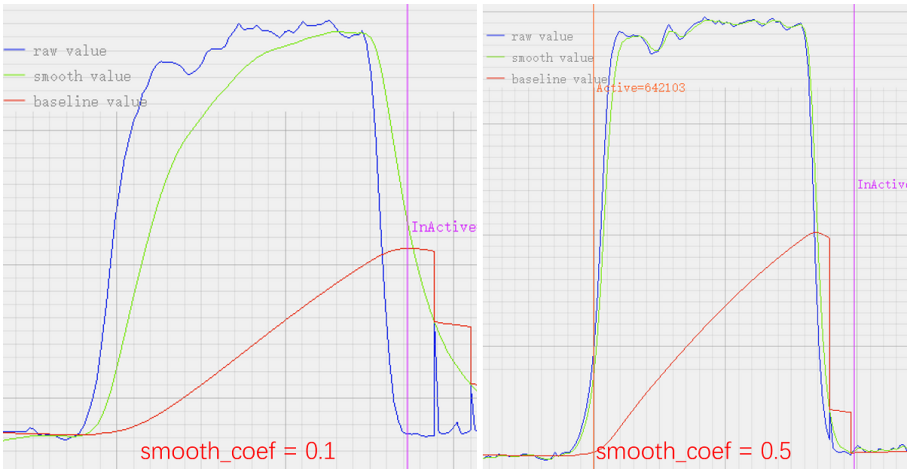
Parameter	Default Value
channel_num	1
channel_list	TOUCH_PAD_NUM8
meas_count	20
smooth_coef	0.7
baseline_coef	0.05
max_p	0.5
min_n	0.05
noise_p	0.1
noise_n	0.2
debounce_p	2
debounce_n	50
reset_p	0
reset_n	50

The following parameter adjustment comparisons will be made based on modifying **only one parameter** from the above parameters:

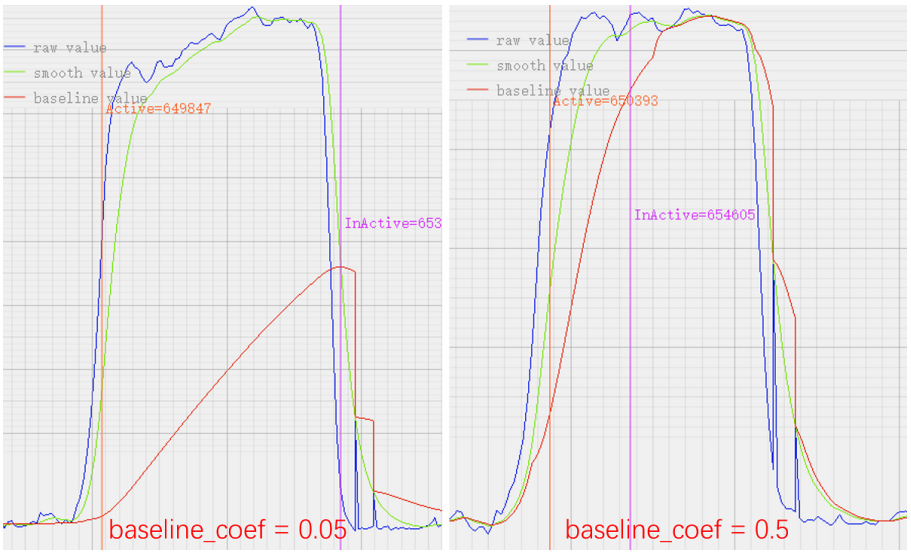
1. Modifying the value of *meas_count* changes the update rate of sensor data, with smaller values resulting in a wider waveform during sensing. The waveform comparison under different *meas_count* is shown below:



2. Modifying the value of *smooth_coef* changes the smoothing effect of the *smooth* waveform. The smaller the *smooth_coef*, the better the smoothing effect, stronger resistance to interference, slower response to *raw* , and vice versa. The waveform comparison under different *smooth_coef* is shown below:



3. Modifying the value of *baseline_coef* changes the updating effect of the *baseline* . Smaller values of *baseline_coef* result in slower response time and longer trigger duration. The waveform comparison under different *baseline_coef* is shown below:

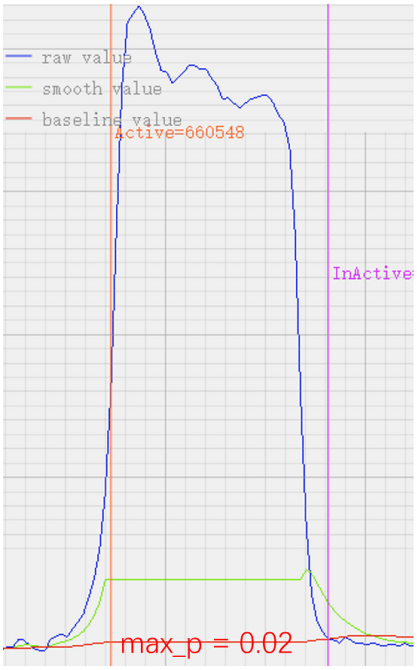


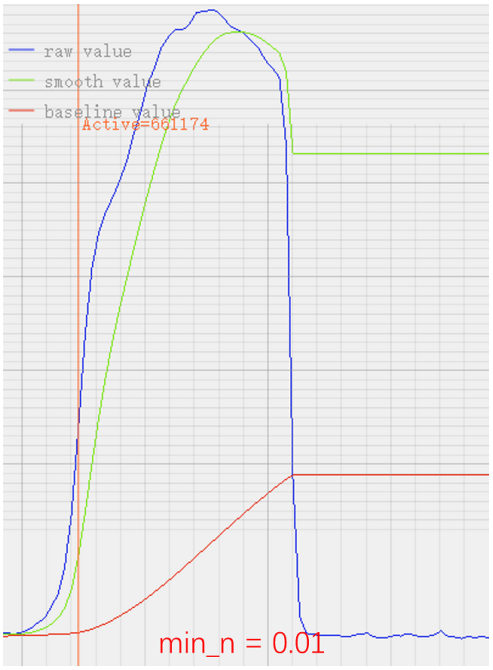
4. Modifying the values of *max_p* and *min_n* will change the update logic of *smooth* and *baseline* . A too small value of *max_p* can cause the *smooth* to be "locked" when the hand approaches the sensing panel, potentially leading to failure to trigger; a too small value of *min_n* can cause both *smooth* and *baseline* to be "locked" when the hand leaves the sensing panel, resulting in failure to release the trigger. The waveform diagram for too small *max_p* and *min_n* values is as follows:

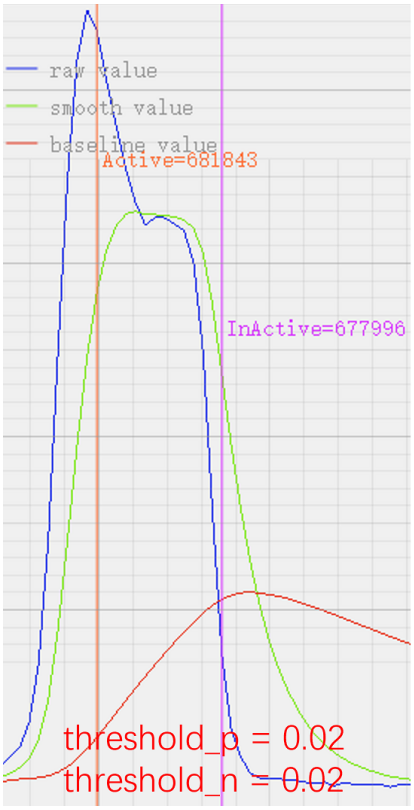
5. Modifying the value of *threshold_p* will change the proximity sensing distance. A smaller value allows for a longer sensing distance but decreases resistance to interference, making false triggers more likely. The waveform comparison under different *threshold_p* values is shown below:

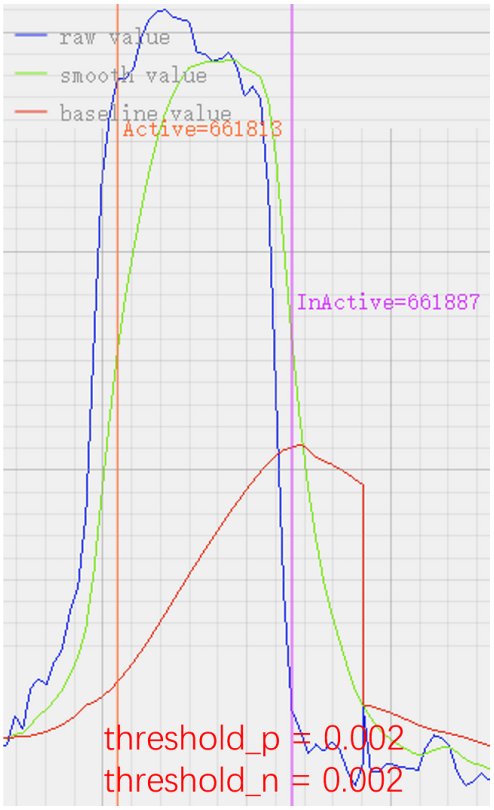
6. Modifying the value of *hysteresis_p* will change the timing of triggering and releasing the trigger, i.e., the hysteresis. A smaller *hysteresis_p* value leads to faster trigger response, and vice versa. The waveform comparison under different *hysteresis_p* values is shown below:

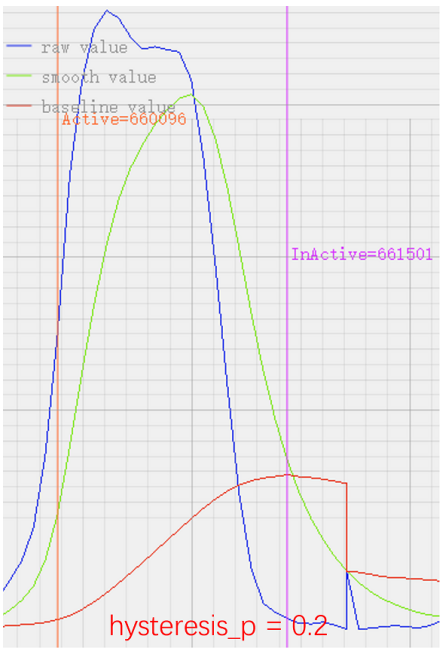
7. Modifying the values of *noise_p* and *noise_n* will change the updating effect of *baseline* . A smaller value of *noise_p* causes *baseline* to follow *smooth* more slowly, resulting in slower trigger response and longer trigger duration, and vice versa. The waveform comparison under different *noise_p* and *noise_n* values is shown below:

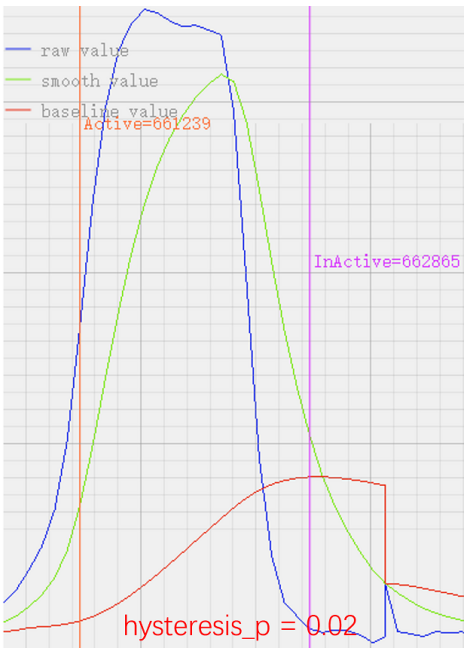

raw value
smooth value
baseline value
Active=660548
InActive
max_p = 0.02

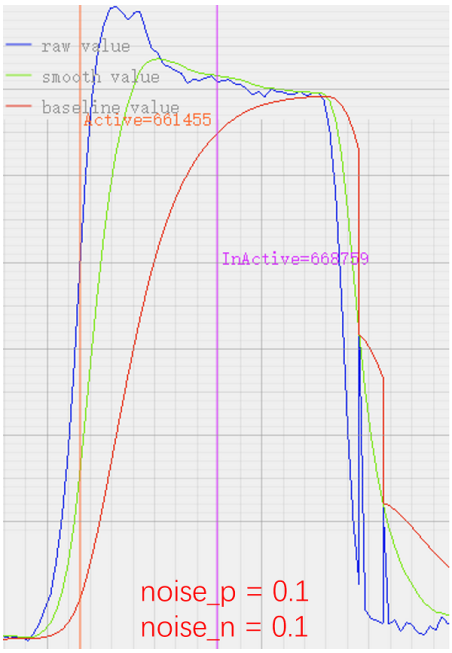

raw value
smooth value
baseline value
Active=661174
min_n = 0.01

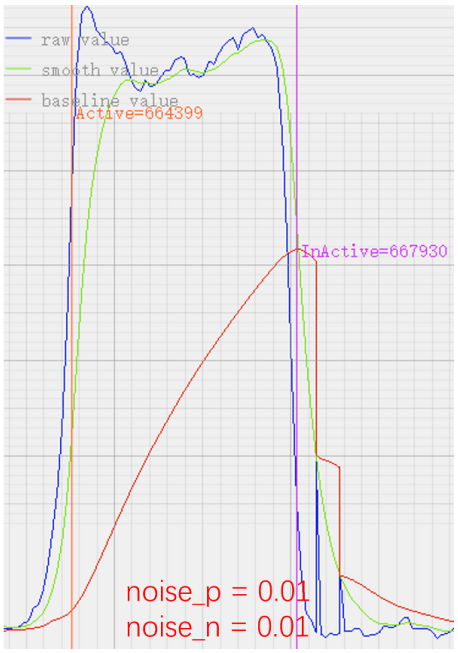

raw value
smooth value
baseline value
Active=681843
InActive=677996
threshold_p = 0.02
threshold_n = 0.02


raw value
smooth value
baseline value
Active=661813
InActive=661887
threshold_p = 0.002
threshold_n = 0.002

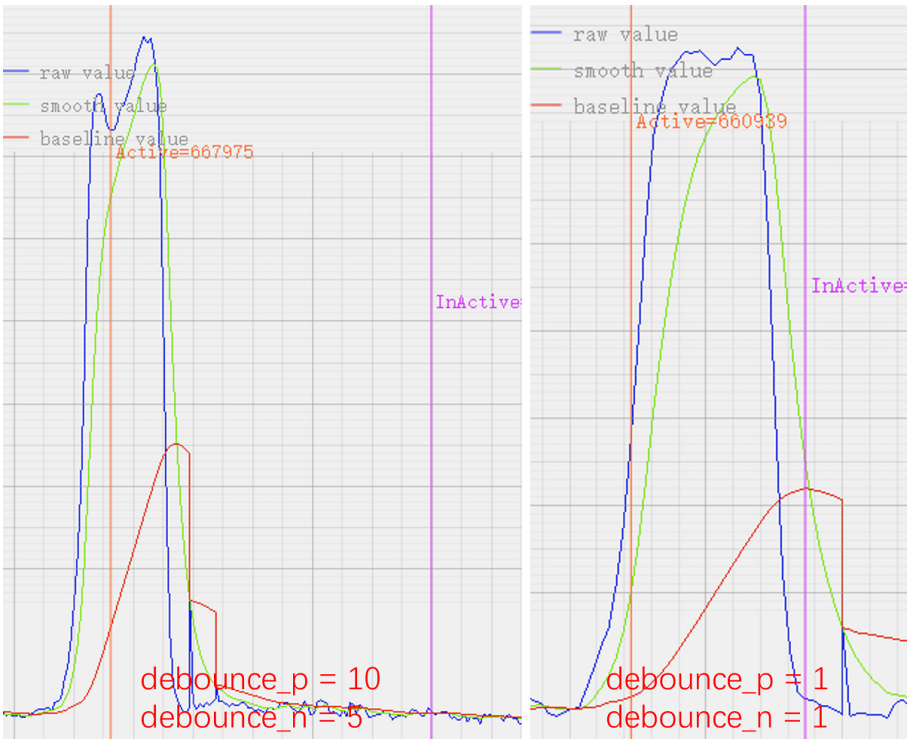

raw value
smooth value
baseline value
Active=660096
InActive=661501
hysteresis_p = 0.2


raw value
smooth value
baseline value
Active=661239
InActive=662865
hysteresis_p = 0.02


raw value
smooth value
baseline value
Active=661455
InActive=668759
noise_p = 0.1
noise_n = 0.1


raw value
smooth value
baseline value
Active=664399
InActive=667930
noise_p = 0.01
noise_n = 0.01

8. Modifying the values of *debounce_p* and *debounce_n* will change the timing of triggering and releasing the trigger and the resistance to interference. A larger *debounce_p* value leads to slower trigger response and stronger resistance to interference, and vice versa; a larger *debounce_n* value leads to slower release of the trigger response and stronger resistance to interference, and vice versa. The values of *debounce_p* and *debounce_n* need to be adjusted in conjunction with *meas_count* ; as *meas_count* decreases, *debounce_p* and *debounce_n* should be appropriately increased. The waveform comparison under different *debounce_p* and *debounce_n* values is shown below:



Note: Achieving the ideal proximity sensing effect requires comprehensive adjustment of multiple parameters, rather than just tweaking one or two parameters.

13.1.5 Examples

- touch/touch_proximity

13.1.6 API Reference

Header File

- components/touch/touch_proximity_sensor/include/touch_proximity_sensor.h

Functions

esp_err_t **touch_proximity_sensor_create** (*touch_proxi_config_t* *config, *touch_proximity_handle_t* *sensor_handle, *proxi_cb_t* cb, void *cb_arg)

Create a touch proximity sensor instance.

Parameters

- **config** –The touch pad channel configuration.
- **sensor_handle** –The handle of the successfully created touch proximity sensor.
- **cb** –Callback function to handle proximity events.
- **cb_arg** –The callback function argument.

Returns

- ESP_OK: Create the touch proximity sensor successfully.
- ESP_ERR_NO_MEM: Failed to create the touch proximity sensor (memory allocation failed).

esp_err_t **touch_proximity_sensor_delete** (*touch_proximity_handle_t* proxi_sensor)

Delete the touch proximity sensor instance.

This function deletes the touch proximity sensor instance associated with the provided sensor handle.

Parameters **proxi_sensor** –Pointer to the handle of the touch proximity sensor instance to be deleted.

Returns

- ESP_OK: Delete the touch proximity sensor instance successfully

esp_err_t **touch_proximity_sensor_get_data** (*touch_proximity_handle_t* handle, uint32_t channel, uint32_t *data)

Get touch proximity sensor data.

Retrieves the smoothed touch sensor reading from the specified channel.

Parameters

- **handle** –**[in]** Touch proximity sensor handle
- **channel** –**[in]** Touch channel number
- **data** –**[out]** Pointer to store the smoothed touch sensor data

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if handle or data is NULL
- ESP_ERR_INVALID_STATE if sensor not initialized
- ESP_ERR_NOT_FOUND if channel not found

esp_err_t **touch_proximity_sensor_get_state** (*touch_proximity_handle_t* handle, uint32_t channel, *proxi_state_t* *state)

Get current state of a proximity channel.

Returns whether the proximity channel is currently considered active (object detected) based on the current sensor readings and configured thresholds.

Parameters

- **handle** –**[in]** Touch proximity sensor handle
- **channel** –**[in]** Touch channel number
- **state** –**[out]** Pointer to store the channel state

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if handle or state is NULL
- ESP_ERR_INVALID_STATE if sensor not initialized
- ESP_ERR_NOT_FOUND if channel not found

esp_err_t **touch_proximity_sensor_handle_events** (*touch_proximity_handle_t* handle)

Handle pending proximity sensor events.

Processes events from the FSM. This function should be called periodically to update proximity states and trigger callbacks.

Parameters **handle** –**[in]** Touch proximity sensor handle

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if handle is NULL
- ESP_ERR_INVALID_STATE if sensor not initialized

Structures

struct **touch_proxi_config_t**

Configuration structure for touch proximity sensor.

Public Members

uint32_t **channel_num**

Number of touch proximity sensor channels

uint32_t ***channel_list**

Touch channel list

float ***channel_threshold**

Threshold for touch detection for each channel

uint32_t **debounce_times**

Number of consecutive readings needed to confirm state change

uint32_t ***channel_gold_value**

Reference values for touch channels

bool **skip_lowlevel_init**

Skip low level initialization when working with existing touch driver

Type Definitions

typedef struct touch_proximity_sensor_t ***touch_proximity_handle_t**

typedef void (***proxi_cb_t**)(uint32_t channel, *proxi_state_t* event, void *cb_arg)

proximity sensor user callback type

Enumerations

enum **proxi_state_t**

Touch Proximity Sensor State.

Values:

enumerator **PROXI_STATE_INACTIVE**

enumerator **PROXI_STATE_ACTIVE**

13.2 Touch Button Sensor

13.2.1 Overview

The `touch_button_sensor` component provides enhanced touch button detection functionality for ESP32 series chips. It uses multiple frequency sampling (for ESP32-P4) and FSM-based processing to provide reliable touch detection even in noisy environments.

Note:

- ESP32/ESP32-S2/ESP32-S3 touch-related components are intended for testing or demo purposes only. Due to the poor anti-interference capability of the touch functionality, it may not pass EMS testing, and therefore, it is not recommended for mass production products.
- This component is currently applicable to ESP32, ESP32-S2, and ESP32-S3, and requires an IDF version greater than or equal to v5.3.

Key Features

- Multi-frequency touch sampling for improved noise immunity
- FSM-based touch detection with configurable thresholds
- Debounce support for reliable state changes
- Support for multiple touch channels
- Callback-based event notification

Dependencies

- touch_sensor_fsm
- touch_sensor_lowlevel

13.2.2 Configuration Parameters

Touch Button Configuration Structure

The touch button can be configured via the `touch_button_config_t` structure:

```
typedef struct {
    uint32_t channel_num;          /*!< Number of touch button sensor channels */
    uint32_t *channel_list;        /*!< Touch channel list */
    float *channel_threshold;       /*!< Threshold for touch detection for each␣
↪channel */
    uint32_t *channel_gold_value;   /*!< (Optional) Reference values for touch␣
↪channels */
    uint32_t debounce_times;        /*!< Number of consecutive readings needed to␣
↪confirm state change */
    bool skip_lowlevel_init;        /*!< Skip low level initialization when␣
↪working with existing touch driver */
} touch_button_config_t;
```

Parameter Descriptions

Parameter	Description	Default Value
channel_num	Number of touch button sensor channels	•
channel_list	Array of touch channel numbers to use	•
channel_threshold	Array of threshold values for each channel	•
channel_gold_value	Reference values for touch channels (optional)	NULL
debounce_times	Consecutive readings for state change	3
skip_lowlevel_init	Skip touch driver initialization if exists	false

13.2.3 API Usage Examples

Create and Initialize

```
touch_button_config_t config = {
    .channel_num = 1,
    .channel_list = (uint32_t[]){8},            // Using touch channel 8
    .channel_threshold = (float[]){0.02},       // 2% change threshold
    .channel_gold_value = NULL,                  // No reference values
    .debounce_times = 3,                        // 3 readings to confirm
    .skip_lowlevel_init = false                 // Initialize touch hardware
};
touch_button_handle_t btn_handle = NULL;
esp_err_t ret = touch_button_sensor_create(&config, &btn_handle, button_state_
↪callback, NULL);
```

State Change Callback

The callback function is called when a touch button state changes:

```
void button_state_callback(touch_button_handle_t handle, uint32_t channel, touch_
↪state_t state, void *arg)
{
    switch (state) {
        case TOUCH_STATE_ACTIVE:
            printf("Button channel %d pressed\n", channel);
            break;
        case TOUCH_STATE_INACTIVE:
            printf("Button channel %d released\n", channel);
            break;
    }
}
```

Event Handling

The touch button sensor component provides an event handling mechanism to process touch events in a non-blocking way. Events should be handled periodically in your application' s main loop or in a dedicated task.

```
// In your main loop or task
while (1) {
    // Process any pending touch events
    touch_button_sensor_handle_events(btn_handle);

    // Add delay to prevent tight loop
    vTaskDelay(pdMS_TO_TICKS(20));  // 20ms interval is typically sufficient
}
```

13.2.4 Examples

- touch/touch_button_sensor

13.2.5 API Reference

Header File

- components/touch/touch_button_sensor/include/touch_button_sensor.h

Functions

esp_err_t **touch_button_sensor_create** (*touch_button_config_t* *config, *touch_button_handle_t* *handle, *touch_cb_t* cb, void *cb_arg)

Create a touch button sensor instance.

This function initializes the touch sensor hardware (unless skip_lowlevel_init is true), sets up FSM instances for touch detection, and registers callbacks for touch events.

Parameters
- **config** –Touch button sensor configuration
- **handle** –Pointer to receive the created touch button sensor handle
- **cb** –Callback function for touch state changes
- **cb_arg** –User data to pass to callback function

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if config, handle, or required config fields are NULL
- ESP_ERR_NO_MEM if memory allocation fails
- ESP_FAIL if touch sensor or FSM initialization fails

esp_err_t **touch_button_sensor_delete** (*touch_button_handle_t* handle)

Delete a touch button sensor instance.

Stops all FSMs, unregisters callbacks, frees resources, and optionally deinitializes the touch sensor hardware.

Parameters **handle** –Touch button sensor handle to delete

Returns
- ESP_OK on success (or if handle is NULL)
- ESP_ERR_INVALID_STATE if sensor state is invalid

esp_err_t **touch_button_sensor_get_data** (*touch_button_handle_t* handle, uint32_t channel, uint32_t channel_alt, uint32_t *data)

Get touch sensor data for a specific channel and frequency.

Retrieves the smoothed touch sensor reading from the specified channel and frequency instance.

Parameters
- **handle** –Touch button sensor handle
- **channel** –Touch channel number

- **channel_alt** –Frequency instance index (0 to SOC_TOUCH_SAMPLE_CFG_NUM-1)
- **data** –Pointer to store the smoothed touch sensor data

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if handle or data is NULL
- ESP_ERR_INVALID_STATE if sensor not initialized
- ESP_ERR_NOT_FOUND if channel not configured

esp_err_t **touch_button_sensor_get_state**(*touch_button_handle_t* handle, uint32_t channel, *touch_state_t* *state)

Get current state of a touch channel.

Returns whether the touch channel is currently considered active (touched) based on the combined state of all frequency instances.

Parameters

- **handle** –Touch button sensor handle
- **channel** –Touch channel number
- **state** –Pointer to store the channel state

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if handle or state is NULL
- ESP_ERR_INVALID_STATE if sensor not initialized
- ESP_ERR_NOT_FOUND if channel not configured

esp_err_t **touch_button_sensor_get_state_bitmap**(*touch_button_handle_t* handle, uint32_t channel, uint32_t *bitmap)

Get active frequency bitmap for a touch channel.

Returns a bitmap indicating which frequency instances have detected a touch on the specified channel. Each bit corresponds to one frequency.

Parameters

- **handle** –Touch button sensor handle
- **channel** –Touch channel number
- **bitmap** –Pointer to store the frequency activation bitmap

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if handle or bitmap is NULL
- ESP_ERR_INVALID_STATE if sensor not initialized
- ESP_ERR_NOT_FOUND if channel not configured

esp_err_t **touch_button_sensor_handle_events**(*touch_button_handle_t* handle)

Handle pending touch sensor events.

Processes events from all FSM instances. This function should be called periodically to update touch states and trigger callbacks.

Parameters **handle** –Touch button sensor handle

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if handle is NULL
- ESP_ERR_INVALID_STATE if sensor not initialized

Structures

struct **touch_button_config_t**

Configuration structure for touch button sensor.

Public Members

uint32_t **channel_num**

Number of touch button sensor channels

uint32_t ***channel_list**

Touch channel list

float ***channel_threshold**

Threshold for touch detection for each channel

uint32_t ***channel_gold_value**

(Optional) Reference values for touch channels

uint32_t **debounce_times**

Number of consecutive readings needed to confirm state change

bool **skip_lowlevel_init**

Skip low level initialization when working with existing touch driver

Type Definitions

typedef struct touch_button_sensor_t ***touch_button_handle_t**

typedef void (***touch_cb_t**)(*touch_button_handle_t* handle, uint32_t channel, *touch_state_t* state, void *cb_arg)

Touch sensor callback function type.

Param handle Touch button sensor handle
Param channel Channel number that triggered the event
Param state Current state of the touch sensor
Param cb_arg User data passed to the callback

Enumerations

enum **touch_state_t**

Touch sensor state.

Values:

enumerator **TOUCH_STATE_INACTIVE**

Touch sensor is inactive

enumerator **TOUCH_STATE_ACTIVE**

Touch sensor is active

13.3 Touch Button Sensor

13.3.1 Overview

The `touch_slider_sensor` component provides enhanced touch slider detection functionality for ESP32 series chips.

Note:

- ESP32/ESP32-S2/ESP32-S3 touch-related components are intended for testing or demo purposes only. Due to the poor anti-interference capability of the touch functionality, it may not pass EMS testing, and therefore, it is not recommended for mass production products.
- This component is currently applicable to ESP32, ESP32-S2, and ESP32-S3, and requires an IDF version greater than or equal to v5.3.

Key Features

- FSM-based touch detection with configurable thresholds
- Support for slider gesture detection
- Callback-based event notification

Dependencies

- touch_sensor_fsm
- touch_sensor_lowlevel

13.3.2 Configuration Parameters

Touch Slider Configuration Structure

The touch button can be configured via the *touch_slider_config_t* structure:

```
typedef struct {
    uint32_t channel_num;          /*!< Number of touch slider sensor channels */
    uint32_t *channel_list;        /*!< Touch channel list */
    float *channel_threshold;      /*!< Threshold for touch detection for each␣
↪channel */
    uint32_t *channel_gold_value; /*!< (Optional) Reference values for touch␣
↪channels */
    uint32_t debounce_times;       /*!< Number of consecutive readings needed to␣
↪confirm state change */
    uint32_t filter_reset_times;   /*!< Number of consecutive readings to reset␣
↪position filter */
    uint32_t position_range;        /*!< Maximum position value of touch slider,␣
↪range [0, position_range]. Higher values provide better position resolution */
    uint8_t calculate_window;     /*!< Window size for position calculation (should␣
↪be <= channel_num). Set to 0 for auto-default: 2 for 2-channel, 3 for 3+␣
↪channels */
    float swipe_threshold;         /*!< The threshold for identifying swiping */
    float swipe_hysterisis;        /*!< The hysterisis for identifying swiping */
    float swipe_alpha;             /*!< Filter parameter for estimating speed */
    bool skip_lowlevel_init;       /*!< Skip low level initialization when working␣
↪with existing touch driver */
} touch_slider_config_t;
```

Parameter Descriptions

Parameter	Description	Default Value
channel_num	Number of touch slider sensor channels	•
channel_list	Array of touch channel numbers to use	•
channel_threshold	Array of threshold values for each channel	•
channel_gold_value	Reference values for touch channels (optional)	NULL
debounce_times	Consecutive readings for state change	3
filter_reset_times	Consecutive readings to reset position filter	•
position_range	Maximum slider position value (higher = better resolution)	•
calculate_window	Window size for position calculation	Auto (0)
swipe_threshold	Speed threshold for identifying swiping	•
swipe_hysterisis	Speed hysteresis for identifying swiping	•
swipe_alpha	Filter parameter for estimating swipe speed	•
skip_lowlevel_init	Skip touch driver initialization if exists	false

Calculate Window Configuration

The `calculate_window` parameter determines how many adjacent touch channels are used for position calculation. This affects the precision and noise immunity of the slider.

Default Values (when set to 0):

- **2 channels**: `calculate_window = 2` (uses all available channels)
- **3+ channels**: `calculate_window = 3` (optimal balance between precision and noise immunity)

Manual Configuration:

- You can explicitly set any value from 2 to `channel_num`
- Smaller values (2): Higher sensitivity but more susceptible to noise
- Larger values (3+): Better noise immunity but potentially lower resolution
- For high-precision applications: Recommended value is `min(3, channel_num)`
- **Note**: Minimum value is 2, as slider position calculation requires at least 2 adjacent channels

Backward Compatibility:

- Setting `calculate_window = 0` enables automatic default selection
- Existing code that doesn't initialize this field will use optimal defaults
- Explicitly set values continue to work as before

13.3.3 API Usage Examples

Create and Initialize

```
uint32_t channel_list[] = {2, 4, 6, 12, 10, 8};
float threshold[] = {0.005f, 0.005f, 0.005f, 0.005f, 0.005f, 0.005f};
touch_slider_config_t config = {
    .channel_num = 6,
    .channel_list = channel_list,
    .channel_threshold = threshold,
    .filter_reset_times = 5,
    .position_range = 10000,
    .swipe_alpha = 0.9,
    .swipe_threshold = 50,
    .swipe_hysterisis = 40,
    .channel_gold_value = NULL,
    .debounce_times = 0,
    .calculate_window = 0,  // Use default value (auto-selected as 3 for 6␣
↪channels)
    .skip_lowlevel_init = false
};

// Test successful creation
TEST_ASSERT_EQUAL(ESP_OK, touch_slider_sensor_create(&config, &s_touch_slider,␣
↪touch_slider_position_callback, touch_slider_event_callback, NULL));
TEST_ASSERT_NOT_NULL(s_touch_slider);
```

Event Callback

The callback function is called when slide is detected. Gesture may be determined based on the swiping speed or displacement.

```
static void touch_slider_event_callback(touch_slider_handle_t handle, touch_slider_
↪event_t event, int32_t data, void *cb_arg)
{
    if (event == TOUCH_SLIDER_EVENT_RIGHT_SWIPE) {
        printf("Right swipe (speed)\n");
    } else if (event == TOUCH_SLIDER_EVENT_LEFT_SWIPE) {
        printf("Left swipe (speed)\n");
    } else if (event == TOUCH_SLIDER_EVENT_RELEASE) {
        printf("Slide %ld\n", data);
        if (data > 1000)
        {
            printf("Right swipe (displacement)\n");
        }
        else if (data < -1000)
        {
            printf("Left swipe (displacement)\n");
        }
    } else if (event == TOUCH_SLIDER_EVENT_POSITION)
    {
        printf("pos,%" PRId64 ",%lu\n", get_time_in_ms(), data);
    }
}
```

Event Handling

The touch button sensor component provides an event handling mechanism to process touch events in a non-blocking way. Events should be handled periodically in your application' s main loop or in a dedicated task.

```
// In your main loop or task
while (1) {
    // Process any pending touch events
    touch_slider_sensor_handle_events(s_touch_slider);

    // Add delay to prevent tight loop
    vTaskDelay(pdMS_TO_TICKS(20));  // 20ms interval is typically sufficient
}
```

13.3.4 Examples

- touch/touch_slider_sensor

13.3.5 API Reference

Header File

- components/touch/touch_slider_sensor/include/touch_slider_sensor.h

Functions

esp_err_t **touch_slider_sensor_create** (*touch_slider_config_t* *config, *touch_slider_handle_t* *handle, *touch_slider_event_cb_t* cb, void *cb_arg)

Create a touch slider sensor instance.

This function initializes the touch sensor hardware (unless skip_lowlevel_init is true), sets up FSM instances for touch detection, and registers callbacks for touch events.

Note: The calculate_window parameter determines how many adjacent channels are used for position calculation. For backward compatibility, if set to 0, default values will be used:

- 2 channels: window_size = 2 (use all channels)
- 3+ channels: window_size = 3 (optimal balance between precision and noise immunity)
- For high-precision applications: window_size = min(3, channel_num)

Parameters
- **config** –Touch slider sensor configuration
- **handle** –Pointer to receive the created touch slider sensor handle
- **cb** –Callback function for touch slider events
- **cb_arg** –User data to pass to callback function

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if config, handle, or required config fields are NULL, or if calculate_window is less than 2 or greater than channel_num (0 is acceptable for auto-default)
- ESP_ERR_NO_MEM if memory allocation fails
- ESP_FAIL if touch sensor or FSM initialization fails

esp_err_t **touch_slider_sensor_delete** (*touch_slider_handle_t* handle)

Delete a touch slider sensor instance.

Stops all FSMs, unregisters callbacks, frees resources, and optionally deinitializes the touch sensor hardware.

Parameters **handle** –Touch slider sensor handle to delete

Returns

- ESP_OK on success (or if handle is NULL)
- ESP_ERR_INVALID_STATE if sensor state is invalid

esp_err_t **touch_slider_sensor_get_data** (*touch_slider_handle_t* handle, uint32_t channel, uint32_t channel_alt, uint32_t *data)

Get touch sensor data for a specific channel and frequency.

Retrieves the smoothed touch sensor reading from the specified channel and frequency instance.

Parameters
- **handle** –Touch slider sensor handle
- **channel** –Touch channel number
- **channel_alt** –Frequency instance index (0 to SOC_TOUCH_SAMPLE_CFG_NUM-1)
- **data** –Pointer to store the smoothed touch sensor data

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if handle or data is NULL
- ESP_ERR_INVALID_STATE if sensor not initialized
- ESP_ERR_NOT_FOUND if channel not configured

void **touch_slider_sensor_get_state** (*touch_slider_handle_t* handle, bool *pressed, uint32_t *pos, float *speed)

Get touch sensor data for a specific channel and frequency.

Retrieves the smoothed touch sensor reading from the specified channel and frequency instance.

Parameters
- **handle** –Touch slider sensor handle
- **pressed** –Touch slider is pressed or not
- **pos** –Touch slider speed
- **speed** –Touch slider speed

esp_err_t **touch_slider_sensor_handle_events** (*touch_slider_handle_t* handle)

Handle pending touch sensor events.

Processes events from all FSM instances. This function should be called periodically to update touch states and trigger callbacks.

Parameters **handle** –Touch slider sensor handle

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if handle is NULL
- ESP_ERR_INVALID_STATE if sensor not initialized

Structures

struct **touch_slider_config_t**

Configuration structure for touch slider sensor.

Public Members

uint32_t **channel_num**

Number of touch slider sensor channels

uint32_t ***channel_list**

Touch channel list

float ***channel_threshold**

Threshold for touch detection for each channel

uint32_t ***channel_gold_value**

(Optional) Reference values for touch channels

uint32_t **debounce_times**

Number of consecutive readings needed to confirm state change

uint32_t **filter_reset_times**

Number of consecutive readings to reset position filter

uint32_t **position_range**

Maximum position value of touch slider, range [0, position_range]. Higher values provide better position resolution

uint8_t **calculate_window**

Window size for position calculation (should be <= channel_num). Set to 0 for auto-default: 2 for 2-channel, 3 for 3+ channels

float **swipe_threshold**

The speed threshold for identifying swiping

float **swipe_hysterisis**

The speed hysterisis for identifying swiping

float **swipe_alpha**

Filter parameter for estimating speed

bool **skip_lowlevel_init**

Skip low level initialization when working with existing touch driver

Type Definitions

typedef struct touch_slider_sensor_t ***touch_slider_handle_t**

typedef void (***touch_slider_event_cb_t**)(*touch_slider_handle_t* handle, *touch_slider_event_t* event, int32_t data, void *cb_arg)

Touch slider sensor event callback function type.

Param handle Touch slider sensor handle
Param event Touch slider event
Param data Touch slider data. Position if event == TOUCH_SLIDER_EVENT_POSITION, displacement if event == TOUCH_SLIDER_EVENT_RELEASE
Param cb_arg User data passed to the callback

Enumerations

enum **touch_slider_event_t**

Touch slider event.

Values:

enumerator **TOUCH_SLIDER_EVENT_NONE**

Touch slider is stationary

enumerator **TOUCH_SLIDER_EVENT_RIGHT_SWIPE**

Touch slider is swiped right (from 0 to position_range)

enumerator **TOUCH_SLIDER_EVENT_LEFT_SWIPE**

Touch slider is swiped left (from position_range to 0)

enumerator **TOUCH_SLIDER_EVENT_RELEASE**

Touch slider is released

enumerator **TOUCH_SLIDER_EVENT_POSITION**

Touch slider position is calculated

Chapter 14

Storage

14.1 Storage Media

Supported storage media list:

Name	Key Features	Application Scenarios	Capacity	Transmission Mode	Speed	Driver	Notes
SPI Flash	Can be shared with code, no additional cost	Parameter storage, text, image storage	MB	SPI	40/80 MHz 4-line	SPI Flash Driver	
SD Card	Large capacity, pluggable	Audio, video file storage	GB	SDIO/SPI	20/40 MHz 1/4-line	SD/SDIO/MMC Driver for SD	*1
eMMC	Large capacity, high-speed read/write	Audio, video file storage	GB	SDIO	20/40 MHz 1/4/8-line	SD/SDIO/MMC Driver for eMMC	*2
USB Flash Drive	Large capacity, pluggable	Audio, video file storage	GB	USB	12 Mbps/480 Mbps	USB Host Driver	*3
EEPROM	Byte-addressable, low cost	Parameter storage	MB	I2C	100 ~ 400 kHz	eeprom driver	

Note:

- *1. SD card SDIO mode is only available on ESP chips that support SDIO host peripherals (such as ESP32, ESP32-S3);
- *2. Only available on ESP chips that support SDIO host peripherals (such as ESP32, ESP32-S3);
- *3. Only available on ESP chips that support USB-OTG peripherals (such as ESP32-S2, ESP32-S3).

14.1.1 SPI Flash

ESP series chips use NOR Flash by default to store user code and data. Flash can be integrated into modules or chips, with capacities typically of 4 MB, 8 MB, or 16 MB.

Users can use partition tables to partition Flash. Based on the functional division of partition tables, the main Flash designated space can be used as non-volatile storage (NVS) space to store application parameters, or mounted to file systems (FatFS, etc.) to store text, images and other files.

Flash chips support dual-line (DOUT/DIO) and quad-line (QOUT/QIO) operation modes, and can be configured to work in 40 MHz or 80 MHz modes. Since the main Flash chip can be used directly for data storage without additional storage chips, it is particularly suitable for applications with small capacity requirements (MB), high integration requirements, and cost sensitivity.

Note: For ESP-IDF v4.0 and above, the SPI Flash component not only supports using the main Flash for user data storage, but also supports connecting to a second external Flash chip for data storage.

Reference Documentation:

- SPI Flash API
- NVS Flash API
- FatFS API

Example Programs:

- storage/nvs: Use NVS for parameter storage;
- storage/fatfs: Use FatFS for file storage.

14.1.2 SD Card

ESP series chips support using SDIO interface or SPI interface to access SD cards. The SDIO interface supports 1-line, 4-line, or 8-line modes, supporting default rate of 20 MHz or high-speed rate of 40 MHz. It should be noted that the SDIO interface generally can only use fixed pins. The SPI interface supports specifying arbitrary IOs for SD cards through GPIO matrix, and can support access to multiple SD cards through CS pins. The SPI interface is more flexible in hardware design, but has lower access rates compared to the SDIO interface.

The `SD/SDIO/MMC Driver` in ESP-IDF provides protocol layer encapsulation based on the two access modes of SD cards, providing SD card initialization interfaces and protocol layer APIs. SD cards have the characteristics of large capacity and pluggability, and are widely used in application scenarios such as smart speakers and electronic photo albums that have large capacity storage requirements.

Reference Documentation:

- SD/SDIO/MMC Driver: Provides SD card initialization interfaces and protocol layer APIs;
- SDMMC Host Driver: Provides SDIO mode interface implementation;
- SD SPI Host Driver: Provides SPI mode interface implementation;
- When using SPI or 1-bit mode, please note the pin pull-up requirements.

Example Programs:

- storage/sd_card: Access SD card using FAT file system.

14.1.3 eMMC

eMMC (embedded MMC) memory chips have similar protocols to SD cards and can use the same driver SD/SDIO/MMC Driver. However, it should be noted that eMMC chips can only use SDIO mode and do not support SPI mode.

eMMC is generally soldered to the motherboard in chip form, with higher integration than SD cards, suitable for wearable devices and other scenarios that have large capacity storage requirements while having certain requirements for system integration.

Reference Documentation:

- SD/SDIO/MMC Driver;
- Supported eMMC Speed Modes.

Example Programs:

- storage/emmc: Access eMMC storage using FAT file system.

14.1.4 EEPROM

EEPROM (such as AT24C0X series) is a 1024-16384 bit serial electrically erasable memory (can also run in read-only mode by controlling pin levels), its storage space is generally distributed by `word`, with each `word` containing `8-bit` space. EEPROM is byte-addressable, with simple read/write operations, particularly suitable for saving configuration parameters. After optimization, it can also be applied to industrial and commercial scenarios with certain requirements for power consumption and reliability.

Adapted EEPROM Chips:

Name	Function	Bus	Vendor	Datasheet	Driver
AT24C01/02	1024/2048 bits EEPROM	I2C	Atmel	Datasheet	AT24C02 driver

Frequently Asked Questions (FAQ)

- Please refer to: ESP-FAQ Storage Section

14.2 File System

Supported file systems list:

Table 1: File System Feature Comparison

Key Features	NVS Library	FAT File System	SPIFFS File System	LittleFS File System
Characteristics	Key-value pair storage, secure interface	Operating system support, strong compatibility	Designed for embedded development, low resource usage	Low resource usage, fast read, write, erase speed
Application Scenarios	Parameter storage	Audio/video, file storage	Audio/video, file storage	File storage
Storage Media	SPI NOR Flash	SPI NOR Flash, SD/MMC card, USB flash drive	SPI NOR Flash	SPI NOR Flash, SD/MMC card
Capacity	KB-MB	GB	≤ 128 MB	< 128 MB
Directory Support	✗	✓	✗	✓
Wear Levelling	✓	Optional	✓	✓
Read/Write Efficiency	High	Medium	Medium	High
Resource Usage	Low	Medium	Low	Very Low
Power-down Protection	✓	✗	✗	✓
Encryption	✓	✓	✗	✓

Note:

- ✅: Supports this feature
- ❌: Does not support this feature
- Read/write efficiency and resource usage are relative comparisons, actual performance depends on specific configuration and usage scenarios.

14.2.1 NVS Library

The Non-Volatile Storage (NVS) library is mainly used to read and write data stored in Flash NVS partitions. Data in NVS is saved in key-value pairs, where keys are ASCII strings and values can be integers, strings, or binary data (BLOB) types.

Main Features:

- **Key-value pair storage**: Supports multiple data types including integers, strings, BLOB
- **Power-down protection**: Atomic updates, ensuring data consistency
- **Encryption support**: Supports AES-XTS encryption
- **Wear levelling**: Built-in wear levelling mechanism

Usage Limitations:

- **Applicable scenarios**: Only suitable for configuration data, not suitable for frequent writes or large data
- **Partition size**: Recommended not to exceed 128 KB
- **Space requirements**: Requires sufficient space to store both old and new data simultaneously

If you need to store larger BLOBs or strings, please consider using the FAT file system based on the wear levelling library.

Reference Documentation:

- Non-Volatile Storage Library.
- For mass production, you can use the NVS Partition Generator Tool.

Example Programs:

- Write a single integer value: storage/nvs_rw_value.
- Write binary large object: storage/nvs_rw_blob.

14.2.2 FAT File System

ESP-IDF uses the FatFs library to implement support for the FAT file system. FatFs is a file system layer independent of platform and storage media, providing access to physical devices (such as Flash, SD card, USB flash drive) through a unified interface. Users can directly call FatFs interface operations, or use most of the FatFs library functionality through C standard library and POSIX API via VFS (Virtual File System).

Main Features:

- **Wide compatibility**: Compatible with PC and other platforms, supports standard FAT format
- **Multiple storage media**: Supports SPI Flash, SD/MMC cards, USB flash drives and other storage media
- **Directory support**: Supports multi-level directory structure, suitable for storing large numbers of files
- **Encryption support**: Supports partition encryption (Flash Encryption)

Usage Limitations:

- **Power-down recovery**: Relatively weak power-down recovery capability
- **Partition size**: Minimum 8 sectors required when wear levelling is enabled

Reference Documentation:

- Using FatFs with VFS.
- FATFS Image Generation and Parsing Tool.

Example Programs:

- storage/sd_card: Access SD card using FAT file system.
- storage/ext_flash_fatfs: Access external Flash chip using FAT file system.
- peripherals/usb/host/msc: Access USB flash drive using FAT file system (requires ESP32-S2/ESP32-S3).

14.2.3 SPIFFS File System

SPIFFS is an embedded file system dedicated to SPI NOR Flash, natively supporting wear levelling, file system consistency checks and other functions. Users can directly call the POSIX-style interfaces provided by SPIFFS, or operate most of SPIFFS functionality through VFS.

Main Features:

- **Embedded optimization**: Designed specifically for NOR Flash, low RAM usage
- **Static wear levelling**: Built-in wear levelling algorithm
- **Power-down recovery**: Certain degree of post-power-down repair functionality

Usage Limitations:

- **No directory support**: Only supports flat file structure
- **Capacity limitation**: Maximum support for 128 MB Flash
- **Performance degradation**: Significant performance degradation when usage exceeds 70%
- **Development stopped**: Maintenance has been discontinued

Reference Documentation:

- SPIFFS File System.
- SPIFFS Image Generation Tool.

Example Programs:

- storage/spiffs: SPIFFS usage example.
- storage/spiffsgen: Demonstrates how to use the SPIFFS image generation tool to automatically create SPIFFS images from host folders during the build process.

14.2.4 LittleFS File System

LittleFS is a block-based file system designed specifically for microcontrollers and embedded devices, natively supporting wear levelling, file system consistency checks, power-down protection and other functions.

Main Features:

- **Excellent power-down recovery**: Fault-safe features, strong power-down protection capability
- **Dynamic wear levelling**: Adaptive wear levelling algorithm
- **Extremely low RAM usage**: Fixed and extremely low RAM usage
- **Multiple storage media**: Supports SPI Flash and SD/MMC cards
- **Complete directory support**: Supports directory and subdirectory structure

Usage Limitations:

- **Platform compatibility**: Less compatible with other platforms than FAT (mainly for embedded use)
- **Capacity recommendation**: Recommended to be less than 128 MB for optimal performance
- **Third-party maintenance**: Needs to be obtained through ESP Component Registry
- **Documentation resources**: Fewer documentation resources compared to FAT file system

LittleFS is currently recommended for general application scenarios, especially applications with high power-down protection requirements.

Reference Documentation:

- LittleFS File System Component Repository .
- LittleFS File System Component Usage Guide .

Example Programs:

- storage/littlefs: LittleFS usage example.

14.2.5 Virtual File System (VFS)

The ESP-IDF Virtual File System (VFS) component can provide a unified interface for different file systems (FAT, SPIFFS), and can also provide file-like read/write operation interfaces for device drivers.

Reference Documentation:

- Virtual File System Component.

14.2.6 Storage Security

When selecting and using file systems, please note the following security-related matters:

- **Data encryption**: NVS and FAT file systems support data encryption, LittleFS also supports encryption functionality, SPIFFS currently does not support encryption.
- **Power-down protection**: NVS and LittleFS have good power-down protection mechanisms, FAT and SPIFFS may have data corruption risks during power-down events.
- **Integrity checks**: It is recommended to regularly perform file system integrity checks, especially in production environments.

Reference Documentation:

- Storage Security.

14.2.7 File System Design Recommendations

- Please refer to: File handling design considerations.

Frequently Asked Questions (FAQ)

- Please refer to: ESP-FAQ Storage Section

Chapter 15

Motor

15.1 BLDC Motor

15.1.1 Overview of Bldc Motor Control

This guide includes the following content:

- *Overview of Bldc Motor*: Advantages of bldc motor
- *Drive Method*: Drive hardware methods
- *Control Method*: Various ways to control bldc motor

Overview of Bldc Motor

Brushless Direct Current (BLDC) motor is a type of synchronous motor and can be configured as single-phase, two-phase, or three-phase. This article discusses three-phase bldc motor.

BLDC motor do not use brushes for commutation but instead use electronic commutation, offering the following advantages:

- Better speed-torque characteristics
- Fast dynamic response
- High efficiency
- Long lifespan
- No operational noise
- Wide speed range

BLDC motor consists of two parts: the stator and the rotor:

- The stator is the armature with coil windings and has three star-connected stator windings distributed along the stator circumference to form evenly distributed magnetic poles.
- The rotor is made of permanent magnets, typically with 2 to 8 magnetic poles, alternating between north and south poles.

Fig. 1: BLDC Brushless Motor

If a fixed DC current is applied to the motor, it will only generate a constant magnetic field and cannot rotate. By appropriately sequencing the stator phase energization, a rotating magnetic field is produced. The rotor' s inherent magnetic poles follow the rotating magnetic field of the stator in an orderly manner, achieving rotation.

Note: Ideally, the torque peak occurs when the two magnetic fields are perpendicular, and is weakest when they are parallel.

Important Parameters:

- *KV value (rpm/V)*: Indicates the specific speed of the bldc motor at a specific operating voltage.

$$MaximumIdleSpeed(rpm) = KV * operatingvoltage$$

- *Torque (Nm)*: The driving torque generated by the rotor to drive mechanical loads.
- *Speed (rpm)*: The motor' s rotational speed per minute.
- *Maximum current (A)*: The maximum current that can be safely sustained.
- *Pole pairs Pp*: The number of magnets on the rotor divided by 2. This can be determined by manually rotating the motor one full turn with a small voltage applied to any two phases and counting the number of resistances felt. If 6 resistances are felt, the pole pair number is 6.
- *Phase inductance LS (H)*: The inductance across the stator windings when the motor is stationary, with the phase inductance being half of it:

$$LS = LL/2$$

- *Phase resistance RS (Ω)*: The resistance between two phases measured with a multimeter, with the phase resistance being half of it:

$$RS = RL/2$$

Drive Method

BLDC motors are generally driven by an inverter circuit composed of 6 MOSFETs. By appropriately switching the upper and lower arm devices, a rotating magnetic field is produced on the stator.

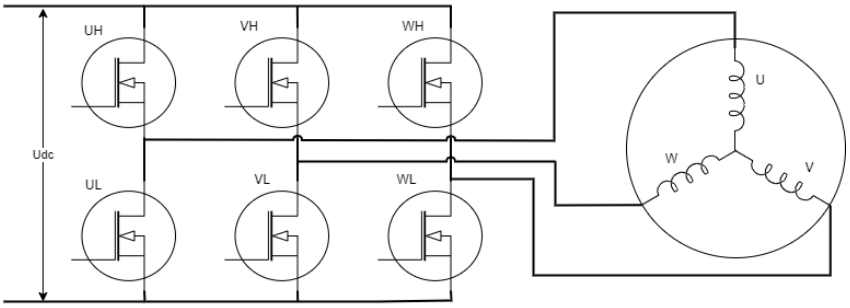


Fig. 2: BLDC Three-phase Inverter Circuit

Through the inverter circuit shown in the figure, the rotor magnet can rotate in cycles by sequentially switching the current. The rotor completes one full rotation every 6 current switches. This demonstrates the method of conducting two bridge arms.

Note: The upper and lower bridge arms cannot be turned on simultaneously; otherwise, a short circuit will occur. Therefore, dead time control must be introduced to prevent the upper and lower bridge arms of the same phase from being turned on simultaneously.

Table 1: Bridge Arm Conduction and Current Flow

Upper Arm Conduction	Lower Arm Conduction	Phase Current A	Phase Current B	Phase Current C
UH	WL	DC+	Floating	DC-
UH	VL	DC+	DC-	Floating
WH	VL	Floating	DC-	DC+
WH	UL	DC-	Floating	DC+
VH	UL	DC-	DC+	Floating
VH	WL	Floating	DC+	DC-

To make the motor' s rotational speed controllable, the control signal applied to the upper arm can be set as a PWM signal, and by adjusting the PWM duty cycle, the rotational speed can be controlled.

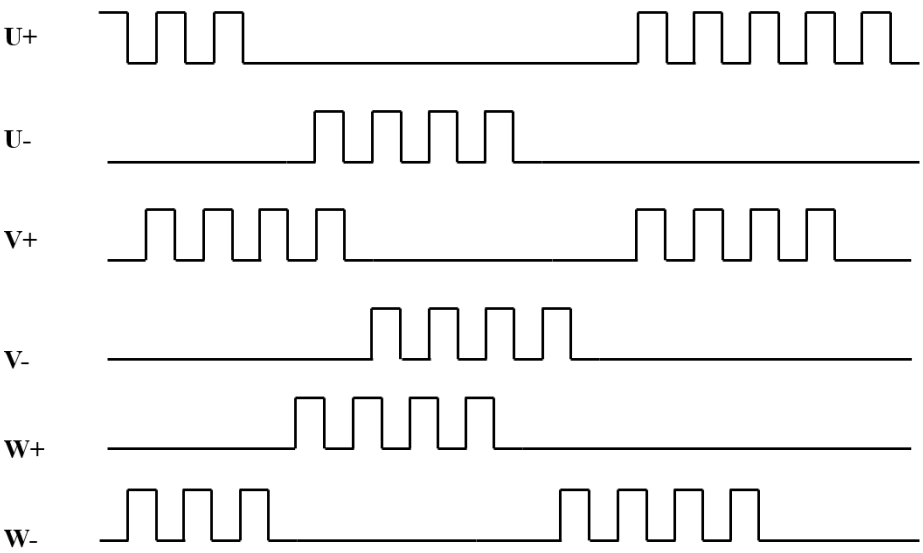


Fig. 3: BLDC PWM Speed Control

Control Method

In actual motor control, it is necessary to obtain the rotor position and calculate the next bridge arm to be turned on in order to make the motor rotate. There are generally two methods to obtain the rotor position: sensor-based detection and sensorless detection.

Sensor-based Hall Effect In bldc motor, three switch-type Hall effect sensors are typically used to detect the rotor's position, installed 120° apart, as shown below.

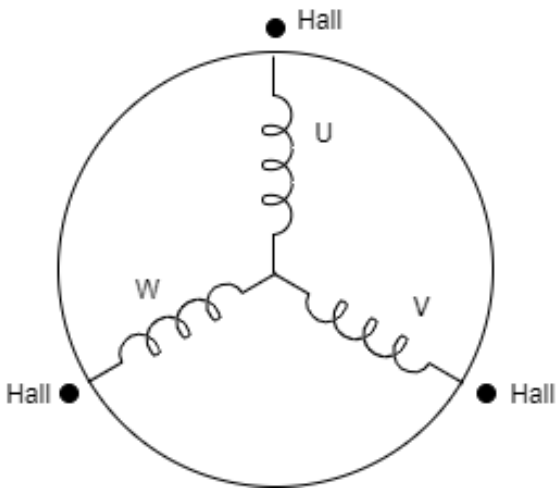


Fig. 4: BLDC Hall Sensor Installation

When the N pole approaches Hall sensor a, a outputs a high level (1), and when the N pole moves away from a, a outputs a low level. The same applies to the other sensors. When the rotor completes one full rotation, it produces the following waveform.

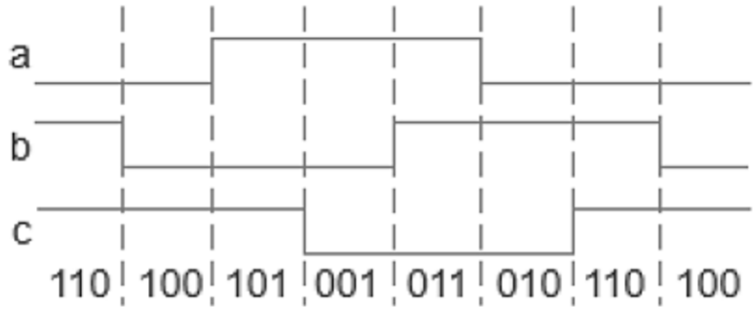


Fig. 5: BLDC Hall Sensor Waveform

By interpreting the output of the Hall sensors, the current position of the rotor is determined. The "two-two conduction" method is used to make the motor rotate, but it has the following drawbacks:

- The sensors are expensive and need to be installed on the motor during manufacturing, increasing installation and wiring costs.
- If the sensors fail, the motor cannot continue to operate.
- Using sensors that do not meet practical needs may result in sensor failure.

Therefore, sensorless control schemes for bldc motor have become mainstream.

Sensorless Detection In some micro-motor systems, installing position sensors negatively affects the motor' s size and cost. Therefore, sensorless position detection is crucial. Sensorless control strategies mainly include the back-EMF method, inductance method, and freewheeling diode method. Among them, the back-EMF method is one of the most widely used and mature schemes.

Back-EMF Back-EMF, according to Lenz' s Law, is opposite in direction to the main voltage provided to the windings. The polarity of back-EMF is opposite to the excitation voltage. Back-EMF mainly depends on three factors:

- Rotor angular speed
- Magnetic field produced by the rotor magnets
- Number of turns in the stator winding

$$BEMF = NlrB\omega$$

For motors, the rotor magnetic field and the number of turns in the stator winding are fixed, so in actual operation, the only factor determining back-EMF is angular speed, or rotor speed. During each commutation, one winding receives positive voltage, the second receives negative voltage, and the third remains open-circuited.

By detecting the zero-crossing points of the back-EMF in each phase winding, the rotor' s six positions can be obtained within one electrical cycle. The zero-crossing point of the back-EMF in the non-conducting phase delayed by 30° electrical angle is the commutation point.

Note: When the motor speed is extremely slow, the amplitude of the back-EMF is very low, making it difficult to detect the zero-crossing point.

There are two ways to detect the zero-crossing point based on back-EMF:

- *Sensorless Square Wave Motor Control Based on ADC Sampling* ADC Sampling Zero-crossing Detection
- *Sensorless Square Wave Motor Control Based on Comparator Detection* Comparator Zero-crossing Detection

In addition, there is a sensorless FOC scheme based on phase current acquisition:

- Dual-resistor Sensorless FOC Scheme (to be updated)

15.1.2 Sensorless Square Wave Motor Control Based on ADC Sampling

This guide includes the following content:

Table of Contents

- *Initial Position Detection*
- *Sensorless Control of BLDC Based on ADC Scheme*
  - *Back EMF Definition*
  - *Zero-crossing Sampling Principle of ADC Scheme*
  - *Zero-crossing Sampling Hardware of ADC Scheme*

Initial Position Detection

From the balanced voltage equation of one phase of a BLDC motor $u_a = Ri_a + L_a \frac{di_a}{dt} + e_a$, we know that when the motor is stationary, the back EMF is zero, so the armature current is:

$$i = \frac{U}{R}\left(1 - \frac{1}{e^{\tau}}\right)$$

By analyzing the above formula, we can accurately determine the position range of the rotor by applying a high-frequency voltage to generate the corresponding pulse current and comparing the magnitude of the pulse current.

To obtain the initial position of the rotor, the *esp_sensorless_bldc_control* component sequentially applies voltage pulses at startup, obtains the current pulses on the sampling resistor, and compares the magnitudes of the 6 vectors to determine the interval of the maximum vector.

Table 2: Pulse Injection Sequence

Order	U	V	W
1	Udc	Udc	GND
2	GND	GND	Udc
3	Udc	GND	Udc
4	GND	Udc	GND
5	GND	Udc	Udc
6	Udc	GND	GND

Note:

1. In the stationary state, specific voltage vectors are injected into the BLDC motor, each vector for a fixed time T_s, to ensure the magnitude of the injected current.
2. The bus current is sampled at the end of the voltage vector injection.
3. Sequentially inject the remaining voltage vectors, compare the current values under each voltage vector, determine the maximum current identifier, and obtain the initial position of the rotor.

Sensorless Control of BLDC Based on ADC Scheme

Back EMF Definition When the BLDC motor rotates, each winding generates a back EMF voltage. According to Lenz' s law, the polarity of the back EMF is opposite to the main voltage. The back EMF calculation formula is:

$$BEMF = NlrB\omega$$

where N is the number of turns of the winding, l is the length of the rotor, r is the inner radius of the rotor, B is the magnetic field of the rotor, and ω is the angular velocity.

When the motor is fixed, the parameters of the motor winding and rotor are fixed. The back EMF of the motor is only proportional to the angular velocity.

The following figure shows the current and back EMF waveforms during one electrical cycle of the motor.

Zero-crossing Sampling Principle of ADC Scheme When the BLDC motor rotates, the zero-crossing point of the back EMF occurs in the floating phase. By detecting the phase-to-ground voltage of each phase and comparing it with the DC bus voltage, the zero-crossing event occurs when the terminal voltage equals half of the DC bus voltage. In the ADC-based zero-crossing detection scheme, the terminal voltage and the DC bus voltage are measured simultaneously and compared to obtain the zero-crossing signal.

Zero-crossing Sampling Hardware of ADC Scheme To simplify the calculation process, the same voltage divider ratio is used for both the terminal voltage and the DC bus voltage. In the 12V motor control scheme, a voltage divider ratio of $1/21$ is used to keep the DC bus voltage and terminal voltage within the V_{ref} range of the ESP32 series chips.

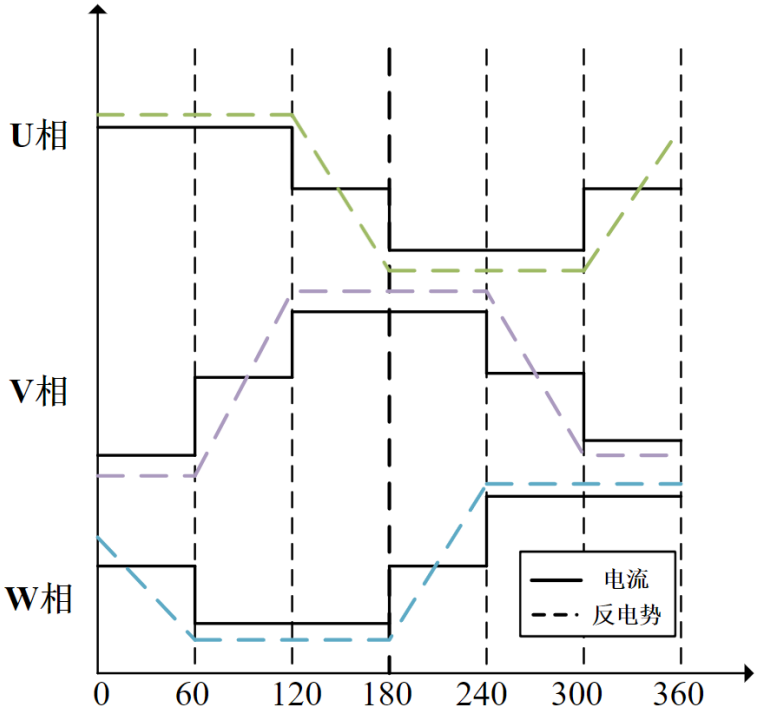


Fig. 6: Current and Back EMF Waveforms

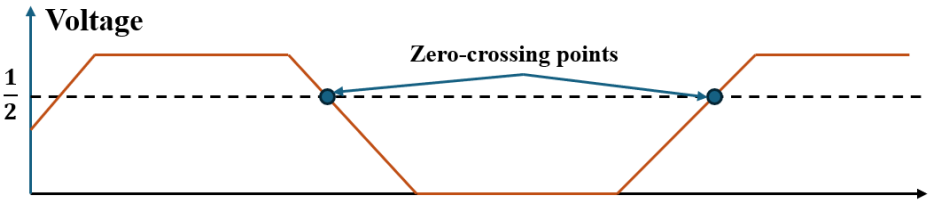


Fig. 7: Implementation of ADC Zero-crossing Detection

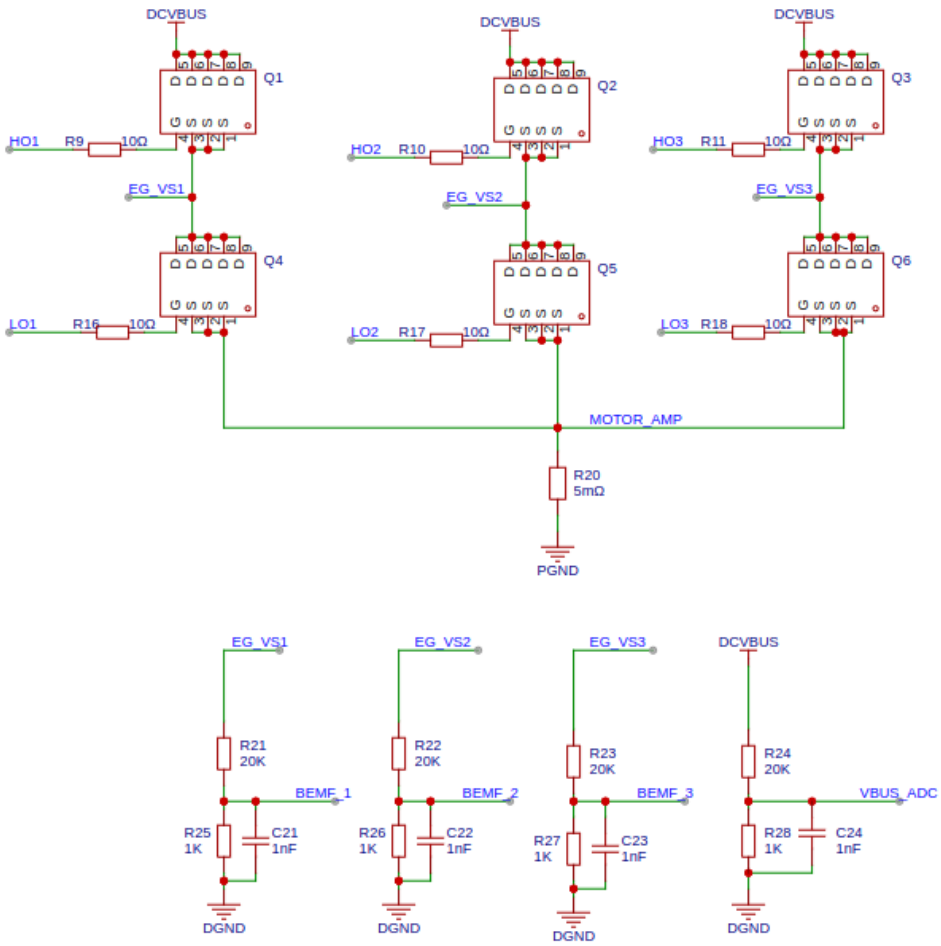


Fig. 8: ADC Zero-crossing Detection Hardware

Note: Note that the voltage needs to be converted to a range that the ESP32 ADC can collect. Please refer to: ESP32 ADC

15.1.3 Sensorless Square Wave Motor Control Based on Comparator Detection

This guide includes the following content:

Table of Contents

- *Sensorless Control of BLDC Based on Comparator Scheme*
  - *Back EMF Definition*
  - *Zero-crossing Sampling Principle of Comparator Scheme*
  - *Zero-crossing Sampling Hardware of Comparator Scheme*

Sensorless Control of BLDC Based on Comparator Scheme

Back EMF Definition When the BLDC motor rotates, each winding generates a back EMF voltage. According to Lenz' s law, the polarity of the back EMF is opposite to the main voltage. The back EMF calculation formula is:

$$BEMF = NlrB\omega$$

where N is the number of turns of the winding, l is the length of the rotor, r is the inner radius of the rotor, B is the magnetic field of the rotor, and ω is the angular velocity.

When the motor is fixed, the parameters of the motor winding and rotor are fixed. The back EMF of the motor is only proportional to the angular velocity.

The following figure shows the current and back EMF waveforms during one electrical cycle of the motor.

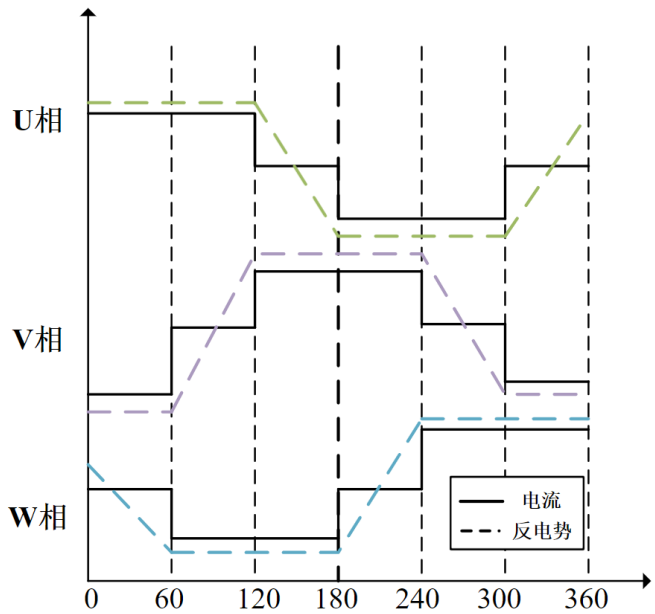


Fig. 9: Current and Back EMF Waveforms

Zero-crossing Sampling Principle of Comparator Scheme When the BLDC motor rotates, the zero-crossing point of the back EMF occurs in the floating phase. By detecting the phase-to-ground voltage of each phase and comparing it with the neutral point voltage, the zero-crossing event occurs when the terminal voltage changes from greater than to less than the neutral point voltage or from less than to greater than the neutral point voltage. However, generally, the neutral point is not accessible in BLDC motors, making direct measurement of the neutral point voltage impossible. In the comparator-based zero-crossing detection scheme, the three-phase windings are connected to a common point through resistors of equal value to reconstruct the neutral point, and the zero-crossing signal is obtained by comparing the neutral point with the terminal voltage using a comparator.

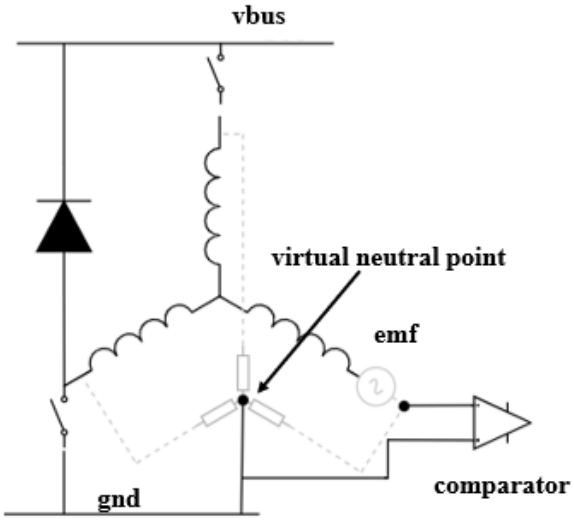


Fig. 10: Comparator Zero-crossing Principle

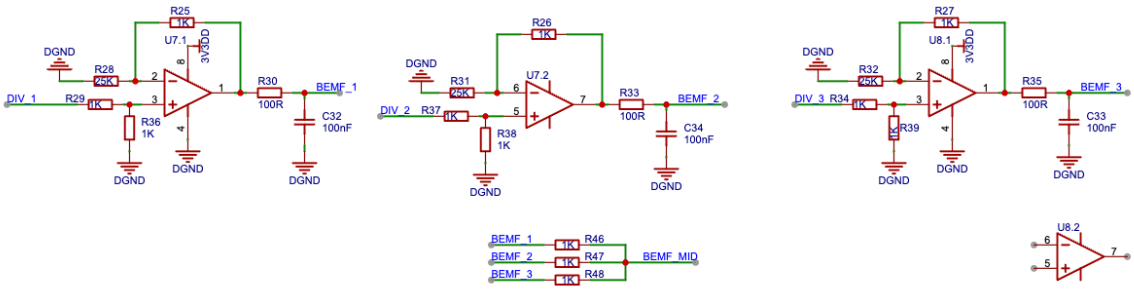


Fig. 11: Comparator Zero-crossing Hardware

Zero-crossing Sampling Hardware of Comparator Scheme Using resistors of equal value to connect each phase to construct a virtual neutral point. Taking the U phase as an example, the U phase back EMF and the neutral point are compared using a comparator to output the zero-crossing signal.

Each phase back EMF has situations where it changes from positive to negative and from negative to positive, resulting in six zero-crossing states for the three phases. For ease of program processing, *esp_sensorless_bldc_control* maps the detected six states to the next commutation action:

Table 3: Forward Mapping Table

Order	ZERO U	ZERO U	ZERO U	U Phase State	V Phase State	W Phase State
↑	0	0	1	Upper Open Lower Close	Upper Close Lower Open	Upper Close Lower Close
↑	0	1	1	Upper Open Lower Close	Upper Close Lower Close	Upper Close Lower Open
↑	0	1	0	Upper Close Lower Close	Upper Open Lower Close	Upper Close Lower Open
↑	1	1	0	Upper Close Lower Open	Upper Open Lower Close	Upper Close Lower Close
↑	1	0	0	Upper Close Lower Open	Upper Close Lower Close	Upper Open Lower Close
↑	1	0	1	Upper Close Lower Close	Upper Close Lower Open	Upper Open Lower Close

Table 4: Reverse Mapping Table

Order	ZERO U	ZERO U	ZERO U	U Phase State	V Phase State	W Phase State
↓	0	1	0	Upper Close Lower Open	Upper Open Lower Close	Upper Close Lower Close
↓	1	1	0	Upper Close Lower Open	Upper Close Lower Close	Upper Open Lower Close
↓	1	0	0	Upper Close Lower Close	Upper Close Lower Open	Upper Open Lower Close
↓	1	0	1	Upper Open Lower Close	Upper Close Lower Open	Upper Close Lower Close
↓	0	0	1	Upper Open Lower Close	Upper Close Lower Close	Upper Close Lower Open
↓	0	1	1	Upper Close Lower Close	Upper Open Lower Close	Upper Close Lower Open

15.1.4 ESP Sensorless BLDC Control Components

This guide includes the following content:

Table of Contents

- *INJECT*
- *ALIGNMENT*
- *DRAG*
- *CLOSED_LOOP*
  - *Zero-crossing Detection Based on ADC Sampling*
  - *Zero-crossing Detection Based on Comparator*
  - *Advance Commutation*
- *Stall Protection*
- *Speed Control*
- *API Reference*
  - *Header File*
  - *Functions*
  - *Structures*
  - *Type Definitions*
  - *Enumerations*
  - *Header File*

– *Macros*

The *esp_sensorless_bldc_control* component is a sensorless BLDC square wave control library based on the ESP32 series chips. It currently supports the following features:

- Zero-crossing detection based on ADC sampling
- Zero-crossing detection supported by comparators
- Rotor initial phase detection based on the pulse method
- Stall protection

This article mainly explains how to use the *esp_sensorless_bldc_control* component for brushless motor development and does not cover the principles. For more information on the principles, please refer to:

- *Overview of Bldc Motor Control* Overview of Brushless Motor Control
- *Sensorless Square Wave Motor Control Based on ADC Sampling* Zero-crossing Detection Based on ADC Sampling
- *Sensorless Square Wave Motor Control Based on Comparator Detection* Zero-crossing Detection Based on Comparators

The sensorless square wave control process can be mainly divided into the following parts:

- INJECT: Injection phase, obtaining the initial phase through high-frequency voltage pulses `INJECT`
- ALIGNMENT: Alignment phase, fixing the rotor to the initial phase `ALIGNMENT`
- DRAG: Drag phase, rotating the rotor through six-step commutation `DRAG`
- CLOSED_LOOP: Sensorless closed-loop control, commuting by detecting back EMF zero-crossing points `CLOSED_LOOP`
- BLOCKED: Motor stall `BLOCKED`
- STOP: Motor stop `STOP`
- FAULT: Motor fault `FAULT`

Next, the specific processes of each part and the parameters to be noted will be introduced.

INJECT

The initial phase of the motor is obtained through pulse injection, and the bus current is collected at the low end of the inverter circuit. As shown in the figure below:

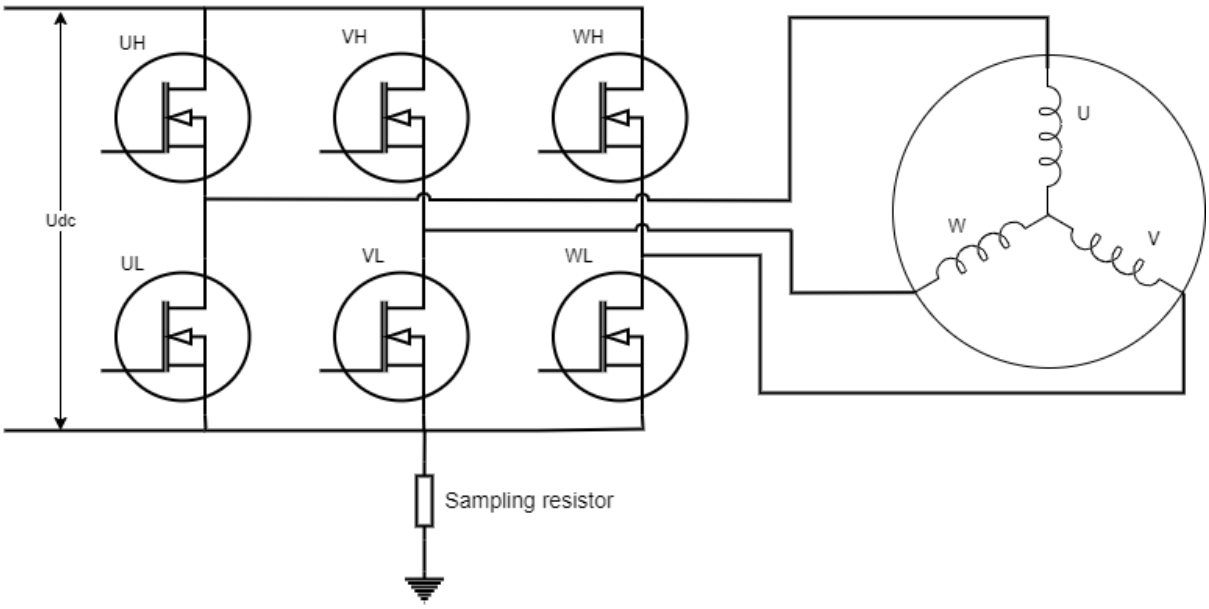


Fig. 12: BLDC Bus Current Collection

Note: Since the current cannot be collected directly, a sampling resistor is used to convert the current into voltage. Note that the voltage needs to be converted to a range that the ESP32 ADC can collect. Please refer to: ESP32 ADC

Since the current only exists when both the upper and lower tubes are conducting, ADC sampling needs to be performed when the upper tube is conducting. Configure MCPWM in rising and falling mode and sample when the counter reaches the peak to accurately collect the bus voltage.

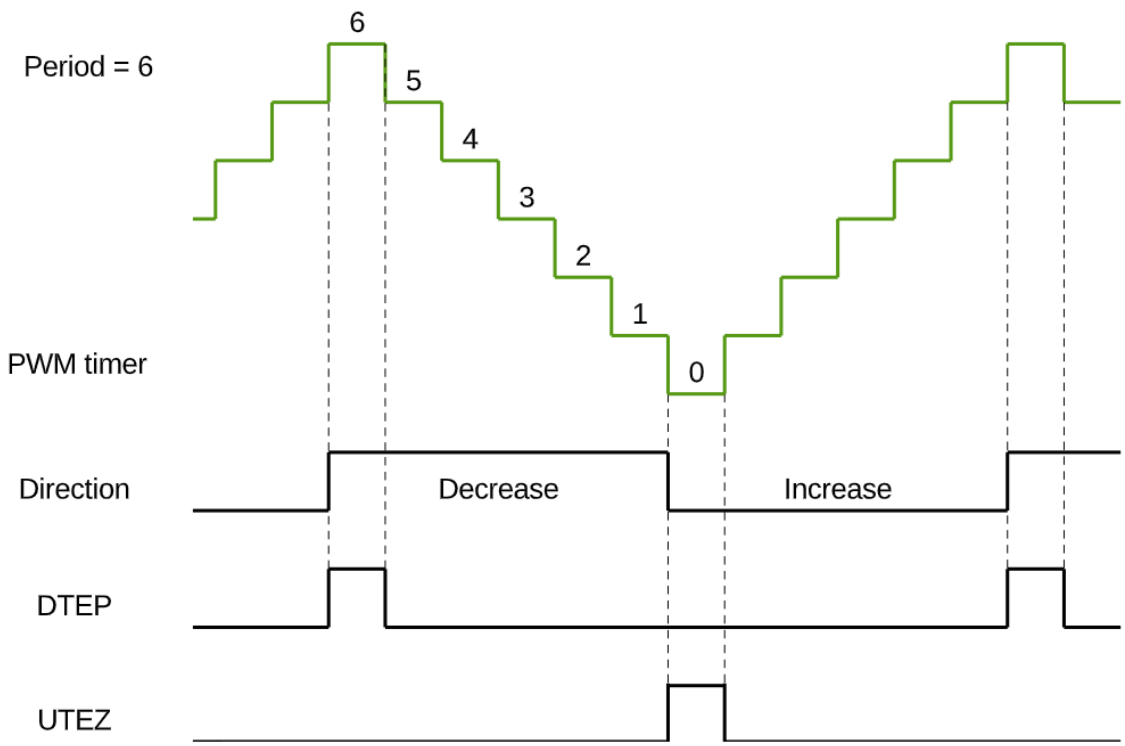


Fig. 13: MCPWM Rising and Falling Mode

Note: The LEDC driver does not support callback triggering at high levels, so the INJECT mode cannot be used with the LEDC driving method.

`INJECT_ENABLE` is set to 1 to enable INJECT mode, otherwise, it is disabled. The default is 0. The PWM generation mode must be MCPWM.

`INJECT_DUTY` is the injected voltage size, generally using high duty cycle injection.

`CHARGE_TIME` is the inductance charging time and pulse injection time, which affects the accuracy of initial phase detection. If this value is too small, the collected ADC value will be 0; if it is too large, the ADC value will be too high. Manually rotate the motor, and it is best to obtain stable phases 1-6 in one turn without errors phases 0 and 7.

ALIGNMENT

To ensure the brushless motor can start normally, it is necessary to determine the position of the rotor when it is stationary. In practical applications, this is done by energizing a set of windings for a certain period, fixing the rotor in a specific phase, and preparing for subsequent forced commutation.

`ALIGNMENTNMS` Alignment time, if too long, it will overcurrent. If too short, the rotor may not be aligned to the correct phase.

`ALIGNMENTDUTY` Alignment force.

DRAG

The rotor is dragged through six-step commutation, using a boost and frequency increase method. Gradually increase the voltage and commutation frequency to give the motor an initial speed with a significant back EMF. The motor should drag smoothly without noise or stuttering. The drag time does not need to be too long.

`RAMP_TIM_STA` Initial delay time for dragging.

`RAMP_TIM_END` Minimum delay time for dragging.

`RAMP_TIM_STEP` Step increment for drag time.

`RAMP_DUTY_STA` Initial duty cycle for dragging.

`RAMP_DUTY_END` Maximum duty cycle for dragging.

`RAMP_DUTY_INC` Step increment for the duty cycle.

Note: The strong drag parameters need to be tuned in the motor' s working environment, and the no-load parameters may not apply to the loaded condition.

CLOSED_LOOP

Zero-crossing Detection Based on ADC Sampling ADC sampling detects the zero-crossing point by collecting the floating phase voltage and motor power supply voltage, and sampling must be performed when the upper tube is conducting.

Note: ADC zero-crossing detection must use MCPWM as the driver.

`ENTER_CLOSE_TIME` Sets the time to enter the closed loop. By default, the closed loop control can be entered after a period of strong drag.

`ZERO_REPEAT_TIME` The zero-crossing point is considered valid after being detected continuously N times.

`AVOID_CONTINUE_CURRENT_TIME` After commutation, there will be an impact of continuous current. Delay detection to avoid continuous current.

Zero-crossing Detection Based on Comparator Comparator zero-crossing detection compares the floating phase back EMF with the bus voltage using hardware comparators. The zero-crossing signal is detected by the GPIO pin. Due to many noise points in the actual process, multiple detections are needed to confirm the zero-crossing point.

`ZERO_STABLE_FLAG_CNT` Enter sensorless control after multiple stable zero-crossing signals are detected.

`ZERO_CROSS_DETECTION_ACCURACY` Continuous detection of the same signal N times is considered a stable signal. 0xFF means 8 times, 0XFFFF means 16 times. The current maximum supported filtering times are 0xFFFFFFFF. If it still cannot enter the closed loop state, check for hardware issues.

Note: Hardware troubleshooting directions mainly include checking whether the filter capacitors for the three-phase terminal voltage and comparator output are set reasonably.

Advance Commutation The zero-crossing signal generally arrives 30° before commutation. Once the zero-crossing signal is detected, a 30° delay is required. However, during motor rotation, due to variable electrical cycles, software filtering, and delays, a slight compensation for the commutation time is necessary.

`ZERO_CROSS_ADVANCE` Advance commutation time, the advance angle is 180 / ZERO_CROSS_ADVANCE, default is 6.

Note: The commutation angle is not better the earlier it is. Use an oscilloscope to observe whether the calculated commutation angle matches the actual commutation angle.

Stall Protection

If the motor does not commutate for a long time, it is considered stalled. At this point, the motor stops running and enters stall protection status.

Speed Control

The speed is controlled by PID to achieve the set speed.

SPEED_KP P value of speed control.

SPEED_KI I value of speed control.

SPEED_KD D value of speed control.

SPEED_MIN_INTEGRAL Minimum integral value of speed control.

SPEED_MAX_INTEGRAL Maximum integral value of speed control.

SPEED_MIN_OUTPUT Minimum output value of speed control.

SPEED_MAX_OUTPUT Maximum output value of speed control, not exceeding the maximum duty cycle.

SPEED_CAL_TYPE Position PID or Incremental PID.

SPEED_MAX_RPM Maximum RPM.

SPEED_MIN_RPM Minimum RPM.

MAX_SPEED_MEASUREMENT_FACTOR To avoid erroneous speed detection, if the detected speed exceeds this set factor, it is considered an erroneous speed detection.

API Reference

Header File

- components/motor/esp_sensorless_bldc_control/include/bldc_control.h

Functions

ESP_EVENT_DECLARE_BASE (BLDC_CONTROL_EVENT)

using esp_event_handler_register() to register BLDC_CONTROL_EVENT

esp event name

esp_err_t **bldc_control_init** (*bldc_control_handle_t* *handle, *bldc_control_config_t* *config)

init bldc control

Parameters
- **handle** –pointer to bldc control handle
- **config** –pointer to bldc control config

Returns ESP_ERR_INVALID_ARG if handle or config is NULL ESP_ERR_NO_MEM if memory allocation failed ESP_OK on success ESP_FAIL on other errors

esp_err_t **bldc_control_deinit** (*bldc_control_handle_t* *handle)

deinit bldc control

Parameters **handle** –pointer to bldc control handle ESP_ERR_INVALID_ARG if handle or config is NULL ESP_OK on success ESP_FAIL on other errors

esp_err_t **bldc_control_start** (*bldc_control_handle_t* *handle, uint32_t expect_speed_rpm)

motor start

Parameters
- **handle** –pointer to bldc control handle
- **expect_speed_rpm** –expect speed in rpm. This parameter does not work in case of open-loop control

Returns ESP_OK on success

esp_err_t **bldc_control_stop** (*bldc_control_handle_t* *handle)

motor stop

Parameters **handle** –pointer to bldc control handle
Returns ESP_FAIL if motor stop failed ESP_OK on success

dir_enum_t **bldc_control_get_dir** (*bldc_control_handle_t* *handle)

get current motor direction

Parameters **handle** –pointer to bldc control handle
Returns dir_enum_t current motor direction

esp_err_t **bldc_control_set_dir** (*bldc_control_handle_t* *handle, dir_enum_t dir)

set motor direction

Parameters
- **handle** –pointer to bldc control handle
- **dir** –motor direction

Returns ESP_OK on success

int **bldc_control_get_duty** (*bldc_control_handle_t* *handle)

get current motor pwm duty

Parameters **handle** –pointer to bldc control handle
Returns int current motor pwm duty

esp_err_t **bldc_control_set_duty** (*bldc_control_handle_t* *handle, uint16_t duty)

set motor pwm duty, Closed-loop speed control without calls

Parameters
- **handle** –pointer to bldc control handle
- **duty** –motor pwm duty

Returns ESP_OK on success

int **bldc_control_get_speed_rpm** (*bldc_control_handle_t* *handle)

get current RPM

Parameters **handle** –pointer to bldc control handle
Returns int current RPM

esp_err_t **bldc_control_set_speed_rpm** (*bldc_control_handle_t* *handle, int speed_rpm)

set motor RPM

Parameters
- **handle** –pointer to bldc control handle
- **speed_rpm** –motor RPM

Returns ESP_OK on success

Structures

struct **bldc_debug_config_t**

Debug configuration, when activated, will periodically invoke the debug_operation.

Public Members

uint8_t **if_debug**

set 1 to open debug mode

esp_err_t (*__debug_operation__)(void *handle)

debug operation

struct **bldc_control_config_t**

BLDC Control Configuration.

Public Members

speed_mode_t **speed_mode**

Speed Mode

control_mode_t **control_mode**

Control Mode

alignment_mode_t **alignment_mode**

Alignment Mode

bldc_six_step_config_t **six_step_config**

six-step phase change config

bldc_zero_cross_comparer_config_t **zero_cross_comparer_config**

Comparator detects zero crossing config

bldc_debug_config_t **debug_config**

debug config

Type Definitions

typedef void ***bldc_control_handle_t**

bldc control handle

Enumerations

enum **bldc_control_event_t**

Values:

enumerator **BLDC_CONTROL_START**

BLDC control start event

enumerator **BLDC_CONTROL_ALIGNMENT**

BLDC control alignment event

enumerator **BLDC_CONTROL_DRAG**

BLDC control drag event

enumerator **BLDC_CONTROL_STOP**

BLDC control stop event

enumerator **BLDC_CONTROL_CLOSED_LOOP**

BLDC control closed loop event

enumerator **BLDC_CONTROL_BLOCKED**

BLDC control blocked event

enum **speed_mode_t**

Values:

enumerator **SPEED_OPEN_LOOP**

Open-loop speed control, speed control by setting pwm duty, poor load carrying capacity

enumerator **SPEED_CLOSED_LOOP**

Closed-loop speed control, rotational speed control via PID, high load carrying capacity

enum **control_mode_t**

Values:

enumerator **BLDC_SIX_STEP**

six-step phase change

enumerator **BLDC_FOC**

foc phase change, not supported yet

Header File

- components/motor/esp_sensorless_bldc_control/user_cfg/bldc_user_cfg.h

Macros

BLDC_LEDC

BLDC_MCPWM

PWM_MODE

Configure the generation of PWM.

Configure the generation of PWM.

MCPWM_CLK_SRC

MCPWM Settings.

Number of count ticks within a period time_us = 1,000,000 / MCPWM_CLK_SRC

MCPWM_PERIOD

pwm_cycle_us = 1,000,000 / MCPWM_CLK_SRC * MCPWM_PERIOD

FREQ_HZ

LEDC Settings.

DUTY_RES

Set duty resolution to 11 bits

ALARM_COUNT_US

No changes should be made here.

DUTY_MAX

PWM_DUTYCYCLE_05

PWM_DUTYCYCLE_10

PWM_DUTYCYCLE_15

PWM_DUTYCYCLE_20

PWM_DUTYCYCLE_25

PWM_DUTYCYCLE_30

PWM_DUTYCYCLE_40

PWM_DUTYCYCLE_50

PWM_DUTYCYCLE_60

PWM_DUTYCYCLE_80

PWM_DUTYCYCLE_90

PWM_DUTYCYCLE_100

INJECT_ENABLE

Pulse injection-related parameters.

Note: Used to detect the initial phase of the motor; MCPWM peripheral support is necessary. Whether to enable pulse injection.

INJECT_DUTY

Injected torque.

CHARGE_TIME

Capacitor charging and injection time.

ALIGNMENTNMS

Parameters related to motor alignment. Used to lock the motor in a specific phase before strong dragging.

Duration of alignment, too short may not reach the position, too long may cause the motor to overheat.

ALIGNMENTDUTY

alignment torque.

RAMP_TIM_STA

Setting parameters for strong dragging. The principle of strong dragging is to increase the control frequency and intensity.

Note: If the control cycle speeds up, corresponding reductions should be made to the RAMP_TIM_STA, RAMP_TIM_END, RAMP_TIM_STEP The start step time for climbing. A smaller value results in faster startup but may lead to overcurrent issues.

RAMP_TIM_END

The end step time for climbing, adjusted based on the load. If loaded, this value should be relatively larger.

RAMP_TIM_STEP

Decremental increment for climbing step time—adjusted in accordance with RAMP_TIM_STA.

RAMP_DUTY_STA

The starting torque for climbing.

RAMP_DUTY_END

The ending torque for climbing.

RAMP_DUTY_INC

The incremental torque step for climbing—too small a value may result in failure to start, while too large a value may lead to overcurrent issues.

ENTER_CLOSE_TIME

ADC parameters for zero-crossing detection; please do not delete if not in use.

Enter the closed-loop state delay times.

ZERO_REPEAT_TIME

Change phase after detecting zero-crossing signals continuously for several times.

AVOID_CONTINUE_CURRENT_TIME

Avoiding Continuous Current

ZERO_STABLE_FLAG_CNT

Comparator parameters for zero-crossing detection; please do not delete if not in use.

After stable detection for multiple revolutions, it is considered to enter a sensorless state.

ZERO_CROSS_DETECTION_ACCURACY

Count a valid comparator value every consecutive detection for how many times.

ZERO_CROSS_ADVANCE

Common parameter for compensated commutation time calculation.

Advance switching at zero-crossing, switching angle = 180°/ZERO_CROSS_ADVANCE. angle compensation should be provided. >= 6

POLE_PAIR

Motor parameter settings.

Number of pole pairs in the motor.

BASE_VOLTAGE

Rated voltage.

BASE_SPEED

Rated speed unit: rpm.

SPEED_KP

Closed-loop PID parameters for speed.

P

SPEED_KI

I

SPEED_KD

D

SPEED_MIN_INTEGRAL

Minimum integral saturation limit.

SPEED_MAX_INTEGRAL

Maximum integral saturation limit.

SPEED_MIN_OUTPUT

Minimum PWM duty cycle output.

SPEED_MAX_OUTPUT

Maximum PWM duty cycle output.

SPEED_CAL_TYPE

0 Incremental 1 Positional

SPEED_MAX_RPM

Speed parameter settings.

Maximum speed.

SPEED_MIN_RPM

Minimum speed.

MAX_SPEED_MEASUREMENT_FACTOR

Supports a measured speed range from 0 to 1.2 times SpeedMAX. Large values could prevent proper filtering of incorrect data.

15.2 Servo

This component uses the LEDC peripheral to generate PWM signals for independent control of servos with up to 16 channels (ESP32 chips support 16 channels and ESP32-S2 chips support 8 channels) at a selectable frequency of 50 ~ 400 Hz. When using this layer of APIs, users only need to specify the servo group, channel and target angle to realize the angle control of a servo.

Generally, there is a reference signal inside the servo generating a fixed period and pulse width, which is used to compare with the input PWM signal to output a voltage difference so as to control the rotation direction and angle of a motor. A common 180 angular rotation servo usually takes 20 ms (50 Hz) as a clock period and 0.5 ~ 2.5 ms as its high level pulse, making it rotates between 0 ~ 180 degrees.

This component can be used in scenarios with lower control accuracy requirements, such as toy cars, remote control robots, home automation, etc.

15.2.1 Instructions

1. Initialization: Use `servo_init()` to initialize a channel. Please note that ESP32 contains two sets of channels as `LEDC_LOW_SPEED_MODE` and `LEDC_HIGH_SPEED_MODE`, while some chip may only support one channel. The configuration items in this step mainly include maximum angle, signal frequency, and minimum and maximum input pulse width to calculate the correspondence between angle and duty cycle; as well as pins and channels to specify the correspondence with chip pins and LEDC channels, respectively;
2. Set a target angle: use `servo_write_angle()` to specify the servo group, channel and target angle so as to realize angle control of the servo;
3. Read the current angle: you can use `servo_read_angle()` to read the current angle of the servo. Please note that this is a theoretical number calculated based on the input signal;
4. De-initialization: you can use `servo_deinit()` to de-initialize a group of channels when a group of servos is used any more.

15.2.2 Application Example

```
servo_config_t servo_cfg = {
    .max_angle = 180,
    .min_width_us = 500,
    .max_width_us = 2500,
    .freq = 50,
    .timer_number = LEDC_TIMER_0,
    .channels = {
        .servo_pin = {
            SERVO_CH0_PIN,
            SERVO_CH1_PIN,
            SERVO_CH2_PIN,
            SERVO_CH3_PIN,
            SERVO_CH4_PIN,
            SERVO_CH5_PIN,
            SERVO_CH6_PIN,
            SERVO_CH7_PIN,
        },
        .ch = {
```

(continues on next page)

(continued from previous page)

```
            LEDC_CHANNEL_0,
            LEDC_CHANNEL_1,
            LEDC_CHANNEL_2,
            LEDC_CHANNEL_3,
            LEDC_CHANNEL_4,
            LEDC_CHANNEL_5,
            LEDC_CHANNEL_6,
            LEDC_CHANNEL_7,
        },
    },
    .channel_number = 8,
} ;
iot_servo_init(LEDC_LOW_SPEED_MODE, &servo_cfg);

float angle = 100.0f;

// Set angle to 100 degree
iot_servo_write_angle(LEDC_LOW_SPEED_MODE, 0, angle);

// Get current angle of servo
iot_servo_read_angle(LEDC_LOW_SPEED_MODE, 0, &angle);

//deinit servo
iot_servo_deinit(LEDC_LOW_SPEED_MODE);
```

15.2.3 API Reference

Header File

- components/motor/servo/include/iot_servo.h

Functions

esp_err_t **iot_servo_init** (ledc_mode_t speed_mode, const *servo_config_t* *config)

Initialize ledc to control the servo.

Parameters
- **speed_mode** –Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **config** –Pointer of servo configure struct

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Configure ledc failed

esp_err_t **iot_servo_deinit** (ledc_mode_t speed_mode)

Deinitialize ledc for servo.

Parameters **speed_mode** –Select the LEDC channel group with specified speed mode.

Returns
- ESP_OK Success

esp_err_t **iot_servo_write_angle** (ledc_mode_t speed_mode, uint8_t channel, float angle)

Set the servo motor to a certain angle.

Note: This API is not thread-safe

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode.
- **channel** –LEDC channel, select from ledc_channel_t
- **angle** –The angle to go

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **iot_servo_read_angle**(ledc_mode_t speed_mode, uint8_t channel, float *angle)

Read current angle of one channel.

Parameters

- **speed_mode** –Select the LEDC channel group with specified speed mode.
- **channel** –LEDC channel, select from ledc_channel_t
- **angle** –Current angle of the channel

Returns

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Structures

struct **servo_channel_t**

Configuration of servo motor channel.

Public Members

gpio_num_t **servo_pin**[LEDC_CHANNEL_MAX]

Pin number of pwm output

ledc_channel_t **ch**[LEDC_CHANNEL_MAX]

The ledc channel which used

struct **servo_config_t**

Configuration of servo motor.

Public Members

uint16_t **max_angle**

Servo max angle

uint16_t **min_width_us**

Pulse width corresponding to minimum angle, which is usually 500us

uint16_t **max_width_us**

Pulse width corresponding to maximum angle, which is usually 2500us

uint32_t **freq**

PWM frequency

ledc_timer_t **timer_number**

Timer number of ledc

servo_channel_t **channels**

Channels to use

uint8_t **channel_number**

Total channel number

15.3 ESP SimpleFOC

`esp_simplefoc` is an FOC component based on Arduino-FOC, developed specifically for ESP chips that support LEDC and MCPWM functions. It supports the following features:

- Voltage control support
- Motor control parameters can be adjusted and configured through SimpleFOCStudio
- Compatible with Arduino-FOC control routines
- Uses IQMath for FOC computation acceleration
- Allows manual selection of LEDC or MCPWM peripherals to control MOSFET drivers, supporting up to four brushless motor controls

FOC (Field-Oriented Control) is a field-oriented control algorithm for brushless DC motors that can produce smooth torque, speed, and position control. For a more user-friendly explanation, refer to the Arduino-FOC official documentation.

15.3.1 Using SimpleFOC in Your Project

1. Use the command `idf.py add_dependency` to add the component to your project:

```
idf.py add-dependency "espressif/esp_simplefoc"
```

2. Use `idf.py menuconfig` to configure FreeRTOS `(Top) → Component config → FreeRTOS → Kernel`:

- `configTICK_RATE_HZ`: The tick rate frequency for FreeRTOS, defaulting to 100, needs to be set to 1000 in FOC scenarios.

3. Configure motor parameters, MOSFET driver control parameters and pins, angle sensors, and controllers.

Motor Parameters

- `pp`: Pole pair count, which you need to set according to the actual three-phase brushless motor parameters.
- `R`: Motor phase resistance, defaults to `NOT_SET`.
- `KV`: Motor KV value, defaults to `NOT_SET`.
- `L`: Motor phase inductance, defaults to `NOT_SET`.

For a three-phase brushless motor with 14 pole pairs, the corresponding motor initialization would be:

```
BLDCMotor motor = BLDCMotor(14);
```

MOSFET Driver Control Parameters and Pins

- `pwm pin`: PWM output pin, which you need to set according to the actual MOSFET driver circuit.
- `enable pin`: MOSFET driver enable pin. If your MOSFET driver needs manual enabling, fill in the corresponding GPIO manually. Defaults to *NOT_SET*.
- `voltage_power_supply`: Power supply voltage for the MOSFET driver.
- `voltage_limit`: Voltage limit for the MOSFET driver.

For a 3PWM mode 12V MOSFET driver, the corresponding driver initialization and parameters are:

```
BLDCDriver3PWM driver = BLDCDriver3PWM(4, 5, 6);
driver.voltage_power_supply = 12;
driver.voltage_limit = 11;
driver.init({1, 2, 3});
motor.linkDriver(&driver);
```

Angle Sensor

You need to initialize the angle sensor according to the actual sensor you are using. Currently supported angle sensors are:

- `as5048a`: Supports SPI interface for angle data reading.
- `mt6701`: Supports I2C and SPI interfaces for angle data reading.
- `as5600`: Supports I2C interface for angle data reading.

For initializing the AS5600 angle sensor and binding it to the motor:

```
AS5600 as5600 = AS5600(I2C_NUM_0, GPIO_NUM_1, GPIO_NUM_2);
as5600.init();
motor.linkSensor(&as5600);
```

Controller

`esp_simplefoc` supports various control methods. You can choose based on the actual application requirements:

- `torque`: Torque control.
- `velocity`: Speed control.
- `angle`: Angle control.
- `velocity_openloop`: Open-loop speed control.
- `angle_openloop`: Open-loop angle control.

> **Note:** For hardware verification stages, you may prefer to select an open-loop control scheme for quick hardware validation.

4. FOC Initialization and Control Loop

After completing the above parameter settings, you need to run `init` and `initFOC` to perform motor initialization and calibration, then start `loopFOC`:

```
motor.init();
motor.initFOC();
while (1) {
    motor.loopFOC();
    motor.move(target_value);
    command.run();
}
```

> **Note:** If the pole pair count estimated by `initFOC` does not match the actual pole pair count entered, please check the motor' s pole pairs and minimize the distance between the magnetic ring and the angle sensor as much as possible.

5. Run `idf.py build flash` for the initial download, then observe the motor' s actual running performance.

15.3.2 API Reference

The API interface of `esp_simplefoc` is consistent with Arduino-FOC, and you can refer to the Arduino-FOC API documentation.

Chapter 16

Solutions

16.1 4G Solutions

16.1.1 IOT ETH Ethernet

iot_eth provides a complete solution for connecting 4G modules to Ethernet, offering an abstraction layer for the underlying hardware and a default network interface based on LWIP. It also supports user-defined network interfaces.

Features

- Advanced factory mode, supporting various underlying connection methods
- Default support for LWIP network interface, also supports user-defined network interfaces

User Guide

1. First, initialize the *driver* layer to provide lower-level hardware abstraction

```
static esp_err_t drv_init(iot_eth_driver_t *driver)
{
    ESP_LOGI(TAG, "network interface init success");
    return ESP_OK;
}

static esp_err_t drv_set_mediator(iot_eth_driver_t *driver, iot_eth_mediator_t␣
↪*mediator)
{
    net_ctx_t *net = __containerof(driver, net_ctx_t, base);
    net->mediator = mediator;
    ESP_LOGI(TAG, "network interface set mediator success");
    return ESP_OK;
}

static esp_err_t usbh_rndis_transmit(iot_eth_driver_t *h, uint8_t *buffer, size_t␣
↪buflen)
```

(continues on next page)

(continued from previous page)

```
{
    ESP_LOGI(TAG, "network interface transmit success");
    return ESP_OK;
}

static esp_err_t drv_get_addr(iot_eth_driver_t *driver, uint8_t *mac)
{
    ESP_LOGI(TAG, "network interface get mac success");
    return ESP_OK;
}

static esp_err_t drv_deinit(iot_eth_driver_t *driver)
{
    ESP_LOGI(TAG, "network interface deinit success");
    free(driver);
    return ESP_OK;
}

static iot_eth_driver_t * new_rndis_eth_driver()
{
    net_ctx_t *net = calloc(1, sizeof(net_ctx_t));
    net->base.name = "test drv";
    net->base.init = drv_init;
    net->base.deinit = drv_deinit;
    net->base.set_mediator = drv_set_mediator;
    net->base.get_addr = drv_get_addr;
    net->base.transmit = usbh_rndis_transmit;
    return &net->base;
}
```

2. Initialize the *iot_eth* layer

```
iot_eth_config_t eth_cfg = {
    .driver = rndis_handle,
    .stack_input = NULL,
};

iot_eth_handle_t eth_handle = NULL;
ret = iot_eth_install(&eth_cfg, &eth_handle);
if (ret != ESP_OK) {
    ESP_LOGE(TAG, "Failed to install USB RNDIS driver");
    return;
}
```

3. Start the *iot_eth* layer

```
ret = iot_eth_start(eth_handle);
if (ret != ESP_OK) {
    ESP_LOGE(TAG, "Failed to start eth driver");
    return;
}
```

Using Other Network Interfaces

1. Transmission: Call *iot_eth_transmit* to send data.
2. Reception: Configure the *stack_input* callback function in *iot_eth_config_t*. This callback is called when data is received.
3. Get the MAC address of the 4G module: Call *iot_eth_get_addr* to retrieve the MAC address.

Supported Network Interfaces

- USB interface-based RNDIS *USB RNDIS Host Driver*
- USB interface-based PPP *USB PPP*
- USB interface-based ECM *USB ECM Host Driver*

API Reference

Header File

- components/iot_eth/include/iot_eth.h

Functions

esp_err_t **iot_eth_install**(const *iot_eth_config_t* *config, *iot_eth_handle_t* *handle)

esp_err_t **iot_eth_uninstall**(*iot_eth_handle_t* handle)

esp_err_t **iot_eth_start**(*iot_eth_handle_t* handle)

esp_err_t **iot_eth_stop**(*iot_eth_handle_t* handle)

esp_err_t **iot_eth_transmit**(*iot_eth_handle_t* handle, void *data, size_t len)

esp_err_t **iot_eth_get_addr**(*iot_eth_handle_t* handle, uint8_t *mac)

esp_err_t **iot_eth_update_input_path**(*iot_eth_handle_t* handle, *static_input_cb_t* stack_input, void *user_data)

Structures

struct **iot_eth_config_t**

Ethernet configuration structure.

This structure holds the configuration for initializing an Ethernet instance.

Public Members

iot_eth_driver_t ***driver**

Pointer to the Ethernet driver

static_input_cb_t **stack_input**

Function pointer for stack input

Type Definitions

typedef void ***iot_eth_handle_t**

iot_eth event base declaration

Ethernet handle type

This is a handle to an Ethernet instance, used for managing Ethernet operations.

typedef esp_err_t (***static_input_cb_t**)(*iot_eth_handle_t* handle, uint8_t *data, size_t len, void *user_data)

Static input callback function type.

This type defines a function pointer for a static input callback. It takes an Ethernet handle, a pointer to data, the length of the data, and a user data pointer as arguments.

Enumerations

enum **iot_eth_event_t**

iot_eth event declarations

Values:

enumerator **IOT_ETH_EVENT_START**

IOT_ETH driver start

enumerator **IOT_ETH_EVENT_STOP**

IOT_ETH driver stop

enumerator **IOT_ETH_EVENT_CONNECTED**

IOT_ETH got a valid link

enumerator **IOT_ETH_EVENT_DISCONNECTED**

IOT_ETH lost a valid link

Chapter 17

Security & Encryption

17.1 Flash Encryption

Flash Encryption is an important security feature of ESP32 series chips, used to protect the confidentiality of firmware and data stored in external Flash. When Flash Encryption is enabled, the content in Flash is transparently encrypted, preventing plaintext firmware and data from being obtained through physical reading.

Key features of Flash Encryption:

- **Transparent encryption and decryption**: Automatic handling of encryption/decryption when accessed through MMU cache
- **Hardware implementation**: Encryption keys are stored in eFuse and cannot be directly accessed by software
- **XTS-AES algorithm**: Uses a secure encryption algorithm designed specifically for storage devices
- **Selective partition encryption**: Specific partition types can be designated for encryption

For more detailed information, refer to: ESP-IDF Flash Encryption Documentation

17.1.1 Flash Encryption and Secure Boot

Flash Encryption and Secure Boot are complementary security features:

- **Flash Encryption**: Protects data confidentiality, prevents reading Flash contents
- **Secure Boot**: Ensures code integrity, prevents running unauthorized firmware

It is recommended to enable both features in production environments for optimal security protection. For details, refer to: Flash Encryption and Secure Boot

17.1.2 Encryption Scope

Flash Encryption automatically encrypts the following content:

- Second-stage bootloader
- All app-type partitions
- Partition table
- NVS key partition
- Partitions marked as "encrypted" by the user

Important Note: NVS partition encryption and Flash Encryption are two independent features. NVS encryption feature should be used. See: NVS Encryption Documentation

Detailed explanation of encryption scope: Encrypted Partitions

17.1.3 Flash Encryption Feature Support List

The following is a comparison table of Flash Encryption features for ESP32 series chips, including encryption algorithms, key length/type, hardware support, key storage method, and key source (simplified description):

Table 1: ESP32 Series Flash Encryption Feature Comparison

Chip Model	Supported Encryption Algorithms	Key Length/Type	Hardware Acceleration/Support	Key Storage Method	Key Source
ESP32	AES-256 (custom implementation)	256-bit	Yes	eFuse	Hardware random number generation
ESP32-S2	XTS-AES-128, XTS-AES-256	256-bit or 512-bit	Yes	eFuse (BLOCK_KEYN)	Hardware random number or host generation
ESP32-S3	XTS-AES-128, XTS-AES-256	256-bit or 512-bit	Yes	eFuse (BLOCK_KEYN)	Hardware random number or host generation
ESP32-C2	XTS-AES-128	256-bit	Yes	eFuse (BLOCK_KEY0)	Hardware random number or host generation
ESP32-C3	XTS-AES-128	256-bit	Yes	eFuse (BLOCK_KEYN)	Hardware random number or host generation
ESP32-C5	XTS-AES-128	256-bit	Yes	eFuse (BLOCK_KEYN) or Key Manager	Hardware random number, host generation, or Key Manager
ESP32-C6	XTS-AES-128	256-bit	Yes	eFuse (BLOCK_KEYN)	Hardware random number or host generation
ESP32-H2	XTS-AES-128	256-bit	Yes	eFuse (BLOCK_KEYN)	Hardware random number or host generation
ESP32-P4	XTS-AES-128, XTS-AES-256	256-bit or 512-bit	Yes	eFuse (BLOCK_KEYN)	Hardware random number or host generation

Notes: - ESP32-C5 supports Key Manager as a key source and storage method ESP32-C5 SoC Capability Macros. - Other chips use eFuse storage, and keys can be generated by hardware random number or by the host and then burned ESP32-S2 Flash Encryption ESP32-C3 Flash Encryption.

17.1.4 Development Mode and Release Mode

Flash Encryption provides two modes:

Development Mode

- Allows multiple burnings of plaintext firmware
- Bootloader automatically handles encryption
- Suitable for development and testing phases
- Relatively lower security

Release Mode

- Disables plaintext firmware burning
- Can only update plaintext firmware through OTA
- Suitable for production environments
- Provides maximum security

Status Check:

- Use esp_flash_encryption_enabled() to check Flash Encryption status
- Use esp_get_flash_encryption_mode() to get encryption mode (Development Mode or Release Mode)

For configuration methods, see: Flash Encryption Settings

17.1.5 Encryption Process

The basic process of Flash Encryption:

1. **First Boot**: Detect eFuse status, initiate encryption process
2. **Key Generation**: Generate random encryption key and store it in eFuse (can be skipped, supports burning custom keys)
3. **In-Place Encryption**: Encrypt the content of specified partitions in Flash
4. **Set Flag**: Mark Flash Encryption as enabled
5. **Restart System**: Begin normal encrypted mode operation

For a complete explanation of the encryption process: The Flash Encryption Process

17.1.6 Application Access to Encrypted Partitions

Applications can transparently access encrypted Flash content:

Reading Encrypted Flash:

- Use esp_partition_read() or esp_flash_read_encrypted() to read plaintext content (automatically decrypted)
- Use esp_flash_read() to read raw encrypted data (no decryption)

Writing Encrypted Flash:

- Use esp_partition_write() to write plaintext content (automatically encrypted)
- Use esp_flash_write_encrypted() to write raw encrypted content (skips automatic encryption)

For more API details: Reading/Writing Data in Encrypted Flash

17.1.7 Reflashing Flash

In Development Mode:

- Use `idf.py encrypted-app-flash` to burn new application plaintext, which will be automatically encrypted during burning
- Use `idf.py encrypted-flash` to burn all partition plaintext, which will be automatically encrypted during burning

In Release Mode:

- Can only update plaintext firmware through OTA
- Can only manually burn ciphertext firmware (only supported when UART ROM Downloads is enabled)

For detailed information: Reflashing Updated Partitions

17.1.8 Disabling Encryption

Supported only in Development Mode (cannot be disabled in Release Mode): Disable Flash Encryption by burning eFuse `SPI_BOOT_CRYPT_CNT` while in encrypted state.

Warning: Each chip has a limited number of encryption toggle times, typically 3 times (off->on->off->on). Please operate with caution.

For detailed steps: Disabling Flash Encryption

17.1.9 Example Code

For complete Flash Encryption usage examples, please refer to:

- ESP-IDF Flash Encryption Example
- Security Features Comprehensive Example

These examples demonstrate:

- Flash Encryption status check
- Encrypted partition read/write operations
- NVS and FATFS usage in encrypted environments
- Configuration methods for Development Mode and Release Mode

17.1.10 Best Practices

1. **Use Release Mode for production environments**
2. **Use unique keys for each device**
3. **Use in conjunction with Secure Boot**
4. **Plan partition encryption strategy reasonably**
5. **Test OTA update process**

For more best practices: Flash Encryption Best Practices

17.1.11 Frequently Asked Questions (FAQ)

- Please refer to: ESP-FAQ Security Section

17.2 Secure Boot

Secure Boot is a critical security mechanism in ESP32 series chips that ensures devices only execute authorized firmware through digital signature verification during boot process and OTA updates. This mechanism establishes a complete chain of trust from bootloader to application, providing comprehensive code integrity protection for devices.

Key features of Secure Boot:

- **Code Integrity Protection**: Ensures firmware code has not been tampered with or replaced through cryptographic verification, preventing execution of unauthorized malicious programs
- **Complete Chain of Trust**: Establishes an end-to-end verification mechanism from bootloader to application, ensuring every step in the boot process is verified
- **Multiple Signature Algorithms**: Supports RSA-PSS or ECDSA as signature algorithms, with hardware acceleration for efficient verification
- **Hardware Security Storage**: Stores signature key digests in protected eFuse, providing strict hardware-level read/write protection mechanisms

For more detailed information, refer to: ESP-IDF Secure Boot v2 Documentation

17.2.1 Secure Boot and Flash Encryption

Secure Boot and Flash Encryption are complementary security features:

- **Secure Boot**: Ensures code integrity, prevents running unauthorized firmware
- **Flash Encryption**: Protects data confidentiality, prevents reading Flash code and data

It is recommended to enable both features in production environments for optimal security protection. For details, refer to: Secure Boot and Flash Encryption

17.2.2 Secure Boot Chain of Trust

Secure Boot verifies the following software components:

- First-stage bootloader (ROM code) as chip-embedded code that cannot be modified or compromised, serving as the foundation of the trust chain
- First-stage bootloader verifies the integrity and authenticity of the second-stage bootloader through digital signature
- Second-stage bootloader verifies the integrity and authenticity of the application binary through digital signature
- Application verifies the digital signature of new firmware during OTA update process, ensuring update security

Detailed verification process: Secure Boot v2 Process

17.2.3 Secure Boot Feature Support List

Table 2: Secure Boot Feature Support List

Chip Model	Supported Algorithms (Curve/Bits)	Signature Verification Time (CPU Frequency)	Maximum Number of Public Keys	Notes/Special Instructions
ESP32 (ECO V3+)	RSA-3072 (RSA-PSS, Secure Boot V2) AES (V1)	Not specifically stated	1	Only supports 1 public key, Secure Boot V2 requires ECO V3 or above
ESP32-S2	RSA-3072 (RSA-PSS)	Not specifically stated	3	Up to 3 public keys, revocable
ESP32-S3	RSA-3072 (RSA-PSS)	Not specifically stated	3	Up to 3 public keys, revocable
ESP32-C2	ECDSA-192 (NISTP192) ECDSA-256 (NISTP256, recommended)	Not specifically stated	1	Only supports ECDSA, and only 1 public key ESP32-C2 Kconfig Reference
ESP32-C3 (v0.3 and above)	RSA-3072 (RSA-PSS)	Not specifically stated	3	Up to 3 public keys, revocable
ESP32-C5	RSA-3072 ECDSA-256 (P-256) ECDSA-384 (P-384)	RSA-3072: 12.1 ms ECDSA-256: 5.6 ms ECDSA-384: 20.6 ms (48 MHz)	3	Only one algorithm can be selected ESP32-C5 Secure Boot v2
ESP32-C6	RSA-3072 ECDSA-256 (P-256)	RSA-3072: 10.2 ms ECDSA-256: 83.9 ms (40 MHz)	3	Only one algorithm can be selected ESP32-C6 Secure Boot v2
ESP32-H2	RSA-3072 ECDSA-256 (P-256)	RSA-3072: 18.3 ms ECDSA-256: 76.2 ms (32 MHz)	3	Only one algorithm can be selected ESP32-H2 Secure Boot v2
ESP32-P4	RSA-3072 ECDSA-256 (P-256)	RSA-3072: 14.8 ms ECDSA-256: 61.1 ms (40 MHz)	3	Only one algorithm can be selected ESP32-P4 Secure Boot v2

Notes: - "Maximum Number of Public Keys" refers to the number of public keys that can be stored in eFuse, some chips support key revocation. - "Signature Verification Time" refers to the time required for the ROM bootloader to verify the bootloader image signature, actual boot time may vary slightly. - For more detailed configuration and usage recommendations, please refer to the official documentation for the corresponding chip.

17.2.4 Signature and Key Management

In information security, digital signature is an integrity verification mechanism based on asymmetric encryption. Its core purpose is to: prove the authenticity of the data source (Authenticity), and ensure that data has not been tampered with during transmission or storage (Integrity).

The signature process uses a pair of keys:

- Private Key: Used to sign data, the private key must be kept strictly confidential, stored in a secure environment (local firmware development environment or remote signing server).
- Public Key: Used to verify the legitimacy of signatures, the public key can be safely distributed to any entity that needs to verify signatures.

Secure Boot v2 supports various asymmetric encryption algorithms, depending on the chip model:

- **RSA-PSS**: RSA-3072 key, suitable for scenarios requiring fast boot
- **ECDSA**: Supports P-256 and P-192 curves, shorter key length but longer verification time (limited by current ECDSA hardware acceleration capabilities)

17.2.5 Signature Block Format

The signature block is a critical structure in ESP32-S3 Secure Boot v2 used to verify image integrity and source trustworthiness. It is a fixed-format data block appended to the end of the bootloader or application image, containing signature information, public keys, and auxiliary verification fields.

- The signature block mainly contains the SHA-256 hash value of the image, public key, signature result, etc.
- The signature block itself occupies 1216 bytes, located at the 4KB-aligned boundary at the end of the image, and has CRC32 checksum
- Each image can have up to 3 signature blocks, used to support multi-key verification and revocation mechanisms

Detailed signature block definition: Signature Block Format

17.2.6 Enabling Hardware Secure Boot

Method 1: Enable via configuration items (menuconfig):

The simplest way to enable Secure Boot v2 in ESP-IDF is through `menuconfig` configuration items. When building the `bootloader`, the build system automatically inserts the signature process when compiling the `bootloader` and `app` images. During first boot, the bootloader automatically burns the eFuse to enable Secure Boot and revokes unused key slots. This method is suitable for the development phase, with automated and highly integrated processes, but since key revocation is irreversible, it should be used with caution in mass production.

1. Open the project configuration menu `Security features` and enable `Enable hardware Secure Boot in bootloader`
2. Select the `Secure Boot v2` signature scheme, depending on the chip, the signature algorithm options are `RSA` or `ECDSA`
3. Specify the signing key file path, the key file (e.g., RSA 3072) can be generated using the `idf.py secure-generate-signing-key` or `openssl genrsa` command
4. Using the `idf.py build` command will directly compile, align, pad, and generate the `app` image with attached signature blocks
5. Use the `idf.py bootloader` command to generate a bootloader image with Secure Boot enabled and signature blocks attached
6. For security reasons, by default, a bootloader with Secure Boot enabled needs to be burned separately using the `esptool.py write_flash` command, `idf.py flash` can only burn application and partition table partitions
7. Restart the device, the bootloader will automatically burn the eFuse to enable Secure Boot on first run, write the calculated public key digest of the signature block to eFuse, and revoke unused key slots
8. In subsequent boot processes, the public key digest in eFuse will be used to verify the signatures of bootloader and app images, ensuring the device only runs firmware signed with the corresponding private key

Notes: After enabling Secure Boot, the bootloader image size will increase and may require readjustment of partition table size. The private key must be safeguarded; if the private key is leaked, attackers can generate malicious firmware with valid signatures, and if the key is lost, the device cannot update firmware.

Method 2: Enable via external tools:

Another method is to manually configure eFuse before burning using the `espefuse.py` tool, including writing public key digest, setting key usage, enabling the `Secure Boot` flag, etc. This method does not trigger the bootloader's automatic key revocation logic, thus preserving unused key slots, facilitating future key rotation or OTA signature updates. It is suitable for mass production environments, with high security and fine control, but complex operations requiring strict key management and burning processes. Additionally, you can use remote signing or external HSM to generate signature blocks, further enhancing private key security.

1. Generate private key (can be generated locally or in a remote environment): Create a private key for signing firmware
2. Generate public key digest: Calculate the SHA-256 hash of the public key
3. Burn digest: Write the public key digest to specific eFuse area
4. Enable Secure Boot: Set related eFuse flag bits
5. Revoke unused key slots: Prevent unauthorized keys from being added
6. Burn security configuration: Set other security-related eFuse bits
7. Configure project: Disable automatic signing to use external signing tools
8. Sign images: Sign bootloader and app
9. Burn images: Write signed firmware to the device
10. Enable secure download: Finally lock all security configurations

For detailed configuration methods and instructions, see: Enable v2 Externally

17.2.7 Key Revocation

For chips with multiple public keys burned in, keys can be revoked to prevent exploiting leaked or unused keys. After revoking a key, firmware signed with that key will fail verification, thereby protecting the device from potential attacks.

Implementation Prerequisites:

Key revocation functionality requires the following prerequisites:

1. The chip must support multiple public keys (>1) and have key revocation capability
2. At least two public key digests are burned into eFuse at factory (e.g., key #0 and key #1)
3. The second-stage bootloader (bootloader) uses a multi-signature mechanism, signed with private keys corresponding to key #0 and key #1
4. The application (app) is signed using only one of the keys (e.g., key #0)

For more information on multiple signatures, please refer to: Multiple Keys

Implementation Steps:

1. OTA firmware signed with a new key (e.g., key #1) is written to the backup partition after verification by the current application
2. After the new application starts, it verifies the second-stage bootloader (bootloader) signature is normal (confirming bootloader key #1 is available)
3. Call esp_ota_revoke_secure_boot_public_key() to revoke the old key (key #0 revoked)
4. After the old key (key #0) is revoked, firmware signed with that key will fail verification
5. Both the second-stage bootloader (bootloader) and application (app) use the new key (key #1) for signature verification

Key revocation details: Key Revocation

17.2.8 Restrictions After Enabling

Security Restrictions:

- Once hardware Secure Boot is enabled, there is no way to disable it
- Updated bootloaders or applications must be signed with matching keys
- The USB-OTG USB stack is disabled, preventing updates via serial emulation or DFU
- Further eFuse read protection is disabled, preventing attackers from read-protecting the Secure Boot public key digest

Debug Interfaces:

- JTAG interface is disabled by default
- UART download mode switches to secure mode

Detailed restriction information: Restrictions After Secure Boot is Enabled

17.2.9 Software Signature Verification

Software signature verification provides a lightweight signature verification mechanism, using the same signature scheme as hardware Secure Boot, but only verifies new images during OTA updates. It is suitable for scenarios sensitive to boot speed or with lower physical security requirements, but does not provide complete boot chain protection.

Software Signature vs Hardware Secure Boot:

Table 3: Software Signature vs Hardware Secure Boot

Item	⍰ Secure Boot v2 (Hardware)	⍰ Software Signature Verification (No Secure Boot)
Boot-time Verification	✓ Verifies bootloader and app image signatures	✗ Does not verify current app, assumes it is trusted
OTA Update Verification	✓ Uses keys in eFuse to verify new app signature	✓ Uses public key in current app' s signature block to verify new app
Root of Trust	Public key digest burned in eFuse	Public key in the signature block of current running app
eFuse Configuration Requirements	Must burn SECURE_BOOT_EN and KEY_DIGESTx	No need to burn Secure Boot related eFuse
Key Revocation Mechanism	✓ Supports KEY_REVOKE_x and aggressive revocation policy	✗ Does not support key revocation
Signature Block Support Count	✓ Up to 3 signature blocks, supports multi-key verification	✗ Only uses the first signature block, ignores others
Tamper Resistance	✓ Prevents Flash replacement or malicious code injection	✗ Cannot prevent physical attacks or Flash replacement
Boot Performance	✗ Has signature verification overhead during boot	✓ Faster boot, no verification delay
Development Convenience	✗ Irreversible after enabling eFuse, requires careful operation	✓ No irreversible operations, suitable for development and debugging
Applicable Scenarios	Mass production deployment, high-security requirements devices	Development testing, boot speed sensitive, physically controlled environments
Key Management Flexibility	✓ Supports multiple key rotation and revocation	✗ Only relies on public key of current app, cannot rotate
Recommendation	✓ Officially recommended for formal products	⚠ Use only with clear threat model, needs careful evaluation

Configuration Method:

Enable the `CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT` option.

For details, refer to: Signed App Verification Without Hardware Secure Boot

17.2.10 Example Code

For complete Secure Boot usage examples, please refer to:

- ESP-IDF Secure Boot Example
- Security Features Comprehensive Example

These examples demonstrate:

- Secure Boot status check
- Signature key generation and management
- Multi-key signing and revocation
- Use in conjunction with Flash Encryption

17.2.11 Best Practices

1. **Use high-quality entropy sources to generate signing keys**
2. **Always keep signing keys private**
3. **Avoid third-party observation of key generation or signing process**
4. **Enable all Secure Boot configuration options**
5. **Use in conjunction with Flash Encryption**
6. **Use multiple keys to reduce single point of failure**
7. **Establish key rotation policy**

For more best practices: Secure Boot Best Practices

17.2.12 Frequently Asked Questions (FAQ)

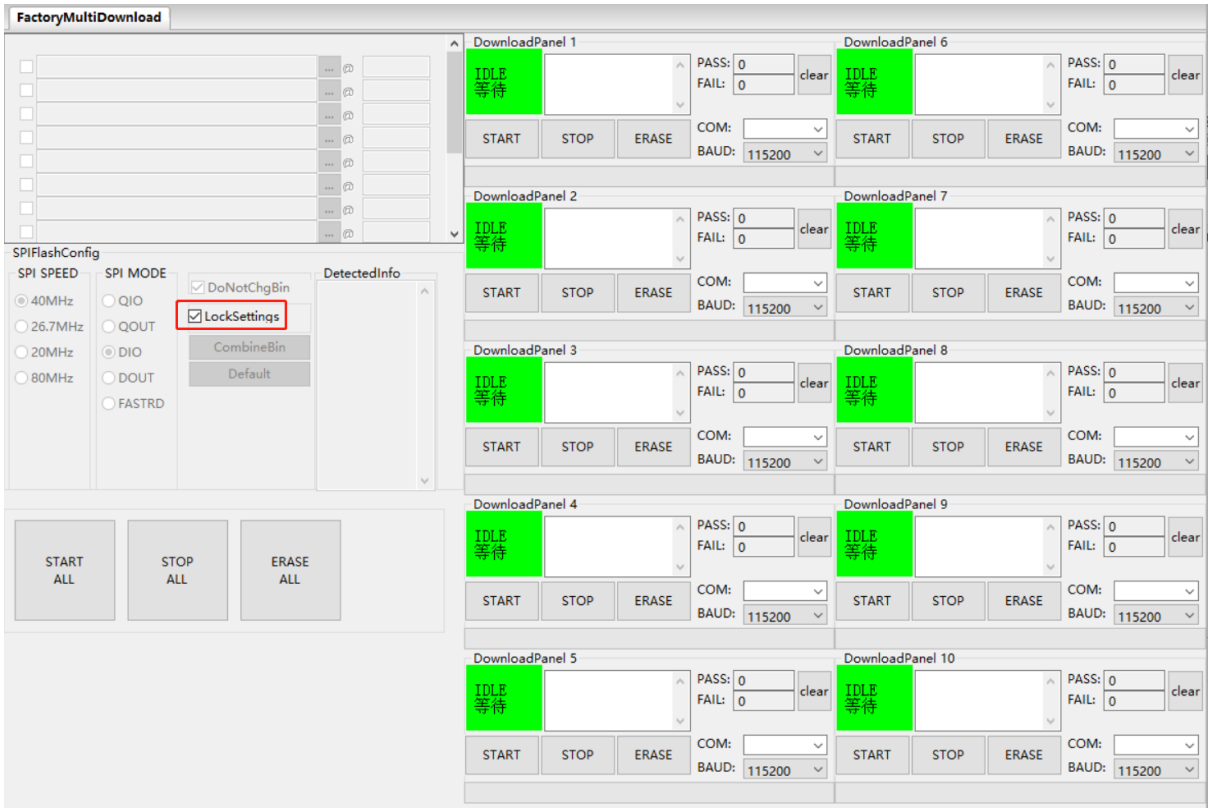
- Please refer to: ESP-FAQ Security Section

17.3 Mass Production

This document compiles various tools for mass production of Espressif chips, including flashing, testing, fixture specifications, network configuration, and RF tools.

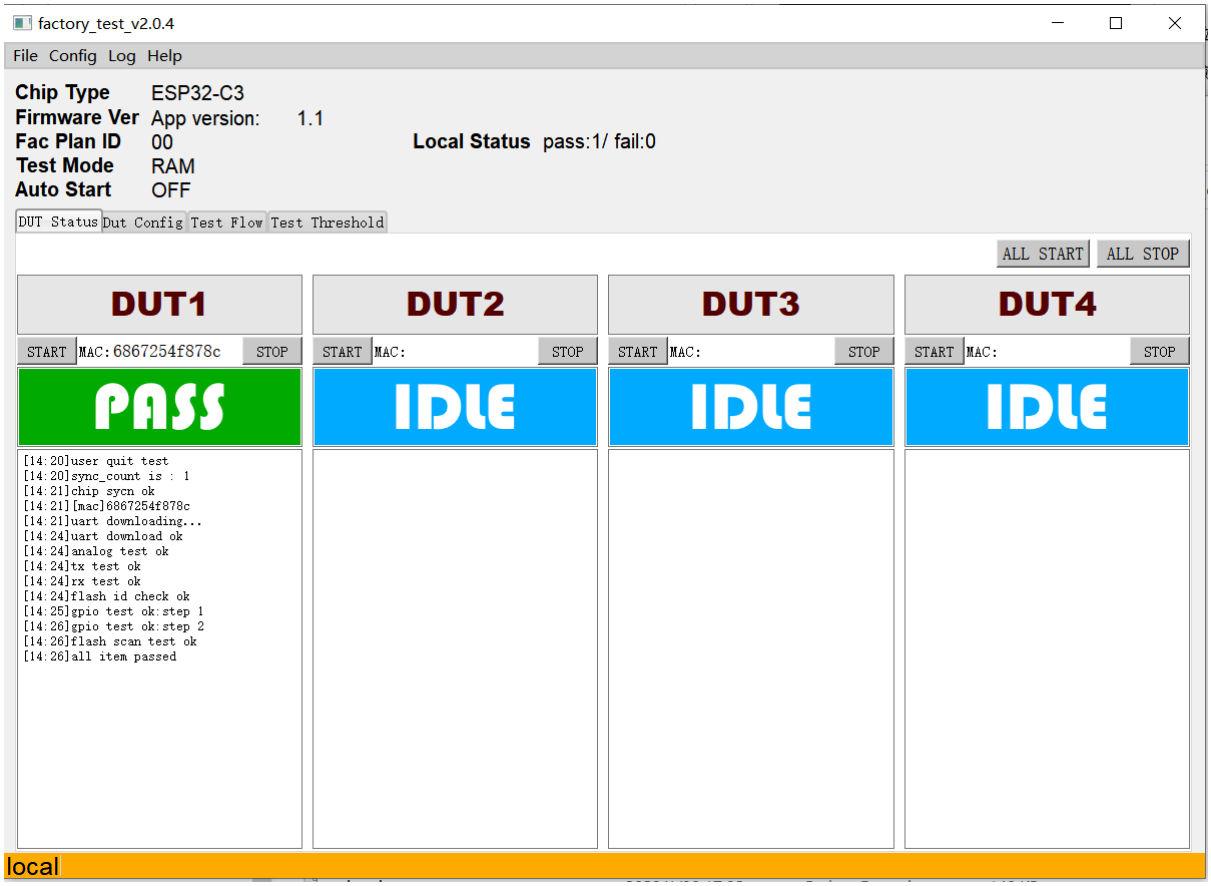
17.3.1 Flash Download Tool

This tool is designed for production environments to efficiently flash production firmware to chip Flash, supporting parallel flashing of multiple chips and various flashing parameter configurations. It solves problems such as firmware updates, batch flashing, and automated testing in mass production, improving production efficiency and reducing human error. It is applicable for production line firmware deployment, batch upgrades, and debugging processes. Refer to ESP32 Production Stage.



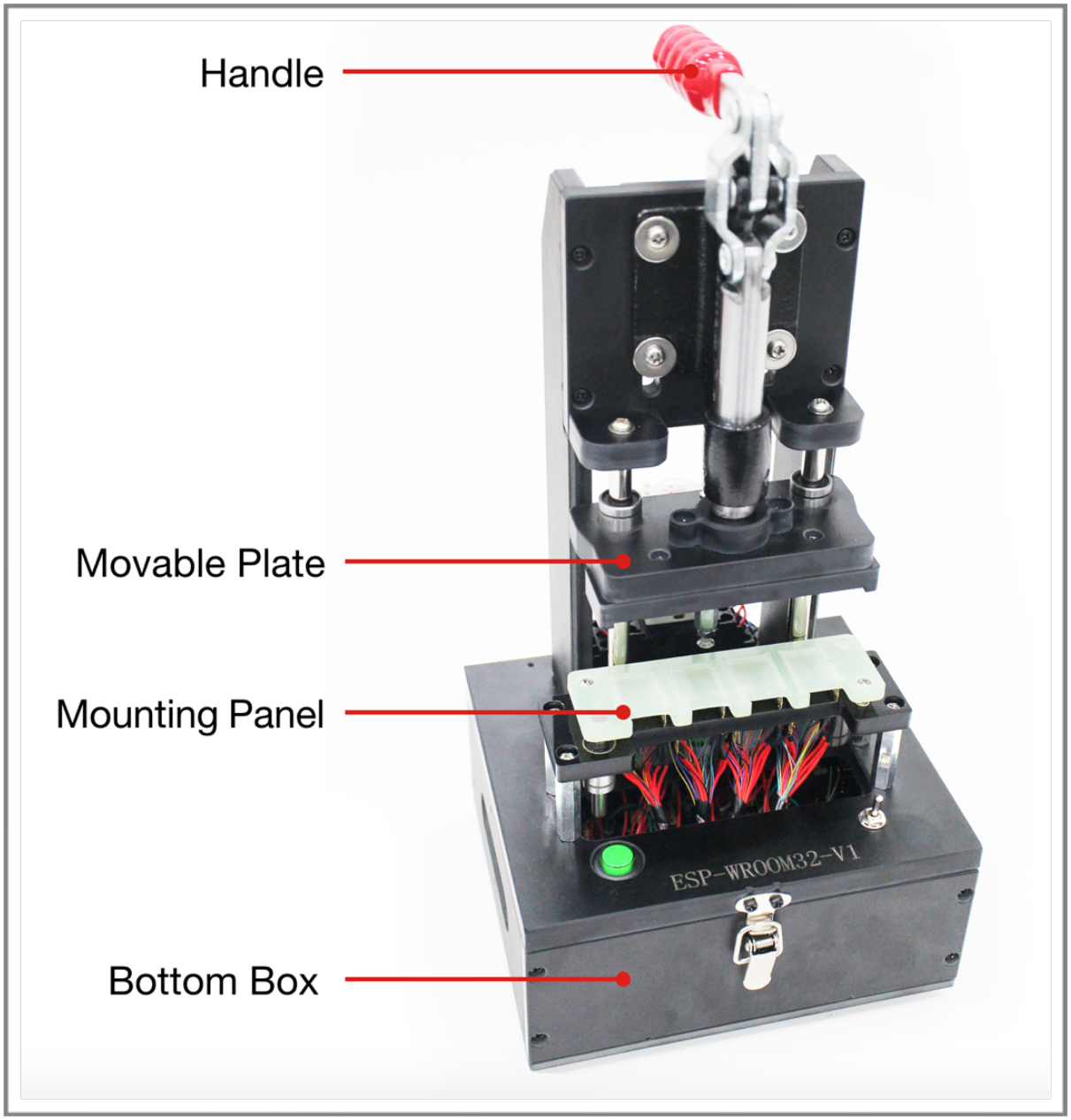
17.3.2 Espressif Production Test Tool

This tool is a one-to-four production line testing host that uses "RAM run test" method, supporting automated configuration and execution of test procedures. Features include RF performance testing, GPIO connectivity testing, firmware version verification, Flash read/write/ID testing, etc. It automatically generates test logs and statistical reports, and supports test bin, threshold and system configuration management. It solves problems such as low production line testing efficiency, cumbersome manual operations, and difficult data statistics, suitable for large-scale production testing and production line quality control. Refer to ESP32 Production Testing Guide - Production Testing Tool.



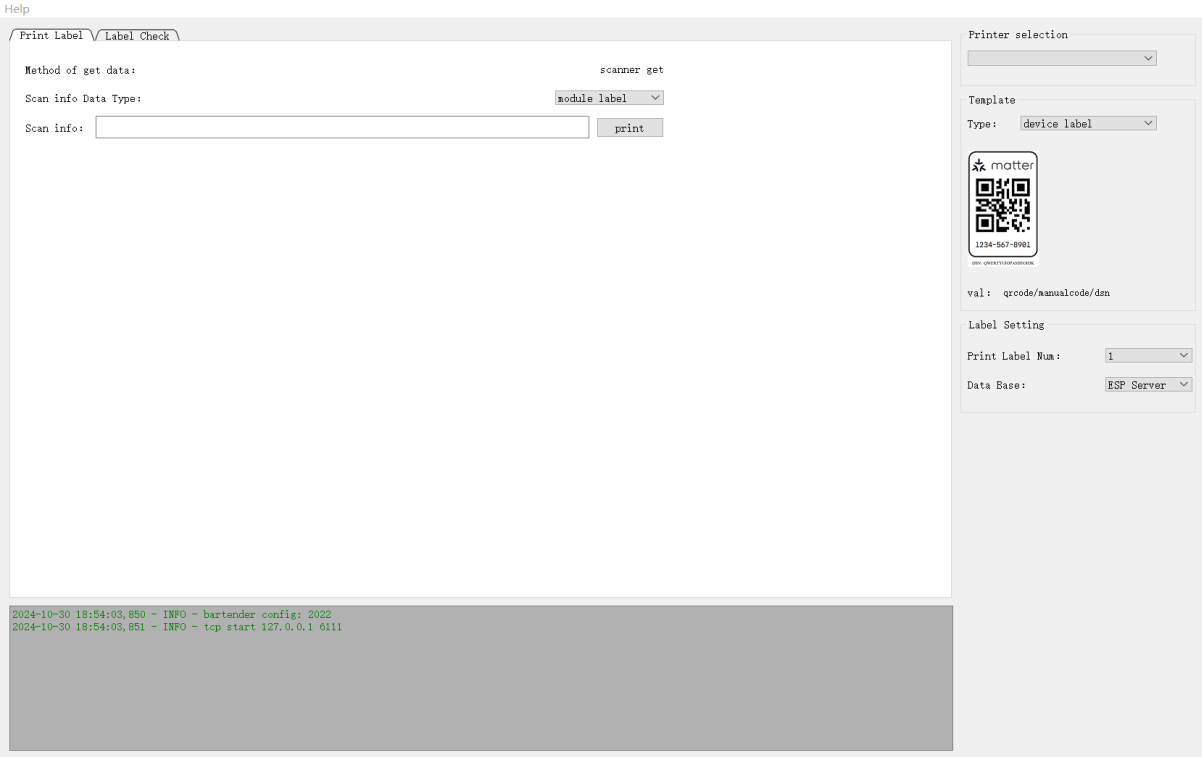
17.3.3 Module Fixture Design Specifications

This specification guides the structural design and manufacturing process of production test fixtures, ensuring consistency and reliability of test fixtures during mass production. By standardizing fixture design, it reduces test errors and defect rates caused by fixture differences, improving test efficiency and product quality. It is applicable for production test fixture development and production line fixture selection. Refer to ESP32 Production Stage.



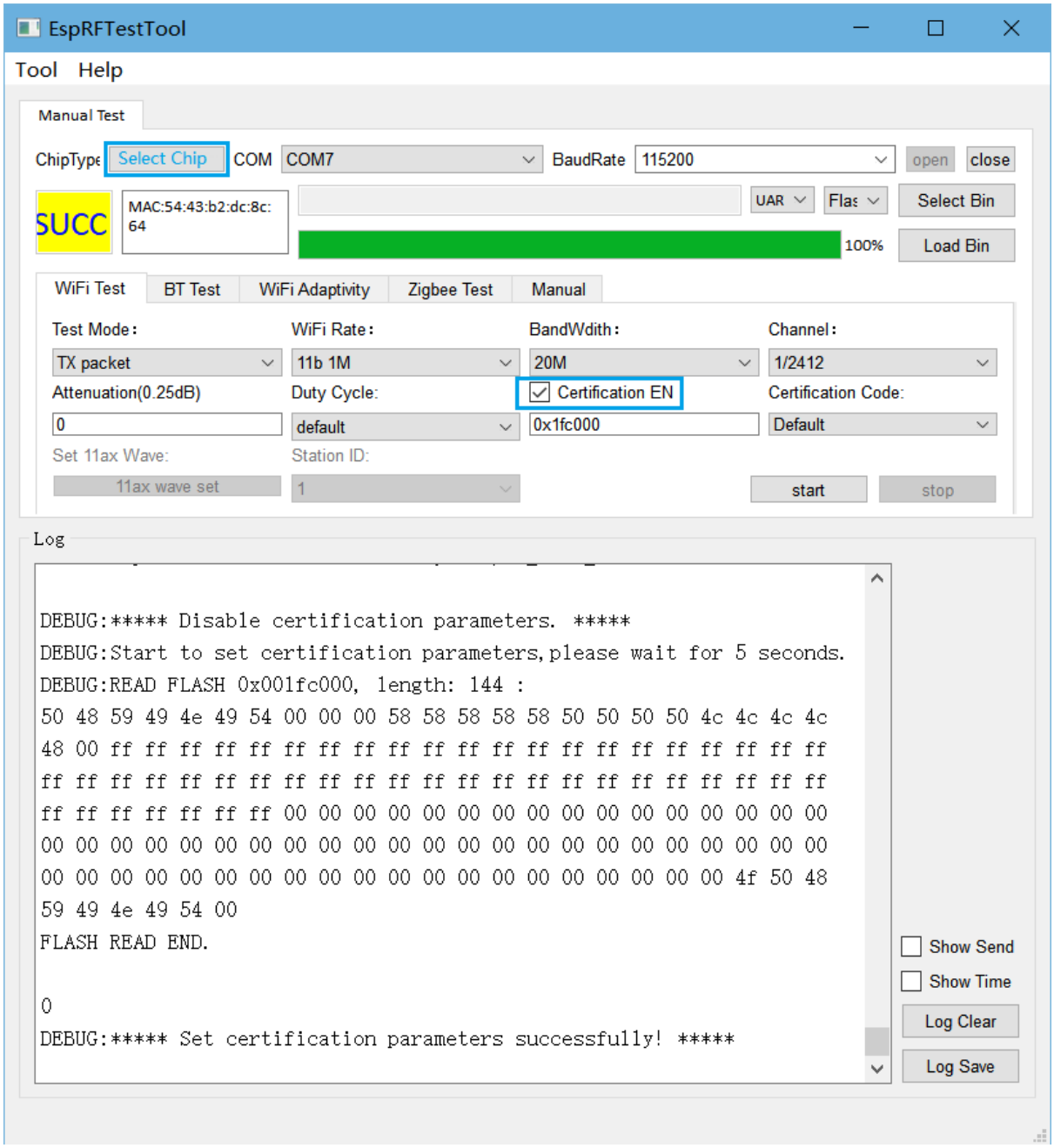
17.3.4 Matter QR Code Generation Tool

This tool is used to generate network configuration QR codes for Matter devices, supporting batch generation and custom parameter configuration. It solves problems such as rapid network connection of production line devices, automated network configuration, and batch verification, improving network connection efficiency and consistency of mass-produced devices. It is applicable for Matter device production, production line network configuration, and network connection verification scenarios. Refer to ESP32 Production Stage.



17.3.5 EspRFTestTool Package

- EspRFTestTool: A host computer tool for RF testing, supporting Wi-Fi non-signaling testing, Bluetooth/BLE non-signaling testing, and Wi-Fi adaptive testing. This tool integrates serial port configuration, fast firmware flashing, test parameter configuration, and status log windows, helping developers quickly validate key RF indicators during chip development stage, discover and locate RF performance issues, and improve product reliability. It is suitable for early chip debugging, RF performance evaluation, and pre-certification verification. See download link at EspRFTestTool Package.
- DownloadTool: Used to flash test firmware to chips, supporting both RAM and Flash flashing modes, and allowing specification of flashing address. This tool primarily solves the problem of quick firmware deployment before RF testing, ensuring test environment consistency and reducing manual operation errors. It is suitable for RF test preparation, firmware version switching, and batch flashing requirements. Refer to ESP32 Tool Package.
- PowerLimitTool: Used to generate customized phy_init_data firmware, supporting flexible configuration of RF power limit parameters. This tool helps users meet RF regulatory requirements in different countries/regions, improving product compliance, and can be used with RF testing and certification processes. It is suitable for certification preparation and power adjustment for special application scenarios. Refer to ESP32 Tool Package.


EspRFTestTool
Tool Help
Manual Test
ChipType Select Chip
COM COM7
BaudRate 115200
open close
SUCC
MAC:54:43:b2:dc:8c:64
UAR Flas
Select Bin
100%
Load Bin
WiFi Test BT Test WiFi Adaptivity Zigbee Test Manual
Test Mode:
TX packet
WiFi Rate:
11b 1M
BandWdith:
20M
Channel:
1/2412
Attenuation(0.25dB)
0
Duty Cycle:
default
Certification EN
0x1fc000
Certification Code:
Default
Set 11ax Wave:
11ax wave set
Station ID:
1
start
stop
Log
DEBUG:***** Disable certification parameters. *****
DEBUG:Start to set certification parameters,please wait for 5 seconds.
DEBUG:READ FLASH 0x001fc000, length: 144 :
50 48 59 49 4e 49 54 00 00 00 58 58 58 58 58 50 50 50 50 4c 4c 4c 4c
48 00 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 4f 50 48
59 49 4e 49 54 00
FLASH READ END.
0
DEBUG:***** Set certification parameters successfully! *****
Show Send
Show Time
Log Clear
Log Save

Chapter 18

Electrical&Lighting Solution

18.1 Lightbulb Driver

The Lightbulb driver component encapsulates several commonly used dimming schemes for bulb lights, using an abstraction layer to manage these schemes, making it easier for developers to integrate into their applications. Currently, all ESP32 series chips are fully supported

18.1.1 Dimming scheme

PWM Dimming scheme

PWM dimming is a technique that controls LED brightness by adjusting the pulse width. The core idea is to vary the duty cycle of the current pulse (i.e., the proportion of high-level time within the entire cycle time). When the duty cycle is high, the LED receives current for a longer duration, resulting in higher brightness; conversely, a low duty cycle shortens the current duration, producing lower brightness.

All ESP chips support PWM output using hardware LEDC driver or MCPWM driver. It is recommended to use the LEDC driver as it supports hardware fading, with configurable frequency and duty cycle, and a maximum resolution of 20 bits. The following types of PWM dimming drivers are currently supported:

- RGB + C/W
- RGB + CCT/Brightnes

PWM scheme use cases Use the PWM scheme to light up a 3-channel RGB bulb with PWM frequency set to 4000Hz. Use software color mixing, enable the fade function, and set the power-on color to red.

```
lightbulb_config_t config = {
    // 1. Select PWM output and configure parameters
    .type = DRIVER_ESP_PWM,
    .driver_conf.pwm.freq_hz = 4000,

    // 2. capability selection: enable/disable according to your needs
    .capability.enable_fade = true,
    .capability.fade_time_ms = 800,
    .capability.enable_lowpower = false,
    /* Enable this feature if your white light output is hardware-controlled␣
↪independently rather than software color-mixed */
```

(continues on next page)

(continued from previous page)

```
    .capability.enable_hardware_cct = true,
    .capability.enable_status_storage = true,
    /* Configure the LED combination based on your LED aluminum substrate */
    .capability.led_beads = LED_BEADS_3CH_RGB,
    .capability.storage_cb = NULL,
    .capability.sync_change_brightness_value = true,

    // 3. Configure hardware pins for PWM output
    .io_conf.pwm_io.red = 25,
    .io_conf.pwm_io.green = 26,
    .io_conf.pwm_io.blue = 27,

    // 4. Limits parameter
    .external_limit = NULL,

    // 5. Color calibration parameters
    .gamma_conf = NULL,

    // 6. Initialize lighting parameters; if 'on' is set, the lightbulb will light␣
↪up during driver initialization
    .init_status.mode = WORK_COLOR,
    .init_status.on = true,
    .init_status.hue = 0,
    .init_status.saturation = 100,
    .init_status.value = 100,
};
lightbulb_init(&config);
```

I2C Dimming Scheme

I2C Dimming : Control commands are sent via the I2C bus to the dimming chip, adjusting the LED brightness by changing the output current of the dimming chip. The I2C bus consists of two lines: the data line (SDA) and the clock line (SCL). All ESP chips support dimming chips using hardware I2C communication. The following dimming chips are currently supported:

- SM2135EH
- SM2X35EGH (SM2235EGH/SM2335EGH)
- BP57x8D (BP5758/BP5758D/BP5768)
- BP1658CJ
- KP18058

I2C Dimming Use Case Use the I2C driver to control the BP5758D chip to light up a 5-channel RGBCW bulb. Set the I2C communication rate to 300KHz, with RGB LEDs driven at 50mA and CW LEDs at 30mA. Use software color mixing, enable the fade function, and set the power-on color to red.

```
lightbulb_config_t config = {
    // 1. Select the required chip and configure parameters. Each chip has␣
↪different configuration parameters, so please refer carefully to the chip manual.
    .type = DRIVER_BP57x8D,
    .driver_conf.bp57x8d.freq_khz = 300,
    .driver_conf.bp57x8d.enable_iic_queue = true,
    .driver_conf.bp57x8d.iic_clk = 4,
    .driver_conf.bp57x8d.iic_sda = 5,
    .driver_conf.bp57x8d.current = {50, 50, 50, 30, 30},

    // 2. capability selection: enable/disable according to your needs
    .capability.enable_fade = true,
    .capability.fade_time_ms = 800,
```

(continues on next page)

(continued from previous page)

```
    .capability.enable_lowpower = false,

    .capability.enable_status_storage = true,
    .capability.led_beads = LED_BEADS_5CH_RGBCW,
    .capability.storage_cb = NULL,
    .capability.sync_change_brightness_value = true,

    // 3. Configure the hardware pins for the IIC chip
    .io_conf.iic_io.red = OUT3,
    .io_conf.iic_io.green = OUT2,
    .io_conf.iic_io.blue = OUT1,
    .io_conf.iic_io.cold_white = OUT5,
    .io_conf.iic_io.warm_yellow = OUT4,

    // 4. Limits parameter
    .external_limit = NULL,

    // 5. Color calibration parameters
    .gamma_conf = NULL,

    // 6. Initialize lighting parameters; if 'on' is set, the lightbulb will light␣
↪up during driver initialization
    .init_status.mode = WORK_COLOR,
    .init_status.on = true,
    .init_status.hue = 0,
    .init_status.saturation = 100,
    .init_status.value = 100,
};
lightbulb_init(&config);
```

Single-wire dimming scheme

A single-wire dimming scheme is a method of controlling LED brightness through a single communication line. The core idea is to adjust LED brightness by sending control signals through a specific communication protocol. This scheme can be implemented on ESP chips using the RMT peripheral or SPI peripheral. SPI is recommended for controlling LED communication. The following LED types are currently supported:

- WS2812

WS2812 Use Case Use the SPI driver to light up 10 WS2812 LEDs, enable the fade function, and set the power-on color to red.

```
lightbulb_config_t config = {
    // 1. Select WS2812 output and configure parameters
    .type = DRIVER_WS2812,
    .driver_conf.ws2812.led_num = 10,
    .driver_conf.ws2812.ctrl_io = 4,

    // 2. capability selection: enable/disable according to your needs
    .capability.enable_fade = true,
    .capability.fade_time_ms = 800,
    .capability.enable_status_storage = true,

    /* For WS2812, only LED_BEADS_3CH_RGB can be selected */
    .capability.led_beads = LED_BEADS_3CH_RGB,
    .capability.storage_cb = NULL,

    // 4. Limits parameter
    .external_limit = NULL,
```

(continues on next page)

(continued from previous page)

```
    // 5. Color calibration parameters
    .gamma_conf = NULL,

    // 6. Initialize lighting parameters; if 'on' is set, the lightbulb will light␣
↪up during driver initialization
    .init_status.mode = WORK_COLOR,
    .init_status.on = true,
    .init_status.hue = 0,
    .init_status.saturation = 100,
    .init_status.value = 100,
};
lightbulb_init(&config);
```

18.1.2 Fade principle

The fade effect in the bulb light component is implemented through software. Each channel records the current output value, target value, step size, and remaining steps. When the API is used to set a color, it updates the target value, step size, step count, and enables a fade timer. This timer triggers a callback every 12ms, where each channel' s steps are checked. As long as there are steps remaining, the current value is adjusted by the step size and updated in the underlying driver. When all channels have reached zero steps, the fade is complete, and the timer is stopped.

18.1.3 Low-power implementation process

To pass certifications like T20 for low power consumption, after optimizing the light board' s power supply, some low-power configurations are needed in the software. Apart from the settings mentioned in the Low-Power Mode Usage Guide the driver logic also requires adjustments. In the lightbulb component, low-power adjustments have been added for both PWM and I2C dimming schemes. The specific logic involves the I2C scheme using the dimming chip' s low-power command to exit or enter low power when switching the light on or off. For the PWM scheme, ESP32 requires power lock management with the APB clock source to prevent flickering—locking power and disabling dynamic frequency scaling when the light is on, and releasing the lock when off, other chips use the XTAL clock source, so no further measures are necessary.

18.1.4 Color calibration scheme

CCT Mode

The calibration for color temperature mode requires configuring the following structure.

```
union {
    struct {
        uint16_t kelvin_min;
        uint16_t kelvin_max;
    } standard;
    struct {
        lightbulb_cct_mapping_data_t *table;
        int table_size;
    } precise;
} cct_mix_mode;
```

- Standard mode: calibrate the maximum and minimum Kelvin values, use linear interpolation to fill in intermediate values, and then adjust the output ratio of cool and warm LEDs according to the target color temperature.
- Precision mode: calibrate the required output ratios of red, green, blue, cool, and warm LEDs at different color temperatures. Use these calibration points to directly output the corresponding ratios. The more calibration points, the more accurate the color temperature.

Color Mode

The calibration for color mode requires configuring the following structure.

```
union {
    struct {
        lightbulb_color_mapping_data_t *table;
        int table_size;
    } precise;
} color_mix_mode;
```

- Standard Mode: No parameter configuration is required. Internal theoretical algorithms will convert HSV, XYY, and other color models into RGB ratios, and LEDs will light up directly based on this ratio.
- Precision Mode: Calibrate colors using the HSV model. Measure the required output ratios of red, green, blue, cool, and warm LEDs for various hues and saturation levels as calibration points. Use linear interpolation to fill in intermediate values and light up the LEDs based on these calibrated ratios.

18.1.5 Power limiting scheme

Power limiting is used to balance and fine-tune the output current of a specific channel or the overall system to meet power requirements.

To limit the overall power, configure the following structure.

```
typedef struct {
    /* Scale the incoming value
     * range: 10% <= value <= 100%
     * step: 1%
     * default min: 10%
     * default max: 100%
     */
    uint8_t white_max_brightness; /**< Maximum brightness limit for white light␣
↪output. */
    uint8_t white_min_brightness; /**< Minimum brightness limit for white light␣
↪output. */
    uint8_t color_max_value;      /**< Maximum value limit for color light output.␣
↪*/
    uint8_t color_min_value;      /**< Minimum value limit for color light output.␣
↪*/

    /* Dynamically adjust the final power
     * range: 100% <= value <= 500%
     * step: 10%
     */
    uint16_t white_max_power;     /**< Maximum power limit for white light output.␣
↪*/
    uint16_t color_max_power;     /**< Maximum power limit for color light output.␣
↪*/
} lightbulb_power_limit_t;
```

- `white_max_brightness` and `white_min_brightness` are used in color temperature mode to constrain the `brightness` parameter set by the color temperature API within these maximum and minimum values.
- `color_max_value` and `color_min_value` are used in color mode to constrain the `value` parameter set by the color API within these maximum and minimum values.
- `white_max_power` is used to limit power in color temperature mode. The default value is 100, meaning the maximum output power is half of the full power; if set to 200, it can reach the maximum power of the cool and warm LEDs.
- `color_max_power` is used to limit power in color mode. The default value is 100, meaning the maximum output power is one-third of full power; if set to 300, it can reach the maximum power of the red, green, and blue LEDs.

To limit the power of individual LEDs, configure the following structure:

```
typedef struct {
    float balance_coefficient[5]; /**< Array of float coefficients for adjusting
↪the intensity of each color channel (R, G, B, C, W).
                                  These coefficients help in achieving the
↪desired color balance for the light output. */

    float curve_coefficient;      /**< Coefficient for gamma correction. This
↪value is used to modify the luminance levels
                                  to suit the non-linear characteristics of
↪human vision, thus improving the overall
                                  visual appearance of the light. */
} lightbulb_gamma_config_t;
```

- `balance_coefficient` is used to fine-tune the output current of each LED. The final output of the driver will be reduced according to this ratio, with a default value of 1.0, meaning no reduction.
- `curve_coefficient` is used to convert linear changes during fading into curve-based changes.

Note: Modifying `balance_coefficient` will affect the accuracy of color calibration, so this parameter should be adjusted before performing color calibration. This parameter is especially useful for I2C dimming chips that output current in multiples of 5 or 10; if specific currents are needed, this parameter can be used for adjustment.

18.1.6 API Reference

Header File

- components/led/lightbulb_driver/include/lightbulb.h

Functions

esp_err_t **lightbulb_init** (*lightbulb_config_t* *config)

Initialize the lightbulb.

Parameters **config** –Pointer to the configuration parameters for the lightbulb.
Returns esp_err_t

esp_err_t **lightbulb_deinit** (void)

Deinitialize the lightbulb and release resources.

Returns esp_err_t

esp_err_t **lightbulb_set_fade_time** (uint32_t fade_time_ms)

Set lightbulb fade time.

Parameters **fade_time_ms** –Fade time in milliseconds (ms). Range: 100ms - 3000ms.
Returns esp_err_t

esp_err_t **lightbulb_set_fades_function** (bool is_enable)

Enable/Disable the lightbulb fade function.

Parameters **is_enable** –A boolean flag indicating whether to enable (true) or disable (false) the fade function.
Returns esp_err_t

esp_err_t **lightbulb_set_storage_function** (bool is_enable)

Enable/Disable the lightbulb storage function.

Parameters **is_enable** –A boolean flag indicating whether to enable (true) or disable (false) the storage function.

Returns esp_err_t

esp_err_t **lightbulb_update_status** (*lightbulb_status_t* *new_status, bool trigger)

Re-update the lightbulb status.

Parameters
- **new_status** –Pointer to the new status to be applied to the lightbulb.
- **trigger** –A boolean flag indicating whether the update should be triggered immediately.

Returns esp_err_t

bool **lightbulb_get_fades_function_status** (void)

Get lightbulb fade function enabled status.

Returns true if the fade function is enabled.
Returns false if the fade function is disabled.

esp_err_t **lightbulb_hsv2rgb** (uint16_t hue, uint8_t saturation, uint8_t value, uint8_t *red, uint8_t *green, uint8_t *blue)

Convert HSV model to RGB model.

Note: RGB model color depth is 8 bit (0-255).

Parameters
- **hue** –Hue value in the range of 0-360 degrees.
- **saturation** –Saturation value in the range of 0-100.
- **value** –Value (brightness) value in the range of 0-100.
- **red** –Pointer to a variable to store the resulting Red component (0-255).
- **green** –Pointer to a variable to store the resulting Green component (0-255).
- **blue** –Pointer to a variable to store the resulting Blue component (0-255).

Returns esp_err_t

esp_err_t **lightbulb_rgb2hsv** (uint16_t red, uint16_t green, uint16_t blue, uint16_t *hue, uint8_t *saturation, uint8_t *value)

Convert RGB model to HSV model.

Note: RGB model color depth is 8 bit (0-255).

Parameters
- **red** –Red component value in the range of 0-255.
- **green** –Green component value in the range of 0-255.
- **blue** –Blue component value in the range of 0-255.
- **hue** –Pointer to a variable to store the resulting Hue value (0-360 degrees).
- **saturation** –Pointer to a variable to store the resulting Saturation value (0-100).
- **value** –Pointer to a variable to store the resulting Value (brightness) value (0-100).

Returns esp_err_t

esp_err_t **lightbulb_xyy2rgb** (float x, float y, float Y, uint8_t *red, uint8_t *green, uint8_t *blue)

Convert xyY model to RGB model.

Note: Refer: https://www.easyrgb.com/en/convert.php#inputFORM https://www.easyrgb.com/en/math.php

Parameters
- **x** –x coordinate value in the range of 0 to 1.0.
- **y** –y coordinate value in the range of 0 to 1.0.

- **Y** –Y (luminance) value in the range of 0 to 100.0.
- **red** –Pointer to a variable to store the resulting Red component (0-255).
- **green** –Pointer to a variable to store the resulting Green component (0-255).
- **blue** –Pointer to a variable to store the resulting Blue component (0-255).

Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_rgb2xyy**(uint8_t red, uint8_t green, uint8_t blue, float *x, float *y, float *Y)

Convert RGB model to xyY model.

Parameters
- **red** –Red component value in the range of 0 to 255.
- **green** –Green component value in the range of 0 to 255.
- **blue** –Blue component value in the range of 0 to 255.
- **x** –Pointer to a variable to store the resulting x coordinate (0-1.0).
- **y** –Pointer to a variable to store the resulting y coordinate (0-1.0).
- **Y** –Pointer to a variable to store the resulting Y (luminance) value (0-100.0).

Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_kelvin2percentage**(uint16_t kelvin, uint8_t *percentage)

Convert CCT (Color Temperature) kelvin to percentage.

Parameters
- **kelvin** –Default range: 2200k - 7000k (Color Temperature in kelvin).
- **percentage** –Pointer to a variable to store the resulting percentage (0 - 100).

Returns esp_err_t

esp_err_t **lightbulb_percentage2kelvin**(uint8_t percentage, uint16_t *kelvin)

Convert percentage to CCT (Color Temperature) kelvin.

Attention

Parameters
- **percentage** –Percentage value in the range of 0 to 100.
- **kelvin** –Pointer to a variable to store the resulting Color Temperature in kelvin. Default range: 2200k - 7000k.

Returns esp_err_t

esp_err_t **lightbulb_set_hue**(uint16_t hue)

Set the hue value.

Parameters **hue** –Hue value in the range of 0-360 degrees.
Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_set_saturation**(uint8_t saturation)

Set the saturation value.

Parameters **saturation** –Saturation value in the range of 0-100.
Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_set_value**(uint8_t value)

Set the value (brightness) of the lightbulb.

Parameters **value** –Brightness value in the range of 0-100.
Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_set_cct**(uint16_t cct)

Set the color temperature (CCT) of the lightbulb.

Note: Supports using either percentage or Kelvin values.

Parameters **cct** –CCT value in the range of 0-100 or 2200-7000.
Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_set_brightness**(uint8_t brightness)

Set the brightness of the lightbulb.

Parameters **brightness** –Brightness value in the range of 0-100.
Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_set_xyy**(float x, float y, float Y)

Set the xyY color model for the lightbulb.

Attention The xyY color model cannot fully correspond to the HSV color model, so the color may be biased. The grayscale will be recalculated in lightbulb, so we cannot directly operate the underlying driver through the xyY interface.

Parameters
- **x** –x-coordinate value in the range of 0 to 1.0.
- **y** –y-coordinate value in the range of 0 to 1.0.
- **Y** –Y-coordinate (luminance) value in the range of 0 to 100.0.

Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_set_hsv**(uint16_t hue, uint8_t saturation, uint8_t value)

Set the HSV (Hue, Saturation, Value) color model for the lightbulb.

Parameters
- **hue** –Hue value in the range of 0 to 360 degrees.
- **saturation** –Saturation value in the range of 0 to 100.
- **value** –Value (brightness) value in the range of 0 to 100.

Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_set_cctb**(uint16_t cct, uint8_t brightness)

Set the color temperature (CCT) and brightness of the lightbulb.

Note: Supports using either percentage or Kelvin values.

Parameters
- **cct** –CCT value in the range of 0-100 or 2200-7000.
- **brightness** –Brightness value in the range of 0-100.

Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_set_switch**(bool status)

Set the on/off status of the lightbulb.

Parameters **status** –On/off status (true for on, false for off).
Returns esp_err_t An error code indicating the success or failure of the operation.

int16_t **lightbulb_get_hue**(void)

Get the hue value of the lightbulb.

Returns int16_t The hue value in the range of 0 to 360 degrees.

int8_t **lightbulb_get_saturation**(void)

Get the saturation value of the lightbulb.

Returns int8_t The saturation value in the range of 0 to 100.

int8_t **lightbulb_get_value** (void)

Get the value (brightness) of the lightbulb.

Returns int8_t The brightness value in the range of 0 to 100.

int8_t **lightbulb_get_cct_percentage** (void)

Get the color temperature (CCT) percentage of the lightbulb.

Returns int8_t The CCT percentage value in the range of 0 to 100.

int16_t **lightbulb_get_cct_kelvin** (void)

Get the color temperature (CCT) Kelvin value of the lightbulb.

Returns int16_t The CCT Kelvin value in the range of 2200 to 7000.

int8_t **lightbulb_get_brightness** (void)

Get the brightness value of the lightbulb.

Returns int8_t The brightness value in the range of 0 to 100.

bool **lightbulb_get_switch** (void)

Get the on/off status of the lightbulb.

Returns true The lightbulb is on.
Returns false The lightbulb is off.

lightbulb_works_mode_t **lightbulb_get_mode** (void)

Get the work mode of the lightbulb.

Returns lightbulb_works_mode_t The current work mode of the lightbulb.

esp_err_t **lightbulb_get_power_limit** (*lightbulb_power_limit_t* *power_limit)

Get the power limit of the lightbulb.

Parameters **power_limit** –A pointer to a `lightbulb_power_limit_t` structure where the power limit details will be stored.
Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_get_all_detail** (*lightbulb_status_t* *status)

Get all the status details of the lightbulb.

Parameters **status** –A pointer to a `lightbulb_status_t` structure where the status details will be stored.
Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_status_get_from_nvs** (*lightbulb_status_t* *value)

Get the lightbulb status from NVS.

Parameters **value** –Pointer to a `lightbulb_status_t` structure where the stored state will be read into.
Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_status_set_to_nvs** (const *lightbulb_status_t* *value)

Store the lightbulb state to NVS.

Parameters **value** –Pointer to a `lightbulb_status_t` structure representing the current running state to be stored.
Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_status_erase_nvs_storage** (void)

Erase the lightbulb state stored in NVS.

Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_basic_effect_start** (*lightbulb_effect_config_t* *config)

Start some blinking/breathing effects.

Parameters **config** –Pointer to a `lightbulb_effect_config_t` structure containing the configuration for the effect.

Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_basic_effect_stop** (void)

Stop the effect in progress and keep the current lighting output.

Returns esp_err_t An error code indicating the success or failure of the operation.

esp_err_t **lightbulb_basic_effect_stop_and_restore** (void)

Stop the effect in progress and restore the previous lighting output.

Returns esp_err_t An error code indicating the success or failure of the operation.

void **lightbulb_lighting_output_test** (*lightbulb_lighting_unit_t* mask, uint16_t speed_ms)

Used to test lightbulb hardware functionality.

Parameters

- **mask** –A bitmask representing the test unit or combination of test units to be tested.
- **speed_ms** –The switching speed in milliseconds for the lighting patterns.

Structures

struct **lightbulb_cct_mapping_data_t**

CCT (Correlated Color Temperature) mapping data.

Note: This structure is used for precise color temperature calibration. Each color temperature value (cct_kelvin) needs to have a corresponding percentage (cct_percentage) determined, which is used to calibrate the color temperature more accurately. The rgbcw array specifies the current coefficients for the RGBCW channels. These coefficients are instrumental in adjusting the intensity of each color channel (Red, Green, Blue, Cool White, Warm White) to achieve the desired color temperature. They are also used for power limiting to ensure energy efficiency and LED longevity. Therefore, the sum of all values in the rgbcw array must equal 1 to maintain the correct power balance.

Public Members

float **rgbcw**[5]

Array of float coefficients for CCT data (R, G, B, C, W). These coefficients are used to adjust the intensity of each color channel.

uint8_t **cct_percentage**

Percentage representation of the color temperature. Used to calibrate the light' s color temperature within a predefined range.

uint16_t **cct_kelvin**

The specific color temperature value in Kelvin. Used to define the perceived warmth or coolness of the light emitted.

struct **lightbulb_color_mapping_data_t**

Color mode mapping data.

Note: Used for calibrating color accuracy in color mode.

Public Members

float **rgbcw_100**[5]

The RGBCW components required when saturation is 100 at a specific hue.

float **rgbcw_50**[5]

The RGBCW components required when saturation is 50 at a specific hue.

float **rgbcw_0**[5]

The RGBCW components required when saturation is 10 at a specific hue.

uint16_t **hue**

hue.

struct **lightbulb_gamma_config_t**

Gamma correction and color balance configuration.

Note: This structure is used for calibrating the brightness proportions and color balance of a lightbulb. Typically, the human eye perceives changes in brightness in a non-linear manner, which is why gamma correction is necessary. This helps to adjust the brightness to match the non-linear perception of the human eye, ensuring a more natural and visually comfortable light output. The color balance coefficients are used to adjust the intensity of each color channel (Red, Green, Blue, Cool White, Warm White) to achieve the desired color balance and overall light quality.

Public Members

float **balance_coefficient**[5]

Array of float coefficients for adjusting the intensity of each color channel (R, G, B, C, W). These coefficients help in achieving the desired color balance for the light output.

float **color_curve_coefficient**

Coefficient for gamma correction (RGB mode). The default value is 1.0, which is linear.

float **white_curve_coefficient**

Coefficient for gamma correction (CCT mode). The default value is 1.0, which is linear.

struct **lightbulb_status_t**

The working status of the lightbulb.

Attention Both the variable `value` and the variable `brightness` are used to mark light brightness. They respectively indicate the brightness of color light and white light.

Note: Due to the differences in led beads, the percentage does not represent color temperature. The real meaning is the output ratio of cold and warm led beads: 0% lights up only the warm beads, 50% lights up an

equal number of cold and warm beads, and 100% lights up only the cold beads. For simplicity, we can roughly assume that 0% represents the lowest color temperature of the beads, and 100% represents the highest color temperature. The meaning of intermediate percentages varies under different color temperature calibration schemes:

- Standard Mode: The percentage is proportionally mapped between the lowest and highest color temperatures.
- Precise Mode:
  - For hardware CCT schemes (only applicable to PWM-driven): The percentage actually represents the duty cycle of the PWM-driven color temperature channel, with each percentage corresponding to an accurate and real color temperature.
  - For those with CW channels: The percentage represents the proportion of cold and warm beads involved in the output. The increase in percentage does not linearly correspond to the increase in color temperature, and instruments are needed for accurate determination.
  - For those using RGB channels to mix cold and warm colors: The percentage has no real significance and can be ignored. In actual use, it can serve as an index to reference corresponding color temperature values."

Public Members

lightbulb_works_mode_t **mode**

The working mode of the lightbulb (color, white, etc.).

bool **on**

On/off status of the lightbulb.

uint16_t **hue**

Hue value for color light (range: 0-360).

uint8_t **saturation**

Saturation value for color light (range: 0-100).

uint8_t **value**

Brightness value for color light (range: 0-100).

uint8_t **cct_percentage**

0% -> .. -> 100% 2200K -> .. -> 7000K warm -> .. -> cold Cold and warm led bead output ratio (range: 0-100).

uint8_t **brightness**

Brightness value for white light (range: 0-100).

struct **lightbulb_power_limit_t**

Output limit or gain without changing color.

Public Members

uint8_t **white_max_brightness**

Maximum brightness limit for white light output.

uint8_t **white_min_brightness**

Minimum brightness limit for white light output.

uint8_t **color_max_value**

Maximum value limit for color light output.

uint8_t **color_min_value**

Minimum value limit for color light output.

uint16_t **white_max_power**

Maximum power limit for white light output.

uint16_t **color_max_power**

Maximum power limit for color light output.

struct **lightbulb_cct_kelvin_range_t**

Used to map percentages to Kelvin values.

Public Members

uint16_t **min**

Minimum color temperature value in Kelvin, default is 2200 K.

uint16_t **max**

Maximum color temperature value in Kelvin, default is 7000 K.

struct **lightbulb_capability_t**

Some Lightbulb Capability Configuration Options.

Public Members

uint16_t **fade_time_ms**

Fade time in milliseconds (ms), data range: 100ms - 3000ms, default is 800ms.

uint16_t **storage_delay_ms**

Storage delay time in milliseconds (ms), used to mitigate adverse effects of short-term repeated erasing and writing of NVS.

lightbulb_led_beads_comb_t **led_beads**

Configuration for the combination of LED beads. Please select the appropriate type for the onboard LED.

lightbulb_status_storage_cb_t **storage_cb**

Callback function to be called when the lightbulb status starts to be stored.

bool **enable_fade**

Enable this option to use fade effects for color switching instead of direct rapid changes.

bool **enable_lowpower**

Enable low-power regulation when the lights are off.

bool **enable_status_storage**

Enable this option to store the lightbulb state in NVS.

bool **enable_hardware_cct**

Enable this option if your driver uses hardware CCT. Some PWM type drivers may need to set this option.

bool **enable_precise_cct_control**

Enable this option if you need precise CCT control. Must set 'enable_hardware_cct' to false in order to enable it.

bool **enable_precise_color_control**

Enable this option if you need precise Color control.

bool **sync_change_brightness_value**

Enable this option if you need to use a parameter to mark the brightness of the white and color output.

bool **disable_auto_on**

Enable this option if you don't need automatic on when the color/white value is set.

struct **lightbulb_config_t**

Lightbulb Configuration Options.

Public Members

lightbulb_driver_t **type**

Type of the lightbulb driver.

union *lightbulb_config_t*::[anonymous] **driver_conf**

Configuration specific to the lightbulb driver.

uint16_t **kelvin_min**

Minimum Kelvin value.

uint16_t **kelvin_max**

Maximum Kelvin value.

struct *lightbulb_config_t*::[anonymous]::[anonymous] **standard**

Standard Mode

lightbulb_cct_mapping_data_t ***table**

Mixed Color table

int **table_size**

Table size

struct *lightbulb_config_t*::[anonymous]::[anonymous] **precise**

Precise Mode

union *lightbulb_config_t*::[anonymous] **cct_mix_mode**

This configuration is used to set up the CCT (Correlated Color Temperature) calibration scheme. The default mode is the standard mode, which requires no additional configuration. For the precise mode, a color mixing table needs to be configured. To enable this precise CCT control, the 'enable_precise_cct_control' should be set to true in the capability settings.

lightbulb_color_mapping_data_t ***table**

Mixed Color table

struct *lightbulb_config_t*::[anonymous]::[anonymous] **precise**

Precise Mode

union *lightbulb_config_t*::[anonymous] **color_mix_mode**

This configuration is used to set up the color calibration scheme. Measure certain hue and saturation values as calibration points, and use a linear interpolation method for color calibration.

lightbulb_gamma_config_t ***gamma_conf**

Pointer to the gamma configuration data.

lightbulb_power_limit_t ***external_limit**

Pointer to the external power limit configuration.

gpio_num_t **red**

GPIO Pin for the red LED

gpio_num_t **green**

GPIO Pin for the green LED

gpio_num_t **blue**

GPIO Pin for the blue LED

gpio_num_t **cold_cct**

GPIO Pin for the cold or cct LED

gpio_num_t **warm_brightness**

GPIO Pin for the warm or brightness LED

struct *lightbulb_config_t*::[anonymous]::[anonymous] **pwm_io**

Configuration for PWM driver I/O pins.

lightbulb_chip_out_pin_t **red**

IIC dimming driver output channel for red LED

GPIO Pin for the red LED for SM16825E driver

lightbulb_chip_out_pin_t **green**

IIC dimming driver output channel for green LED

GPIO Pin for the green LED for SM16825E driver

lightbulb_chip_out_pin_t **blue**

IIC dimming driver output channel for blue LED

GPIO Pin for the blue LED for SM16825E driver

lightbulb_chip_out_pin_t **cold_white**

IIC dimming driver output channel for cold or white LED

GPIO Pin for the white LED for SM16825E driver

lightbulb_chip_out_pin_t **warm_yellow**

IIC dimming driver output channel for warm or yellow LED

GPIO Pin for the yellow LED for SM16825E driver

struct *lightbulb_config_t*::[anonymous]::[anonymous] **iic_io**

Configuration for IIC driver I/O pins.

struct *lightbulb_config_t*::[anonymous]::[anonymous] **sm16825e_io**

Configuration for SM16825E driver I/O pins.

union *lightbulb_config_t*::[anonymous] **io_conf**

Union for I/O configuration based on the selected driver type.

lightbulb_capability_t **capability**

Lightbulb capability configuration.

lightbulb_status_t **init_status**

Initial status of the lightbulb.

struct **lightbulb_effect_config_t**

Effect function configuration options.

Public Members

lightbulb_effect_t **effect_type**

Type of the effect to be configured.

lightbulb_works_mode_t **mode**

Working mode of the lightbulb during the effect.

uint16_t **hue**

Hue component value for the effect (0-360).

uint8_t **saturation**

Saturation component value for the effect (0-100).

uint16_t **cct**

Color temperature value for the effect.

uint8_t **min_value_brightness**

Minimum brightness level for the effect (0-100).

uint8_t **max_value_brightness**

Maximum brightness level for the effect (0-100).

uint16_t **effect_cycle_ms**

Cycle time for the effect in milliseconds (ms).

int **total_ms**

Total duration of the effect in milliseconds (ms). If greater than 0, enables an auto-stop timer.

void (*__user_cb__)(void)

User-defined callback function to be called when the auto-stop timer triggers.

bool **interrupt_forbidden**

If true, the auto-stop timer can only be stopped by specific interfaces or FreeRTOS triggers.

Type Definitions

typedef esp_err_t (*__lightbulb_status_storage_cb_t__)(*lightbulb_status_t* status)

Function pointer type to store lightbulb status.

Param status The *lightbulb_status_t* structure containing the current status of the lightbulb.
Return esp_err_t

typedef *lightbulb_chip_out_pin_t* **lightbulb_iic_out_pin_t**

IIC dimming chip output port definition.

Enumerations

enum **lightbulb_driver_t**

Supported drivers.

Values:

enumerator **DRIVER_SELECT_INVALID**

enumerator **DRIVER_ESP_PWM**

enumerator **DRIVER_SM2135E**

enumerator **DRIVER_SM2135EH**

enumerator **DRIVER_SM2182E**

enumerator **DRIVER_SM2x35EGH**

enumerator **DRIVER_BP1658CJ**

enumerator **DRIVER_BP57x8D**

enumerator **DRIVER_KP18058**

enumerator **DRIVER_WS2812**

enumerator **DRIVER_SM16825E**

enumerator **DRIVER_SELECT_MAX**

enum **lightbulb_led_beads_comb_t**

Supported LED bead combinations.

Values:

enumerator **LED_BEADS_INVALID**

Invalid LED bead combination.

enumerator **LED_BEADS_1CH_C**

Single-channel: Cold white LED bead.

enumerator **LED_BEADS_1CH_W**

Single-channel: Warm white LED bead.

enumerator **LED_BEADS_2CH_CW**

Two channels: Warm white + cold white LED beads combination.

enumerator **LED_BEADS_3CH_RGB**

Three channels: Red + green + blue LED beads combination.

enumerator **LED_BEADS_4CH_RGBC**

Four channels: Red + green + blue + cold white LED beads combination.

enumerator **LED_BEADS_4CH_RGBCC**

Four channels: Red + green + blue + 2 * cold white LED beads combination.

enumerator **LED_BEADS_4CH_RGBW**

Four channels: Red + green + blue + warm white LED beads combination.

enumerator **LED_BEADS_4CH_RGBWW**

Four channels: Red + green + blue + 2 * warm white LED beads combination.

enumerator **LED_BEADS_5CH_RGBCW**

Five channels: Red + green + blue + cold white + warm white LED beads combination.

enumerator **LED_BEADS_5CH_RGBCC**

Five channels: Red + green + blue + 2 * cold white + RGB mix warm white LED beads combination.

enumerator **LED_BEADS_5CH_RGBWW**

Five channels: Red + green + blue + 2 * warm white + RGB mix cold white LED beads combination.

enumerator **LED_BEADS_5CH_RGBC**

Five channels: Red + green + blue + cold white + RGB mix warm white LED beads combination.

enumerator **LED_BEADS_5CH_RGBW**

Five channels: Red + green + blue + warm white + RGB mix cold white LED beads combination.

enumerator **LED_BEADS_MAX**

Maximum number of supported LED bead combinations.

enum **lightbulb_effect_t**

Supported effects.

Values:

enumerator **EFFECT_BREATH**

enumerator **EFFECT_BLINK**

enum **lightbulb_lighting_unit_t**

Lighting test units.

Values:

enumerator **LIGHTING_RAINBOW**

Rainbow lighting effect.

enumerator **LIGHTING_WARM_TO_COLD**

Transition from warm to cold lighting.

enumerator **LIGHTING_COLD_TO_WARM**

Transition from cold to warm lighting.

enumerator **LIGHTING_BASIC_FIVE**

Basic five lighting colors.

enumerator **LIGHTING_COLOR_MUTUAL_WHITE**

Color and mutual white lighting.

enumerator **LIGHTING_COLOR_EFFECT**

Color-specific lighting effect.

enumerator **LIGHTING_WHITE_EFFECT**

White-specific lighting effect.

enumerator **LIGHTING_ALEXA**

Alexa integration lighting.

enumerator **LIGHTING_COLOR_VALUE_INCREMENT**

Incrementing color values lighting.

enumerator **LIGHTING_WHITE_BRIGHTNESS_INCREMENT**

Incrementing white brightness lighting.

enumerator **LIGHTING_ALL_UNIT**

All lighting units.

enum **lightbulb_works_mode_t**

The working mode of the lightbulb.

Values:

enumerator **WORK_INVALID**

Invalid working mode.

enumerator **WORK_COLOR**

Color mode, where the lightbulb emits colored light.

enumerator **WORK_WHITE**

White mode, where the lightbulb emits white light.

enum **lightbulb_chip_out_pin_t**

Port enumeration names for multiple output channel chips.

Values:

enumerator **OUT1**

Chip output port 1/RED.

enumerator **OUT2**

Chip output port 2/GREEN.

enumerator **OUT3**

Chip output port 3/BLUE.

enumerator **OUT4**

Chip output port 4/WHITE.

enumerator **OUT5**

Chip output port 5/YELLOW.

enumerator **OUT_MAX**

The maximum value for the Chip output port enumeration, this is invalid value.

Chapter 19

Other Resources

19.1 GPIO Expander

With further expansions of the ESP32 chip family, more application scenarios with diverse demands are being introduced, including some that have more requirements on GPIO numbers. The subsequence release of ESP32-S2 and other products have included up to 43 GPIOs, which can greatly alleviate the problem of GPIO resource constraint. If this still could not meet your demand, you can also add GPIO expansion chips to ESP32 to have more GPIO resources, such as using the I2C-based GPIO expansion module MCP23017, which can expand 16 GPIO ports per module and mount up to 8 expansion modules simultaneously thus expanding additional 128 GPIO ports totally.

19.1.1 Adapted Products

Name	Function	Bus	Vendor	Datasheet	Driver
MCP23017	16-bit I/O expander	I2C	Microchip	Datasheet	mcp23017

19.2 ADC Range Extension Solution

Important: The current ADC extended range solution is only applicable to ESP32-S2 and ESP32-S3 chips.

The ESP32-S2/S3 ADC, within its default range, can handle most ADC functionalities such as key input and battery voltage detection through an external voltage divider circuit. However, for applications like NTC measurements, full-scale measurement (0 ~ 3300 mV) may be required. By configuring the registers to adjust the ADC' s offset voltage and applying a nonlinear compensation method for the high-voltage region, we can extend the range of the ESP32-S2/S3 ADC.

The process is as follows:

1. Measure the first voltage value using the default offset
2. If the measured voltage is less than the `set voltage`, directly output the measured voltage as the result
3. If the measured voltage is greater than the `set voltage`, increase the offset voltage, perform a second measurement, apply nonlinear correction, and output the corrected value as the result
4. Restore the offset value once measurement is completed

Overall, during each ADC measurement, there will be 1-2 times ADC reading. For most application scenarios, the measurement delay introduced by this scheme is negligible.

Note: For the ESP32-S2, the `set voltage` is 2600 mV; for the ESP32-S3, the `set voltage` is 2900 mV.

19.2.1 Patch Use Guide

Important: This patch is developed based on ESP-IDF `v4.4.8` and `v5.3.1`. If you need to use this solution with other versions of ESP-IDF, please refer to the patch content and modify ESP-IDF accordingly.

How to Apply a Patch Based on ESP-IDF `v4.4.8`

- ESP32-S2 ADC Extension Patch Download: `esp32s2_adc_range_to_3300.patch`
- ESP32-S3 ADC Extension Patch Download: `esp32s3_adc_range_to_3300.patch`

Using the ESP32-S3 as an example, please load the patch as follows:

1. Please make sure ESP-IDF has been `checked out` to the `v4.4.8`
2. Using command `git am --signoff < esp32s3_adc_range_to_3300.patch` to apply the patch to ESP-IDF
3. Please note that this solution is only effective for the `esp_adc_cal_get_voltage` interface. Users can directly call this interface to obtain the extended readings.

How to Apply a Patch Based on ESP-IDF `v5.3.1`

- ESP32-S2 ADC Extension Patch Download: `esp32s2_adc_range_to_3300_v531.patch`
- ESP32-S3 ADC Extension Patch Download: `esp32s3_adc_range_to_3300_v531.patch`

Using the ESP32-S3 as an example, please load the patch as follows:

1. Please make sure ESP-IDF has been `checked out` to the `v5.3.1`
2. Using command `git am --signoff < esp32s3_adc_range_to_3300_v531.patch` to apply the patch to ESP-IDF
3. Please note that this solution is only effective for the `adc_oneshot_get_calibrated_result` interface. Users can directly call this interface to obtain the extended readings.

19.2.2 API Guide

The method to obtain the voltage value after ADC range extension varies for different versions of ESP-IDF:

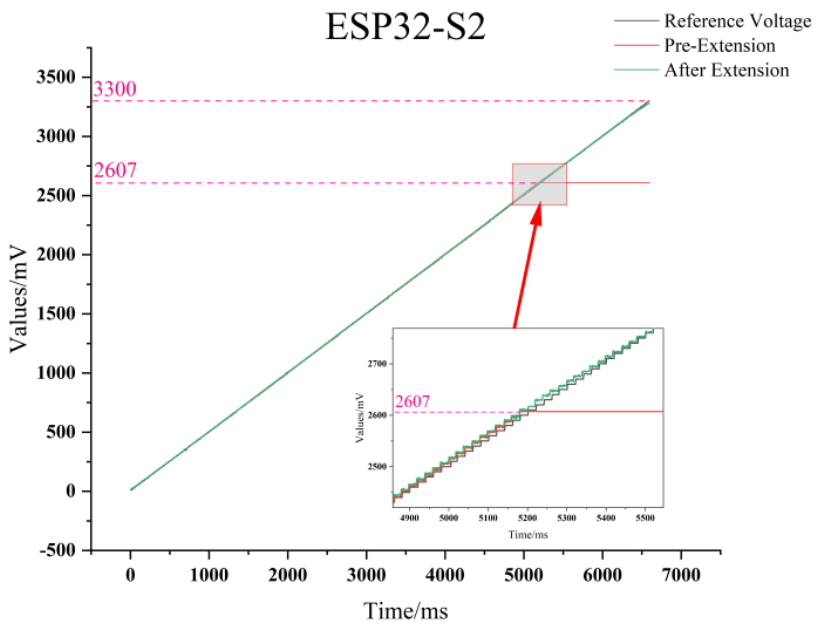
- ESP-IDF `v4.4.8`
  1. To get the range expansion result, users must directly use `esp_adc_cal_get_voltage` to get the voltage of `ADC1` or `ADC2`.
  2. Other APIs of ESP-IDF `v4.4.8` ADC are not affected, and the read results are consistent with the default results
- ESP-IDF `v5.3.1`
  1. To get the range expansion result, users must directly use `adc_oneshot_get_calibrated_result` to get the voltage of `ADC1` or `ADC2`.
  2. Other APIs of ESP-IDF `v5.3.1` ADC are not affected, and the read results are consistent with the default results

The patch enables ADC range extension by default and uses the `float` data type for correction. If you need to disable ADC extension or change the data type used during the correction process, please refer to the following procedure:

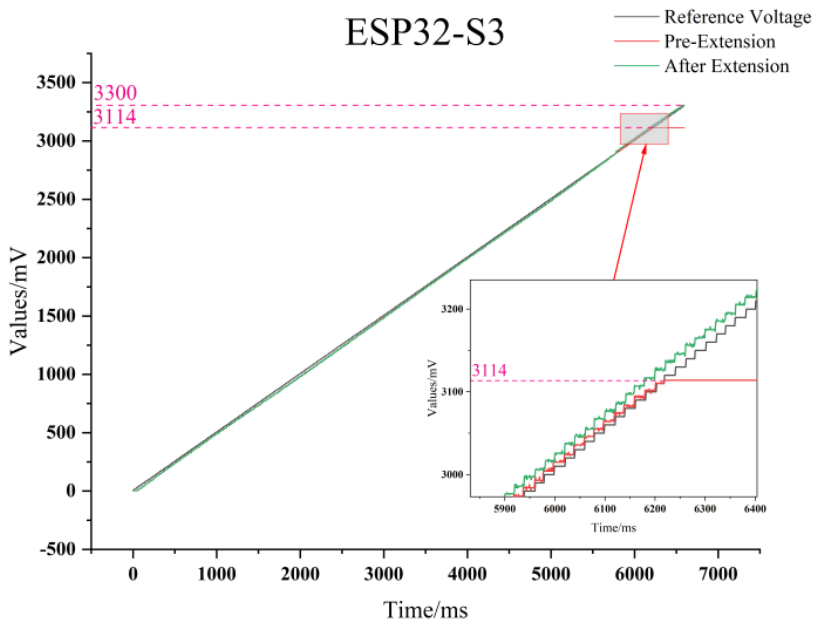
- For ESP-IDF `v4.4.8`: Please use `menuconfig` to modify the settings under `Component config → Driver configurations → ADC configuration → ADC User Code Offset`
- For ESP-IDF `v5.3.1`: Please use `menuconfig` to modify the settings under `Component config → ADC and ADC Calibration → ADC User Code Offset`

19.2.3 Comparison of ADC Range Extension Effects

ESP32-S2 ADC Range Extension Effects Comparison:



ESP32-S3 ADC Range Extension Effects Comparison:



19.3 Zero_Detection

The zero cross detection driver is a component designed to analyze zero cross signals. By examining the period and triggering edges of zero cross signals, it can determine the signal' s validity, invalidity, whether it exceeds the expected frequency range, and if there are signal losses, among other conditions.

The zero cross detection component supports the detection of two types of zero cross signals.

- Square Wave: Inverts the current voltage level when the signal crosses zero.
- Pulse Wave: A pulse generated when the signal crosses zero.

Due to factors such as response delay, this component supports two driving modes, including GPIO interrupt and MCPWM capture.

Users can flexibly configure the component' s drive modes, as well as adjust parameters such as the effective frequency range and the number of valid signal judgments. This provides a high level of flexibility.

The program returns results and data in the form of events, meeting the user' s need for timely processing of signals.

19.3.1 Zero Detection event

Triggering conditions for each zero detection event are enlisted in the table below:

Event	Trigger Condition
SIGNAL_RISING_EDGE	Rising edge
SIGNAL_FALLING_EDGE	Falling edge
SIGNAL_VALID	The number of times the frequency is within the valid range exceeds the valid times.
SIGNAL_INVALID	The number of times the frequency is within the invalid range exceeds the invalid times.
SIGNAL_LOST	No rising or falling edges in the signal within 100ms.
SIGNAL_FREQ_OUT_OF_RANGE	The calculated frequency is outside the set frequency range.

Attention: No blocking operations such as **TaskDelay** are allowed in the call-back function

19.3.2 Configuration

- USE_GPTIMER : Decide whether to use the GPTimer driver; default is to use the ESPTimer

19.3.3 Demonstration

Create a zero detection

```
void IRAM_ATTR zero_detection_event_cb(zero_detect_event_t zero_detect_event, zero_
↪detect_cb_param_t *param, void *usr_data)  //User's callback API
{
    switch (zero_detect_event) {
    case SIGNAL_FREQ_OUT_OF_RANGE:
        ESP_LOGE(TAG, "SIGNAL_FREQ_OUT_OF_RANGE");
        break;
    case SIGNAL_VALID:
        ESP_LOGE(TAG, "SIGNAL_VALID");
```

(continues on next page)

(continued from previous page)

```
        break;
    case SIGNAL_LOST:
        ESP_LOGE(TAG, "SIGNAL_LOST");
        break;
    default:
        break;
    }
}

// Create a zero detection and register call-back
zero_detect_config_t config = {
    .capture_pin = 2,
    .freq_range_max_hz = 65,
    .freq_range_min_hz = 45,  //Hz
    .valid_times = 6,
    .invalid_times = 5,
    .zero_signal_type = SQUARE_WAVE,
    .zero_driver_type = MCPWM_TYPE,
};
zero_detect_handle_t *g_zcds = zero_detect_create(&config);
if(NULL == g_zcds) {
    ESP_LOGE(TAG, "Zero Detection create failed");
}
zero_detect_register_cb(g_zcds, zero_detection_event_cb, NULL);
```

19.3.4 API Reference

Header File

- components/zero_detection/include/zero_detection.h

Functions

zero_detect_handle_t **zero_detect_create** (*zero_detect_config_t* *config)

Create a zero detect target.

Parameters **config** –A zero detect object to config

Returns

- zero_detect_handle_t A zero cross detection object

void **zero_show_data** (*zero_detect_handle_t* zcd_handle)

Show zero detect test data.

Parameters **zcd_handle** –A zero detect handle

void **zero_detect_pause** (*zero_detect_handle_t* zcd_handle)

Pause zero detect.

Parameters **zcd_handle** –A zero detect handle

void **zero_detect_resume** (*zero_detect_handle_t* zcd_handle)

Resume zero detect.

Parameters **zcd_handle** –A zero detect handle

esp_err_t **zero_detect_delete** (*zero_detect_handle_t* zcd_handle)

Delete a zero detect device.

Parameters **zcd_handle** –A zero detect handle

Returns

- ESP_OK Success
- ESP_FAIL Failure

bool **zero_detect_get_power_status** (*zero_detect_handle_t* zcd_handle)

Get relay power status.

Parameters **zcd_handle** –A zero detect handle

Returns

- true Power on
- false Power off

bool **zero_detect_singal_invaild_status** (*zero_detect_handle_t* zcd_handle)

Get signal invalid status.

Parameters **zcd_handle** –A zero detect handle

Returns

- true Signal is invalid
- false Signal is valid

zero_signal_type_t **zero_detect_get_signal_type** (*zero_detect_handle_t* zcd_handle)

Get signal type.

Parameters **zcd_handle** –A zero detect handle

Returns

- SQUARE_WAVE Signal type is square
- PULSE_WAVE Signal type is pulse

esp_err_t **zero_detect_register_cb** (*zero_detect_handle_t* zcd_handle, *zero_cross_cb_t* cb, void *usr_data)

Register zero cross detection callback.

Parameters

- **zcd_handle** –A zero detect handle
- **cb** –A callback function
- **usr_data** –User data

Returns

- ESP_OK Success
- ESP_FAIL Failure

Unions

union **zero_detect_cb_param_t**

#include <zero_detection.h> Event callback parameters union.

Public Members

struct *zero_detect_cb_param_t*::*signal_freq_event_data_t* **signal_freq_event_data**

Signal freq event data struct

struct *zero_detect_cb_param_t*::*signal_valid_event_data_t* **signal_valid_event_data**

Signal valid event data struct

struct *zero_detect_cb_param_t*::*signal_invalid_event_data_t* **signal_invalid_event_data**

Signal invalid event data struct

struct *zero_detect_cb_param_t*::*signal_rising_edge_event_data_t* **signal_rising_edge_event_data**

Signal rising edge event data struct

struct *zero_detect_cb_param_t*::*signal_falling_edge_event_data_t* **signal_falling_edge_event_data**

Signal falling edge event_data

struct **signal_falling_edge_event_data_t**

#include <zero_detection.h> Signal falling edge data return type.

Public Members

uint16_t **valid_count**

Counting when the signal is valid

uint16_t **invalid_count**

Counting when the signal is invalid

uint32_t **full_cycle_us**

Current signal cycle

struct **signal_freq_event_data_t**

#include <zero_detection.h> Signal exceeds frequency range data return type.

Public Members

zero_signal_edge_t **cap_edge**

Trigger edge of zero cross signal

uint32_t **full_cycle_us**

Current signal cycle

struct **signal_invalid_event_data_t**

#include <zero_detection.h> Signal invalid data return type.

Public Members

zero_signal_edge_t **cap_edge**

Trigger edge of zero cross signal

uint32_t **full_cycle_us**

Current signal cycle

uint16_t **invalid_count**

Counting when the signal is invalid

struct **signal_rising_edge_event_data_t**

#include <zero_detection.h> Signal rising edge data return type.

Public Members

uint16_t **valid_count**

Counting when the signal is valid

uint16_t **invalid_count**

Counting when the signal is invalid

uint32_t **full_cycle_us**

Current signal cycle

struct **signal_valid_event_data_t**

#include <zero_detection.h> Signal valid data return type.

Public Members

zero_signal_edge_t **cap_edge**

Trigger edge of zero cross signal

uint32_t **full_cycle_us**

Current signal cycle

uint16_t **valid_count**

Counting when the signal is valid

Structures

struct **zero_detect_config_t**

User config data type.

Public Members

int32_t **capture_pin**

GPIO number for zero cross detect capture

uint16_t **valid_times**

Minimum required number of times for detecting signal validity

uint16_t **invalid_times**

Minimum required number of times for detecting signal invalidity

uint64_t **signal_lost_time_us**

Minimum required duration for detecting signal loss

zero_signal_type_t **zero_signal_type**

Zero Crossing Signal type

zero_driver_type_t **zero_driver_type**

Zero crossing driver type

double **freq_range_max_hz**

Maximum value of the frequency range when determining a valid signal

double **freq_range_min_hz**

Minimum value of the frequency range when determining a valid signal

Macros

ZERO_DETECTION_INIT_CONFIG_DEFAULT()

Type Definitions

typedef void (***zero_cross_cb_t**)(*zero_detect_event_t* zero_detect_event, *zero_detect_cb_param_t* *param, void *usr_data)

Callback format.

typedef void ***zero_detect_handle_t**

Enumerations

enum **zero_detect_event_t**

Zero detection events.

Values:

enumerator **SIGNAL_FREQ_OUT_OF_RANGE**

enumerator **SIGNAL_VALID**

enumerator **SIGNAL_INVALID**

enumerator **SIGNAL_LOST**

enumerator **SIGNAL_RISING_EDGE**

enumerator **SIGNAL_FALLING_EDGE**

enum **zero_signal_type_t**

Zero detection wave type.

Values:

enumerator **SQUARE_WAVE**

enumerator **PULSE_WAVE**

enum **zero_signal_edge_t**

Zero detection signal edge type.

Values:

enumerator **CAP_EDGE_POS**

enumerator **CAP_EDGE_NEG**

enum **zero_driver_type_t**

Zero detection driver type.

Values:

enumerator **GPIO_TYPE**

Use GPIO as driver

19.4 ADC Two-Point Calibration Scheme

Important: The current two-point ADC calibration scheme is only applicable to ESP32 and ESP32-S2 chips

By default, the ESP32 is calibrated at the factory using a reference voltage. To further improve the consistency of ADC measurement results, users can modify the calibration scheme stored in eFuse to use the two-point calibration method.

Similarly, for the ESP32-S2 chip, the factory default is the two-point calibration method (Method 1). However, this method does not take into account calibration data across different ADC attenuation levels. If users wish to perform calibration using data from different attenuation settings, they can modify the calibration scheme in eFuse to use the two-point calibration method (Method 2).

It is important to note that both of the above methods require users to use the espefuse tool to modify the calibration scheme in eFuse and input the corresponding calibration data. This process must be carried out with extreme caution, as writing incorrect calibration data is irreversible and cannot be modified afterward. This is because data stored in eFuse is permanent and cannot be changed once written. Therefore, ensure the accuracy of the data before proceeding to avoid any unnecessary loss.

To avoid the risks associated with modifying eFuse, the `adc_tp_calibration` component provides a method that allows users to perform calibration at the application level. This approach offers the following features:

- Supports inputting calibration parameters at the application level, allowing users to store the calibration data in storage media such as NVS or SD cards.
- Supports the two-point calibration scheme for ESP32 and the Method 2 two-point calibration scheme for ESP32-S2, without interfering with the existing calibration scheme provided by ESP-IDF.

19.4.1 Principle of Two-Point Calibration for ESP32

The two-point calibration for ESP32 requires applying stable voltages of 150 mV and 850 mV to both ADC1 and ADC2 channels. At each voltage level, raw ADC data should be collected using the ADC raw data acquisition interface to obtain stable readings corresponding to the applied voltages.

Input Voltage	ADC1 (ATTEN0)	ADC2 (ATTEN0)
150 mV	A1	A2
850 mV	B1	B2

Note: When collecting raw ADC data at 150 mV and 850 mV, the ADC driver attenuation must be configured to `ADC_ATTEN_DB_0`.

Two-Point Calibration Formula for ESP32 ADC:

$$y = \frac{coeff_a * x + coeff_{around}}{coeff_{scale}} + coeff_b$$

- $coeff_a$: $\frac{(850-150)*atten_{scale}[atten]+(B-A)/2}{B-A}$
- $coeff_b$: $850 - \frac{(850-150)*B+(B-A)/2}{B-A} + atten_{offset}[atten]$
- $coeff_{scale}$: 65536
- $coeff_{around}$: $\frac{coeff_{scale}}{2}$

ADC1	ATTEN0	ATTEN_2_5	ATTEN_6	ATTEN_12
$atten_{scale}$	65504	86975	120389	224310
$atten_{offset}$	0	1	27	54

ADC2	ATTEN0	ATTEN_2_5	ATTEN_6	ATTEN_12
$atten_{scale}$	65467	86861	120416	224708
$atten_{offset}$	0	9	26	66

19.4.2 Principle of Two-Point Calibration (Method 2) for ESP32-S2

The two-point calibration (Method 2) for ESP32-S2 requires applying stable voltages of 600 mV, 800 mV, 1000 mV, and 2000 mV to both ADC1 and ADC2 channels. At each voltage level, raw ADC data should be collected using the ADC raw data acquisition interface to obtain stable readings. Taking ADC1 as an example:

Input Voltage	ATTEN	ADC1 Raw Value
600 mV	ATTEN_0	High[ATTEN_0]
800 mV	ATTEN_2_5	High[ATTEN_2_5]
1000 mV	ATTEN_6	High[ATTEN_6]
2000 mV	ATTEN_12	High[ATTEN_12]

Note: In two-point calibration (Method 2) for ESP32-S2, calibration points must be selected by collecting stable raw ADC data under the corresponding attenuation settings and calibration voltages to ensure accuracy. The High value is used to store the raw ADC data obtained under each specific combination of attenuation and calibration voltage.

The calculation formula for two-point ADC calibration (Method 2) on ESP32-S2 is as follows:

$$y = \frac{x * coeff_a/(coeff_{ascaling}/coeff_{bscaling}) + coeff_b}{coeff_{bscaling}}$$

- $coeff_a$: $coeff_{ascaling} * v_{high}[atten]/High[atten]$
- $coeff_b$: 0
- $coeff_{ascaling}$: 65536
- $coeff_{bscaling}$: 1024

	ATTEN0	ATTEN_2_5	ATTEN_6	ATTEN_12
v_{high}	600	800	1000	2000

19.4.3 API Reference

Header File

- components/utilities/adc_tp_calibration/include/adc_tp_calibration.h

Functions

adc_tp_cali_handle_t **adc_tp_cali_create** (*adc_tp_cali_config_t* *config, adc_atten_t atten)

Create ADC calibration handle.

Parameters
- **config** –Pointer to ADC calibration configuration
- **atten** –ADC attenuation level

Returns adc_tp_cali_handle_t Handle of ADC calibration, NULL if failed

esp_err_t **adc_tp_cali_delete** (*adc_tp_cali_handle_t* *adc_tp_cali_handle)

Delete ADC calibration handle.

Parameters **adc_tp_cali_handle** –Pointer to ADC calibration handle

Returns esp_err_t
- ESP_OK: Always returns ESP_OK

esp_err_t **adc_tp_cali_raw_to_voltage** (*adc_tp_cali_handle_t* adc_tp_cali_handle, int raw_value, int *voltage)

Convert ADC raw value to voltage in millivolts.

Parameters
- **adc_tp_cali_handle** –ADC calibration handle
- **raw_value** –ADC raw value to convert
- **voltage** –Pointer to store the converted voltage in millivolts

Returns esp_err_t
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid argument

Structures

struct **adc_tp_cali_config_t**

ADC calibration configuration structure.

This structure contains the raw ADC values measured at specific voltage points for calibration. The values are different for ESP32 and ESP32-S2.

Public Members

adc_unit_t **adc_unit**

ADC unit (ADC1 or ADC2)

Type Definitions

typedef void ***adc_tp_cali_handle_t**

ADC calibration handle

Chapter 20

Contributions Guide

We welcome contributions to the esp-iot-solution project!

20.1 How to Contribute

Contributions to esp-iot-solution - fixing bugs, adding features, adding documentation - are welcome. We accept contributions via GitHub Pull Requests.

20.2 Before Contributing

Before sending us a Pull Request, please consider this list of points:

- Is the contribution entirely your own work, or already licensed under an Apache License 2.0 compatible Open Source License? If not then we unfortunately cannot accept it.
- Does any new code conform to the esp-idf : *Style Guide* ?
- Have you installed the pre-commit hook for the esp-iot-solution project?
- Does the code documentation follow requirements in Documenting-code ?
- Is the code adequately commented for people to understand how it is structured?
- Are comments and documentation written in clear English, with no spelling or grammar errors?
- If the contribution contains multiple commits, are they grouped together into logical changes (one major change per pull request)? Are any commits with names like "fixed typo" squashed into previous commits?
- If you' re unsure about any of these points, please open the Pull Request anyhow and then ask us for feedback.

20.3 Pull Request Process

After you open the Pull Request, there will probably be some discussion in the comments field of the request itself.

Once the Pull Request is ready to merge, it will first be merged into our internal git system for in-house automated testing.

If this process passes, it will be merged onto the public github repository.

20.4 Legal Part

Before a contribution can be accepted, you will need to sign our contributor-agreement. You will be prompted for this automatically as part of the Pull Request process.

20.5 Related Documents

20.5.1 esp-iot-solution 编码规范

总体原则

- 简洁明了，结构清晰
- 统一风格，易于维护
- 充分注释，易于理解
- 继承 ESP-IDF 已有规范
- 继承第三方代码已有规范

目录结构

- components：总体上按照功能分类，大类下如果存在多级目录，应包含一个 `README` 做综述和索引；
- docs：`rst` 格式文档，包括各个组件的使用指南，API 说明；
- examples：总体上按照与组件对应的功能分类；
- tools：CI 脚本、调试工具。

头文件

- 尽量每一 `.c` 对应一个同名的 `.h` 文件；
- 单个组件存在多个 `.h`，主要对外 `.h` 命名尽量与组件名保持一致；
- 头文件中主要放函数声明，不放函数实现；
- 尽量不在头文件定义任何形式的变量；
- 头文件应按照注释规范，对函数接口进行充分注释；
- 添加宏定义避免重复引用，宏定义名为大写的头文件名加下划线填充：

```
#ifndef _IOT_I2C_BUS_H_
#define _IOT_I2C_BUS_H_

#endif
```

- 函数声明添加 extern "C" 修饰，支持 C/C++ 混合编程：

```
#ifdef __cplusplus
extern "C"
{
#endif

//c code

#ifdef __cplusplus
}
#endif
```

注释

- 安装 VSCODE 插件 Doxygen Documentation Generator 可自动生成注释框架；
- 注释中避免使用单词缩写；
- 函数声明处注释需要描述函数功能、性能或用法，输入和输出参数、函数返回值说明。

自动生成的注释框架：

```
/**
* @brief
*
* @param port
* @param conf
* @return i2c_bus_handle_t
*/
i2c_bus_handle_t iot_i2c_bus_create(i2c_port_t port, i2c_config_t* conf);
```

补充信息和参数方向：

```
/**
* @brief Create an I2C bus instance then return a handle if created successfully.
* @note Each I2C bus works in a singleton mode, which means for an i2c port only␣
↪one group parameter works. When
* iot_i2c_bus_create is called more than one time for the same i2c port, following␣
↪parameter will override the previous one.
*
* @param[in] port I2C port number
* @param[in] conf Pointer to I2C parameters
* @return i2c_bus_handle_t Return the I2C bus handle if created successfully,␣
↪return NULL if failed.
*/
i2c_bus_handle_t iot_i2c_bus_create(i2c_port_t port, i2c_config_t* conf);
```

- 版权声明注释（第三方代码，请保留版权声明信息）

```
/*
 * SPDX-FileCopyrightText: 2022-2023 Espressif Systems (Shanghai) CO LTD
 *
 * SPDX-License-Identifier: Apache-2.0
 */
```

函数规范

- 多处重复使用的代码尽量设计为函数；
- 作用域仅限于当前文件的函数必须声明为静态 `static`；
- 设计使用静态全局变量、静态局部变量的函数时，需要考虑重入问题；
- 尽量在一个固定函数中操作静态全局变量；
- 如果函数存在重入或线程安全问题，需在注释中说明；
- 同一组件内的公有函数名，应保持同一前缀；
- 函数名统一使用 `snake_case` 格式，只使用小写字母，单词之间加 _；
- 函数命名指引（应保持与已有代码风格一致，不严格约束）：

函数名格式	函数示例	说明
iot_type_xxx	iot_sensor_xxx; iot_board_xxx; iot_storage_…	高度抽象的 iot 组件
type_xxx	imu_xxx; light_xxx; eeprom_xxx	对一类外设的抽象
name_xxx	mpu6050_xxx;	底层 driver，由于可能来自第三方，不约束函数名
xxx_creat / xxx_delete		创建和销毁
xxx_read / xxx_write		数据操作

变量规范

- 避免使用全局变量，可声明为静态全局变量，使用 `get_` `set_` 等接口进行变量操作；
- 作用域仅限于当前文件的变量必须声明为静态变量 `static`；
- 静态全局变量请添加 `g_` 前缀，静态局部变量请添加 `s_` 前缀；
- 局部变量设计大小时，应考虑栈溢出的问题；
- 任何变量定义时，必须赋初值；
- 变量功能要明确，避免将单一变量做多个用途；
- 句柄类型变量，在对象销毁后，应重新赋值为 NULL;
- 变量统一使用 `snake_case` 格式，只使用小写字母，单词之间加 _；
- 避免不必要的缩写，例如 `data` 不必缩写为 `dat`；
- 变量应尽量使用有意义的词语，或者已经达成共识的符号或词语缩写；
- 变量命名指引：

类型	规范	示例
全局变量	避免使用	x
静态全局变量	static 标识，g_ 前缀，赋初值	static uint32_t g_connect_num = 0;
静态局部变量	static 标识，s_ 前缀，赋初值	static uint32_t s_connect_num= 0;
迭代计数变量	使用通用的 `i j k`	
常用缩写	abbreviations-in-code	addr,buf ,cfg , cmd, , ctrl,

- 常用缩写列表

缩写	全称	缩写	全称	缩写	全称	缩写	全称
addr	address	id	identifier	len	length	ptr	pointer
buf	buffer	info	information	obj	object	ret	return
cfg	config	hdr	header	param	parameter	temp	temporary、temperature
cmd	command	init	initialize	pos	position	ts	timestamp

类型定义

- 使用加 `snake_case` 格式加 `_t` 后缀

```
typedef int signed_32_bit_t;
```

- 枚举应通过 typedef 通过以下方式定义

```
typedef enum {
    MODULE_FOO_ONE,
    MODULE_FOO_TWO,
    MODULE_FOO_THREE
} module_foo_t;
```

格式和排版规范

该部分继承 ESP-IDF 规范

1. 缩进 每个缩进层使用 **4 个空格**，不要使用制表符进行缩进，将编辑器配置为每次按 tab 键时发出 4 个空格。

2. 垂直间隔 在函数之间放置一个空行，不要以空行开始或结束函数。

```
void function1()
{
    do_one_thing();
    do_another_thing();
                                // INCORRECT, don't place empty line here
}
                                // place empty line here
void function2()
{
                                // INCORRECT, don't use an empty line here
    int var = 0;
    while (var < SOME_CONSTANT) {
        do_stuff(&var);
    }
}
```

只要不严重影响可读性，最大行长度为 120 个字符。

3. 水平间隔 总是在条件和循环关键字之后添加单个空格

```
if (condition) {     // correct
    // ...
}

switch (n) {         // correct
    case 0:
        // ...
}

for(int i = 0; i < CONST; ++i) {     // INCORRECT
    // ...
}
```

在二元操作符两端添加单个空格，一元运算符不需要空格，可以在乘法运算符和除法运算符之间省略空格。

```
const int y = y0 + (x - x0) * (y1 - y0) / (x1 - x0);     // correct

const int y = y0 + (x - x0)*(y1 - y0)/(x1 - x0);         // also okay

int y_cur = -y;                                          // correct
++y_cur;

const int y = y0+(x-x0)*(y1-y0)/(x1-x0);                 // INCORRECT
```

. 和 -> 操作符的周围不需要任何空格。

有时，在一行中添加水平间隔有助于提高代码的可读性。如下，可以添加空格来对齐函数参数:

```
gpio_matrix_in(PIN_CAM_D6,   I2S0I_DATA_IN14_IDX, false);
gpio_matrix_in(PIN_CAM_D7,   I2S0I_DATA_IN15_IDX, false);
gpio_matrix_in(PIN_CAM_HREF, I2S0I_H_ENABLE_IDX,  false);
gpio_matrix_in(PIN_CAM_PCLK, I2S0I_DATA_IN15_IDX, false);
```

- 但是请注意，如果有人添加了一个新行，第一个参数是一个更长的标识符 (例如 PIN_CAM_VSYNC)，它将不适合。因为必须重新对齐其他行，这添加了无意义的更改。因此，尽量少使用这种对齐，特别是如果您希望稍后将新行添加到这列中。
- 不要使用制表符进行水平对齐，不要在行尾添加尾随空格。

4. 括号 函数定义的大括号应该在单独的行上

```
// This is correct:
void function(int arg)
{

}

// NOT like this:
void function(int arg) {

}
```

在函数中，将左大括号与条件语句和循环语句放在同一行

```
if (condition) {
    do_one();
} else if (other_condition) {
    do_two();
}
```

5. 注释 `//` 用于单行注释。对于多行注释，可以在每行上使用 `//`或 `/ * * /` 块注释。

虽然与格式没有直接关系，但下面是一些关于有效使用注释的注意事项。

- 不要使用一个注释来禁用某些功能

```
void init_something()
{
    setup_dma();
    // load_resources();                // WHY is this thing commented, asks the␣
↪reader?
    start_timer();
}
```

- 如果不再需要某些代码，则将其完全删除。如果你需要，你可以随时在 git 历史中查找这个文件。如果您因为临时原因而禁用某些调用，并打算在将来恢复它，则在相邻行上添加解释

```
void init_something()
{
    setup_dma();
    // TODO: we should load resources here, but loader is not fully integrated yet.
    // load_resources();
    start_timer();
}
```

- `#if 0 ... #endif` 块也是如此。如果不使用，请完全删除代码块。否则，添加注释以解释为什么禁用该块。不要使用 `#if 0 ... #endif` 或注释来存储将来可能需要的代码段。
- 不要添加有关作者和更改日期的琐碎注释。您总是可以查找谁使用 git 修改了任何给定的行。例如，此注释在不添加任何有用信息的情况下，使代码混乱不堪：

```
void init_something()
{
    setup_dma();
    // XXX add 2016-09-01
    init_dma_list();
    fill_dma_item(0);
    // end XXX add
    start_timer();
}
```

6. 代码行的结束 commit 中只能包含以 LF（Unix 风格）结尾的文件。

Windows 用户可以将 git 配置为在本地 checkout 是 CRLF（Windows 风格）结尾，通过设置 core.autocrlf 设置来 commit 时以 LF 结尾。Github 有一个关于设置此选项的文档。但是，由于 MSYS2 使用 Unix 样式的行尾，因此在编辑 ESP-IDF 源文件时，通常更容易将文本编辑器配置为使用 LF（Unix 样式）结尾。

如果您在分支中意外地 commit 了 LF 结尾，则可以通过在 MSYS2 或 Unix 终端中运行此命令将它们转换为 Unix（将目录更改为 IDF 工作目录，并预先检查当前是否已 checkout 正确的分支）：

```
git rebase --exec 'git diff-tree --no-commit-id --name-only -r HEAD | xargs␣
↪dos2unix && git commit -a --amend --no-edit --allow-empty' master
```

(请注意，这行代码将在 master 上重新建立基，并在最后更改分支名称以在另一个分支上建立基。)

要更新单个提交，可以运行

```
dos2unix FILENAME
```

然后运行

```
git commit --amend
```

7. 格式化代码 您可以使用 astyle 程序根据上述建议对代码进行格式化。

如果您正在从头开始编写一个文件，或者正在进行完全重写，请随意重新格式化整个文件。如果您正在更改文件的一小部分，不要重新格式化您没有更改的代码。这将帮助其他人检查您的更改。

要重新格式化文件，请运行

```
tools/format.sh components/my_component/file.c
```

CMake 代码风格

- 缩进是 4 个空格
- 最大行长为 120 个字符。分割行时，请尝试尽可能集中于可读性（例如，通过在单独的行上配对关键字/参数对）。
- 不要在 endforeach()、endif() 等后面的可选括号中放入任何内容。
- 对命令、函数和宏名使用小写 (with_underscores)。
- 对于局部作用域的变量，使用小写字母 (with_underscores)。
- 对于全局作用域的变量，使用大写 (WITH_UNDERSCORES)。
- 其他，请遵循 cmake-lint 项目的默认设置。

Index

A

adc_tp_cali_config_t (*C++ struct*), 580
adc_tp_cali_config_t::adc_unit (*C++ member*), 580
adc_tp_cali_create (*C++ function*), 580
adc_tp_cali_delete (*C++ function*), 580
adc_tp_cali_handle_t (*C++ type*), 581
adc_tp_cali_raw_to_voltage (*C++ function*), 580
ALARM_COUNT_US (*C macro*), 521
ALIGNMENTDUTY (*C macro*), 522
ALIGNMENTNMS (*C macro*), 521
audio_codec_adc_cfg_t (*C++ struct*), 366
audio_codec_adc_cfg_t::adc_channel_list (*C++ member*), 366
audio_codec_adc_cfg_t::adc_channel_num (*C++ member*), 366
audio_codec_adc_cfg_t::atten (*C++ member*), 366
audio_codec_adc_cfg_t::conv_frame_size (*C++ member*), 366
audio_codec_adc_cfg_t::handle (*C++ member*), 366
audio_codec_adc_cfg_t::max_store_buf_size (*C++ member*), 366
audio_codec_adc_cfg_t::sample_rate_hz (*C++ member*), 367
audio_codec_adc_cfg_t::unit_id (*C++ member*), 366
audio_codec_new_adc_data (*C++ function*), 366
audio_frame_format (*C++ enum*), 373
audio_frame_format::FORMAT_PCM (*C++ enumerator*), 373
audio_frame_info_t (*C++ struct*), 371
audio_frame_info_t::bits_per_sample (*C++ member*), 371
audio_frame_info_t::channel (*C++ member*), 371
audio_frame_info_t::format (*C++ member*), 371
audio_frame_info_t::sample_rate (*C++ member*), 371
audio_set_clock_cb (*C++ type*), 373
audio_write_cb (*C++ type*), 373
avi_play_end_cb (*C++ type*), 373
avi_player_config_t (*C++ struct*), 372
avi_player_config_t::audio_cb (*C++ member*), 372
avi_player_config_t::audio_set_clock_cb (*C++ member*), 372
avi_player_config_t::avi_play_end_cb (*C++ member*), 372
avi_player_config_t::buffer_size (*C++ member*), 372
avi_player_config_t::coreID (*C++ member*), 372
avi_player_config_t::priority (*C++ member*), 372
avi_player_config_t::stack_in_psram (*C++ member*), 372
avi_player_config_t::stack_size (*C++ member*), 372
avi_player_config_t::user_data (*C++ member*), 372
avi_player_config_t::video_cb (*C++ member*), 372
avi_player_deinit (*C++ function*), 370
avi_player_get_audio_buffer (*C++ function*), 370
avi_player_get_video_buffer (*C++ function*), 370
avi_player_handle_t (*C++ type*), 373
avi_player_init (*C++ function*), 370
avi_player_play_from_file (*C++ function*), 369
avi_player_play_from_memory (*C++ function*), 369
avi_player_play_stop (*C++ function*), 370
AVOID_CONTINUE_CURRENT_TIME (*C macro*), 522

B

BAS_CHR_BATTERY_INFO_FEATURE_BM_NONE (*C macro*), 78
BAS_CHR_BATTERY_INFO_FEATURE_BM_RECHARGE (*C macro*), 78
BAS_CHR_BATTERY_INFO_FEATURE_BM_REPLACE (*C macro*), 78
BAS_CHR_BATTERY_INFO_FEATURE_BM_RFU (*C macro*), 78
BAS_CHR_BATTERY_INFO_FLAGS_BM_AGGREGATION_GROUP (*C macro*), 78
BAS_CHR_BATTERY_INFO_FLAGS_BM_CHEMISTRY (*C macro*), 78

BAS_CHR_BATTERY_INFO_FLAGS_BM_CRITICAL_ENERGY (*C macro*), 78
BAS_CHR_BATTERY_INFO_FLAGS_BM_DESIGNED_CAPACITY (*C macro*), 78
BAS_CHR_BATTERY_INFO_FLAGS_BM_EXPIRATION_DATE (*C macro*), 78
BAS_CHR_BATTERY_INFO_FLAGS_BM_LOW_ENERGY (*C macro*), 78
BAS_CHR_BATTERY_INFO_FLAGS_BM_MANUFACTURE_DATE (*C macro*), 78
BAS_CHR_BATTERY_INFO_FLAGS_BM_NOMINAL_VOLTAGE (*C macro*), 78
BAS_CHR_BATTERY_INFO_FLAGS_BM_NONE (*C macro*), 78
BAS_CHR_BATTERY_INFO_FLAGS_BM_RFU (*C macro*), 78
BAS_CHR_CRITICAL_STATUS_FLAGS_BM_CRITICAL_POWER_LEVEL (*C macro*), 77
BAS_CHR_CRITICAL_STATUS_FLAGS_BM_IMMEDIATE_SERVICE (*C macro*), 77
BAS_CHR_CRITICAL_STATUS_FLAGS_BM_NONE (*C macro*), 77
BAS_CHR_ENERGY_STATUS_FLAGS_BM_AVAILABLE_BATTERY_CAPACITY (*C macro*), 77
BAS_CHR_ENERGY_STATUS_FLAGS_BM_AVAILABLE_ENERGY (*C macro*), 77
BAS_CHR_ENERGY_STATUS_FLAGS_BM_AVAILABLE_LAST_CHARGE (*C macro*), 77
BAS_CHR_ENERGY_STATUS_FLAGS_BM_CHARGE_RATE (*C macro*), 77
BAS_CHR_ENERGY_STATUS_FLAGS_BM_EXTERNAL_SOURCE_POWER (*C macro*), 77
BAS_CHR_ENERGY_STATUS_FLAGS_BM_NONE (*C macro*), 77
BAS_CHR_ENERGY_STATUS_FLAGS_BM_RFU (*C macro*), 77
BAS_CHR_ENERGY_STATUS_FLAGS_BM_VOLTAGE (*C macro*), 77
BAS_CHR_HEALTH_INFO_FLAGS_BM_CYCLE_COUNT (*C macro*), 78
BAS_CHR_HEALTH_INFO_FLAGS_BM_DESIGNED_CHARGE_LIFE (*C macro*), 78
BAS_CHR_HEALTH_INFO_FLAGS_BM_NONE (*C macro*), 77
BAS_CHR_HEALTH_INFO_FLAGS_BM_RFU (*C macro*), 78
BAS_CHR_HEALTH_STATUS_FLAGS_BM_BATTERY_HEALTH_SUMMARY (*C macro*), 77
BAS_CHR_HEALTH_STATUS_FLAGS_BM_CURRENT_TEMPERATURE (*C macro*), 77
BAS_CHR_HEALTH_STATUS_FLAGS_BM_DEEP_DISCHARGE (*C macro*), 77
BAS_CHR_HEALTH_STATUS_FLAGS_BM_NONE (*C macro*), 77
BAS_CHR_HEALTH_STATUS_FLAGS_BM_RCYCLE_COUNT (*C macro*), 77
BAS_CHR_HEALTH_STATUS_FLAGS_BM_RFU (*C macro*), 77
BAS_CHR_LEVEL_STATUS_ASSITIONAL_STATUS_BATTERY_FA (*C macro*), 77
BAS_CHR_LEVEL_STATUS_ASSITIONAL_STATUS_BATTERY_FA (*C macro*), 76
BAS_CHR_LEVEL_STATUS_ASSITIONAL_STATUS_SERVICE_FA (*C macro*), 76
BAS_CHR_LEVEL_STATUS_ASSITIONAL_STATUS_SERVICE_RF (*C macro*), 76
BAS_CHR_LEVEL_STATUS_ASSITIONAL_STATUS_SERVICE_TR (*C macro*), 76
BAS_CHR_LEVEL_STATUS_ASSITIONAL_STATUS_SERVICE_UN (*C macro*), 76
BAS_CHR_LEVEL_STATUS_FLAGS_BM_ADDITIONAL_STATUS (*C macro*), 75
BAS_CHR_LEVEL_STATUS_FLAGS_BM_BATTERY_LEVEL (*C macro*), 75
BAS_CHR_LEVEL_STATUS_FLAGS_BM_IDENTIFY (*C macro*), 75
BAS_CHR_LEVEL_STATUS_FLAGS_BM_NONE (*C macro*), 75
BAS_CHR_LEVEL_STATUS_FLAGS_BM_RFU (*C macro*), 75
BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_CHARGE_L (*C macro*), 76
BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_CHARGE_L (*C macro*), 76
BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_CHARGE_L (*C macro*), 76
BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_CHARGE_L (*C macro*), 76
BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_CHARGE_S (*C macro*), 76
BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_CHARGE_S (*C macro*), 76
BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_CHARGE_S (*C macro*), 76
BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_NOT (*C macro*), 75
BAS_CHR_LEVEL_STATUS_POWER_STATE_BATTERY_SET (*C macro*), 75
BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_FAULT_BAT (*C macro*), 76
BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_FAULT_EXT (*C macro*), 76
BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_FAULT_NON (*C macro*), 76
BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_FAULT_OTH (*C macro*), 76
BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_TYPE_CURR (*C macro*), 76
BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_TYPE_FLOA (*C macro*), 76
BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_TYPE_RFU (*C macro*), 76
BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_TYPE_TRIC (*C macro*), 76

BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_TYPE_UNKNOWN (*C macro*), 76
BAS_CHR_LEVEL_STATUS_POWER_STATE_CHARGE_TYPE_VOLTAGE (*C macro*), 76
BAS_CHR_LEVEL_STATUS_POWER_STATE_WIRED_EXTERNAL_POWER_SOURCE_CONNECTED (*C macro*), 75
BAS_CHR_LEVEL_STATUS_POWER_STATE_WIRED_EXTERNAL_POWER_SOURCE_NOT_CONNECTED (*C macro*), 75
BAS_CHR_LEVEL_STATUS_POWER_STATE_WIRED_EXTERNAL_POWER_SOURCE_RFU (*C macro*), 75
BAS_CHR_LEVEL_STATUS_POWER_STATE_WIRED_EXTERNAL_POWER_SOURCE_UNKNOWN (*C macro*), 75
BAS_CHR_LEVEL_STATUS_POWER_STATE_WIRELESS_EXTERNAL_POWER_SOURCE_CONNECTED (*C macro*), 75
BAS_CHR_LEVEL_STATUS_POWER_STATE_WIRELESS_EXTERNAL_POWER_SOURCE_NOT_CONNECTED (*C macro*), 75
BAS_CHR_LEVEL_STATUS_POWER_STATE_WIRELESS_EXTERNAL_POWER_SOURCE_RFU (*C macro*), 75
BAS_CHR_LEVEL_STATUS_POWER_STATE_WIRELESS_EXTERNAL_POWER_SOURCE_UNKNOWN (*C macro*), 75
BAS_CHR_TIME_STATUS_FLAGS_BM_DISCHARGED_STANDBY (*C macro*), 77
BAS_CHR_TIME_STATUS_FLAGS_BM_NONE (*C macro*), 77
BAS_CHR_TIME_STATUS_FLAGS_BM_RECHARGE (*C macro*), 77
BAS_CHR_TIME_STATUS_FLAGS_BM_RFU (*C macro*), 77
BASE_SPEED (*C macro*), 523
BASE_VOLTAGE (*C macro*), 523
bldc_control_config_t (*C++ struct*), 519
bldc_control_config_t::alignment_mode (*C++ member*), 519
bldc_control_config_t::control_mode (*C++ member*), 519
bldc_control_config_t::debug_config (*C++ member*), 519
bldc_control_config_t::six_step_config (*C++ member*), 519
bldc_control_config_t::speed_mode (*C++ member*), 519
bldc_control_config_t::zero_cross_comparer_config (*C++ member*), 519
bldc_control_deinit (*C++ function*), 517
bldc_control_event_t (*C++ enum*), 519
bldc_control_event_t::BLDC_CONTROL_ALIGNMENT (*C++ enumerator*), 519
bldc_control_event_t::BLDC_CONTROL_BLOCKED (*C++ enumerator*), 520
bldc_control_event_t::BLDC_CONTROL_CLOSED_LOOP (*C++ enumerator*), 520
bldc_control_event_t::BLDC_CONTROL_DRAG (*C++ enumerator*), 519
bldc_control_event_t::BLDC_CONTROL_START (*C++ enumerator*), 519
bldc_control_event_t::BLDC_CONTROL_STOP (*C++ enumerator*), 519
bldc_control_get_dir (*C++ function*), 518
bldc_control_get_duty (*C++ function*), 518
bldc_control_get_speed_rpm (*C++ function*), 518
bldc_control_handle_t (*C++ type*), 519
bldc_control_init (*C++ function*), 517
bldc_control_set_dir (*C++ function*), 518
bldc_control_set_duty (*C++ function*), 518
bldc_control_set_speed_rpm (*C++ function*), 518
bldc_control_start (*C++ function*), 517
bldc_control_stop (*C++ function*), 518
bldc_debug_config_t (*C++ struct*), 518
bldc_debug_config_t::debug_operation (*C++ member*), 519
bldc_debug_config_t::debug_output (*C++ member*), 519
BLDC_EXTERNAL (*C macro*), 520
BLDC_MCPWM (*C macro*), 520
BLE_ANP_CAT_BM_EMAIL (*C macro*), 116
BLE_ANP_CAT_BM_MISSED_CALL (*C macro*), 116
BLE_ANP_CAT_BM_NEWS (*C macro*), 116
BLE_ANP_CAT_BM_NONE (*C macro*), 116
BLE_ANP_CAT_BM_SCHEDULE (*C macro*), 116
BLE_ANP_CAT_BM_SIMPLE_ALERT (*C macro*), 116
BLE_ANP_CAT_BM_SMS (*C macro*), 116
BLE_ANP_CAT_BM_VOICE_MAIL (*C macro*), 116
BLE_ANP_CAT_ID_CALL (*C macro*), 117
BLE_ANP_CAT_ID_EMAIL (*C macro*), 116
BLE_ANP_CAT_ID_MISSED_CALL (*C macro*), 117
BLE_ANP_CAT_ID_NEWS (*C macro*), 117
BLE_ANP_CAT_ID_SCHEDULE (*C macro*), 117
BLE_ANP_CAT_ID_SIMPLE_ALERT (*C macro*), 116
BLE_ANP_CAT_ID_SMS (*C macro*), 117
BLE_ANP_CAT_ID_VOICE_MAIL (*C macro*), 117
BLE_ANP_CAT_NUM (*C macro*), 117
BLE_ANP_CHR_UUID16_ALERT_NOT_CTRL_PT (*C macro*), 116
BLE_ANP_CHR_UUID16_NEW_ALERT (*C macro*), 116
BLE_ANP_CHR_UUID16_SUP_NEW_ALERT_CAT (*C macro*), 116
BLE_ANP_CHR_UUID16_SUP_UNR_ALERT_CAT (*C macro*), 116
BLE_ANP_CHR_UUID16_UNR_ALERT_STAT (*C macro*), 116
BLE_ANP_CMD_DIS_NEW_ALERT_CAT (*C macro*), 117
BLE_ANP_CMD_DIS_UNR_ALERT_CAT (*C macro*), 117
BLE_ANP_CMD_EN_NEW_ALERT_CAT (*C macro*), 117
BLE_ANP_CMD_EN_UNR_ALERT_CAT (*C macro*), 117
BLE_ANP_CMD_NOT_NEW_ALERT_IMMEDIATE (*C macro*), 117

BLE_ANP_CMD_NOT_UNR_ALERT_IMMEDIATE (*C macro*), 117
BLE_ANP_INFO_STR_MAX_LEN (*C macro*), 117
BLE_ANP_NEW_ALERT_MAX_LEN (*C macro*), 117
BLE_ANP_UUID16 (*C macro*), 116
BLE_ANS_CAT_BM_CALL (*C macro*), 64
BLE_ANS_CAT_BM_EMAIL (*C macro*), 64
BLE_ANS_CAT_BM_MISSED_CALL (*C macro*), 64
BLE_ANS_CAT_BM_NEWS (*C macro*), 64
BLE_ANS_CAT_BM_NONE (*C macro*), 64
BLE_ANS_CAT_BM_SCHEDULE (*C macro*), 64
BLE_ANS_CAT_BM_SIMPLE_ALERT (*C macro*), 64
BLE_ANS_CAT_BM_SMS (*C macro*), 64
BLE_ANS_CAT_BM_VOICE_MAIL (*C macro*), 64
BLE_ANS_CAT_ID_CALL (*C macro*), 64
BLE_ANS_CAT_ID_EMAIL (*C macro*), 64
BLE_ANS_CAT_ID_MISSED_CALL (*C macro*), 64
BLE_ANS_CAT_ID_NEWS (*C macro*), 64
BLE_ANS_CAT_ID_SCHEDULE (*C macro*), 64
BLE_ANS_CAT_ID_SIMPLE_ALERT (*C macro*), 64
BLE_ANS_CAT_ID_SMS (*C macro*), 64
BLE_ANS_CAT_ID_VOICE_MAIL (*C macro*), 64
BLE_ANS_CAT_NUM (*C macro*), 64
BLE_ANS_CHR_UUID16_ALERT_NOT_CTRL_PT (*C macro*), 63
BLE_ANS_CHR_UUID16_NEW_ALERT (*C macro*), 63
BLE_ANS_CHR_UUID16_SUP_NEW_ALERT_CAT (*C macro*), 63
BLE_ANS_CHR_UUID16_SUP_UNR_ALERT_CAT (*C macro*), 63
BLE_ANS_CHR_UUID16_UNR_ALERT_STAT (*C macro*), 63
BLE_ANS_CMD_DIS_NEW_ALERT_CAT (*C macro*), 64
BLE_ANS_CMD_DIS_UNR_ALERT_CAT (*C macro*), 64
BLE_ANS_CMD_EN_NEW_ALERT_CAT (*C macro*), 64
BLE_ANS_CMD_EN_UNR_ALERT_CAT (*C macro*), 64
BLE_ANS_CMD_NOT_NEW_ALERT_IMMEDIATE (*C macro*), 64
BLE_ANS_CMD_NOT_UNR_ALERT_IMMEDIATE (*C macro*), 65
BLE_ANS_INFO_STR_MAX_LEN (*C macro*), 65
BLE_ANS_NEW_ALERT_MAX_LEN (*C macro*), 65
BLE_ANS_UUID16 (*C macro*), 63
BLE_BAS_CHR_UUID16_BATTERY_INFO (*C macro*), 75
BLE_BAS_CHR_UUID16_CRITICAL_STATUS (*C macro*), 75
BLE_BAS_CHR_UUID16_ENERGY_STATUS (*C macro*), 75
BLE_BAS_CHR_UUID16_ESTIMATED_SERVICE_DATE (*C macro*), 75
BLE_BAS_CHR_UUID16_HEALTH_INFO (*C macro*), 75
BLE_BAS_CHR_UUID16_HEALTH_STATUS (*C macro*), 75
BLE_BAS_CHR_UUID16_LEVEL (*C macro*), 74
BLE_BAS_CHR_UUID16_LEVEL_STATUS (*C macro*), 75
BLE_BAS_CHR_UUID16_TIME_STATUS (*C macro*), 75
BLE_BAS_UUID16 (*C macro*), 74
BLE_BCS_CHR_UUID16_FEATURE (*C macro*), 80
BLE_BCS_CHR_UUID16_MEASUREMENT (*C macro*), 80
BLE_BCS_FEAT_BASAL_METABOLISM (*C macro*), 80
BLE_BCS_FEAT_BODY_WATER_MASS (*C macro*), 81
BLE_BCS_FEAT_FAT_FREE_MASS (*C macro*), 81
BLE_BCS_FEAT_HEIGHT (*C macro*), 81
BLE_BCS_FEAT_HEIGHT_RESOLUTION (*C macro*), 81
BLE_BCS_FEAT_IMPEDENCE (*C macro*), 81
BLE_BCS_FEAT_MASS_MEASUREMENT_RESOLUTION (*C macro*), 81
BLE_BCS_FEAT_MULTI_USER (*C macro*), 80
BLE_BCS_FEAT_MUSCLE_MASS (*C macro*), 81
BLE_BCS_FEAT_MUSCLE_PERCENTAGE (*C macro*), 80
BLE_BCS_FEAT_SOFT_LEAN_MASS (*C macro*), 81
BLE_BCS_FEAT_TIME_STAMP (*C macro*), 80
BLE_BCS_FEAT_WEIGHT (*C macro*), 81
BLE_BCS_FLAG_BASAL_METABOLISM (*C macro*), 81
BLE_BCS_FLAG_BODY_WATER_MASS (*C macro*), 81
BLE_BCS_FLAG_FAT_FREE_MASS (*C macro*), 81
BLE_BCS_FLAG_HEIGHT (*C macro*), 81
BLE_BCS_FLAG_IMPEDENCE (*C macro*), 81
BLE_BCS_FLAG_MEASUREMENT_UNITS (*C macro*), 81
BLE_BCS_FLAG_MULTI_USER (*C macro*), 81
BLE_BCS_FLAG_MULTIPLE_PACKET (*C macro*), 81
BLE_BCS_FLAG_MUSCLE_MASS (*C macro*), 81
BLE_BCS_FLAG_MUSCLE_PERCENTAGE (*C macro*), 81
BLE_BCS_FLAG_SOFT_LEAN_MASS (*C macro*), 81
BLE_BCS_FLAG_TIME_STAMP (*C macro*), 81
BLE_BCS_FLAG_WEIGHT (*C macro*), 81
BLE_BCS_MAX_VAL_LEN (*C macro*), 80
BLE_BCS_UUID16 (*C macro*), 80
BLE_CONN_GATT_CHR_BROADCAST (*C macro*), 54
BLE_CONN_GATT_CHR_INDICATE (*C macro*), 54
BLE_CONN_GATT_CHR_NOTIFY (*C macro*), 54
BLE_CONN_GATT_CHR_READ (*C macro*), 54
BLE_CONN_GATT_CHR_READ_AUTHEN (*C macro*), 54
BLE_CONN_GATT_CHR_READ_AUTHOR (*C macro*), 54
BLE_CONN_GATT_CHR_READ_ENC (*C macro*), 54

BLE_CONN_GATT_CHR_WRITE (*C macro*), 54
BLE_CONN_GATT_CHR_WRITE_AUTHEN (*C macro*), 55
BLE_CONN_GATT_CHR_WRITE_AUTHOR (*C macro*), 55
BLE_CONN_GATT_CHR_WRITE_ENC (*C macro*), 55
BLE_CONN_GATT_CHR_WRITE_NO_RSP (*C macro*), 54
BLE_CONN_HANDLE_INVALID (*C macro*), 52
BLE_CONN_HANDLE_MAX (*C macro*), 52
BLE_CONN_MGR_ADDR_HEX (*C macro*), 51
BLE_CONN_MGR_ADDR_STR (*C macro*), 51
BLE_CTS_CHANGE_OF_DST_MASK (*C macro*), 84
BLE_CTS_CHANGE_OF_TIME_ZONE_MASK (*C macro*), 84
BLE_CTS_CHR_UUID16_CURRENT_TIME (*C macro*), 84
BLE_CTS_CHR_UUID16_LOCAL_TIME (*C macro*), 84
BLE_CTS_CHR_UUID16_REFERENCE_TIME (*C macro*), 84
BLE_CTS_EXTERNAL_REFERENCE_TIME_UPDATE_MASK (*C macro*), 84
BLE_CTS_MANUAL_TIME_UPDATE_MASK (*C macro*), 84
BLE_CTS_UUID16 (*C macro*), 84
BLE_DIS_CHR_UUID16_FIRMWARE_REVISION (*C macro*), 88
BLE_DIS_CHR_UUID16_HARDWARE_REVISION (*C macro*), 88
BLE_DIS_CHR_UUID16_MANUFACTURER_NAME (*C macro*), 88
BLE_DIS_CHR_UUID16_MODEL_NUMBER (*C macro*), 88
BLE_DIS_CHR_UUID16_PNP_ID (*C macro*), 88
BLE_DIS_CHR_UUID16_REG_CERT (*C macro*), 88
BLE_DIS_CHR_UUID16_SERIAL_NUMBER (*C macro*), 88
BLE_DIS_CHR_UUID16_SOFTWARE_REVISION (*C macro*), 88
BLE_DIS_CHR_UUID16_SYSTEM_ID (*C macro*), 88
BLE_DIS_UUID16 (*C macro*), 88
ble_hci_add_to_accept_list (*C++ function*), 148
BLE_HCI_ADDR_LEN (*C macro*), 150
ble_hci_addr_t (*C++ type*), 150
ble_hci_addr_type_t (*C++ enum*), 151
ble_hci_addr_type_t::BLE_ADDR_TYPE_PUBLIC (*C++ enumerator*), 151
ble_hci_addr_type_t::BLE_ADDR_TYPE_RANDOM (*C++ enumerator*), 151
ble_hci_addr_type_t::BLE_ADDR_TYPE_RPA_PUBLIC (*C++ enumerator*), 151
ble_hci_addr_type_t::BLE_ADDR_TYPE_RPA_RANDOM (*C++ enumerator*), 151
ble_hci_adv_channel_t (*C++ enum*), 152
ble_hci_adv_channel_t::ADV_CHNL_37 (*C++ enumerator*), 152
ble_hci_adv_channel_t::ADV_CHNL_38 (*C++ enumerator*), 152
ble_hci_adv_channel_t::ADV_CHNL_39 (*C++ enumerator*), 152
ble_hci_adv_channel_t::ADV_CHNL_ALL (*C++ enumerator*), 152
ble_hci_adv_filter_t (*C++ enum*), 152
ble_hci_adv_filter_t::ADV_FILTER_ALLOW_SCAN_ANY_C (*C++ enumerator*), 152
ble_hci_adv_filter_t::ADV_FILTER_ALLOW_SCAN_ANY_C (*C++ enumerator*), 153
ble_hci_adv_filter_t::ADV_FILTER_ALLOW_SCAN_WLST_ (*C++ enumerator*), 153
ble_hci_adv_filter_t::ADV_FILTER_ALLOW_SCAN_WLST_ (*C++ enumerator*), 153
ble_hci_adv_param_t (*C++ struct*), 149
ble_hci_adv_param_t::adv_filter_policy (*C++ member*), 150
ble_hci_adv_param_t::adv_int_max (*C++ member*), 149
ble_hci_adv_param_t::adv_int_min (*C++ member*), 149
ble_hci_adv_param_t::adv_type (*C++ member*), 149
ble_hci_adv_param_t::channel_map (*C++ member*), 149
ble_hci_adv_param_t::own_addr_type (*C++ member*), 149
ble_hci_adv_param_t::peer_addr (*C++ member*), 149
ble_hci_adv_param_t::peer_addr_type (*C++ member*), 149
ble_hci_adv_type_t (*C++ enum*), 152
ble_hci_adv_type_t::ADV_TYPE_DIRECT_IND_HIGH (*C++ enumerator*), 152
ble_hci_adv_type_t::ADV_TYPE_DIRECT_IND_LOW (*C++ enumerator*), 152
ble_hci_adv_type_t::ADV_TYPE_IND (*C++ enumerator*), 152
ble_hci_adv_type_t::ADV_TYPE_NONCONN_IND (*C++ enumerator*), 152
ble_hci_adv_type_t::ADV_TYPE_SCAN_IND (*C++ enumerator*), 152
ble_hci_clear_accept_list (*C++ function*), 148
ble_hci_deinit (*C++ function*), 147
ble_hci_dev_type_t (*C++ enum*), 151
ble_hci_dev_type_t::BLE_HCI_DEVICE_TYPE_BLE (*C++ enumerator*), 152
ble_hci_dev_type_t::BLE_HCI_DEVICE_TYPE_BREDR (*C++ enumerator*), 151
ble_hci_dev_type_t::BLE_HCI_DEVICE_TYPE_DUMO (*C++ enumerator*), 152
ble_hci_enable_meta_event (*C++ function*), 147
ble_hci_init (*C++ function*), 147
ble_hci_reset (*C++ function*), 147

ble_hci_scan_cb_t (*C++ type*), 150
ble_hci_scan_param_t (*C++ struct*), 150
ble_hci_scan_param_t::filter_policy (*C++ member*), 150
ble_hci_scan_param_t::own_addr_type (*C++ member*), 150
ble_hci_scan_param_t::scan_interval (*C++ member*), 150
ble_hci_scan_param_t::scan_type (*C++ member*), 150
ble_hci_scan_param_t::scan_window (*C++ member*), 150
ble_hci_scan_result_t (*C++ struct*), 148
ble_hci_scan_result_t::adv_data_len (*C++ member*), 149
ble_hci_scan_result_t::bda (*C++ member*), 149
ble_hci_scan_result_t::ble_addr_type (*C++ member*), 149
ble_hci_scan_result_t::ble_adv (*C++ member*), 149
ble_hci_scan_result_t::dev_type (*C++ member*), 149
ble_hci_scan_result_t::rssi (*C++ member*), 149
ble_hci_scan_result_t::scan_rsp_len (*C++ member*), 149
ble_hci_scan_result_t::search_evt (*C++ member*), 148
ble_hci_scan_type_t (*C++ enum*), 152
ble_hci_scan_type_t::BLE_SCAN_TYPE_ACTIVE (*C++ enumerator*), 152
ble_hci_scan_type_t::BLE_SCAN_TYPE_PASSIVE (*C++ enumerator*), 152
ble_hci_search_evt_t (*C++ enum*), 151
ble_hci_search_evt_t::BLE_HCI_SEARCH_DISCOVERY_COMPL (*C++ enumerator*), 151
ble_hci_search_evt_t::BLE_HCI_SEARCH_DISCOVERY_RESULT (*C++ enumerator*), 151
ble_hci_search_evt_t::BLE_HCI_SEARCH_DISCOVERY_START (*C++ enumerator*), 151
ble_hci_search_evt_t::BLE_HCI_SEARCH_DISCOVERY_STOP (*C++ enumerator*), 151
ble_hci_search_evt_t::BLE_HCI_SEARCH_INQUIRY_COMPL (*C++ enumerator*), 151
ble_hci_search_evt_t::BLE_HCI_SEARCH_INQUIRY_RESULT (*C++ enumerator*), 151
ble_hci_search_evt_t::BLE_HCI_SEARCH_INQUIRY_START (*C++ enumerator*), 151
ble_hci_search_evt_t::BLE_HCI_SEARCH_INQUIRY_STOP (*C++ enumerator*), 151
ble_hci_search_evt_t::BLE_HCI_SEARCH_SEARCH_COMPL (*C++ enumerator*), 151
ble_hci_set_adv_data (*C++ function*), 147
ble_hci_set_adv_enable (*C++ function*), 147
ble_hci_set_adv_param (*C++ function*), 147
ble_hci_set_random_address (*C++ function*), 148
ble_hci_set_register_scan_callback (*C++ function*), 148
ble_hci_set_scan_enable (*C++ function*), 148
ble_hci_set_scan_param (*C++ function*), 147
BLE_HRP_CHR_MERSUREMENT_FLAGS_FORMAT_U16 (*C macro*), 125
BLE_HRP_CHR_MERSUREMENT_FLAGS_FORMAT_U8 (*C macro*), 125
BLE_HRP_CHR_MERSUREMENT_FLAGS_NOT (*C macro*), 125
BLE_HRP_CHR_MERSUREMENT_FLAGS_SET (*C macro*), 125
BLE_HRP_CHR_MERSUREMENT_RR_INTERVAL_MAX_NUM (*C macro*), 125
BLE_HRP_CHR_UUID16_BODY_SENSOR_LOC (*C macro*), 125
BLE_HRP_CHR_UUID16_HEART_RATE_CNTL_POINT (*C macro*), 125
BLE_HRP_CHR_UUID16_MEASUREMENT (*C macro*), 125
BLE_HRP_CMD_MAX (*C macro*), 125
BLE_HRP_CMD_RESET_ENERGY_EXPENDED (*C macro*), 125
BLE_HRP_CMD_RFU (*C macro*), 125
BLE_HRP_FLAGS_BM_ENERGY (*C macro*), 125
BLE_HRP_FLAGS_BM_FORMAT (*C macro*), 125
BLE_HRP_FLAGS_BM_NONE (*C macro*), 125
BLE_HRP_FLAGS_BM_RFU (*C macro*), 125
BLE_HRP_FLAGS_BM_RR_INTERVAL (*C macro*), 125
BLE_HRP_FLAGS_BM_SENSOR_CONTACT_DETECTED (*C macro*), 125
BLE_HRP_FLAGS_BM_SENSOR_CONTACT_SUPPOTRED (*C macro*), 125
BLE_HRP_UUID16 (*C macro*), 125
BLE_HRS_CHR_BODY_SENSOR_LOC_CHEST (*C macro*), 91
BLE_HRS_CHR_BODY_SENSOR_LOC_EAR_LOBE (*C macro*), 91
BLE_HRS_CHR_BODY_SENSOR_LOC_FINGER (*C macro*), 91
BLE_HRS_CHR_BODY_SENSOR_LOC_FOOT (*C macro*), 91
BLE_HRS_CHR_BODY_SENSOR_LOC_HAND (*C macro*), 91
BLE_HRS_CHR_BODY_SENSOR_LOC_MAX (*C macro*), 91
BLE_HRS_CHR_BODY_SENSOR_LOC_OTHER (*C macro*), 91
BLE_HRS_CHR_BODY_SENSOR_LOC_RFU (*C macro*), 91
BLE_HRS_CHR_BODY_SENSOR_LOC_WRIST (*C macro*), 91
BLE_HRS_CHR_MERSUREMENT_RR_INTERVAL_MAX_NUM (*C macro*), 91
BLE_HRS_CHR_UUID16_BODY_SENSOR_LOC (*C macro*), 91
BLE_HRS_CHR_UUID16_HEART_RATE_CNTL_POINT (*C macro*), 91
BLE_HRS_CHR_UUID16_MEASUREMENT (*C

macro), 91
BLE_HRS_UUID16 (*C macro*), 91
BLE_HTP_CHR_TEMPERATURE_FLAGS_NOT (*C macro*), 128
BLE_HTP_CHR_TEMPERATURE_FLAGS_SET (*C macro*), 128
BLE_HTP_CHR_TEMPERATURE_TYPE_ARMPIT (*C macro*), 128
BLE_HTP_CHR_TEMPERATURE_TYPE_BODY (*C macro*), 128
BLE_HTP_CHR_TEMPERATURE_TYPE_EAR (*C macro*), 128
BLE_HTP_CHR_TEMPERATURE_TYPE_FINGER (*C macro*), 128
BLE_HTP_CHR_TEMPERATURE_TYPE_GAST_TRACT (*C macro*), 128
BLE_HTP_CHR_TEMPERATURE_TYPE_MAX (*C macro*), 129
BLE_HTP_CHR_TEMPERATURE_TYPE_MOUTH (*C macro*), 128
BLE_HTP_CHR_TEMPERATURE_TYPE_RECTUM (*C macro*), 128
BLE_HTP_CHR_TEMPERATURE_TYPE_RFU (*C macro*), 128
BLE_HTP_CHR_TEMPERATURE_TYPE_TOE (*C macro*), 128
BLE_HTP_CHR_TEMPERATURE_TYPE_TYMPANUM (*C macro*), 128
BLE_HTP_CHR_TEMPERATURE_UNITS_CELSIUS (*C macro*), 128
BLE_HTP_CHR_TEMPERATURE_UNITS_FAHRENHEIT (*C macro*), 128
BLE_HTP_CHR_UUID16_INTERMEDIATE_TEMPERATURE (*C macro*), 128
BLE_HTP_CHR_UUID16_MEASUREMENT_INTERVAL (*C macro*), 128
BLE_HTP_CHR_UUID16_TEMPERATURE_MEASUREMENT (*C macro*), 128
BLE_HTP_CHR_UUID16_TEMPERATURE_TYPE (*C macro*), 128
BLE_HTP_FLAGS_BM_NONE (*C macro*), 128
BLE_HTP_FLAGS_BM_RFU (*C macro*), 128
BLE_HTP_FLAGS_BM_TEMPERATURE_TYPE (*C macro*), 128
BLE_HTP_FLAGS_BM_TEMPERATURE_UNITS (*C macro*), 128
BLE_HTP_FLAGS_BM_TIME_STAMP (*C macro*), 128
BLE_HTP_UUID16 (*C macro*), 128
BLE_HTS_CHR_TEMPERATURE_FLAGS_NOT (*C macro*), 94
BLE_HTS_CHR_TEMPERATURE_FLAGS_SET (*C macro*), 94
BLE_HTS_CHR_TEMPERATURE_TYPE_ARMPIT (*C macro*), 94
BLE_HTS_CHR_TEMPERATURE_TYPE_BODY (*C macro*), 94
BLE_HTS_CHR_TEMPERATURE_TYPE_EAR (*C macro*), 94
BLE_HTS_CHR_TEMPERATURE_TYPE_FINGER (*C macro*), 94
BLE_HTS_CHR_TEMPERATURE_TYPE_GAST_TRACT (*C macro*), 94
BLE_HTS_CHR_TEMPERATURE_TYPE_MAX (*C macro*), 95
BLE_HTS_CHR_TEMPERATURE_TYPE_MOUTH (*C macro*), 94
BLE_HTS_CHR_TEMPERATURE_TYPE_RECTUM (*C macro*), 94
BLE_HTS_CHR_TEMPERATURE_TYPE_RFU (*C macro*), 94
BLE_HTS_CHR_TEMPERATURE_TYPE_TOE (*C macro*), 95
BLE_HTS_CHR_TEMPERATURE_TYPE_TYMPANUM (*C macro*), 95
BLE_HTS_CHR_TEMPERATURE_UNITS_CELSIUS (*C macro*), 94
BLE_HTS_CHR_TEMPERATURE_UNITS_FAHRENHEIT (*C macro*), 94
BLE_HTS_CHR_UUID16_INTERMEDIATE_TEMPERATURE (*C macro*), 94
BLE_HTS_CHR_UUID16_MEASUREMENT_INTERVAL (*C macro*), 94
BLE_HTS_CHR_UUID16_TEMPERATURE_MEASUREMENT (*C macro*), 94
BLE_HTS_CHR_UUID16_TEMPERATURE_TYPE (*C macro*), 94
BLE_HTS_UUID16 (*C macro*), 94
BLE_MIDI_CHAR_UUID128 (*C macro*), 122
BLE_MIDI_MAX_PKT_LEN (*C macro*), 122
BLE_MIDI_SERVICE_UUID128 (*C macro*), 122
BLE_OTP_EVENT_DATA_MAX_LEN (*C macro*), 144
BLE_OTP_EVENT_OACP_ABORTED (*C macro*), 144
BLE_OTP_EVENT_OACP_RESPONSE (*C macro*), 145
BLE_OTP_EVENT_OACP_STARTED (*C macro*), 144
BLE_OTP_EVENT_OBJECT_CHANGED (*C macro*), 144
BLE_OTP_EVENT_OBJECT_SELECTED (*C macro*), 144
BLE_OTP_EVENT_OLCP_RESPONSE (*C macro*), 145
BLE_OTP_EVENT_OTS_DISCOVERED (*C macro*), 144
BLE_OTP_EVENT_OTS_DISCOVERY_FAILED (*C macro*), 144
BLE_OTP_EVENT_TRANSFER_CHANNEL_CONNECTED (*C macro*), 145
BLE_OTP_EVENT_TRANSFER_CHANNEL_DISCONNECTED (*C macro*), 145
BLE_OTP_EVENT_TRANSFER_COMPLETE (*C macro*), 145
BLE_OTP_EVENT_TRANSFER_DATA_RECEIVED (*C macro*), 145
BLE_OTP_EVENT_TRANSFER_DATA_SENT (*C macro*), 145

BLE_OTP_EVENT_TRANSFER_EOF (*C macro*), 145
BLE_OTP_EVENT_TRANSFER_ERROR (*C macro*), 145
BLE_OTP_L2CAP_COC_MTU_DEFAULT (*C macro*), 144
BLE_OTP_L2CAP_COC_MTU_MIN (*C macro*), 144
BLE_OTP_OBJECT_SIZE_UNKNOWN (*C macro*), 144
BLE_OTP_PSM_DEFAULT (*C macro*), 144
BLE_OTS_ATT_VAL_LEN (*C macro*), 102
BLE_OTS_CHR_UUID16_OBJECT_ACTION_CONTROL_POINT (*C macro*), 103
BLE_OTS_CHR_UUID16_OBJECT_CHANGED (*C macro*), 103
BLE_OTS_CHR_UUID16_OBJECT_FIRST_CREATED (*C macro*), 103
BLE_OTS_CHR_UUID16_OBJECT_ID (*C macro*), 103
BLE_OTS_CHR_UUID16_OBJECT_LAST_MODIFIED (*C macro*), 103
BLE_OTS_CHR_UUID16_OBJECT_LIST_CONTROL_POINT (*C macro*), 103
BLE_OTS_CHR_UUID16_OBJECT_LIST_FILTER (*C macro*), 103
BLE_OTS_CHR_UUID16_OBJECT_NAME (*C macro*), 103
BLE_OTS_CHR_UUID16_OBJECT_PROP (*C macro*), 103
BLE_OTS_CHR_UUID16_OBJECT_SIZE (*C macro*), 103
BLE_OTS_CHR_UUID16_OBJECT_TYPE (*C macro*), 103
BLE_OTS_CHR_UUID16_OTS_FEATURE (*C macro*), 103
BLE_OTS_FILTER_ALLOC_SIZE (*C macro*), 105
BLE_OTS_FILTER_CREATE_TIME (*C macro*), 105
BLE_OTS_FILTER_CURRENT_SIZE (*C macro*), 105
BLE_OTS_FILTER_MARK_OBJECT (*C macro*), 105
BLE_OTS_FILTER_MODIFY_TIME (*C macro*), 105
BLE_OTS_FILTER_NAME_CONTAIN (*C macro*), 105
BLE_OTS_FILTER_NAME_END (*C macro*), 105
BLE_OTS_FILTER_NAME_EXACTLY (*C macro*), 105
BLE_OTS_FILTER_NAME_START (*C macro*), 105
BLE_OTS_FILTER_OBJ_TYPE (*C macro*), 105
BLE_OTS_OACP_ABORT (*C macro*), 103
BLE_OTS_OACP_CALCULATE_CHECKSUM (*C macro*), 103
BLE_OTS_OACP_CREATE (*C macro*), 103
BLE_OTS_OACP_DELETE (*C macro*), 103
BLE_OTS_OACP_EXECUTE (*C macro*), 103
BLE_OTS_OACP_READ (*C macro*), 103
BLE_OTS_OACP_RESPONSE (*C macro*), 103
BLE_OTS_OACP_RSP_CHANNEL_UNAVAILABLE (*C macro*), 104
BLE_OTS_OACP_RSP_INSUFFICIENT_RESOURCES (*C macro*), 104
BLE_OTS_OACP_RSP_INVALID_OBJECT (*C macro*), 104
BLE_OTS_OACP_RSP_INVALID_PARAMETER (*C macro*), 103
BLE_OTS_OACP_RSP_NOT_PERMIT (*C macro*), 104
BLE_OTS_OACP_RSP_NOT_SUPPORT (*C macro*), 103
BLE_OTS_OACP_RSP_OBJECT_LOCK (*C macro*), 104
BLE_OTS_OACP_RSP_OPERATION_FAIL (*C macro*), 104
BLE_OTS_OACP_RSP_SUCCESS (*C macro*), 103
BLE_OTS_OACP_RSP_UNSUPPORT_TYPE (*C macro*), 104
BLE_OTS_OACP_WRITE (*C macro*), 103
BLE_OTS_OBJ_CHG_CONTENT (*C macro*), 105
BLE_OTS_OBJ_CHG_CREATION (*C macro*), 106
BLE_OTS_OBJ_CHG_DELETION (*C macro*), 106
BLE_OTS_OBJ_CHG_METADATA (*C macro*), 106
BLE_OTS_OBJ_CHG_SRC (*C macro*), 105
BLE_OTS_OLCP_CLEAR_MARK (*C macro*), 104
BLE_OTS_OLCP_FIRST (*C macro*), 104
BLE_OTS_OLCP_GO_TO (*C macro*), 104
BLE_OTS_OLCP_LAST (*C macro*), 104
BLE_OTS_OLCP_NEXT (*C macro*), 104
BLE_OTS_OLCP_ORDER (*C macro*), 104
BLE_OTS_OLCP_PREVIOUS (*C macro*), 104
BLE_OTS_OLCP_REQ_NUM_OF_OBJ (*C macro*), 104
BLE_OTS_OLCP_RESPONSE (*C macro*), 104
BLE_OTS_OLCP_RSP_INVALID_PARAMETER (*C macro*), 105
BLE_OTS_OLCP_RSP_NO_OBJECT (*C macro*), 105
BLE_OTS_OLCP_RSP_NOT_SUPPORT (*C macro*), 105
BLE_OTS_OLCP_RSP_OBJECT_ID_NOT_FOUND (*C macro*), 105
BLE_OTS_OLCP_RSP_OPERATION_FAIL (*C macro*), 105
BLE_OTS_OLCP_RSP_OUT_OF_BOUNDS (*C macro*), 105
BLE_OTS_OLCP_RSP_SUCCESS (*C macro*), 105
BLE_OTS_OLCP_RSP_TOO_MANY_OBJECTS (*C macro*), 105
BLE_OTS_OLCP_SORT_BY_FIRST_CREATE_TIME_ASCENDING (*C macro*), 104
BLE_OTS_OLCP_SORT_BY_FIRST_CREATE_TIME_DESCENDING (*C macro*), 105
BLE_OTS_OLCP_SORT_BY_LAST_MODIFY_TIME_ASCENDING (*C macro*), 104
BLE_OTS_OLCP_SORT_BY_LAST_MODIFY_TIME_DESCENDING (*C macro*), 105
BLE_OTS_OLCP_SORT_BY_NAME_ASCENDING (*C macro*), 104
BLE_OTS_OLCP_SORT_BY_NAME_DESCENDING (*C macro*), 104

BLE_OTS_OLCP_SORT_BY_SIZE_ASCENDING (*C macro*), 104
BLE_OTS_OLCP_SORT_BY_SIZE_DESCENDING (*C macro*), 105
BLE_OTS_OLCP_SORT_BY_TYPE_ASCENDING (*C macro*), 104
BLE_OTS_OLCP_SORT_BY_TYPE_DESCENDING (*C macro*), 104
BLE_OTS_UUID16 (*C macro*), 102
BLE_TPS_CHR_UUID16_TX_POWER_LEVEL (*C macro*), 106
BLE_TPS_UUID16 (*C macro*), 106
BLE_UDS_ATT_VAL_LEN (*C macro*), 109
BLE_UDS_CHR_UUID16_DATABASH_CHG (*C macro*), 109
BLE_UDS_CHR_UUID16_REG_USER (*C macro*), 109
BLE_UDS_CHR_UUID16_USER_CTRL (*C macro*), 109
BLE_UDS_CHR_UUID16_USER_INDEX (*C macro*), 109
BLE_UDS_USER_CTRL_CONSENT_OP (*C macro*), 110
BLE_UDS_USER_CTRL_DEL_DATA_OP (*C macro*), 110
BLE_UDS_USER_CTRL_DEL_USER_OP (*C macro*), 110
BLE_UDS_USER_CTRL_LIST_ALL_OP (*C macro*), 110
BLE_UDS_USER_CTRL_POINT_PARAM_LEN (*C macro*), 109
BLE_UDS_USER_CTRL_REG_NEW_OP (*C macro*), 109
BLE_UDS_USER_CTRL_RSP_OP (*C macro*), 110
BLE_UDS_UUID16 (*C macro*), 109
BLE_UUID128_VAL_LEN (*C macro*), 54
BLE_UUID_CMP (*C macro*), 55
BLE_UUID_TYPE (*C macro*), 55
BLE_WSS_BMI_FLAG (*C macro*), 113
BLE_WSS_BMI_MASK (*C macro*), 113
BLE_WSS_CHR_UUID16_WEIGHT_FEATURE (*C macro*), 112
BLE_WSS_CHR_UUID16_WEIGHT_MEASUREMENT (*C macro*), 112
BLE_WSS_HEIGHT_RESOLUTION_0P001_M (*C macro*), 113
BLE_WSS_HEIGHT_RESOLUTION_0P005_M (*C macro*), 113
BLE_WSS_HEIGHT_RESOLUTION_0P01_M (*C macro*), 113
BLE_WSS_HEIGHT_RESOLUTION_NONE (*C macro*), 113
BLE_WSS_HEIGHT_RESOLUTION_OFSET (*C macro*), 113
BLE_WSS_MEASUREMENT_UINTS_FLAG (*C macro*), 113
BLE_WSS_MULTI_USER_MASK (*C macro*), 113
BLE_WSS_TIME_STAMP_FLAG (*C macro*), 113
BLE_WSS_TIMESTAMP_MASK (*C macro*), 112
BLE_WSS_USER_ID_FLAG (*C macro*), 113
BLE_WSS_UUID16 (*C macro*), 112
BLE_WSS_WEIGHT_RESOLUTION_0P005_KG (*C macro*), 113
BLE_WSS_WEIGHT_RESOLUTION_0P01_KG (*C macro*), 113
BLE_WSS_WEIGHT_RESOLUTION_0P02_KG (*C macro*), 113
BLE_WSS_WEIGHT_RESOLUTION_0P05_KG (*C macro*), 113
BLE_WSS_WEIGHT_RESOLUTION_0P1_KG (*C macro*), 113
BLE_WSS_WEIGHT_RESOLUTION_0P2_KG (*C macro*), 113
BLE_WSS_WEIGHT_RESOLUTION_0P5_KG (*C macro*), 113
BLE_WSS_WEIGHT_RESOLUTION_NONE (*C macro*), 113
BLE_WSS_WEIGHT_RESOLUTION_OFSET (*C macro*), 113
BROADCAST_PARAM_LEN (*C macro*), 54
bthome_bin_sensor_id_t (*C++ enum*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_BATT (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_BATT (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_CARB (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_COLD (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_CONN (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_DOOR (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_GARA (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_GAS (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_GENE (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_HEAT (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_LIGH (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_LOCK (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_MOIS (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_MOTI (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_MOVI (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_OCCU (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_OPEN (*C++ enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_PLUG

(C++ *enumerator*), 161
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_POWER (C++ *enumerator*), 160
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_PRESENCE (C++ *enumerator*), 161
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_PROBLEM (C++ *enumerator*), 161
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_RUNNING (C++ *enumerator*), 161
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_SAFETY (C++ *enumerator*), 161
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_SMOKE (C++ *enumerator*), 161
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_SOUND (C++ *enumerator*), 161
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_TAMPER (C++ *enumerator*), 161
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_VIBRATION (C++ *enumerator*), 161
bthome_bin_sensor_id_t::BTHOME_BIN_SENSOR_ID_WINDOW (C++ *enumerator*), 161
bthome_callbacks_t (C++ *struct*), 157
bthome_callbacks_t::load (C++ *member*), 157
bthome_callbacks_t::store (C++ *member*), 157
bthome_create (C++ *function*), 154
bthome_delete (C++ *function*), 154
bthome_device_info_t (C++ *union*), 156
bthome_device_info_t::all (C++ *member*), 156
bthome_device_info_t::bit (C++ *member*), 156
bthome_device_info_t::bthome_version (C++ *member*), 156
bthome_device_info_t::encryption_flag (C++ *member*), 156
bthome_device_info_t::reserved1 (C++ *member*), 156
bthome_device_info_t::reserved2 (C++ *member*), 156
bthome_device_info_t::trigger_based_flag (C++ *member*), 156
bthome_event_id_t (C++ *enum*), 161
bthome_event_id_t::BTHOME_EVENT_ID_BUTTON (C++ *enumerator*), 161
bthome_event_id_t::BTHOME_EVENT_ID_DIMMER (C++ *enumerator*), 161
bthome_free_reports (C++ *function*), 155
bthome_handle_t (C++ *type*), 157
bthome_load_func_t (C++ *type*), 157
bthome_load_params (C++ *function*), 155
bthome_make_adv_data (C++ *function*), 155
bthome_parse_adv_data (C++ *function*), 155
bthome_payload_add_sensor_data (C++ *function*), 155
bthome_payload_adv_add_bin_sensor_data (C++ *function*), 155
bthome_payload_adv_add_evt_data (C++ *function*), 155
bthome_register_callbacks (C++ *function*), 154
bthome_report_t (C++ *struct*), 156
bthome_report_t::data (C++ *member*), 157
bthome_report_t::id (C++ *member*), 157
bthome_report_t::len (C++ *member*), 157
BTHOME_REPORTS_MAX (C *macro*), 157
bthome_reports_t (C++ *struct*), 157
bthome_reports_t::num_reports (C++ *member*), 157
bthome_reports_t::report (C++ *member*), 157
bthome_sensor_id_t (C++ *enum*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_ACCELERATION (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_BATTERY (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_CO2 (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_COUNT (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_COUNT2 (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_COUNT4 (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_CURRENT (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_DEWPOINT (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_DISTANCE (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_DISTANCEM (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_DURATION (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_ENERGY (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_ENERGY4 (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_GAS (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_GAS4 (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_GYROSCOPE (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_HUMIDITY (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_HUMIDITY_PRE (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_ILLUMINANCE (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_MASS (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_MASSLB (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_MOISTURE

(C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_MOISTURE (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_PACKET (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_PM10 (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_PM25 (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_POWER (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_PRESSURE (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_RAW (C++ *enumerator*), 160
bthome_sensor_id_t::BTHOME_SENSOR_ID_ROTATION (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_SPD (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_TEMPERATURE (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_TEMPERATURE_PRECISE (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_TEXT (C++ *enumerator*), 160
bthome_sensor_id_t::BTHOME_SENSOR_ID_TIMESTAMP (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_TVOC (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_UV (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_VOLTAGE (C++ *enumerator*), 158
bthome_sensor_id_t::BTHOME_SENSOR_ID_VOLTAGE1 (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_VOLUME (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_VOLUME1 (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_VOLUME2 (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_VOLUME_STORAGE (C++ *enumerator*), 160
bthome_sensor_id_t::BTHOME_SENSOR_ID_VOLUMEFR (C++ *enumerator*), 159
bthome_sensor_id_t::BTHOME_SENSOR_ID_WATER (C++ *enumerator*), 159
bthome_set_encrypt_key (C++ *function*), 154
bthome_set_local_mac_addr (C++ *function*), 154
bthome_set_peer_mac_addr (C++ *function*), 154
bthome_store_func_t (C++ *type*), 157
bus_handle_t (C++ *type*), 451
button_cb_t (C++ *type*), 416
button_event_args_t (C++ *union*), 415
button_event_args_t::long_press (C++ *member*), 415
button_event_args_t::long_press_t (C++ *struct*), 415
button_event_args_t::long_press_t::press_time (C++ *member*), 415
button_event_args_t::multiple_clicks (C++ *member*), 415
button_event_args_t::multiple_clicks_t (C++ *struct*), 415
button_event_args_t::multiple_clicks_t::clicks (C++ *member*), 416
button_event_t (C++ *enum*), 416
button_event_t::BUTTON_DOUBLE_CLICK (C++ *enumerator*), 416
button_event_t::BUTTON_EVENT_MAX (C++ *enumerator*), 417
button_event_t::BUTTON_LONG_PRESS_HOLD (C++ *enumerator*), 417
button_event_t::BUTTON_LONG_PRESS_START (C++ *enumerator*), 416
button_event_t::BUTTON_LONG_PRESS_UP (C++ *enumerator*), 417
button_event_t::BUTTON_MULTIPLE_CLICK (C++ *enumerator*), 416
button_event_t::BUTTON_NONE_PRESS (C++ *enumerator*), 417
button_event_t::BUTTON_PRESS_DOWN (C++ *enumerator*), 416
button_event_t::BUTTON_PRESS_END (C++ *enumerator*), 417
button_event_t::BUTTON_PRESS_REPEAT (C++ *enumerator*), 416
button_event_t::BUTTON_PRESS_REPEAT_DONE (C++ *enumerator*), 416
button_event_t::BUTTON_PRESS_UP (C++ *enumerator*), 416
button_event_t::BUTTON_SINGLE_CLICK (C++ *enumerator*), 416
button_param_t (C++ *enum*), 417
button_param_t::BUTTON_LONG_PRESS_TIME_MS (C++ *enumerator*), 417
button_param_t::BUTTON_PARAM_MAX (C++ *enumerator*), 417
button_param_t::BUTTON_SHORT_PRESS_TIME_MS (C++ *enumerator*), 417
button_power_save_cb_t (C++ *type*), 416
button_power_save_config_t (C++ *struct*), 416
button_power_save_config_t::enter_power_save_cb (C++ *member*), 416
button_power_save_config_t::usr_data (C++ *member*), 416

C

CHARGE_TIME (C *macro*), 521
control_mode_t (C++ *enum*), 520
control_mode_t::BLDC_FOC (C++ *enumerator*), 520

control_mode_t::BLDC_SIX_STEP (*C++ enumerator*), 520

D

dac_audio_config_t (*C++ struct*), 363
dac_audio_config_t::bits_per_sample (*C++ member*), 363
dac_audio_config_t::dac_mode (*C++ member*), 363
dac_audio_config_t::dma_buf_count (*C++ member*), 364
dac_audio_config_t::dma_buf_len (*C++ member*), 364
dac_audio_config_t::i2s_num (*C++ member*), 363
dac_audio_config_t::max_data_size (*C++ member*), 364
dac_audio_config_t::sample_rate (*C++ member*), 363
dac_audio_deinit (*C++ function*), 362
dac_audio_init (*C++ function*), 362
dac_audio_set_param (*C++ function*), 362
dac_audio_set_volume (*C++ function*), 363
dac_audio_start (*C++ function*), 362
dac_audio_stop (*C++ function*), 362
dac_audio_write (*C++ function*), 363
DEFAULT_AUDIO_CODEC_ADC_MONO_CFG (*C macro*), 367
DEFAULT_AUDIO_CODEC_ADC_STEREO_CFG (*C macro*), 367
DEFAULT_TINYUF2_NVS_CONFIG (*C macro*), 356
DEFAULT_TINYUF2_OTA_CONFIG (*C macro*), 356
DUTY_MAX (*C macro*), 521
DUTY_RES (*C macro*), 521

E

ENTER_CLOSE_TIME (*C macro*), 522
esp_bcs_val_t (*C++ struct*), 79
esp_bcs_val_t::basal_metabolism (*C++ member*), 80
esp_bcs_val_t::bcs_flag (*C++ member*), 79
esp_bcs_val_t::body_water_mass (*C++ member*), 80
esp_bcs_val_t::day (*C++ member*), 79
esp_bcs_val_t::fat_free_mass (*C++ member*), 80
esp_bcs_val_t::height (*C++ member*), 80
esp_bcs_val_t::hours (*C++ member*), 79
esp_bcs_val_t::impedance (*C++ member*), 80
esp_bcs_val_t::minutes (*C++ member*), 79
esp_bcs_val_t::month (*C++ member*), 79
esp_bcs_val_t::muscle_mass (*C++ member*), 80
esp_bcs_val_t::muscle_percentage (*C++ member*), 80
esp_bcs_val_t::seconds (*C++ member*), 79
esp_bcs_val_t::soft_lean_mass (*C++ member*), 80
esp_bcs_val_t::timestamp (*C++ member*), 79
esp_bcs_val_t::user_id (*C++ member*), 80
esp_bcs_val_t::weight (*C++ member*), 80
esp_bcs_val_t::year (*C++ member*), 79
ESP_BLE_ADV_DATA_LEN_MAX (*C macro*), 150
esp_ble_anp_data_t (*C++ struct*), 115
esp_ble_anp_data_t::cat_id (*C++ member*), 115
esp_ble_anp_data_t::cat_info (*C++ member*), 116
esp_ble_anp_data_t::count (*C++ member*), 115
esp_ble_anp_data_t::new_alert_val (*C++ member*), 116
esp_ble_anp_data_t::unr_alert_stat (*C++ member*), 115
esp_ble_anp_data_t::[anonymous] (*C++ member*), 116
esp_ble_anp_deinit (*C++ function*), 115
esp_ble_anp_get_new_alert (*C++ function*), 114
esp_ble_anp_get_unr_alert (*C++ function*), 114
esp_ble_anp_init (*C++ function*), 115
esp_ble_anp_option_t (*C++ enum*), 117
esp_ble_anp_option_t::BLE_ANP_OPT_DISABLE (*C++ enumerator*), 117
esp_ble_anp_option_t::BLE_ANP_OPT_ENABLE (*C++ enumerator*), 117
esp_ble_anp_option_t::BLE_ANP_OPT_RECOVER (*C++ enumerator*), 117
esp_ble_anp_set_new_alert (*C++ function*), 114
esp_ble_anp_set_unr_alert (*C++ function*), 115
esp_ble_ans_get_new_alert (*C++ function*), 62
esp_ble_ans_get_unread_alert (*C++ function*), 63
esp_ble_ans_init (*C++ function*), 63
esp_ble_ans_set_new_alert (*C++ function*), 63
esp_ble_ans_set_unread_alert (*C++ function*), 63
esp_ble_bas_battery_info_t (*C++ struct*), 72
esp_ble_bas_battery_info_t::aggregation_group (*C++ member*), 74
esp_ble_bas_battery_info_t::chemistry (*C++ member*), 74
esp_ble_bas_battery_info_t::critical_energy (*C++ member*), 73
esp_ble_bas_battery_info_t::designed_capacity (*C++ member*), 73
esp_ble_bas_battery_info_t::en_aggregation_group (*C++ member*), 73
esp_ble_bas_battery_info_t::en_chemistry (*C++ member*), 73
esp_ble_bas_battery_info_t::en_critical_energy

(C++ member), 73
esp_ble_bas_battery_info_t::en_designed_capacity (C++ member), 72
esp_ble_bas_battery_info_t::en_expiration_date (C++ member), 72
esp_ble_bas_battery_info_t::en_low_energy (C++ member), 73
esp_ble_bas_battery_info_t::en_manufacture_date (C++ member), 72
esp_ble_bas_battery_info_t::en_nominalvoltage (C++ member), 73
esp_ble_bas_battery_info_t::expiration_date (C++ member), 73
esp_ble_bas_battery_info_t::features (C++ member), 73
esp_ble_bas_battery_info_t::flags (C++ member), 73
esp_ble_bas_battery_info_t::flags_reserved (C++ member), 73
esp_ble_bas_battery_info_t::low_energy (C++ member), 73
esp_ble_bas_battery_info_t::manufacture_date (C++ member), 73
esp_ble_bas_battery_info_t::nominalvoltage (C++ member), 74
esp_ble_bas_battery_info_t::recharge_able (C++ member), 73
esp_ble_bas_battery_info_t::replace_able (C++ member), 73
esp_ble_bas_battery_info_t::reserved (C++ member), 73
esp_ble_bas_critical_status_t (C++ struct), 69
esp_ble_bas_critical_status_t::critical_power_rate (C++ member), 69
esp_ble_bas_critical_status_t::immediate_service (C++ member), 69
esp_ble_bas_critical_status_t::reserved (C++ member), 69
esp_ble_bas_critical_status_t::status (C++ member), 69
esp_ble_bas_data_t (C++ struct), 74
esp_ble_bas_data_t::battery_info (C++ member), 74
esp_ble_bas_data_t::battery_level (C++ member), 74
esp_ble_bas_data_t::critical_status (C++ member), 74
esp_ble_bas_data_t::energy_status (C++ member), 74
esp_ble_bas_data_t::estimated_service_date (C++ member), 74
esp_ble_bas_data_t::health_info (C++ member), 74
esp_ble_bas_data_t::health_status (C++ member), 74
esp_ble_bas_data_t::level_status (C++ member), 74
esp_ble_bas_data_t::time_status (C++ member), 74
esp_ble_bas_energy_status_t (C++ struct), 69
esp_ble_bas_energy_status_t::available_battery_ca (C++ member), 70
esp_ble_bas_energy_status_t::available_energy (C++ member), 70
esp_ble_bas_energy_status_t::available_energy_las (C++ member), 70
esp_ble_bas_energy_status_t::charge_rate (C++ member), 70
esp_ble_bas_energy_status_t::en_available_battery (C++ member), 70
esp_ble_bas_energy_status_t::en_available_energy (C++ member), 70
esp_ble_bas_energy_status_t::en_available_energy_ (C++ member), 70
esp_ble_bas_energy_status_t::en_charge_rate (C++ member), 70
esp_ble_bas_energy_status_t::en_external_source_p (C++ member), 70
esp_ble_bas_energy_status_t::en_voltage (C++ member), 70
esp_ble_bas_energy_status_t::external_source_powe (C++ member), 70
esp_ble_bas_energy_status_t::flags (C++ member), 70
esp_ble_bas_energy_status_t::reserved (C++ member), 70
esp_ble_bas_energy_status_t::voltage (C++ member), 70
esp_ble_bas_get_battery_info (C++ function), 67
esp_ble_bas_get_battery_level (C++ function), 65
esp_ble_bas_get_critical_status (C++ function), 66
esp_ble_bas_get_energy_status (C++ function), 66
esp_ble_bas_get_estimated_date (C++ function), 66
esp_ble_bas_get_health_info (C++ function), 67
esp_ble_bas_get_health_status (C++ function), 67
esp_ble_bas_get_level_status (C++ function), 65
esp_ble_bas_get_time_status (C++ function), 66
esp_ble_bas_health_info_t (C++ struct), 72
esp_ble_bas_health_info_t::cycle_count_designed_l (C++ member), 72
esp_ble_bas_health_info_t::en_cycle_count_designe (C++ member), 72
esp_ble_bas_health_info_t::flags (C++ member), 72
esp_ble_bas_health_info_t::max_designed_operating

(C++ member), 72
esp_ble_bas_health_info_t::min_designed_operating_temperature
(C++ member), 72
esp_ble_bas_health_info_t::min_max_designed_operating_temperature
(C++ member), 72
esp_ble_bas_health_info_t::reserved
(C++ member), 72
esp_ble_bas_health_status_t (C++ struct), 71
esp_ble_bas_health_status_t::battery_health_summary
(C++ member), 71
esp_ble_bas_health_status_t::current_temperature
(C++ member), 72
esp_ble_bas_health_status_t::cycle_count
(C++ member), 71
esp_ble_bas_health_status_t::deep_discharge_counts
(C++ member), 72
esp_ble_bas_health_status_t::en_battery_health_summary
(C++ member), 71
esp_ble_bas_health_status_t::en_current_temperature
(C++ member), 71
esp_ble_bas_health_status_t::en_cycle_count
(C++ member), 71
esp_ble_bas_health_status_t::en_deep_discharge_counts
(C++ member), 71
esp_ble_bas_health_status_t::flags
(C++ member), 71
esp_ble_bas_health_status_t::reserved
(C++ member), 71
esp_ble_bas_init (C++ function), 67
esp_ble_bas_level_status_t (C++ struct), 68
esp_ble_bas_level_status_t::additional_status
(C++ member), 69
esp_ble_bas_level_status_t::battery
(C++ member), 68
esp_ble_bas_level_status_t::battery_charge_level
(C++ member), 68
esp_ble_bas_level_status_t::battery_charge_state
(C++ member), 68
esp_ble_bas_level_status_t::battery_charge_type
(C++ member), 68
esp_ble_bas_level_status_t::battery_fault
(C++ member), 69
esp_ble_bas_level_status_t::battery_level
(C++ member), 69
esp_ble_bas_level_status_t::charging_fault_reason
(C++ member), 68
esp_ble_bas_level_status_t::en_additional_status
(C++ member), 68
esp_ble_bas_level_status_t::en_battery_level
(C++ member), 68
esp_ble_bas_level_status_t::en_identifier
(C++ member), 68
esp_ble_bas_level_status_t::flags
(C++ member), 68
esp_ble_bas_level_status_t::flags_reserved
(C++ member), 68
esp_ble_bas_level_status_t::identifier
(C++ member), 69
esp_ble_bas_level_status_t::operating_temperature
(C++ member), 69
esp_ble_bas_level_status_t::power_state
(C++ member), 69
esp_ble_bas_level_status_t::power_state_reserved
(C++ member), 69
esp_ble_bas_level_status_t::reserved
(C++ member), 69
esp_ble_bas_level_status_t::service_required
(C++ member), 69
esp_ble_bas_level_status_t::wired_external_power_
(C++ member), 68
esp_ble_bas_level_status_t::wireless_external_pow
(C++ member), 68
esp_ble_bas_set_battery_info (C++ function), 67
esp_ble_bas_set_battery_level (C++ function), 65
esp_ble_bas_set_critical_status (C++ function), 66
esp_ble_bas_set_energy_status (C++ function), 66
esp_ble_bas_set_estimated_date (C++ function), 66
esp_ble_bas_set_health_info (C++ function), 67
esp_ble_bas_set_health_status (C++ function), 67
esp_ble_bas_set_level_status (C++ function), 65
esp_ble_bas_set_time_status (C++ function), 66
esp_ble_bas_time_status_t (C++ struct), 70
esp_ble_bas_time_status_t::discharged
(C++ member), 71
esp_ble_bas_time_status_t::discharged_standby
(C++ member), 71
esp_ble_bas_time_status_t::en_discharged_standby
(C++ member), 71
esp_ble_bas_time_status_t::en_recharged
(C++ member), 71
esp_ble_bas_time_status_t::flags (C++ member), 71
esp_ble_bas_time_status_t::recharged
(C++ member), 71
esp_ble_bas_time_status_t::reserved
(C++ member), 71
esp_ble_bcs_init (C++ function), 79
esp_ble_bcs_set_measurement (C++ function), 79
esp_ble_conn_add_svc (C++ function), 39
esp_ble_conn_addr_type_t (C++ enum), 58
esp_ble_conn_addr_type_t::ESP_BLE_CONN_ADDR_PUBLI
(C++ enumerator), 58
esp_ble_conn_addr_type_t::ESP_BLE_CONN_ADDR_RANDO
(C++ enumerator), 58
esp_ble_conn_adv_conn_mode_t (C++ enum), 57
esp_ble_conn_adv_conn_mode_t::ESP_BLE_CONN_ADV_CO

(C++ *enumerator*), 57
esp_ble_conn_adv_conn_mode_t::ESP_BLE_CONN_ADV_CONN_MODE_DIR (C++ *enumerator*), 57
esp_ble_conn_adv_conn_mode_t::ESP_BLE_CONN_ADV_CONN_MODE_UND (C++ *enumerator*), 57
ESP_BLE_CONN_ADV_DATA_MAX_LEN (*C macro*), 52
esp_ble_conn_adv_data_set (*C++ function*), 33
esp_ble_conn_adv_disc_mode_t (*C++ enum*), 57
esp_ble_conn_adv_disc_mode_t::ESP_BLE_CONN_ADV_DISC_MODE_LIM (C++ *enumerator*), 58
esp_ble_conn_adv_disc_mode_t::ESP_BLE_CONN_ADV_DISC_MODE_NON (C++ *enumerator*), 57
esp_ble_conn_adv_disc_mode_t::ESP_BLE_CONN_ADV_DISC_MODE_UND (C++ *enumerator*), 57
esp_ble_conn_adv_params_set (*C++ function*), 33
esp_ble_conn_adv_params_t (*C++ struct*), 47
esp_ble_conn_adv_params_t::adv_event_properties (*C++ member*), 47
esp_ble_conn_adv_params_t::adv_handle (*C++ member*), 47
esp_ble_conn_adv_params_t::channel_map (*C++ member*), 47
esp_ble_conn_adv_params_t::conn_mode (*C++ member*), 47
esp_ble_conn_adv_params_t::disc_mode (*C++ member*), 47
esp_ble_conn_adv_params_t::ext_adv_cap (*C++ member*), 48
esp_ble_conn_adv_params_t::filter_policy (*C++ member*), 48
esp_ble_conn_adv_params_t::itvl_max (*C++ member*), 47
esp_ble_conn_adv_params_t::itvl_min (*C++ member*), 47
esp_ble_conn_adv_params_t::own_addr_type (*C++ member*), 48
esp_ble_conn_adv_params_t::peer_addr (*C++ member*), 48
esp_ble_conn_adv_params_t::peer_addr_type (*C++ member*), 48
esp_ble_conn_adv_params_t::primary_phy (*C++ member*), 48
esp_ble_conn_adv_params_t::primary_phy_options (*C++ member*), 48
esp_ble_conn_adv_params_t::scan_req_notif (*C++ member*), 48
esp_ble_conn_adv_params_t::secondary_adv_max_skip (*C++ member*), 48
esp_ble_conn_adv_params_t::secondary_phy (*C++ member*), 48
esp_ble_conn_adv_params_t::secondary_phy_options (*C++ member*), 48
esp_ble_conn_adv_params_t::sid (*C++ member*), 48
esp_ble_conn_adv_params_t::tx_power (*C++ member*), 48
esp_ble_conn_adv_start (*C++ function*), 34
esp_ble_conn_adv_stop (*C++ function*), 34
ESP_BLE_CONN_ADV_TYPE_APPEARANCE (*C macro*), 51
ESP_BLE_CONN_ADV_TYPE_FLAGS (*C macro*), 51
ESP_BLE_CONN_ADV_TYPE_MANUFACTURER (*C macro*), 51
ESP_BLE_CONN_ADV_TYPE_NAME_COMPLETE (*C macro*), 51
ESP_BLE_CONN_ADV_TYPE_NAME_SHORT (*C macro*), 51
ESP_BLE_CONN_ADV_TYPE_SERVICE_DATA128 (*C macro*), 51
ESP_BLE_CONN_ADV_TYPE_SERVICE_DATA16 (*C macro*), 51
ESP_BLE_CONN_ADV_TYPE_SERVICE_DATA32 (*C macro*), 51
ESP_BLE_CONN_ADV_TYPE_TX_POWER (*C macro*), 51
ESP_BLE_CONN_ADV_TYPE_UUID128_COMPLETE (*C macro*), 51
ESP_BLE_CONN_ADV_TYPE_UUID128_INCOMP (*C macro*), 51
ESP_BLE_CONN_ADV_TYPE_UUID16_COMPLETE (*C macro*), 51
ESP_BLE_CONN_ADV_TYPE_UUID16_INCOMP (*C macro*), 51
ESP_BLE_CONN_ADV_TYPE_UUID32_COMPLETE (*C macro*), 51
ESP_BLE_CONN_ADV_TYPE_UUID32_INCOMP (*C macro*), 51
esp_ble_conn_cb_t (*C++ type*), 56
esp_ble_conn_cccd_update_t (*C++ struct*), 46
esp_ble_conn_cccd_update_t::char_handle (*C++ member*), 46
esp_ble_conn_cccd_update_t::conn_handle (*C++ member*), 46
esp_ble_conn_cccd_update_t::indicate_enable (*C++ member*), 46
esp_ble_conn_cccd_update_t::notify_enable (*C++ member*), 46
esp_ble_conn_cccd_update_t::uuid (*C++ member*), 46
esp_ble_conn_cccd_update_t::uuid_type (*C++ member*), 46
esp_ble_conn_character_t (*C++ struct*), 42
esp_ble_conn_character_t::flag (*C++ member*), 43
esp_ble_conn_character_t::name (*C++ member*), 42
esp_ble_conn_character_t::type (*C++ member*), 42
esp_ble_conn_character_t::uuid (*C++ member*), 43
esp_ble_conn_character_t::uuid_fn (*C++ member*), 43

esp_ble_conn_config_t (*C++ struct*), 43
esp_ble_conn_config_t::adv_uuid128 (*C++ member*), 44
esp_ble_conn_config_t::adv_uuid16 (*C++ member*), 44
esp_ble_conn_config_t::adv_uuid_type (*C++ member*), 44
esp_ble_conn_config_t::broadcast_data (*C++ member*), 43
esp_ble_conn_config_t::device_name (*C++ member*), 43
esp_ble_conn_config_t::extended_adv_data (*C++ member*), 44
esp_ble_conn_config_t::extended_adv_len (*C++ member*), 43
esp_ble_conn_config_t::extended_adv_rsp_data (*C++ member*), 44
esp_ble_conn_config_t::extended_adv_rsp_len (*C++ member*), 44
esp_ble_conn_config_t::include_service_uuid (*C++ member*), 44
esp_ble_conn_config_t::periodic_adv_data (*C++ member*), 44
esp_ble_conn_config_t::periodic_adv_len (*C++ member*), 43
esp_ble_conn_config_t::remote_name (*C++ member*), 43
esp_ble_conn_config_t::[anonymous] (*C++ member*), 44
esp_ble_conn_connect (*C++ function*), 29
esp_ble_conn_connect_to_addr (*C++ function*), 29
esp_ble_conn_data_t (*C++ struct*), 44
esp_ble_conn_data_t::data (*C++ member*), 44
esp_ble_conn_data_t::data_len (*C++ member*), 44
esp_ble_conn_data_t::type (*C++ member*), 44
esp_ble_conn_data_t::uuid (*C++ member*), 44
esp_ble_conn_data_t::write_conn_id (*C++ member*), 44
esp_ble_conn_deinit (*C++ function*), 28
esp_ble_conn_desc_t (*C++ enum*), 59
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_AGG_FORMAT (*C++ enumerator*), 60
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_CLIENT_CONFIG (*C++ enumerator*), 60
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_COMPLETE_BLOCK (*C++ enumerator*), 60
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_DIGITAL (*C++ enumerator*), 60
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_ENV_SENS_CONFIG (*C++ enumerator*), 60
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_ENV_SENS_MEASURE (*C++ enumerator*), 60
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_ENV_SENS_TRIGGER (*C++ enumerator*), 60
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_EXTENDED (*C++ enumerator*), 59
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_EXTREPORT (*C++ enumerator*), 60
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_PRE_FORMAT (*C++ enumerator*), 60
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_REPORT (*C++ enumerator*), 60
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_SERVER_CON (*C++ enumerator*), 60
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_TIME_TRIGG (*C++ enumerator*), 60
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_UNKNOWN (*C++ enumerator*), 59
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_USER (*C++ enumerator*), 60
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_VALID_RANG (*C++ enumerator*), 60
esp_ble_conn_desc_t::ESP_BLE_CONN_DESC_VALUE_TRIG (*C++ enumerator*), 60
esp_ble_conn_disconnect (*C++ function*), 29
esp_ble_conn_disconnect_by_handle (*C++ function*), 30
esp_ble_conn_event_data_t (*C++ struct*), 49
esp_ble_conn_event_data_t::action (*C++ member*), 50
esp_ble_conn_event_data_t::authenticated (*C++ member*), 50
esp_ble_conn_event_data_t::cccd_update (*C++ member*), 50
esp_ble_conn_event_data_t::channel_id (*C++ member*), 50
esp_ble_conn_event_data_t::conn_handle (*C++ member*), 49
esp_ble_conn_event_data_t::conn_param_update (*C++ member*), 50
esp_ble_conn_event_data_t::connected (*C++ member*), 49
esp_ble_conn_event_data_t::data_receive (*C++ member*), 50
esp_ble_conn_event_data_t::disconnected (*C++ member*), 49
esp_ble_conn_event_data_t::enc_change (*C++ member*), 50
esp_ble_conn_event_data_t::encrypted (*C++ member*), 50
esp_ble_conn_event_data_t::mtu (*C++ member*), 50
esp_ble_conn_event_data_t::mtu_update (*C++ member*), 50
esp_ble_conn_event_data_t::params (*C++ member*), 50
esp_ble_conn_event_data_t::passkey (*C++ member*), 51
esp_ble_conn_event_data_t::passkey_action (*C++ member*), 51
esp_ble_conn_event_data_t::peer_addr

(*C++ member*), 49
esp_ble_conn_event_data_t::peer_addr_type (*C++ member*), 49
esp_ble_conn_event_data_t::periodic_report (*C++ member*), 50
esp_ble_conn_event_data_t::periodic_sync (*C++ member*), 50
esp_ble_conn_event_data_t::periodic_sync_lost (*C++ member*), 50
esp_ble_conn_event_data_t::reason (*C++ member*), 49
esp_ble_conn_event_data_t::scan_result (*C++ member*), 50
esp_ble_conn_event_data_t::status (*C++ member*), 50
esp_ble_conn_event_t (*C++ enum*), 58
esp_ble_conn_event_t::ESP_BLE_CONN_EVENT_CCCD_UPDATE (*C++ enumerator*), 59
esp_ble_conn_event_t::ESP_BLE_CONN_EVENT_CONN_PARAM_UPDATE (*C++ enumerator*), 59
esp_ble_conn_event_t::ESP_BLE_CONN_EVENT_CONNECTED (*C++ enumerator*), 59
esp_ble_conn_event_t::ESP_BLE_CONN_EVENT_DATA_RECEIVED (*C++ enumerator*), 59
esp_ble_conn_event_t::ESP_BLE_CONN_EVENT_DISCOVER_COMPLETE (*C++ enumerator*), 59
esp_ble_conn_event_t::ESP_BLE_CONN_EVENT_DISCONNECTED (*C++ enumerator*), 59
esp_ble_conn_event_t::ESP_BLE_CONN_EVENT_ENC_CHANGE (*C++ enumerator*), 59
esp_ble_conn_event_t::ESP_BLE_CONN_EVENT_MTU (*C++ enumerator*), 59
esp_ble_conn_event_t::ESP_BLE_CONN_EVENT_PASSKEY_ACTION (*C++ enumerator*), 59
esp_ble_conn_event_t::ESP_BLE_CONN_EVENT_PERIODIC_REPORT (*C++ enumerator*), 59
esp_ble_conn_event_t::ESP_BLE_CONN_EVENT_PERIODIC_SYNC (*C++ enumerator*), 59
esp_ble_conn_event_t::ESP_BLE_CONN_EVENT_PERIODIC_SYNC_LOST (*C++ enumerator*), 59
esp_ble_conn_event_t::ESP_BLE_CONN_EVENT_SCAN_RESULT (*C++ enumerator*), 59
esp_ble_conn_event_t::ESP_BLE_CONN_EVENT_START (*C++ enumerator*), 58
esp_ble_conn_event_t::ESP_BLE_CONN_EVENT_STOP (*C++ enumerator*), 58
esp_ble_conn_event_t::ESP_BLE_CONN_EVENT_UNKNOWN (*C++ enumerator*), 58
esp_ble_conn_find_conn_handle_by_peer_addr (*C++ function*), 31
esp_ble_conn_find_peer_addr_by_conn_handle (*C++ function*), 31
esp_ble_conn_get_conn_handle (*C++ function*), 30
esp_ble_conn_get_conn_handle_by_addr (*C++ function*), 30
esp_ble_conn_get_disconnect_reason (*C++ function*), 31
esp_ble_conn_get_disconnect_reason_by_handle (*C++ function*), 32
esp_ble_conn_get_mtu (*C++ function*), 30
esp_ble_conn_get_mtu_by_handle (*C++ function*), 30
esp_ble_conn_get_peer_addr (*C++ function*), 31
esp_ble_conn_get_peer_addr_by_handle (*C++ function*), 31
esp_ble_conn_get_phy (*C++ function*), 32
esp_ble_conn_init (*C++ function*), 28
esp_ble_conn_l2cap_coc_accept (*C++ function*), 28
esp_ble_conn_l2cap_coc_chan_info_t (*C++ struct*), 40
esp_ble_conn_l2cap_coc_chan_info_t::dcid (*C++ member*), 40
esp_ble_conn_l2cap_coc_chan_info_t::our_coc_mtu (*C++ member*), 41
esp_ble_conn_l2cap_coc_chan_info_t::our_l2cap_mtu (*C++ member*), 40
esp_ble_conn_l2cap_coc_chan_info_t::peer_coc_mtu (*C++ member*), 41
esp_ble_conn_l2cap_coc_chan_info_t::peer_l2cap_mtu (*C++ member*), 40
esp_ble_conn_l2cap_coc_chan_info_t::psm (*C++ member*), 40
esp_ble_conn_l2cap_coc_chan_info_t::scid (*C++ member*), 40
esp_ble_conn_l2cap_coc_chan_t (*C++ type*), 55
esp_ble_conn_l2cap_coc_connect (*C++ function*), 36
esp_ble_conn_l2cap_coc_create_server (*C++ function*), 35
esp_ble_conn_l2cap_coc_disconnect (*C++ function*), 37
esp_ble_conn_l2cap_coc_event (*C++ struct*), 41
esp_ble_conn_l2cap_coc_event::accept (*C++ member*), 42
esp_ble_conn_l2cap_coc_event::chan (*C++ member*), 42
esp_ble_conn_l2cap_coc_event::conn_handle (*C++ member*), 42
esp_ble_conn_l2cap_coc_event::connect (*C++ member*), 42
esp_ble_conn_l2cap_coc_event::disconnect (*C++ member*), 42
esp_ble_conn_l2cap_coc_event::peer_sdu_size (*C++ member*), 42
esp_ble_conn_l2cap_coc_event::receive (*C++ member*), 42
esp_ble_conn_l2cap_coc_event::reconfigured (*C++ member*), 42
esp_ble_conn_l2cap_coc_event::sdu (*C++ member*), 42
esp_ble_conn_l2cap_coc_event::status

(C++ member), 42
esp_ble_conn_l2cap_coc_event::tx_unstalled (C++ member), 42
esp_ble_conn_l2cap_coc_event::type (C++ member), 41
esp_ble_conn_l2cap_coc_event_cb_t (C++ type), 56
esp_ble_conn_l2cap_coc_event_t (C++ type), 55
esp_ble_conn_l2cap_coc_event_type_t (C++ enum), 57
esp_ble_conn_l2cap_coc_event_type_t::ESP_BLE_CONN_L2CAP_COC_EVENT_ACCEPT (C++ enumerator), 57
esp_ble_conn_l2cap_coc_event_type_t::ESP_BLE_CONN_L2CAP_COC_EVENT_CONNECTED (C++ enumerator), 57
esp_ble_conn_l2cap_coc_event_type_t::ESP_BLE_CONN_L2CAP_COC_EVENT_DATA_RECEIVED (C++ enumerator), 57
esp_ble_conn_l2cap_coc_event_type_t::ESP_BLE_CONN_L2CAP_COC_EVENT_DISCONNECTED (C++ enumerator), 57
esp_ble_conn_l2cap_coc_event_type_t::ESP_BLE_CONN_L2CAP_COC_EVENT_PEER_RECONFIGURED (C++ enumerator), 57
esp_ble_conn_l2cap_coc_event_type_t::ESP_BLE_CONN_L2CAP_COC_EVENT_RECONFIG_COMPLETED (C++ enumerator), 57
esp_ble_conn_l2cap_coc_event_type_t::ESP_BLE_CONN_L2CAP_COC_EVENT_TX_UNSTALLED (C++ enumerator), 57
esp_ble_conn_l2cap_coc_get_chan_info (C++ function), 37
esp_ble_conn_l2cap_coc_mem_init (C++ function), 28
esp_ble_conn_l2cap_coc_mem_release (C++ function), 28
esp_ble_conn_l2cap_coc_recv_ready (C++ function), 37
esp_ble_conn_l2cap_coc_sdu_t (C++ struct), 41
esp_ble_conn_l2cap_coc_sdu_t::data (C++ member), 41
esp_ble_conn_l2cap_coc_sdu_t::len (C++ member), 41
esp_ble_conn_l2cap_coc_send (C++ function), 36
esp_ble_conn_mtu_update (C++ function), 34
esp_ble_conn_notify (C++ function), 38
esp_ble_conn_notify_by_handle (C++ function), 38
esp_ble_conn_numcmp_reply (C++ function), 35
esp_ble_conn_params_t (C++ struct), 46
esp_ble_conn_params_t::itvl_max (C++ member), 47
esp_ble_conn_params_t::itvl_min (C++ member), 46
esp_ble_conn_params_t::latency (C++ member), 47
esp_ble_conn_params_t::max_ce_len (C++ member), 47
esp_ble_conn_params_t::min_ce_len (C++ member), 47
esp_ble_conn_params_t::supervision_timeout (C++ member), 47
esp_ble_conn_parse_adv_data (C++ function), 38
esp_ble_conn_passkey_reply (C++ function), 35
esp_ble_conn_periodic_adv_data_set (C++ function), 33
esp_ble_conn_periodic_report_t (C++ struct), 45
esp_ble_conn_periodic_report_t::data (C++ member), 46
esp_ble_conn_periodic_report_t::data_length (C++ member), 45
esp_ble_conn_periodic_report_t::data_status (C++ member), 45
esp_ble_conn_periodic_report_t::rssi (C++ member), 45
esp_ble_conn_periodic_report_t::sync_handle (C++ member), 45
esp_ble_conn_periodic_report_t::tx_power (C++ member), 45
esp_ble_conn_periodic_sync_lost_t (C++ struct), 46
esp_ble_conn_periodic_sync_lost_t::reason (C++ member), 46
esp_ble_conn_periodic_sync_lost_t::sync_handle (C++ member), 46
esp_ble_conn_periodic_sync_t (C++ struct), 45
esp_ble_conn_periodic_sync_t::adv_addr (C++ member), 45
esp_ble_conn_periodic_sync_t::adv_clk_accuracy (C++ member), 45
esp_ble_conn_periodic_sync_t::adv_phy (C++ member), 45
esp_ble_conn_periodic_sync_t::per_adv_ival (C++ member), 45
esp_ble_conn_periodic_sync_t::sid (C++ member), 45
esp_ble_conn_periodic_sync_t::status (C++ member), 45
esp_ble_conn_periodic_sync_t::sync_handle (C++ member), 45
ESP_BLE_CONN_PHY_CODED_OPT_ANY (C macro), 55
ESP_BLE_CONN_PHY_CODED_OPT_S2 (C macro), 55
ESP_BLE_CONN_PHY_CODED_OPT_S8 (C macro), 55
ESP_BLE_CONN_PHY_MASK_1M (C macro), 55
ESP_BLE_CONN_PHY_MASK_2M (C macro), 55
ESP_BLE_CONN_PHY_MASK_ALL (C macro), 55
ESP_BLE_CONN_PHY_MASK_CODED (C macro), 55
esp_ble_conn_phy_t (C++ enum), 58
esp_ble_conn_phy_t::ESP_BLE_CONN_PHY_1M (C++ enumerator), 58
esp_ble_conn_phy_t::ESP_BLE_CONN_PHY_2M

(C++ *enumerator*), 58
esp_ble_conn_phy_t::ESP_BLE_CONN_PHY_CODED (C++ *enumerator*), 58
esp_ble_conn_read (C++ *function*), 38
esp_ble_conn_read_by_handle (C++ *function*), 38
esp_ble_conn_register_scan_callback (C++ *function*), 37
esp_ble_conn_remove_svc (C++ *function*), 40
esp_ble_conn_scan_cb_t (C++ *type*), 55
esp_ble_conn_scan_filt_policy_t (C++ *enum*), 58
esp_ble_conn_scan_filt_policy_t::ESP_BLE_CONN_SCAN_FILT_NO_WL (C++ *enumerator*), 58
esp_ble_conn_scan_filt_policy_t::ESP_BLE_CONN_SCAN_FILT_USE_WL (C++ *enumerator*), 58
esp_ble_conn_scan_params_set (C++ *function*), 34
esp_ble_conn_scan_params_t (C++ *struct*), 48
esp_ble_conn_scan_params_t::filter_duplicates (C++ *member*), 49
esp_ble_conn_scan_params_t::filter_policy (C++ *member*), 49
esp_ble_conn_scan_params_t::itvl (C++ *member*), 49
esp_ble_conn_scan_params_t::limited (C++ *member*), 49
esp_ble_conn_scan_params_t::own_addr_type (C++ *member*), 48
esp_ble_conn_scan_params_t::passive (C++ *member*), 49
esp_ble_conn_scan_params_t::scan_phys (C++ *member*), 49
esp_ble_conn_scan_params_t::window (C++ *member*), 49
esp_ble_conn_scan_result_t (C++ *struct*), 41
esp_ble_conn_scan_result_t::addr (C++ *member*), 41
esp_ble_conn_scan_result_t::addr_type (C++ *member*), 41
esp_ble_conn_scan_result_t::adv_data (C++ *member*), 41
esp_ble_conn_scan_result_t::adv_data_len (C++ *member*), 41
esp_ble_conn_scan_result_t::rssi (C++ *member*), 41
esp_ble_conn_scan_rsp_data_set (C++ *function*), 33
esp_ble_conn_scan_start (C++ *function*), 34
esp_ble_conn_scan_stop (C++ *function*), 34
esp_ble_conn_security_initiate (C++ *function*), 35
esp_ble_conn_set_data_len (C++ *function*), 33
esp_ble_conn_set_local_passkey (C++ *function*), 34
esp_ble_conn_set_mtu (C++ *function*), 29
esp_ble_conn_set_preferred_phy (C++ *function*), 32
esp_ble_conn_sm_action_t (C++ *enum*), 56
esp_ble_conn_sm_action_t::ESP_BLE_CONN_SM_ACT_DISC (C++ *enumerator*), 56
esp_ble_conn_sm_action_t::ESP_BLE_CONN_SM_ACT_INPUT (C++ *enumerator*), 56
esp_ble_conn_sm_action_t::ESP_BLE_CONN_SM_ACT_NONE (C++ *enumerator*), 56
esp_ble_conn_sm_action_t::ESP_BLE_CONN_SM_ACT_NUMCMP (C++ *enumerator*), 56
esp_ble_conn_sm_action_t::ESP_BLE_CONN_SM_ACT_OOB (C++ *enumerator*), 56
esp_ble_conn_start (C++ *function*), 29
esp_ble_conn_stop (C++ *function*), 29
esp_ble_conn_subscribe (C++ *function*), 39
esp_ble_conn_subscribe_by_handle (C++ *function*), 39
esp_ble_conn_svc_t (C++ *struct*), 43
esp_ble_conn_svc_t::nu_lookup (C++ *member*), 43
esp_ble_conn_svc_t::nu_lookup_count (C++ *member*), 43
esp_ble_conn_svc_t::type (C++ *member*), 43
esp_ble_conn_svc_t::uuid (C++ *member*), 43
esp_ble_conn_update_params (C++ *function*), 32
esp_ble_conn_uuid_t (C++ *union*), 40
esp_ble_conn_uuid_t::uuid128 (C++ *member*), 40
esp_ble_conn_uuid_t::uuid16 (C++ *member*), 40
esp_ble_conn_uuid_t::uuid32 (C++ *member*), 40
esp_ble_conn_uuid_type_t (C++ *enum*), 60
esp_ble_conn_uuid_type_t::BLE_CONN_UUID_TYPE_128 (C++ *enumerator*), 61
esp_ble_conn_uuid_type_t::BLE_CONN_UUID_TYPE_16 (C++ *enumerator*), 61
esp_ble_conn_uuid_type_t::BLE_CONN_UUID_TYPE_32 (C++ *enumerator*), 61
esp_ble_conn_write (C++ *function*), 39
esp_ble_conn_write_by_handle (C++ *function*), 39
esp_ble_cts_cur_time_t (C++ *struct*), 83
esp_ble_cts_cur_time_t::adjust_reason (C++ *member*), 83
esp_ble_cts_cur_time_t::day (C++ *member*), 83
esp_ble_cts_cur_time_t::day_of_week (C++ *member*), 83
esp_ble_cts_cur_time_t::fractions_256 (C++ *member*), 83
esp_ble_cts_cur_time_t::hours (C++ *member*), 83
esp_ble_cts_cur_time_t::minutes (C++ *member*), 83
esp_ble_cts_cur_time_t::month (C++ *member*), 83

esp_ble_cts_cur_time_t::seconds (C++ member), 83
esp_ble_cts_cur_time_t::timestamp (C++ member), 83
esp_ble_cts_cur_time_t::year (C++ member), 83
esp_ble_cts_get_current_time (C++ function), 82
esp_ble_cts_get_local_time (C++ function), 82
esp_ble_cts_get_reference_time (C++ function), 82
esp_ble_cts_init (C++ function), 83
esp_ble_cts_local_time_t (C++ struct), 83
esp_ble_cts_local_time_t::dst_offset (C++ member), 84
esp_ble_cts_local_time_t::timezone (C++ member), 84
esp_ble_cts_ref_time_t (C++ struct), 84
esp_ble_cts_ref_time_t::days_since_update (C++ member), 84
esp_ble_cts_ref_time_t::hours_since_update (C++ member), 84
esp_ble_cts_ref_time_t::time_accuracy (C++ member), 84
esp_ble_cts_ref_time_t::time_source (C++ member), 84
esp_ble_cts_set_current_time (C++ function), 82
esp_ble_cts_set_local_time (C++ function), 82
esp_ble_cts_set_reference_time (C++ function), 82
esp_ble_dis_data (C++ struct), 87
esp_ble_dis_data::firmware_revision (C++ member), 88
esp_ble_dis_data::hardware_revision (C++ member), 88
esp_ble_dis_data::manufacturer_name (C++ member), 88
esp_ble_dis_data::model_number (C++ member), 87
esp_ble_dis_data::pnp_id (C++ member), 88
esp_ble_dis_data::serial_number (C++ member), 88
esp_ble_dis_data::software_revision (C++ member), 88
esp_ble_dis_data::system_id (C++ member), 88
esp_ble_dis_data_t (C++ type), 89
esp_ble_dis_get_firmware_revision (C++ function), 85
esp_ble_dis_get_hardware_revision (C++ function), 86
esp_ble_dis_get_manufacturer_name (C++ function), 86
esp_ble_dis_get_model_number (C++ function), 85
esp_ble_dis_get_pnp_id (C++ function), 87
esp_ble_dis_get_serial_number (C++ function), 85
esp_ble_dis_get_software_revision (C++ function), 86
esp_ble_dis_get_system_id (C++ function), 86
esp_ble_dis_init (C++ function), 87
esp_ble_dis_pnp (C++ struct), 87
esp_ble_dis_pnp::pid (C++ member), 87
esp_ble_dis_pnp::src (C++ member), 87
esp_ble_dis_pnp::ver (C++ member), 87
esp_ble_dis_pnp::vid (C++ member), 87
esp_ble_dis_pnp_t (C++ type), 89
esp_ble_dis_set_firmware_revision (C++ function), 85
esp_ble_dis_set_hardware_revision (C++ function), 86
esp_ble_dis_set_manufacturer_name (C++ function), 86
esp_ble_dis_set_model_number (C++ function), 85
esp_ble_dis_set_pnp_id (C++ function), 87
esp_ble_dis_set_serial_number (C++ function), 85
esp_ble_dis_set_software_revision (C++ function), 86
esp_ble_dis_set_system_id (C++ function), 86
esp_ble_hrp_data_t (C++ struct), 124
esp_ble_hrp_data_t::detected (C++ member), 124
esp_ble_hrp_data_t::energy (C++ member), 124
esp_ble_hrp_data_t::energy_val (C++ member), 124
esp_ble_hrp_data_t::flags (C++ member), 124
esp_ble_hrp_data_t::format (C++ member), 124
esp_ble_hrp_data_t::heartrate (C++ member), 124
esp_ble_hrp_data_t::interval (C++ member), 124
esp_ble_hrp_data_t::interval_buf (C++ member), 124
esp_ble_hrp_data_t::reserved (C++ member), 124
esp_ble_hrp_data_t::supported (C++ member), 124
esp_ble_hrp_data_t::u16 (C++ member), 124
esp_ble_hrp_data_t::u8 (C++ member), 124
esp_ble_hrp_deinit (C++ function), 124
esp_ble_hrp_get_ctrl (C++ function), 123
esp_ble_hrp_get_location (C++ function), 123
esp_ble_hrp_init (C++ function), 123
esp_ble_hrp_set_ctrl (C++ function), 123

esp_ble_hrs_get_hrm (*C++ function*), 89
esp_ble_hrs_get_location (*C++ function*), 89
esp_ble_hrs_hrm_t (*C++ struct*), 90
esp_ble_hrs_hrm_t::detected (*C++ member*), 90
esp_ble_hrs_hrm_t::energy (*C++ member*), 90
esp_ble_hrs_hrm_t::energy_val (*C++ member*), 90
esp_ble_hrs_hrm_t::flags (*C++ member*), 90
esp_ble_hrs_hrm_t::format (*C++ member*), 90
esp_ble_hrs_hrm_t::heartrate (*C++ member*), 90
esp_ble_hrs_hrm_t::interval (*C++ member*), 90
esp_ble_hrs_hrm_t::interval_buf (*C++ member*), 90
esp_ble_hrs_hrm_t::reserved (*C++ member*), 90
esp_ble_hrs_hrm_t::supported (*C++ member*), 90
esp_ble_hrs_hrm_t::u16 (*C++ member*), 90
esp_ble_hrs_hrm_t::u8 (*C++ member*), 90
esp_ble_hrs_init (*C++ function*), 89
esp_ble_hrs_set_hrm (*C++ function*), 89
esp_ble_hrs_set_location (*C++ function*), 89
esp_ble_htp_data_t (*C++ struct*), 126
esp_ble_htp_data_t::celsius (*C++ member*), 127
esp_ble_htp_data_t::day (*C++ member*), 127
esp_ble_htp_data_t::fahrenheit (*C++ member*), 127
esp_ble_htp_data_t::flags (*C++ member*), 127
esp_ble_htp_data_t::hours (*C++ member*), 127
esp_ble_htp_data_t::location (*C++ member*), 127
esp_ble_htp_data_t::minutes (*C++ member*), 127
esp_ble_htp_data_t::month (*C++ member*), 127
esp_ble_htp_data_t::reserved (*C++ member*), 127
esp_ble_htp_data_t::seconds (*C++ member*), 127
esp_ble_htp_data_t::temperature (*C++ member*), 127
esp_ble_htp_data_t::temperature_type (*C++ member*), 127
esp_ble_htp_data_t::temperature_unit (*C++ member*), 126
esp_ble_htp_data_t::time_stamp (*C++ member*), 127
esp_ble_htp_data_t::timestamp (*C++ member*), 127
esp_ble_htp_data_t::year (*C++ member*), 127
esp_ble_htp_deinit (*C++ function*), 126
esp_ble_htp_get_measurement_interval (*C++ function*), 126
esp_ble_htp_get_temp_type (*C++ function*), 126
esp_ble_htp_init (*C++ function*), 126
esp_ble_htp_set_measurement_interval (*C++ function*), 126
esp_ble_hts_get_intermediate_temp (*C++ function*), 92
esp_ble_hts_get_measurement_interval (*C++ function*), 92
esp_ble_hts_get_measurement_temp (*C++ function*), 92
esp_ble_hts_get_temp_type (*C++ function*), 92
esp_ble_hts_init (*C++ function*), 92
esp_ble_hts_set_intermediate_temp (*C++ function*), 92
esp_ble_hts_set_measurement_interval (*C++ function*), 92
esp_ble_hts_set_measurement_temp (*C++ function*), 92
esp_ble_hts_set_temp_type (*C++ function*), 92
esp_ble_hts_temp_t (*C++ struct*), 93
esp_ble_hts_temp_t::celsius (*C++ member*), 93
esp_ble_hts_temp_t::day (*C++ member*), 93
esp_ble_hts_temp_t::fahrenheit (*C++ member*), 93
esp_ble_hts_temp_t::flags (*C++ member*), 93
esp_ble_hts_temp_t::hours (*C++ member*), 93
esp_ble_hts_temp_t::location (*C++ member*), 94
esp_ble_hts_temp_t::minutes (*C++ member*), 93
esp_ble_hts_temp_t::month (*C++ member*), 93
esp_ble_hts_temp_t::reserved (*C++ member*), 93
esp_ble_hts_temp_t::seconds (*C++ member*), 94
esp_ble_hts_temp_t::temperature (*C++ member*), 93
esp_ble_hts_temp_t::temperature_type (*C++ member*), 93
esp_ble_hts_temp_t::temperature_unit (*C++ member*), 93
esp_ble_hts_temp_t::time_stamp (*C++ member*), 93
esp_ble_hts_temp_t::timestamp (*C++ member*), 94
esp_ble_hts_temp_t::year (*C++ member*), 93
esp_ble_midi_event_cb_t (*C++ type*), 122

esp_ble_midi_event_type_t (*C++ enum*), 122
esp_ble_midi_event_type_t::ESP_BLE_MIDI_EVENT_NORMAL (*C++ enumerator*), 123
esp_ble_midi_event_type_t::ESP_BLE_MIDI_EVENT_SYSEX_OVERFLOW (*C++ enumerator*), 123
esp_ble_midi_get_timestamp_ms (*C++ function*), 121
esp_ble_midi_is_notify_enabled (*C++ function*), 121
esp_ble_midi_on_bep_received (*C++ function*), 121
esp_ble_midi_profile_deinit (*C++ function*), 118
esp_ble_midi_profile_init (*C++ function*), 118
esp_ble_midi_register_event_cb (*C++ function*), 119
esp_ble_midi_register_rx_cb (*C++ function*), 118
esp_ble_midi_rx_cb_t (*C++ type*), 122
esp_ble_midi_send (*C++ function*), 119
esp_ble_midi_send_cc (*C++ function*), 120
esp_ble_midi_send_multi (*C++ function*), 120
esp_ble_midi_send_note_off (*C++ function*), 120
esp_ble_midi_send_note_on (*C++ function*), 119
esp_ble_midi_send_pitch_bend (*C++ function*), 120
esp_ble_midi_send_raw_midi (*C++ function*), 119
esp_ble_midi_send_sysex (*C++ function*), 121
esp_ble_midi_set_notify_enabled (*C++ function*), 121
ESP_BLE_MIDI_TIMESTAMP_MAX (*C macro*), 122
esp_ble_otp_client_abort (*C++ function*), 137
esp_ble_otp_client_calculate_checksum (*C++ function*), 137
esp_ble_otp_client_clear_mark (*C++ function*), 132
esp_ble_otp_client_create_object (*C++ function*), 135
esp_ble_otp_client_delete_object (*C++ function*), 136
esp_ble_otp_client_disconnect_transfer_channel (*C++ function*), 138
esp_ble_otp_client_discover_all_start (*C++ function*), 134
esp_ble_otp_client_discover_by_filter (*C++ function*), 134
esp_ble_otp_client_discover_next (*C++ function*), 134
esp_ble_otp_client_discover_ots (*C++ function*), 130
esp_ble_otp_client_execute_object (*C++ function*), 137
esp_ble_otp_client_read_directory_listing (*C++ function*), 135
esp_ble_otp_client_read_feature (*C++ function*), 130
esp_ble_otp_client_read_object (*C++ function*), 136
esp_ble_otp_client_read_object_info (*C++ function*), 133
esp_ble_otp_client_request_num_objects (*C++ function*), 132
esp_ble_otp_client_resume_read_checksum (*C++ function*), 129
esp_ble_otp_client_resume_write_checksum (*C++ function*), 130
esp_ble_otp_client_resume_write_current_size (*C++ function*), 129
esp_ble_otp_client_select_by_id (*C++ function*), 131
esp_ble_otp_client_select_by_index (*C++ function*), 132
esp_ble_otp_client_select_directory_listing (*C++ function*), 135
esp_ble_otp_client_select_first (*C++ function*), 130
esp_ble_otp_client_select_last (*C++ function*), 131
esp_ble_otp_client_select_next (*C++ function*), 131
esp_ble_otp_client_select_previous (*C++ function*), 131
esp_ble_otp_client_send_data (*C++ function*), 138
esp_ble_otp_client_set_filter (*C++ function*), 133
esp_ble_otp_client_set_sort_order (*C++ function*), 132
esp_ble_otp_client_write_name (*C++ function*), 133
esp_ble_otp_client_write_object (*C++ function*), 136
esp_ble_otp_client_write_prop (*C++ function*), 134
esp_ble_otp_config_t (*C++ struct*), 142
esp_ble_otp_config_t::auto_discover_ots (*C++ member*), 142
esp_ble_otp_config_t::l2cap_coc_mtu (*C++ member*), 142
esp_ble_otp_config_t::psm (*C++ member*), 142
esp_ble_otp_config_t::role (*C++ member*), 142
esp_ble_otp_deinit (*C++ function*), 140
esp_ble_otp_event_data_t (*C++ union*), 140
esp_ble_otp_event_data_t::change (*C++ member*), 140
esp_ble_otp_event_data_t::chunk_offset (*C++ member*), 141
esp_ble_otp_event_data_t::conn_handle (*C++ member*), 140

esp_ble_otp_event_data_t::data (C++ member), 141
esp_ble_otp_event_data_t::data_len (C++ member), 141
esp_ble_otp_event_data_t::error (C++ member), 142
esp_ble_otp_event_data_t::feature (C++ member), 140
esp_ble_otp_event_data_t::oacp_aborted (C++ member), 140
esp_ble_otp_event_data_t::oacp_response (C++ member), 141
esp_ble_otp_event_data_t::oacp_started (C++ member), 140
esp_ble_otp_event_data_t::object_changed (C++ member), 140
esp_ble_otp_event_data_t::object_info (C++ member), 140
esp_ble_otp_event_data_t::object_selected (C++ member), 140
esp_ble_otp_event_data_t::olcp_response (C++ member), 141
esp_ble_otp_event_data_t::op_code (C++ member), 140
esp_ble_otp_event_data_t::ots_discovered (C++ member), 140
esp_ble_otp_event_data_t::response (C++ member), 141
esp_ble_otp_event_data_t::success (C++ member), 141
esp_ble_otp_event_data_t::total_len (C++ member), 141
esp_ble_otp_event_data_t::transfer_channel_connected (C++ member), 141
esp_ble_otp_event_data_t::transfer_channel_disconnected (C++ member), 141
esp_ble_otp_event_data_t::transfer_complete (C++ member), 141
esp_ble_otp_event_data_t::transfer_data_received (C++ member), 141
esp_ble_otp_event_data_t::transfer_data_sent (C++ member), 141
esp_ble_otp_event_data_t::transfer_eof (C++ member), 141
esp_ble_otp_event_data_t::transfer_error (C++ member), 142
esp_ble_otp_event_data_t::transfer_info (C++ member), 141
esp_ble_otp_init (C++ function), 139
esp_ble_otp_oacp_rsp_t (C++ struct), 143
esp_ble_otp_oacp_rsp_t::req_op_code (C++ member), 143
esp_ble_otp_oacp_rsp_t::rsp_code (C++ member), 143
esp_ble_otp_oacp_rsp_t::rsp_parameter (C++ member), 143
esp_ble_otp_object_info_t (C++ struct), 142
esp_ble_otp_object_info_t::first_created (C++ member), 142
esp_ble_otp_object_info_t::last_modified (C++ member), 143
esp_ble_otp_object_info_t::object_id (C++ member), 142
esp_ble_otp_object_info_t::object_name (C++ member), 142
esp_ble_otp_object_info_t::object_prop (C++ member), 142
esp_ble_otp_object_info_t::object_size (C++ member), 142
esp_ble_otp_object_info_t::object_type (C++ member), 142
esp_ble_otp_olcp_rsp_t (C++ struct), 143
esp_ble_otp_olcp_rsp_t::num_objects (C++ member), 143
esp_ble_otp_olcp_rsp_t::req_op_code (C++ member), 143
esp_ble_otp_olcp_rsp_t::rsp_code (C++ member), 143
esp_ble_otp_olcp_rsp_t::rsp_parameter (C++ member), 143
esp_ble_otp_role_t (C++ enum), 145
esp_ble_otp_role_t::BLE_OTP_ROLE_CLIENT (C++ enumerator), 145
esp_ble_otp_role_t::BLE_OTP_ROLE_SERVER (C++ enumerator), 145
esp_ble_otp_server_disconnect_transfer_channel (C++ function), 139
esp_ble_otp_server_register_oacp_callback (C++ function), 138
esp_ble_otp_server_register_olcp_callback (C++ function), 138
esp_ble_otp_server_send_data (C++ function), 139
esp_ble_otp_server_send_oacp_response (C++ function), 139
esp_ble_otp_server_send_olcp_response (C++ function), 139
esp_ble_otp_server_set_feature (C++ function), 138
esp_ble_otp_transfer_info_t (C++ struct), 143
esp_ble_otp_transfer_info_t::channel (C++ member), 143
esp_ble_otp_transfer_info_t::conn_handle (C++ member), 144
esp_ble_otp_transfer_info_t::is_read (C++ member), 144
esp_ble_otp_transfer_info_t::mtu (C++ member), 144
esp_ble_otp_transfer_info_t::object_id (C++ member), 144
esp_ble_otp_transfer_info_t::psm (C++ member), 144
esp_ble_otp_write_mode_t (C++ enum), 145
esp_ble_otp_write_mode_t::BLE_OTP_WRITE_MODE_APPE (C++ enumerator), 146

esp_ble_otp_write_mode_t::BLE_OTP_WRITE_MODE_OVERWRITE (C++ enumerator), 146
esp_ble_otp_write_mode_t::BLE_OTP_WRITE_MODE_PATCH (C++ enumerator), 146
esp_ble_otp_write_mode_t::BLE_OTP_WRITE_MODE_TRUNCATE (C++ enumerator), 146
esp_ble_ots_change_t (C++ struct), 102
esp_ble_ots_change_t::flag (C++ member), 102
esp_ble_ots_change_t::object_id (C++ member), 102
esp_ble_ots_deinit (C++ function), 98
esp_ble_ots_feature_t (C++ struct), 99
esp_ble_ots_feature_t::abort_op (C++ member), 99
esp_ble_ots_feature_t::appending_op (C++ member), 99
esp_ble_ots_feature_t::calculate_op (C++ member), 99
esp_ble_ots_feature_t::clear_mark_op (C++ member), 99
esp_ble_ots_feature_t::create_op (C++ member), 99
esp_ble_ots_feature_t::delete_op (C++ member), 99
esp_ble_ots_feature_t::execute_op (C++ member), 99
esp_ble_ots_feature_t::goto_op (C++ member), 99
esp_ble_ots_feature_t::oacp (C++ member), 99
esp_ble_ots_feature_t::olcp (C++ member), 100
esp_ble_ots_feature_t::order_op (C++ member), 99
esp_ble_ots_feature_t::patching_op (C++ member), 99
esp_ble_ots_feature_t::read_op (C++ member), 99
esp_ble_ots_feature_t::req_num_op (C++ member), 99
esp_ble_ots_feature_t::truncation_op (C++ member), 99
esp_ble_ots_feature_t::write_op (C++ member), 99
esp_ble_ots_filter_t (C++ struct), 102
esp_ble_ots_filter_t::filter (C++ member), 102
esp_ble_ots_filter_t::filter_parameter (C++ member), 102
esp_ble_ots_get_change (C++ function), 98
esp_ble_ots_get_feature (C++ function), 95
esp_ble_ots_get_filter (C++ function), 98
esp_ble_ots_get_first_create_time (C++ function), 96
esp_ble_ots_get_id (C++ function), 97
esp_ble_ots_get_last_modify_time (C++ function), 96
esp_ble_ots_get_name (C++ function), 95
esp_ble_ots_get_oacp (C++ function), 97
esp_ble_ots_get_olcp (C++ function), 97
esp_ble_ots_get_prop (C++ function), 97
esp_ble_ots_get_size (C++ function), 96
esp_ble_ots_get_type (C++ function), 96
esp_ble_ots_id_t (C++ struct), 100
esp_ble_ots_id_t::id (C++ member), 100
esp_ble_ots_init (C++ function), 98
esp_ble_ots_oacp_t (C++ struct), 101
esp_ble_ots_oacp_t::op_code (C++ member), 101
esp_ble_ots_oacp_t::parameter (C++ member), 101
esp_ble_ots_olcp_rsp_t (C++ struct), 102
esp_ble_ots_olcp_rsp_t::req_code (C++ member), 102
esp_ble_ots_olcp_rsp_t::rsp_code (C++ member), 102
esp_ble_ots_olcp_rsp_t::rsp_parameter (C++ member), 102
esp_ble_ots_olcp_t (C++ struct), 101
esp_ble_ots_olcp_t::op_code (C++ member), 101
esp_ble_ots_olcp_t::parameter (C++ member), 102
esp_ble_ots_prop_t (C++ struct), 100
esp_ble_ots_prop_t::append_prop (C++ member), 101
esp_ble_ots_prop_t::delete_prop (C++ member), 101
esp_ble_ots_prop_t::execute_prop (C++ member), 101
esp_ble_ots_prop_t::mark_prop (C++ member), 101
esp_ble_ots_prop_t::patch_prop (C++ member), 101
esp_ble_ots_prop_t::read_prop (C++ member), 101
esp_ble_ots_prop_t::truncate_prop (C++ member), 101
esp_ble_ots_prop_t::write_prop (C++ member), 101
esp_ble_ots_set_change (C++ function), 98
esp_ble_ots_set_feature (C++ function), 95
esp_ble_ots_set_filter (C++ function), 98
esp_ble_ots_set_first_create_time (C++ function), 96
esp_ble_ots_set_id (C++ function), 97
esp_ble_ots_set_last_modify_time (C++ function), 97
esp_ble_ots_set_name (C++ function), 95
esp_ble_ots_set_oacp (C++ function), 97
esp_ble_ots_set_olcp (C++ function), 98
esp_ble_ots_set_prop (C++ function), 97
esp_ble_ots_set_size (C++ function), 96
esp_ble_ots_set_type (C++ function), 96
esp_ble_ots_size_t (C++ struct), 100

esp_ble_ots_size_t::allocated_size (*C++ member*), 100
esp_ble_ots_size_t::current_size (*C++ member*), 100
esp_ble_ots_utc_t (*C++ struct*), 100
esp_ble_ots_utc_t::day (*C++ member*), 100
esp_ble_ots_utc_t::hours (*C++ member*), 100
esp_ble_ots_utc_t::minutes (*C++ member*), 100
esp_ble_ots_utc_t::month (*C++ member*), 100
esp_ble_ots_utc_t::seconds (*C++ member*), 100
esp_ble_ots_utc_t::year (*C++ member*), 100
ESP_BLE_SCAN_RSP_DATA_LEN_MAX (*C macro*), 150
esp_ble_tps_get_tx_power_level (*C++ function*), 106
esp_ble_tps_init (*C++ function*), 106
esp_ble_tps_set_tx_power_level (*C++ function*), 106
esp_ble_uds_db_t (*C++ struct*), 108
esp_ble_uds_db_t::db_buf (*C++ member*), 108
esp_ble_uds_db_t::len (*C++ member*), 108
esp_ble_uds_get_db_incre (*C++ function*), 107
esp_ble_uds_get_reg_user (*C++ function*), 107
esp_ble_uds_get_user_ctrl (*C++ function*), 107
esp_ble_uds_init (*C++ function*), 108
esp_ble_uds_reg_user_t (*C++ struct*), 108
esp_ble_uds_reg_user_t::first_segment (*C++ member*), 108
esp_ble_uds_reg_user_t::flag (*C++ member*), 109
esp_ble_uds_reg_user_t::header (*C++ member*), 108
esp_ble_uds_reg_user_t::last_segment (*C++ member*), 108
esp_ble_uds_reg_user_t::rolling_segment_number (*C++ member*), 108
esp_ble_uds_reg_user_t::user_index (*C++ member*), 109
esp_ble_uds_reg_user_t::user_name (*C++ member*), 109
esp_ble_uds_reg_user_t::user_name_present (*C++ member*), 108
esp_ble_uds_reg_user_t::user_name_truncated (*C++ member*), 109
esp_ble_uds_set_db_incre (*C++ function*), 107
esp_ble_uds_set_reg_user (*C++ function*), 108
esp_ble_uds_set_user_ctrl (*C++ function*), 107
esp_ble_uds_user_ctrl_t (*C++ struct*), 109
esp_ble_uds_user_ctrl_t::op_code (*C++ member*), 109
esp_ble_uds_user_ctrl_t::param_len (*C++ member*), 109
esp_ble_uds_user_ctrl_t::parameter (*C++ member*), 109
esp_ble_wss_feature_t (*C++ struct*), 111
esp_ble_wss_feature_t::bmi (*C++ member*), 111
esp_ble_wss_feature_t::h_resolution (*C++ member*), 111
esp_ble_wss_feature_t::height (*C++ member*), 111
esp_ble_wss_feature_t::timestamp (*C++ member*), 111
esp_ble_wss_feature_t::user_id (*C++ member*), 111
esp_ble_wss_feature_t::w_resolution (*C++ member*), 111
esp_ble_wss_feature_t::weight (*C++ member*), 111
esp_ble_wss_get_measurement (*C++ function*), 110
esp_ble_wss_init (*C++ function*), 110
esp_ble_wss_measurement_t (*C++ struct*), 111
esp_ble_wss_measurement_t::bmi (*C++ member*), 112
esp_ble_wss_measurement_t::bmi_height_present (*C++ member*), 111
esp_ble_wss_measurement_t::day (*C++ member*), 112
esp_ble_wss_measurement_t::flag (*C++ member*), 111
esp_ble_wss_measurement_t::height (*C++ member*), 112
esp_ble_wss_measurement_t::height_resolution (*C++ member*), 112
esp_ble_wss_measurement_t::hours (*C++ member*), 112
esp_ble_wss_measurement_t::measurement_unit (*C++ member*), 111
esp_ble_wss_measurement_t::minutes (*C++ member*), 112
esp_ble_wss_measurement_t::month (*C++ member*), 112
esp_ble_wss_measurement_t::seconds (*C++ member*), 112
esp_ble_wss_measurement_t::time_present (*C++ member*), 111
esp_ble_wss_measurement_t::timestamp (*C++ member*), 112
esp_ble_wss_measurement_t::user_id (*C++ member*), 112
esp_ble_wss_measurement_t::user_present (*C++ member*), 111
esp_ble_wss_measurement_t::weight (*C++ member*), 112

esp_ble_wss_measurement_t::weight_resolution (C++ member), 112
esp_ble_wss_measurement_t::year (C++ member), 112
esp_ble_wss_set_measurement (C++ function), 110
ESP_EVENT_DECLARE_BASE (C++ function), 517
ESP_IOT_ATT_ALREADY_OPEN (C macro), 53
ESP_IOT_ATT_APP_RSP (C macro), 54
ESP_IOT_ATT_AUTH_FAIL (C macro), 53
ESP_IOT_ATT_BUSY (C macro), 53
ESP_IOT_ATT_CANCEL (C macro), 53
ESP_IOT_ATT_CCC_CFG_ERR (C macro), 54
ESP_IOT_ATT_CMD_STARTED (C macro), 53
ESP_IOT_ATT_CONGESTED (C macro), 53
ESP_IOT_ATT_DATABASE_OUT_OF_SYNC (C macro), 52
ESP_IOT_ATT_DB_FULL (C macro), 53
ESP_IOT_ATT_DUP_REG (C macro), 53
ESP_IOT_ATT_ENCRYPED_MITM (C macro), 53
ESP_IOT_ATT_ENCRYPED_NO_MITM (C macro), 53
ESP_IOT_ATT_ERR_UNLIKELY (C macro), 52
ESP_IOT_ATT_ERROR (C macro), 53
ESP_IOT_ATT_ILLEGAL_PARAMETER (C macro), 53
ESP_IOT_ATT_INSUF_AUTHENTICATION (C macro), 52
ESP_IOT_ATT_INSUF_AUTHORIZATION (C macro), 52
ESP_IOT_ATT_INSUF_ENCRYPTION (C macro), 52
ESP_IOT_ATT_INSUF_KEY_SIZE (C macro), 52
ESP_IOT_ATT_INSUF_RESOURCE (C macro), 52
ESP_IOT_ATT_INTERNAL_ERROR (C macro), 53
ESP_IOT_ATT_INVALID_ATTR_LEN (C macro), 52
ESP_IOT_ATT_INVALID_CFG (C macro), 53
ESP_IOT_ATT_INVALID_HANDLE (C macro), 52
ESP_IOT_ATT_INVALID_OFFSET (C macro), 52
ESP_IOT_ATT_INVALID_PDU (C macro), 52
ESP_IOT_ATT_MORE (C macro), 53
ESP_IOT_ATT_NO_RESOURCES (C macro), 53
ESP_IOT_ATT_NOT_ENCRYPTED (C macro), 53
ESP_IOT_ATT_NOT_FOUND (C macro), 52
ESP_IOT_ATT_NOT_LONG (C macro), 52
ESP_IOT_ATT_OUT_OF_RANGE (C macro), 54
ESP_IOT_ATT_PENDING (C macro), 53
ESP_IOT_ATT_PRC_IN_PROGRESS (C macro), 54
ESP_IOT_ATT_PREPARE_Q_FULL (C macro), 52
ESP_IOT_ATT_READ_NOT_PERMIT (C macro), 52
ESP_IOT_ATT_REQ_NOT_SUPPORTED (C macro), 52
ESP_IOT_ATT_SERVICE_STARTED (C macro), 53
ESP_IOT_ATT_STACK_RSP (C macro), 53
ESP_IOT_ATT_SUCCESS (C macro), 52
ESP_IOT_ATT_UNKNOWN_ERROR (C macro), 54
ESP_IOT_ATT_UNSUPPORT_GRP_TYPE (C macro), 52
ESP_IOT_ATT_VALUE_NOT_ALLOWED (C macro), 53
ESP_IOT_ATT_WRITE_NOT_PERMIT (C macro), 52
ESP_IOT_ATT_WRONG_STATE (C macro), 53
esp_lv_adapter_config_t (C++ struct), 254
esp_lv_adapter_config_t::stack_in_psram (C++ member), 255
esp_lv_adapter_config_t::task_core_id (C++ member), 255
esp_lv_adapter_config_t::task_max_delay_ms (C++ member), 255
esp_lv_adapter_config_t::task_min_delay_ms (C++ member), 255
esp_lv_adapter_config_t::task_priority (C++ member), 255
esp_lv_adapter_config_t::task_stack_size (C++ member), 254
esp_lv_adapter_config_t::tick_period_ms (C++ member), 255
ESP_LV_ADAPTER_DEFAULT_CONFIG (C macro), 256
ESP_LV_ADAPTER_DEFAULT_STACK_SIZE (C macro), 255
ESP_LV_ADAPTER_DEFAULT_TASK_CORE_ID (C macro), 256
ESP_LV_ADAPTER_DEFAULT_TASK_MAX_DELAY_MS (C macro), 256
ESP_LV_ADAPTER_DEFAULT_TASK_MIN_DELAY_MS (C macro), 256
ESP_LV_ADAPTER_DEFAULT_TASK_PRIORITY (C macro), 255
ESP_LV_ADAPTER_DEFAULT_TICK_PERIOD_MS (C macro), 256
esp_lv_adapter_deinit (C++ function), 251
esp_lv_adapter_dummy_draw_blit (C++ function), 254
esp_lv_adapter_dummy_draw_callbacks_t (C++ struct), 255
esp_lv_adapter_dummy_draw_callbacks_t::on_color_t (C++ member), 255
esp_lv_adapter_dummy_draw_callbacks_t::on_disable (C++ member), 255
esp_lv_adapter_dummy_draw_callbacks_t::on_enable (C++ member), 255
esp_lv_adapter_dummy_draw_callbacks_t::on_vsync (C++ member), 255
esp_lv_adapter_get_dummy_draw_enabled (C++ function), 254
esp_lv_adapter_init (C++ function), 251
esp_lv_adapter_is_initialized (C++ function), 251
esp_lv_adapter_lock (C++ function), 251
esp_lv_adapter_pause (C++ function), 252
esp_lv_adapter_refresh_now (C++ function), 252

esp_lv_adapter_resume (*C++ function*), 252
esp_lv_adapter_set_dummy_draw (*C++ function*), 253
esp_lv_adapter_set_dummy_draw_callbacks (*C++ function*), 254
esp_lv_adapter_sleep_prepare (*C++ function*), 252
esp_lv_adapter_sleep_recover (*C++ function*), 253
esp_lv_adapter_start (*C++ function*), 251
esp_lv_adapter_unlock (*C++ function*), 252
esp_lv_decoder_deinit (*C++ function*), 257
esp_lv_decoder_handle_t (*C++ type*), 258
esp_lv_decoder_init (*C++ function*), 257
esp_lv_fs_desc_deinit (*C++ function*), 259
esp_lv_fs_desc_init (*C++ function*), 259
esp_lv_fs_handle_t (*C++ type*), 259
esp_mcp_mgr_config_t (*C++ struct*), 396
esp_mcp_mgr_config_t::config (*C++ member*), 396
esp_mcp_mgr_config_t::instance (*C++ member*), 396
esp_mcp_mgr_config_t::transport (*C++ member*), 396
esp_mcp_mgr_deinit (*C++ function*), 390
esp_mcp_mgr_handle_t (*C++ type*), 396
esp_mcp_mgr_init (*C++ function*), 390
esp_mcp_mgr_perform_handle (*C++ function*), 392
esp_mcp_mgr_post_info_init (*C++ function*), 392
esp_mcp_mgr_post_tools_call (*C++ function*), 393
esp_mcp_mgr_post_tools_list (*C++ function*), 393
esp_mcp_mgr_register_endpoint (*C++ function*), 391
esp_mcp_mgr_req_destroy_response (*C++ function*), 393
esp_mcp_mgr_req_destroy_response_by_id (*C++ function*), 394
esp_mcp_mgr_req_handle (*C++ function*), 392
esp_mcp_mgr_req_t (*C++ struct*), 394
esp_mcp_mgr_req_t::args_json (*C++ member*), 395
esp_mcp_mgr_req_t::call (*C++ member*), 395
esp_mcp_mgr_req_t::cb (*C++ member*), 394
esp_mcp_mgr_req_t::cursor (*C++ member*), 394
esp_mcp_mgr_req_t::ep_name (*C++ member*), 394
esp_mcp_mgr_req_t::init (*C++ member*), 394
esp_mcp_mgr_req_t::list (*C++ member*), 395
esp_mcp_mgr_req_t::name (*C++ member*), 394
esp_mcp_mgr_req_t::protocol_version (*C++ member*), 394
esp_mcp_mgr_req_t::tool_name (*C++ member*), 395
esp_mcp_mgr_req_t::u (*C++ member*), 395
esp_mcp_mgr_req_t::user_ctx (*C++ member*), 394
esp_mcp_mgr_req_t::version (*C++ member*), 394
esp_mcp_mgr_resp_cb_t (*C++ type*), 396
esp_mcp_mgr_start (*C++ function*), 391
esp_mcp_mgr_stop (*C++ function*), 391
esp_mcp_mgr_unregister_endpoint (*C++ function*), 391
esp_mcp_transport_handle_t (*C++ type*), 396
esp_mcp_transport_s (*C++ struct*), 395
esp_mcp_transport_s::create_config (*C++ member*), 395
esp_mcp_transport_s::deinit (*C++ member*), 395
esp_mcp_transport_s::delete_config (*C++ member*), 395
esp_mcp_transport_s::init (*C++ member*), 395
esp_mcp_transport_s::register_endpoint (*C++ member*), 395
esp_mcp_transport_s::request (*C++ member*), 396
esp_mcp_transport_s::start (*C++ member*), 395
esp_mcp_transport_s::stop (*C++ member*), 395
esp_mcp_transport_s::unregister_endpoint (*C++ member*), 395
esp_mcp_transport_t (*C++ type*), 397
esp_msc_ota (*C++ function*), 318
esp_msc_ota_abort (*C++ function*), 318
esp_msc_ota_begin (*C++ function*), 317
esp_msc_ota_config_t (*C++ struct*), 319
esp_msc_ota_config_t::buffer_size (*C++ member*), 319
esp_msc_ota_config_t::bulk_flash_erase (*C++ member*), 319
esp_msc_ota_config_t::ota_bin_path (*C++ member*), 319
esp_msc_ota_config_t::skip_msc_connect_wait (*C++ member*), 319
esp_msc_ota_config_t::wait_msc_connect (*C++ member*), 319
esp_msc_ota_end (*C++ function*), 318
esp_msc_ota_event_t (*C++ enum*), 320
esp_msc_ota_event_t::ESP_MSC_OTA_ABORT (*C++ enumerator*), 320
esp_msc_ota_event_t::ESP_MSC_OTA_FAILED (*C++ enumerator*), 320
esp_msc_ota_event_t::ESP_MSC_OTA_FINISH (*C++ enumerator*), 320
esp_msc_ota_event_t::ESP_MSC_OTA_GET_IMG_DESC (*C++ enumerator*), 320
esp_msc_ota_event_t::ESP_MSC_OTA_READY_UPDATE (*C++ enumerator*), 320
esp_msc_ota_event_t::ESP_MSC_OTA_START

(C++ *enumerator*), 320
esp_msc_ota_event_t::ESP_MSC_OTA_UPDATE_ROOT_PARTITION (C++ *enumerator*), 320
esp_msc_ota_event_t::ESP_MSC_OTA_VERIFY_CHIP_ID (C++ *enumerator*), 320
esp_msc_ota_event_t::ESP_MSC_OTA_WRITE_FLASH (C++ *enumerator*), 320
esp_msc_ota_get_img_desc (C++ *function*), 319
esp_msc_ota_get_status (C++ *function*), 319
esp_msc_ota_handle_t (C++ *type*), 320
esp_msc_ota_is_complete_data_received (C++ *function*), 319
esp_msc_ota_perform (C++ *function*), 318
esp_msc_ota_set_msc_connect_state (C++ *function*), 319
esp_msc_ota_status_t (C++ *enum*), 320
esp_msc_ota_status_t::ESP_MSC_OTA_BEGIN (C++ *enumerator*), 320
esp_msc_ota_status_t::ESP_MSC_OTA_IN_PROGRESS (C++ *enumerator*), 320
esp_msc_ota_status_t::ESP_MSC_OTA_INIT (C++ *enumerator*), 320
esp_msc_ota_status_t::ESP_MSC_OTA_SUCCESS (C++ *enumerator*), 321
esp_restart_from_tinyuf2 (C++ *function*), 355
esp_tinyuf2_current_state (C++ *function*), 355
esp_tinyuf2_install (C++ *function*), 355
esp_tinyuf2_uninstall (C++ *function*), 355
ESP_USB_DEVICE_MATCH_ID_ANY (C *macro*), 334

F

FLAG_UAC_MIC_SUSPEND_AFTER_START (C *macro*), 315
FLAG_UAC_SPK_SUSPEND_AFTER_START (C *macro*), 315
FLAG_UVC_SUSPEND_AFTER_START (C *macro*), 314
FPS2INTERVAL (C *macro*), 314
frame_data_t (C++ *struct*), 371
frame_data_t::audio_info (C++ *member*), 372
frame_data_t::data (C++ *member*), 371
frame_data_t::data_bytes (C++ *member*), 371
frame_data_t::type (C++ *member*), 371
frame_data_t::video_info (C++ *member*), 372
frame_data_t::[anonymous] (C++ *member*), 372
FRAME_INTERVAL_FPS_10 (C *macro*), 314
FRAME_INTERVAL_FPS_15 (C *macro*), 314
FRAME_INTERVAL_FPS_20 (C *macro*), 314
FRAME_INTERVAL_FPS_30 (C *macro*), 314
FRAME_INTERVAL_FPS_5 (C *macro*), 314
FRAME_RESOLUTION_ANY (C *macro*), 314
frame_type_t (C++ *enum*), 373
frame_type_t::FRAME_TYPE_AUDIO (C++ *enumerator*), 373
frame_type_t::FRAME_TYPE_VIDEO (C++ *enumerator*), 373
FREQ_HZ (C *macro*), 520
fs_cfg_t (C++ *struct*), 259
fs_cfg_t::fs_assets (C++ *member*), 259
fs_cfg_t::fs_letter (C++ *member*), 259
fs_cfg_t::fs_nums (C++ *member*), 259

G

gpio_pad_select_gpio (C *macro*), 14

I

i2c_bus_cmd_begin (C++ *function*), 12
i2c_bus_create (C++ *function*), 8
i2c_bus_delete (C++ *function*), 9
i2c_bus_device_create (C++ *function*), 9
i2c_bus_device_delete (C++ *function*), 9
i2c_bus_device_get_address (C++ *function*), 9
i2c_bus_device_handle_t (C++ *type*), 14
i2c_bus_get_created_device_num (C++ *function*), 9
i2c_bus_get_current_clk_speed (C++ *function*), 9
i2c_bus_get_internal_bus_handle (C++ *function*), 9
i2c_bus_handle_t (C++ *type*), 14
i2c_bus_read_bit (C++ *function*), 10
i2c_bus_read_bits (C++ *function*), 10
i2c_bus_read_byte (C++ *function*), 10
i2c_bus_read_bytes (C++ *function*), 10
i2c_bus_read_reg16 (C++ *function*), 12
i2c_bus_scan (C++ *function*), 9
i2c_bus_write_bit (C++ *function*), 11
i2c_bus_write_bits (C++ *function*), 12
i2c_bus_write_byte (C++ *function*), 11
i2c_bus_write_bytes (C++ *function*), 11
i2c_bus_write_reg16 (C++ *function*), 12
i2c_cmd_handle_t (C++ *type*), 14
i2c_config_t (C++ *struct*), 13
i2c_config_t::clk_flags (C++ *member*), 13
i2c_config_t::clk_speed (C++ *member*), 13
i2c_config_t::master (C++ *member*), 13
i2c_config_t::mode (C++ *member*), 13
i2c_config_t::scl_io_num (C++ *member*), 13
i2c_config_t::scl_pullup_en (C++ *member*), 13
i2c_config_t::sda_io_num (C++ *member*), 13
i2c_config_t::sda_pullup_en (C++ *member*), 13
i2s_lcd_acquire (C++ *function*), 20
i2s_lcd_config_t (C++ *struct*), 20
i2s_lcd_config_t::buffer_size (C++ *member*), 21

i2s_lcd_config_t::clk_freq (*C++ member*), 21
i2s_lcd_config_t::data_width (*C++ member*), 20
i2s_lcd_config_t::i2s_port (*C++ member*), 21
i2s_lcd_config_t::pin_data_num (*C++ member*), 20
i2s_lcd_config_t::pin_num_cs (*C++ member*), 20
i2s_lcd_config_t::pin_num_rs (*C++ member*), 21
i2s_lcd_config_t::pin_num_wr (*C++ member*), 20
i2s_lcd_config_t::swap_data (*C++ member*), 21
i2s_lcd_driver_deinit (*C++ function*), 19
i2s_lcd_driver_init (*C++ function*), 19
i2s_lcd_handle_t (*C++ type*), 21
i2s_lcd_release (*C++ function*), 20
i2s_lcd_write (*C++ function*), 20
i2s_lcd_write_cmd (*C++ function*), 19
i2s_lcd_write_command (*C++ function*), 20
i2s_lcd_write_data (*C++ function*), 19
ina236_clear_mask (*C++ function*), 464
ina236_config_t (*C++ struct*), 465
ina236_config_t::alert_cb (*C++ member*), 465
ina236_config_t::alert_en (*C++ member*), 465
ina236_config_t::alert_pin (*C++ member*), 465
ina236_config_t::bus (*C++ member*), 465
ina236_config_t::dev_addr (*C++ member*), 465
ina236_create (*C++ function*), 464
ina236_delete (*C++ function*), 464
ina236_get_current (*C++ function*), 464
ina236_get_voltage (*C++ function*), 464
ina236_handle_t (*C++ type*), 465
INA236_I2C_ADDRESS_DEFAULT (*C macro*), 465
INJECT_DUTY (*C macro*), 521
INJECT_ENABLE (*C macro*), 521
int236_alert_cb_t (*C++ type*), 465
iot_button_count_cb (*C++ function*), 413
iot_button_count_event_cb (*C++ function*), 413
iot_button_delete (*C++ function*), 412
iot_button_get_event (*C++ function*), 413
iot_button_get_event_str (*C++ function*), 413
iot_button_get_key_level (*C++ function*), 414
iot_button_get_long_press_hold_cnt (*C++ function*), 414
iot_button_get_pressed_time (*C++ function*), 414
iot_button_get_repeat (*C++ function*), 414
iot_button_get_ticks_time (*C++ function*), 414
iot_button_print_event (*C++ function*), 414
iot_button_register_cb (*C++ function*), 412
iot_button_register_power_save_cb (*C++ function*), 415
iot_button_resume (*C++ function*), 414
iot_button_set_param (*C++ function*), 414
iot_button_stop (*C++ function*), 415
iot_button_unregister_cb (*C++ function*), 413
iot_eth_config_t (*C++ struct*), 531
iot_eth_config_t::driver (*C++ member*), 531
iot_eth_config_t::stack_input (*C++ member*), 531
iot_eth_event_t (*C++ enum*), 532
iot_eth_event_t::IOT_ETH_EVENT_CONNECTED (*C++ enumerator*), 532
iot_eth_event_t::IOT_ETH_EVENT_DISCONNECTED (*C++ enumerator*), 532
iot_eth_event_t::IOT_ETH_EVENT_START (*C++ enumerator*), 532
iot_eth_event_t::IOT_ETH_EVENT_STOP (*C++ enumerator*), 532
iot_eth_get_addr (*C++ function*), 531
iot_eth_handle_t (*C++ type*), 531
iot_eth_install (*C++ function*), 531
iot_eth_new_usb_ecm (*C++ function*), 345
iot_eth_new_usb_rndis (*C++ function*), 338
iot_eth_start (*C++ function*), 531
iot_eth_stop (*C++ function*), 531
iot_eth_transmit (*C++ function*), 531
iot_eth_uninstall (*C++ function*), 531
iot_eth_update_input_path (*C++ function*), 531
iot_knob_clear_count_value (*C++ function*), 429
iot_knob_create (*C++ function*), 428
iot_knob_delete (*C++ function*), 428
iot_knob_get_count_value (*C++ function*), 429
iot_knob_get_event (*C++ function*), 429
iot_knob_register_cb (*C++ function*), 428
iot_knob_resume (*C++ function*), 429
iot_knob_stop (*C++ function*), 429
iot_knob_unregister_cb (*C++ function*), 429
iot_sensor_create (*C++ function*), 455
iot_sensor_delete (*C++ function*), 455
iot_sensor_handler_register (*C++ function*), 455
iot_sensor_handler_register_with_type (*C++ function*), 456
iot_sensor_handler_unregister (*C++ function*), 456
iot_sensor_handler_unregister_with_type (*C++ function*), 456
iot_sensor_scan (*C++ function*), 455

iot_sensor_start (*C++ function*), 455
iot_sensor_stop (*C++ function*), 455
iot_servo_deinit (*C++ function*), 525
iot_servo_init (*C++ function*), 525
iot_servo_read_angle (*C++ function*), 526
iot_servo_write_angle (*C++ function*), 525
iot_usbh_ecm_config_t (*C++ struct*), 346
iot_usbh_ecm_config_t::match_id_list (*C++ member*), 346
iot_usbh_rndis_config_t (*C++ struct*), 338
iot_usbh_rndis_config_t::match_id_list (*C++ member*), 338
ir_learn_add_list_node (*C++ function*), 438
ir_learn_add_sub_list_node (*C++ function*), 438
ir_learn_cfg_t (*C++ struct*), 439
ir_learn_cfg_t::callback (*C++ member*), 440
ir_learn_cfg_t::clk_src (*C++ member*), 439
ir_learn_cfg_t::learn_count (*C++ member*), 439
ir_learn_cfg_t::learn_gpio (*C++ member*), 439
ir_learn_cfg_t::resolution (*C++ member*), 439
ir_learn_cfg_t::task_affinity (*C++ member*), 440
ir_learn_cfg_t::task_priority (*C++ member*), 440
ir_learn_cfg_t::task_stack (*C++ member*), 440
ir_learn_check_valid (*C++ function*), 438
ir_learn_clean_data (*C++ function*), 438
ir_learn_clean_sub_data (*C++ function*), 438
ir_learn_handle_t (*C++ type*), 440
ir_learn_list_t (*C++ struct*), 439
ir_learn_list_t::cmd_sub_node (*C++ member*), 439
ir_learn_new (*C++ function*), 437
ir_learn_print_raw (*C++ function*), 438
ir_learn_restart (*C++ function*), 437
ir_learn_result_cb (*C++ type*), 440
ir_learn_state_t (*C++ enum*), 440
ir_learn_state_t::IR_LEARN_STATE_END (*C++ enumerator*), 440
ir_learn_state_t::IR_LEARN_STATE_EXIT (*C++ enumerator*), 440
ir_learn_state_t::IR_LEARN_STATE_FAIL (*C++ enumerator*), 440
ir_learn_state_t::IR_LEARN_STATE_READY (*C++ enumerator*), 440
ir_learn_state_t::IR_LEARN_STATE_STEP (*C++ enumerator*), 440
ir_learn_stop (*C++ function*), 437
ir_learn_sub_list_t (*C++ struct*), 439
ir_learn_sub_list_t::symbols (*C++ member*), 439
ir_learn_sub_list_t::timediff (*C++ member*), 439

K

keyboard_btn_callback_t (*C++ type*), 423
keyboard_btn_cb_config_t (*C++ struct*), 422
keyboard_btn_cb_config_t::callback (*C++ member*), 422
keyboard_btn_cb_config_t::event (*C++ member*), 422
keyboard_btn_cb_config_t::event_data (*C++ member*), 422
keyboard_btn_cb_config_t::user_data (*C++ member*), 422
keyboard_btn_cb_handle_t (*C++ type*), 423
keyboard_btn_config_t (*C++ struct*), 422
keyboard_btn_config_t::active_level (*C++ member*), 423
keyboard_btn_config_t::core_id (*C++ member*), 423
keyboard_btn_config_t::debounce_ticks (*C++ member*), 423
keyboard_btn_config_t::enable_power_save (*C++ member*), 423
keyboard_btn_config_t::input_gpio_num (*C++ member*), 423
keyboard_btn_config_t::input_gpios (*C++ member*), 423
keyboard_btn_config_t::output_gpio_num (*C++ member*), 423
keyboard_btn_config_t::output_gpios (*C++ member*), 422
keyboard_btn_config_t::priority (*C++ member*), 423
keyboard_btn_config_t::ticks_interval (*C++ member*), 423
keyboard_btn_data_t (*C++ struct*), 421
keyboard_btn_data_t::input_index (*C++ member*), 421
keyboard_btn_data_t::output_index (*C++ member*), 421
keyboard_btn_event_data_t (*C++ union*), 421
keyboard_btn_event_data_t::combination (*C++ member*), 421
keyboard_btn_event_data_t::combination_t (*C++ struct*), 421
keyboard_btn_event_data_t::combination_t::key_dat (*C++ member*), 421
keyboard_btn_event_data_t::combination_t::key_num (*C++ member*), 421
keyboard_btn_event_t (*C++ enum*), 423
keyboard_btn_event_t::KBD_EVENT_COMBINATION (*C++ enumerator*), 424
keyboard_btn_event_t::KBD_EVENT_MAX (*C++ enumerator*), 424
keyboard_btn_event_t::KBD_EVENT_PRESSED (*C++ enumerator*), 423
keyboard_btn_handle_t (*C++ type*), 423
keyboard_btn_report_t (*C++ struct*), 422

keyboard_btn_report_t::key_change_num (C++ member), 422
keyboard_btn_report_t::key_data (C++ member), 422
keyboard_btn_report_t::key_pressed_num (C++ member), 422
keyboard_btn_report_t::key_release_data (C++ member), 422
keyboard_btn_report_t::key_release_num (C++ member), 422
keyboard_button_create (C++ function), 419
keyboard_button_delete (C++ function), 420
keyboard_button_get_gpio_by_index (C++ function), 420
keyboard_button_get_index_by_gpio (C++ function), 420
keyboard_button_register_cb (C++ function), 420
keyboard_button_unregister_cb (C++ function), 420
knob_cb_t (C++ type), 430
knob_config_t (C++ struct), 429
knob_config_t::default_direction (C++ member), 429
knob_config_t::enable_power_save (C++ member), 430
knob_config_t::gpio_encoder_a (C++ member), 429
knob_config_t::gpio_encoder_b (C++ member), 430
knob_event_t (C++ enum), 430
knob_event_t::KNOB_EVENT_MAX (C++ enumerator), 430
knob_event_t::KNOB_H_LIM (C++ enumerator), 430
knob_event_t::KNOB_L_LIM (C++ enumerator), 430
knob_event_t::KNOB_LEFT (C++ enumerator), 430
knob_event_t::KNOB_NONE (C++ enumerator), 430
knob_event_t::KNOB_RIGHT (C++ enumerator), 430
knob_event_t::KNOB_ZERO (C++ enumerator), 430
knob_handle_t (C++ type), 430

L

LCD_CMD_LEV (C macro), 21
LCD_DATA_LEV (C macro), 21
LED_INDICATOR_CHECK (C macro), 241
LED_INDICATOR_CHECK_WARNING (C macro), 241
led_indicator_delete (C++ function), 238
led_indicator_get_brightness (C++ function), 239
led_indicator_get_hsv (C++ function), 240
led_indicator_get_rgb (C++ function), 240
led_indicator_preempt_start (C++ function), 239
led_indicator_preempt_stop (C++ function), 239
led_indicator_set_brightness (C++ function), 240
led_indicator_set_color_temperature (C++ function), 241
led_indicator_set_hsv (C++ function), 240
led_indicator_set_on_off (C++ function), 239
led_indicator_set_rgb (C++ function), 240
led_indicator_start (C++ function), 238
led_indicator_stop (C++ function), 239
lightbulb_basic_effect_start (C++ function), 556
lightbulb_basic_effect_stop (C++ function), 557
lightbulb_basic_effect_stop_and_restore (C++ function), 557
lightbulb_capability_t (C++ struct), 560
lightbulb_capability_t::disable_auto_on (C++ member), 561
lightbulb_capability_t::enable_fade (C++ member), 560
lightbulb_capability_t::enable_hardware_cct (C++ member), 561
lightbulb_capability_t::enable_lowpower (C++ member), 560
lightbulb_capability_t::enable_precise_cct_contro (C++ member), 561
lightbulb_capability_t::enable_precise_color_cont (C++ member), 561
lightbulb_capability_t::enable_status_storage (C++ member), 561
lightbulb_capability_t::fade_time_ms (C++ member), 560
lightbulb_capability_t::led_beads (C++ member), 560
lightbulb_capability_t::storage_cb (C++ member), 560
lightbulb_capability_t::storage_delay_ms (C++ member), 560
lightbulb_capability_t::sync_change_brightness_va (C++ member), 561
lightbulb_cct_kelvin_range_t (C++ struct), 560
lightbulb_cct_kelvin_range_t::max (C++ member), 560
lightbulb_cct_kelvin_range_t::min (C++ member), 560
lightbulb_cct_mapping_data_t (C++ struct), 557
lightbulb_cct_mapping_data_t::cct_kelvin (C++ member), 557
lightbulb_cct_mapping_data_t::cct_percentage (C++ member), 557
lightbulb_cct_mapping_data_t::rgbcw

(C++ member), 557
lightbulb_chip_out_pin_t (C++ enum), 567
lightbulb_chip_out_pin_t::OUT1 (C++ enumerator), 567
lightbulb_chip_out_pin_t::OUT2 (C++ enumerator), 567
lightbulb_chip_out_pin_t::OUT3 (C++ enumerator), 567
lightbulb_chip_out_pin_t::OUT4 (C++ enumerator), 567
lightbulb_chip_out_pin_t::OUT5 (C++ enumerator), 567
lightbulb_chip_out_pin_t::OUT_MAX (C++ enumerator), 567
lightbulb_color_mapping_data_t (C++ struct), 557
lightbulb_color_mapping_data_t::hue (C++ member), 558
lightbulb_color_mapping_data_t::rgbcw_01 (C++ member), 558
lightbulb_color_mapping_data_t::rgbcw_100 (C++ member), 558
lightbulb_color_mapping_data_t::rgbcw_50 (C++ member), 558
lightbulb_config_t (C++ struct), 561
lightbulb_config_t::blue (C++ member), 562, 563
lightbulb_config_t::capability (C++ member), 563
lightbulb_config_t::cct_mix_mode (C++ member), 562
lightbulb_config_t::cold_cct (C++ member), 562
lightbulb_config_t::cold_white (C++ member), 563
lightbulb_config_t::color_mix_mode (C++ member), 562
lightbulb_config_t::driver_conf (C++ member), 561
lightbulb_config_t::external_limit (C++ member), 562
lightbulb_config_t::gamma_conf (C++ member), 562
lightbulb_config_t::green (C++ member), 562
lightbulb_config_t::iic_io (C++ member), 563
lightbulb_config_t::init_status (C++ member), 563
lightbulb_config_t::io_conf (C++ member), 563
lightbulb_config_t::kelvin_max (C++ member), 561
lightbulb_config_t::kelvin_min (C++ member), 561
lightbulb_config_t::precise (C++ member), 561, 562
lightbulb_config_t::pwm_io (C++ member), 562
lightbulb_config_t::red (C++ member), 562
lightbulb_config_t::sm16825e_io (C++ member), 563
lightbulb_config_t::standard (C++ member), 561
lightbulb_config_t::table (C++ member), 561, 562
lightbulb_config_t::table_size (C++ member), 561
lightbulb_config_t::type (C++ member), 561
lightbulb_config_t::warm_brightness (C++ member), 562
lightbulb_config_t::warm_yellow (C++ member), 563
lightbulb_deinit (C++ function), 552
lightbulb_driver_t (C++ enum), 564
lightbulb_driver_t::DRIVER_BP1658CJ (C++ enumerator), 565
lightbulb_driver_t::DRIVER_BP57x8D (C++ enumerator), 565
lightbulb_driver_t::DRIVER_ESP_PWM (C++ enumerator), 564
lightbulb_driver_t::DRIVER_KP18058 (C++ enumerator), 565
lightbulb_driver_t::DRIVER_SELECT_INVALID (C++ enumerator), 564
lightbulb_driver_t::DRIVER_SELECT_MAX (C++ enumerator), 565
lightbulb_driver_t::DRIVER_SM16825E (C++ enumerator), 565
lightbulb_driver_t::DRIVER_SM2135E (C++ enumerator), 564
lightbulb_driver_t::DRIVER_SM2135EH (C++ enumerator), 564
lightbulb_driver_t::DRIVER_SM2182E (C++ enumerator), 564
lightbulb_driver_t::DRIVER_SM2x35EGH (C++ enumerator), 564
lightbulb_driver_t::DRIVER_WS2812 (C++ enumerator), 565
lightbulb_effect_config_t (C++ struct), 563
lightbulb_effect_config_t::cct (C++ member), 563
lightbulb_effect_config_t::effect_cycle_ms (C++ member), 564
lightbulb_effect_config_t::effect_type (C++ member), 563
lightbulb_effect_config_t::hue (C++ member), 563
lightbulb_effect_config_t::interrupt_forbidden (C++ member), 564
lightbulb_effect_config_t::max_value_brightness (C++ member), 564
lightbulb_effect_config_t::min_value_brightness (C++ member), 564
lightbulb_effect_config_t::mode (C++

member), 563
lightbulb_effect_config_t::saturation (C++ member), 563
lightbulb_effect_config_t::total_ms (C++ member), 564
lightbulb_effect_config_t::user_cb (C++ member), 564
lightbulb_effect_t (C++ enum), 566
lightbulb_effect_t::EFFECT_BLINK (C++ enumerator), 566
lightbulb_effect_t::EFFECT_BREATH (C++ enumerator), 566
lightbulb_gamma_config_t (C++ struct), 558
lightbulb_gamma_config_t::balance_coefficient (C++ member), 558
lightbulb_gamma_config_t::color_curve_coefficient (C++ member), 558
lightbulb_gamma_config_t::white_curve_coefficient (C++ member), 558
lightbulb_get_all_detail (C++ function), 556
lightbulb_get_brightness (C++ function), 556
lightbulb_get_cct_kelvin (C++ function), 556
lightbulb_get_cct_percentage (C++ function), 556
lightbulb_get_fades_function_status (C++ function), 553
lightbulb_get_hue (C++ function), 555
lightbulb_get_mode (C++ function), 556
lightbulb_get_power_limit (C++ function), 556
lightbulb_get_saturation (C++ function), 555
lightbulb_get_switch (C++ function), 556
lightbulb_get_value (C++ function), 555
lightbulb_hsv2rgb (C++ function), 553
lightbulb_iic_out_pin_t (C++ type), 564
lightbulb_init (C++ function), 552
lightbulb_kelvin2percentage (C++ function), 554
lightbulb_led_beads_comb_t (C++ enum), 565
lightbulb_led_beads_comb_t::LED_BEADS_1CH_C (C++ enumerator), 565
lightbulb_led_beads_comb_t::LED_BEADS_1CH_W (C++ enumerator), 565
lightbulb_led_beads_comb_t::LED_BEADS_2CH_CW (C++ enumerator), 565
lightbulb_led_beads_comb_t::LED_BEADS_3CH_RGB (C++ enumerator), 565
lightbulb_led_beads_comb_t::LED_BEADS_4CH_RGBC (C++ enumerator), 565
lightbulb_led_beads_comb_t::LED_BEADS_4CH_RGBCC (C++ enumerator), 565
lightbulb_led_beads_comb_t::LED_BEADS_4CH_RGBW (C++ enumerator), 565
lightbulb_led_beads_comb_t::LED_BEADS_4CH_RGBWW (C++ enumerator), 565
lightbulb_led_beads_comb_t::LED_BEADS_5CH_RGBC (C++ enumerator), 566
lightbulb_led_beads_comb_t::LED_BEADS_5CH_RGBCC (C++ enumerator), 565
lightbulb_led_beads_comb_t::LED_BEADS_5CH_RGBCW (C++ enumerator), 565
lightbulb_led_beads_comb_t::LED_BEADS_5CH_RGBW (C++ enumerator), 566
lightbulb_led_beads_comb_t::LED_BEADS_5CH_RGBWW (C++ enumerator), 566
lightbulb_led_beads_comb_t::LED_BEADS_INVALID (C++ enumerator), 565
lightbulb_led_beads_comb_t::LED_BEADS_MAX (C++ enumerator), 566
lightbulb_lighting_output_test (C++ function), 557
lightbulb_lighting_unit_t (C++ enum), 566
lightbulb_lighting_unit_t::LIGHTING_ALEXA (C++ enumerator), 566
lightbulb_lighting_unit_t::LIGHTING_ALL_UNIT (C++ enumerator), 567
lightbulb_lighting_unit_t::LIGHTING_BASIC_FIVE (C++ enumerator), 566
lightbulb_lighting_unit_t::LIGHTING_COLD_TO_WARM (C++ enumerator), 566
lightbulb_lighting_unit_t::LIGHTING_COLOR_EFFECT (C++ enumerator), 566
lightbulb_lighting_unit_t::LIGHTING_COLOR_MUTUAL_ (C++ enumerator), 566
lightbulb_lighting_unit_t::LIGHTING_COLOR_VALUE_I (C++ enumerator), 567
lightbulb_lighting_unit_t::LIGHTING_RAINBOW (C++ enumerator), 566
lightbulb_lighting_unit_t::LIGHTING_WARM_TO_COLD (C++ enumerator), 566
lightbulb_lighting_unit_t::LIGHTING_WHITE_BRIGHTN (C++ enumerator), 567
lightbulb_lighting_unit_t::LIGHTING_WHITE_EFFECT (C++ enumerator), 566
lightbulb_percentage2kelvin (C++ function), 554
lightbulb_power_limit_t (C++ struct), 559
lightbulb_power_limit_t::color_max_power (C++ member), 560
lightbulb_power_limit_t::color_max_value (C++ member), 560
lightbulb_power_limit_t::color_min_value (C++ member), 560
lightbulb_power_limit_t::white_max_brightness (C++ member), 559
lightbulb_power_limit_t::white_max_power (C++ member), 560
lightbulb_power_limit_t::white_min_brightness (C++ member), 559
lightbulb_rgb2hsv (C++ function), 553
lightbulb_rgb2xyy (C++ function), 554

lightbulb_set_brightness (*C++ function*), 555
lightbulb_set_cct (*C++ function*), 554
lightbulb_set_cctb (*C++ function*), 555
lightbulb_set_fade_time (*C++ function*), 552
lightbulb_set_fades_function (*C++ function*), 552
lightbulb_set_hsv (*C++ function*), 555
lightbulb_set_hue (*C++ function*), 554
lightbulb_set_saturation (*C++ function*), 554
lightbulb_set_storage_function (*C++ function*), 552
lightbulb_set_switch (*C++ function*), 555
lightbulb_set_value (*C++ function*), 554
lightbulb_set_xyy (*C++ function*), 555
lightbulb_status_erase_nvs_storage (*C++ function*), 556
lightbulb_status_get_from_nvs (*C++ function*), 556
lightbulb_status_set_to_nvs (*C++ function*), 556
lightbulb_status_storage_cb_t (*C++ type*), 564
lightbulb_status_t (*C++ struct*), 558
lightbulb_status_t::brightness (*C++ member*), 559
lightbulb_status_t::cct_percentage (*C++ member*), 559
lightbulb_status_t::hue (*C++ member*), 559
lightbulb_status_t::mode (*C++ member*), 559
lightbulb_status_t::on (*C++ member*), 559
lightbulb_status_t::saturation (*C++ member*), 559
lightbulb_status_t::value (*C++ member*), 559
lightbulb_update_status (*C++ function*), 553
lightbulb_works_mode_t (*C++ enum*), 567
lightbulb_works_mode_t::WORK_COLOR (*C++ enumerator*), 567
lightbulb_works_mode_t::WORK_INVALID (*C++ enumerator*), 567
lightbulb_works_mode_t::WORK_WHITE (*C++ enumerator*), 567
lightbulb_xyy2rgb (*C++ function*), 553

M

MAX_BLE_DEVNAME_LEN (*C macro*), 54
MAX_SPEED_MEASUREMENT_FACTOR (*C macro*), 523
MCPWM_CLK_SRC (*C macro*), 520
MCPWM_PERIOD (*C macro*), 520
mic_callback_t (*C++ type*), 315
mic_frame_t (*C++ struct*), 311
mic_frame_t::bit_resolution (*C++ member*), 311
mic_frame_t::data (*C++ member*), 311
mic_frame_t::data_bytes (*C++ member*), 311
mic_frame_t::samples_frequence (*C++ member*), 311
MIN (*C macro*), 55
mmap_assets_config_t (*C++ struct*), 265
mmap_assets_config_t::app_bin_check (*C++ member*), 265
mmap_assets_config_t::checksum (*C++ member*), 265
mmap_assets_config_t::flags (*C++ member*), 266
mmap_assets_config_t::full_check (*C++ member*), 265
mmap_assets_config_t::max_files (*C++ member*), 265
mmap_assets_config_t::metadata_check (*C++ member*), 266
mmap_assets_config_t::mmap_enable (*C++ member*), 265
mmap_assets_config_t::partition_label (*C++ member*), 265
mmap_assets_config_t::reserved (*C++ member*), 266
mmap_assets_config_t::use_fs (*C++ member*), 265
mmap_assets_copy_by_index (*C++ function*), 264
mmap_assets_copy_mem (*C++ function*), 264
mmap_assets_del (*C++ function*), 264
mmap_assets_get_height (*C++ function*), 265
mmap_assets_get_mem (*C++ function*), 264
mmap_assets_get_name (*C++ function*), 264
mmap_assets_get_size (*C++ function*), 264
mmap_assets_get_stored_files (*C++ function*), 265
mmap_assets_get_width (*C++ function*), 265
mmap_assets_handle_t (*C++ type*), 266
mmap_assets_new (*C++ function*), 263

N

ntc_circuit_mode_t (*C++ enum*), 463
ntc_circuit_mode_t::CIRCUIT_MODE_NTC_GND (*C++ enumerator*), 463
ntc_circuit_mode_t::CIRCUIT_MODE_NTC_VCC (*C++ enumerator*), 463
ntc_config_t (*C++ struct*), 462
ntc_config_t::atten (*C++ member*), 463
ntc_config_t::b_value (*C++ member*), 463
ntc_config_t::channel (*C++ member*), 463
ntc_config_t::circuit_mode (*C++ member*), 463
ntc_config_t::fixed_ohm (*C++ member*), 463
ntc_config_t::r25_ohm (*C++ member*), 463
ntc_config_t::unit (*C++ member*), 463
ntc_config_t::vdd_mv (*C++ member*), 463
ntc_dev_create (*C++ function*), 462
ntc_dev_delete (*C++ function*), 462
ntc_dev_get_adc_handle (*C++ function*), 462

ntc_dev_get_temperature (*C++ function*), 462
ntc_device_handle_t (*C++ type*), 463
NULL_I2C_DEV_ADDR (*C macro*), 14
NULL_I2C_MEM_16BIT_ADDR (*C macro*), 13
NULL_I2C_MEM_ADDR (*C macro*), 13
NULL_SPI_CS_PIN (*C macro*), 18
nvs_modified_cb_t (*C++ type*), 356

O

OpenAI (*C++ struct*), 387
OpenAI_Audio_Input_Format (*C++ enum*), 389
OpenAI_Audio_Input_Format::OPENAI_AUDIO_INPUT_FORMAT_M4A (*C++ enumerator*), 389
OpenAI_Audio_Input_Format::OPENAI_AUDIO_INPUT_FORMAT_MP3 (*C++ enumerator*), 389
OpenAI_Audio_Input_Format::OPENAI_AUDIO_INPUT_FORMAT_MP4 (*C++ enumerator*), 389
OpenAI_Audio_Input_Format::OPENAI_AUDIO_INPUT_FORMAT_MPEG (*C++ enumerator*), 389
OpenAI_Audio_Input_Format::OPENAI_AUDIO_INPUT_FORMAT_MPGA (*C++ enumerator*), 389
OpenAI_Audio_Input_Format::OPENAI_AUDIO_INPUT_FORMAT_WAV (*C++ enumerator*), 389
OpenAI_Audio_Input_Format::OPENAI_AUDIO_INPUT_FORMAT_WEBM (*C++ enumerator*), 389
OpenAI_Audio_Output_Format (*C++ enum*), 389
OpenAI_Audio_Output_Format::OPENAI_AUDIO_OUTPUT_FORMAT_AAC (*C++ enumerator*), 389
OpenAI_Audio_Output_Format::OPENAI_AUDIO_OUTPUT_FORMAT_FLAC (*C++ enumerator*), 390
OpenAI_Audio_Output_Format::OPENAI_AUDIO_OUTPUT_FORMAT_MAX (*C++ enumerator*), 390
OpenAI_Audio_Output_Format::OPENAI_AUDIO_OUTPUT_FORMAT_MP3 (*C++ enumerator*), 389
OpenAI_Audio_Output_Format::OPENAI_AUDIO_OUTPUT_FORMAT_OPUS (*C++ enumerator*), 389
OpenAI_Audio_Output_Format::OPENAI_AUDIO_OUTPUT_FORMAT_PCM (*C++ enumerator*), 390
OpenAI_Audio_Output_Format::OPENAI_AUDIO_OUTPUT_FORMAT_WAV (*C++ enumerator*), 390
OpenAI_Audio_Response_Format (*C++ enum*), 389
OpenAI_Audio_Response_Format::OPENAI_AUDIO_RESPONSE_FORMAT_JSON (*C++ enumerator*), 389
OpenAI_Audio_Response_Format::OPENAI_AUDIO_RESPONSE_FORMAT_SRT (*C++ enumerator*), 389
OpenAI_Audio_Response_Format::OPENAI_AUDIO_RESPONSE_FORMAT_TEXT (*C++ enumerator*), 389
OpenAI_Audio_Response_Format::OPENAI_AUDIO_RESPONSE_FORMAT_VERBOSE_JSON (*C++ enumerator*), 389
OpenAI_Audio_Response_Format::OPENAI_AUDIO_RESPONSE_FORMAT_VTT (*C++ enumerator*), 389
OpenAI_AudioSpeech (*C++ struct*), 386
OpenAI_AudioSpeech::setModel (*C++ member*), 386
OpenAI_AudioSpeech::setResponseFormat (*C++ member*), 386
OpenAI_AudioSpeech::setSpeed (*C++ member*), 386
OpenAI_AudioSpeech::setVoice (*C++ member*), 386
OpenAI_AudioSpeech::speech (*C++ member*), 386
OpenAI_AudioSpeech::speechStream (*C++ member*), 386
OpenAI_AudioSpeech_t (*C++ type*), 388
OpenAI_AudioTranscription (*C++ struct*), 385
OpenAI_AudioTranscription::file (*C++ member*), 386
OpenAI_AudioTranscription::setLanguage (*C++ member*), 385
OpenAI_AudioTranscription::setPrompt (*C++ member*), 385
OpenAI_AudioTranscription::setResponseFormat (*C++ member*), 385
OpenAI_AudioTranscription::setTemperature (*C++ member*), 385
OpenAI_AudioTranscription_t (*C++ type*), 388
OpenAI_AudioTranslation (*C++ struct*), 387
OpenAI_AudioTranslation::file (*C++ member*), 387
OpenAI_AudioTranslation::setPrompt (*C++ member*), 387
OpenAI_AudioTranslation::setResponseFormat (*C++ member*), 387
OpenAI_AudioTranslation::setTemperature (*C++ member*), 387
OpenAI_AudioTranslation_t (*C++ type*), 388
OpenAI_ChatCompletion (*C++ struct*), 381
OpenAI_ChatCompletion::clearConversation (*C++ member*), 382
OpenAI_ChatCompletion::message (*C++ member*), 382
OpenAI_ChatCompletion::multiModalMessage (*C++ member*), 382
OpenAI_ChatCompletion::setFrequencyPenalty (*C++ member*), 381
OpenAI_ChatCompletion::setMaxTokens (*C++ member*), 381
OpenAI_ChatCompletion::setModel (*C++ member*), 381
OpenAI_ChatCompletion::setPresencePenalty (*C++ member*), 381
OpenAI_ChatCompletion::setStop (*C++ member*), 381
OpenAI_ChatCompletion::setSystem (*C++ member*), 381
OpenAI_ChatCompletion::setTemperature (*C++ member*), 381
OpenAI_ChatCompletion::setTopP (*C++ member*), 381
OpenAI_ChatCompletion::setUser (*C++ member*), 382
OpenAI_ChatCompletion_t (*C++ type*), 388

OpenAI_Completion (*C++ struct*), 379
OpenAI_Completion::prompt (*C++ member*), 380
OpenAI_Completion::setBestOf (*C++ member*), 380
OpenAI_Completion::setEcho (*C++ member*), 380
OpenAI_Completion::setFrequencyPenalty (*C++ member*), 380
OpenAI_Completion::setMaxTokens (*C++ member*), 379
OpenAI_Completion::setModel (*C++ member*), 379
OpenAI_Completion::setN (*C++ member*), 380
OpenAI_Completion::setPresencePenalty (*C++ member*), 380
OpenAI_Completion::setStop (*C++ member*), 380
OpenAI_Completion::setTemperature (*C++ member*), 379
OpenAI_Completion::setTopP (*C++ member*), 380
OpenAI_Completion::setUser (*C++ member*), 380
OpenAI_Completion_t (*C++ type*), 388
OpenAI_Edit (*C++ struct*), 382
OpenAI_Edit::process (*C++ member*), 383
OpenAI_Edit::setModel (*C++ member*), 382
OpenAI_Edit::setN (*C++ member*), 383
OpenAI_Edit::setTemperature (*C++ member*), 382
OpenAI_Edit::setTopP (*C++ member*), 382
OpenAI_Edit_t (*C++ type*), 388
OpenAI_EmbeddingData_t (*C++ struct*), 376
OpenAI_EmbeddingData_t::data (*C++ member*), 376
OpenAI_EmbeddingData_t::len (*C++ member*), 376
OpenAI_EmbeddingResponse (*C++ struct*), 376
OpenAI_EmbeddingResponse::deleteResponse (*C++ member*), 377
OpenAI_EmbeddingResponse::getData (*C++ member*), 377
OpenAI_EmbeddingResponse::getError (*C++ member*), 377
OpenAI_EmbeddingResponse::getLen (*C++ member*), 376
OpenAI_EmbeddingResponse::getUsage (*C++ member*), 376
OpenAI_EmbeddingResponse_t (*C++ type*), 387
OpenAI_Image_Response_Format (*C++ enum*), 389
OpenAI_Image_Response_Format::OPENAI_IMAGE_RESPONSE_FORMAT_B64_JSON (*C++ enumerator*), 389
OpenAI_Image_Response_Format::OPENAI_IMAGE_RESPONSE_FORMAT_URL (*C++ enumerator*), 389
OpenAI_Image_Size (*C++ enum*), 388
OpenAI_Image_Size::OPENAI_IMAGE_SIZE_1024x1024 (*C++ enumerator*), 388
OpenAI_Image_Size::OPENAI_IMAGE_SIZE_256x256 (*C++ enumerator*), 388
OpenAI_Image_Size::OPENAI_IMAGE_SIZE_512x512 (*C++ enumerator*), 388
OpenAI_ImageEdit (*C++ struct*), 384
OpenAI_ImageEdit::image (*C++ member*), 385
OpenAI_ImageEdit::setN (*C++ member*), 385
OpenAI_ImageEdit::setPrompt (*C++ member*), 384
OpenAI_ImageEdit::setResponseFormat (*C++ member*), 385
OpenAI_ImageEdit::setSize (*C++ member*), 384
OpenAI_ImageEdit::setUser (*C++ member*), 385
OpenAI_ImageEdit_t (*C++ type*), 388
OpenAI_ImageGeneration (*C++ struct*), 383
OpenAI_ImageGeneration::prompt (*C++ member*), 383
OpenAI_ImageGeneration::setN (*C++ member*), 383
OpenAI_ImageGeneration::setResponseFormat (*C++ member*), 383
OpenAI_ImageGeneration::setSize (*C++ member*), 383
OpenAI_ImageGeneration::setUser (*C++ member*), 383
OpenAI_ImageGeneration_t (*C++ type*), 388
OpenAI_ImageResponse (*C++ struct*), 377
OpenAI_ImageResponse::deleteResponse (*C++ member*), 378
OpenAI_ImageResponse::getData (*C++ member*), 378
OpenAI_ImageResponse::getError (*C++ member*), 378
OpenAI_ImageResponse::getLen (*C++ member*), 378
OpenAI_ImageResponse_t (*C++ type*), 388
OpenAI_ImageVariation (*C++ struct*), 383
OpenAI_ImageVariation::image (*C++ member*), 384
OpenAI_ImageVariation::setN (*C++ member*), 384
OpenAI_ImageVariation::setResponseFormat (*C++ member*), 384
OpenAI_ImageVariation::setSize (*C++ member*), 384
OpenAI_ImageVariation::setUser (*C++ member*), 384
OpenAI_ImageVariation_t (*C++ type*), 388
OpenAI_ModerationResponse (*C++ struct*), 377
OpenAI_ModerationResponse::deleteResponse (*C++ member*), 377
OpenAI_ModerationResponse::getData (*C++ member*), 377
OpenAI_ModerationResponse::getError (*C++ member*), 377

OpenAI_ModerationResponse::getLen (*C++ member*), 377
OpenAI_ModerationResponse_t (*C++ type*), 387
OpenAI_SpeechResponse (*C++ struct*), 379
OpenAI_SpeechResponse::deleteResponse (*C++ member*), 379
OpenAI_SpeechResponse::getData (*C++ member*), 379
OpenAI_SpeechResponse::getLen (*C++ member*), 379
OpenAI_SpeechResponse_t (*C++ type*), 388
OpenAI_StreamCallback (*C++ type*), 387
OpenAI_StringResponse (*C++ struct*), 378
OpenAI_StringResponse::deleteResponse (*C++ member*), 379
OpenAI_StringResponse::getData (*C++ member*), 378
OpenAI_StringResponse::getError (*C++ member*), 378
OpenAI_StringResponse::getLen (*C++ member*), 378
OpenAI_StringResponse::getUsage (*C++ member*), 378
OpenAI_StringResponse_t (*C++ type*), 388
OpenAI_t (*C++ type*), 388
OpenAIChangeBaseURL (*C++ function*), 376
OpenAICreate (*C++ function*), 376
OpenAIDelete (*C++ function*), 376

P

POLE_PAIR (*C macro*), 523
portTICK_RATE_MS (*C macro*), 14
power_measure_calibrate_current (*C++ function*), 472
power_measure_calibrate_power (*C++ function*), 472
power_measure_calibrate_voltage (*C++ function*), 472
power_measure_config_t (*C++ struct*), 473
power_measure_config_t::enable_energy_detection (*C++ member*), 473
power_measure_config_t::overcurrent (*C++ member*), 473
power_measure_config_t::overvoltage (*C++ member*), 473
power_measure_config_t::undervoltage (*C++ member*), 473
power_measure_delete (*C++ function*), 470
power_measure_get_active_power (*C++ function*), 471
power_measure_get_apparent_power (*C++ function*), 471
power_measure_get_current (*C++ function*), 471
power_measure_get_current_multiplier (*C++ function*), 473
power_measure_get_energy (*C++ function*), 471
power_measure_get_power_factor (*C++ function*), 471
power_measure_get_power_multiplier (*C++ function*), 473
power_measure_get_voltage (*C++ function*), 470
power_measure_get_voltage_multiplier (*C++ function*), 472
power_measure_handle_t (*C++ type*), 473
power_measure_reset_energy_calculation (*C++ function*), 471
proxi_cb_t (*C++ type*), 484
proxi_state_t (*C++ enum*), 484
proxi_state_t::PROXI_STATE_ACTIVE (*C++ enumerator*), 484
proxi_state_t::PROXI_STATE_INACTIVE (*C++ enumerator*), 484
pwm_audio_channel_t (*C++ enum*), 361
pwm_audio_channel_t::PWM_AUDIO_CH_MAX (*C++ enumerator*), 361
pwm_audio_channel_t::PWM_AUDIO_CH_MONO (*C++ enumerator*), 361
pwm_audio_channel_t::PWM_AUDIO_CH_STEREO (*C++ enumerator*), 361
pwm_audio_config_t (*C++ struct*), 360
pwm_audio_config_t::duty_resolution (*C++ member*), 361
pwm_audio_config_t::gpio_num_left (*C++ member*), 361
pwm_audio_config_t::gpio_num_right (*C++ member*), 361
pwm_audio_config_t::ledc_channel_left (*C++ member*), 361
pwm_audio_config_t::ledc_channel_right (*C++ member*), 361
pwm_audio_config_t::ledc_timer_sel (*C++ member*), 361
pwm_audio_config_t::ringbuf_len (*C++ member*), 361
pwm_audio_deinit (*C++ function*), 359
pwm_audio_get_param (*C++ function*), 360
pwm_audio_get_status (*C++ function*), 360
pwm_audio_get_volume (*C++ function*), 360
pwm_audio_init (*C++ function*), 358
pwm_audio_set_param (*C++ function*), 359
pwm_audio_set_sample_rate (*C++ function*), 360
pwm_audio_set_volume (*C++ function*), 360
pwm_audio_start (*C++ function*), 359
pwm_audio_status_t (*C++ enum*), 361
pwm_audio_status_t::PWM_AUDIO_STATUS_BUSY (*C++ enumerator*), 361
pwm_audio_status_t::PWM_AUDIO_STATUS_IDLE (*C++ enumerator*), 361
pwm_audio_status_t::PWM_AUDIO_STATUS_UN_INIT (*C++ enumerator*), 361

pwm_audio_stop (*C++ function*), 359
pwm_audio_write (*C++ function*), 359
PWM_DUTYCYCLE_05 (*C macro*), 521
PWM_DUTYCYCLE_10 (*C macro*), 521
PWM_DUTYCYCLE_100 (*C macro*), 521
PWM_DUTYCYCLE_15 (*C macro*), 521
PWM_DUTYCYCLE_20 (*C macro*), 521
PWM_DUTYCYCLE_25 (*C macro*), 521
PWM_DUTYCYCLE_30 (*C macro*), 521
PWM_DUTYCYCLE_40 (*C macro*), 521
PWM_DUTYCYCLE_50 (*C macro*), 521
PWM_DUTYCYCLE_60 (*C macro*), 521
PWM_DUTYCYCLE_80 (*C macro*), 521
PWM_DUTYCYCLE_90 (*C macro*), 521
PWM_MODE (*C macro*), 520

R

RAMP_DUTY_END (*C macro*), 522
RAMP_DUTY_INC (*C macro*), 522
RAMP_DUTY_STA (*C macro*), 522
RAMP_TIM_END (*C macro*), 522
RAMP_TIM_STA (*C macro*), 522
RAMP_TIM_STEP (*C macro*), 522

S

sensor_command_t (*C++ enum*), 452
sensor_command_t::COMMAND_MAX (*C++ enumerator*), 452
sensor_command_t::COMMAND_SELF_TEST (*C++ enumerator*), 452
sensor_command_t::COMMAND_SET_MODE (*C++ enumerator*), 452
sensor_command_t::COMMAND_SET_ODR (*C++ enumerator*), 452
sensor_command_t::COMMAND_SET_POWER (*C++ enumerator*), 452
sensor_command_t::COMMAND_SET_RANGE (*C++ enumerator*), 452
sensor_config_t (*C++ struct*), 457
sensor_config_t::addr (*C++ member*), 457
sensor_config_t::bus (*C++ member*), 457
sensor_config_t::intr_pin (*C++ member*), 457
sensor_config_t::intr_type (*C++ member*), 457
sensor_config_t::min_delay (*C++ member*), 457
sensor_config_t::mode (*C++ member*), 457
sensor_config_t::range (*C++ member*), 457
sensor_config_t::type (*C++ member*), 457
sensor_data_event_id_t (*C++ enum*), 453
sensor_data_event_id_t::SENSOR_ACCE_DATA_READY (*C++ enumerator*), 453
sensor_data_event_id_t::SENSOR_BARO_DATA_READY (*C++ enumerator*), 454
sensor_data_event_id_t::SENSOR_CURRENT_DATA_READY (*C++ enumerator*), 454
sensor_data_event_id_t::SENSOR_EVENT_ID_END (*C++ enumerator*), 454
sensor_data_event_id_t::SENSOR_FORCE_DATA_READY (*C++ enumerator*), 454
sensor_data_event_id_t::SENSOR_GYRO_DATA_READY (*C++ enumerator*), 453
sensor_data_event_id_t::SENSOR_HR_DATA_READY (*C++ enumerator*), 454
sensor_data_event_id_t::SENSOR_HUMI_DATA_READY (*C++ enumerator*), 454
sensor_data_event_id_t::SENSOR_LIGHT_DATA_READY (*C++ enumerator*), 454
sensor_data_event_id_t::SENSOR_MAG_DATA_READY (*C++ enumerator*), 454
sensor_data_event_id_t::SENSOR_NOISE_DATA_READY (*C++ enumerator*), 454
sensor_data_event_id_t::SENSOR_PROXI_DATA_READY (*C++ enumerator*), 454
sensor_data_event_id_t::SENSOR_RGBW_DATA_READY (*C++ enumerator*), 454
sensor_data_event_id_t::SENSOR_STEP_DATA_READY (*C++ enumerator*), 454
sensor_data_event_id_t::SENSOR_TEMP_DATA_READY (*C++ enumerator*), 454
sensor_data_event_id_t::SENSOR_TVOC_DATA_READY (*C++ enumerator*), 454
sensor_data_event_id_t::SENSOR_UV_DATA_READY (*C++ enumerator*), 454
sensor_data_event_id_t::SENSOR_VOLTAGE_DATA_READY (*C++ enumerator*), 454
sensor_data_group_t (*C++ struct*), 451
sensor_data_group_t::number (*C++ member*), 451
sensor_data_group_t::sensor_data (*C++ member*), 451
sensor_data_t (*C++ struct*), 449
sensor_data_t::acce (*C++ member*), 449
sensor_data_t::baro (*C++ member*), 450
sensor_data_t::current (*C++ member*), 450
sensor_data_t::data (*C++ member*), 451
sensor_data_t::event_id (*C++ member*), 449
sensor_data_t::force (*C++ member*), 450
sensor_data_t::gyro (*C++ member*), 450
sensor_data_t::hr (*C++ member*), 450
sensor_data_t::humidity (*C++ member*), 450
sensor_data_t::light (*C++ member*), 450
sensor_data_t::mag (*C++ member*), 450
sensor_data_t::min_delay (*C++ member*), 449
sensor_data_t::noise (*C++ member*), 450
sensor_data_t::proximity (*C++ member*), 450
sensor_data_t::rgbw (*C++ member*), 450
sensor_data_t::sensor_addr (*C++ member*), 449
sensor_data_t::sensor_name (*C++ member*), 449
sensor_data_t::sensor_type (*C++ member*),

449
sensor_data_t::step (*C++ member*), 450
sensor_data_t::temperature (*C++ member*), 450
sensor_data_t::timestamp (*C++ member*), 449
sensor_data_t::tvoc (*C++ member*), 450
sensor_data_t::uv (*C++ member*), 450
sensor_data_t::voltage (*C++ member*), 450
sensor_device_impl_t (*C++ type*), 451
sensor_driver_handle_t (*C++ type*), 451
SENSOR_EVENT_ANY_ID (*C macro*), 451
sensor_event_handler_instance_t (*C++ type*), 458
sensor_event_handler_t (*C++ type*), 458
sensor_event_id_t (*C++ enum*), 453
sensor_event_id_t::SENSOR_EVENT_COMMON_END (*C++ enumerator*), 453
sensor_event_id_t::SENSOR_STARTED (*C++ enumerator*), 453
sensor_event_id_t::SENSOR_STOPED (*C++ enumerator*), 453
sensor_handle_t (*C++ type*), 458
SENSOR_HUB_DETECT_FN (*C macro*), 458
sensor_hub_detect_fn_t (*C++ struct*), 457
sensor_hub_detect_fn_t::fn (*C++ member*), 457
sensor_info_t (*C++ struct*), 456
sensor_info_t::name (*C++ member*), 456
sensor_info_t::sensor_type (*C++ member*), 457
sensor_mode_t (*C++ enum*), 452
sensor_mode_t::MODE_DEFAULT (*C++ enumerator*), 452
sensor_mode_t::MODE_INTERRUPT (*C++ enumerator*), 453
sensor_mode_t::MODE_MAX (*C++ enumerator*), 453
sensor_mode_t::MODE_POLLING (*C++ enumerator*), 453
sensor_power_mode_t (*C++ enum*), 452
sensor_power_mode_t::POWER_MAX (*C++ enumerator*), 452
sensor_power_mode_t::POWER_MODE_SLEEP (*C++ enumerator*), 452
sensor_power_mode_t::POWER_MODE_WAKEUP (*C++ enumerator*), 452
sensor_range_t (*C++ enum*), 453
sensor_range_t::RANGE_DEFAULT (*C++ enumerator*), 453
sensor_range_t::RANGE_MAX (*C++ enumerator*), 453
sensor_range_t::RANGE_MEDIUM (*C++ enumerator*), 453
sensor_range_t::RANGE_MIN (*C++ enumerator*), 453
sensor_type_t (*C++ enum*), 451
sensor_type_t::HUMITURE_ID (*C++ enumerator*), 451
sensor_type_t::IMU_ID (*C++ enumerator*), 451
sensor_type_t::LIGHT_SENSOR_ID (*C++ enumerator*), 452
sensor_type_t::NULL_ID (*C++ enumerator*), 451
sensor_type_t::SENSOR_TYPE_MAX (*C++ enumerator*), 452
servo_channel_t (*C++ struct*), 526
servo_channel_t::ch (*C++ member*), 526
servo_channel_t::servo_pin (*C++ member*), 526
servo_config_t (*C++ struct*), 526
servo_config_t::channel_number (*C++ member*), 527
servo_config_t::channels (*C++ member*), 527
servo_config_t::freq (*C++ member*), 526
servo_config_t::max_angle (*C++ member*), 526
servo_config_t::max_width_us (*C++ member*), 526
servo_config_t::min_width_us (*C++ member*), 526
servo_config_t::timer_number (*C++ member*), 526
SPEED_CAL_TYPE (*C macro*), 523
SPEED_KD (*C macro*), 523
SPEED_KI (*C macro*), 523
SPEED_KP (*C macro*), 523
SPEED_MAX_INTEGRAL (*C macro*), 523
SPEED_MAX_OUTPUT (*C macro*), 523
SPEED_MAX_RPM (*C macro*), 523
SPEED_MIN_INTEGRAL (*C macro*), 523
SPEED_MIN_OUTPUT (*C macro*), 523
SPEED_MIN_RPM (*C macro*), 523
speed_mode_t (*C++ enum*), 520
speed_mode_t::SPEED_CLOSED_LOOP (*C++ enumerator*), 520
speed_mode_t::SPEED_OPEN_LOOP (*C++ enumerator*), 520
spi_bus_create (*C++ function*), 15
spi_bus_delete (*C++ function*), 15
spi_bus_device_create (*C++ function*), 16
spi_bus_device_delete (*C++ function*), 16
spi_bus_device_handle_t (*C++ type*), 18
spi_bus_handle_t (*C++ type*), 18
spi_bus_transfer_byte (*C++ function*), 16
spi_bus_transfer_bytes (*C++ function*), 16
spi_bus_transfer_reg16 (*C++ function*), 17
spi_bus_transfer_reg32 (*C++ function*), 17
spi_bus_transmit_begin (*C++ function*), 16
spi_config_t (*C++ struct*), 17
spi_config_t::max_transfer_sz (*C++ member*), 17
spi_config_t::miso_io_num (*C++ member*), 17
spi_config_t::mosi_io_num (*C++ member*),

17
spi_config_t::sclk_io_num (*C++ member*), 17
spi_device_config_t (*C++ struct*), 17
spi_device_config_t::clock_speed_hz (*C++ member*), 18
spi_device_config_t::cs_io_num (*C++ member*), 18
spi_device_config_t::mode (*C++ member*), 18
state_callback_t (*C++ type*), 315
static_input_cb_t (*C++ type*), 531
stream_ctrl_t (*C++ enum*), 316
stream_ctrl_t::CTRL_MAX (*C++ enumerator*), 316
stream_ctrl_t::CTRL_NONE (*C++ enumerator*), 316
stream_ctrl_t::CTRL_RESUME (*C++ enumerator*), 316
stream_ctrl_t::CTRL_SUSPEND (*C++ enumerator*), 316
stream_ctrl_t::CTRL_UAC_MUTE (*C++ enumerator*), 316
stream_ctrl_t::CTRL_UAC_VOLUME (*C++ enumerator*), 316

T

tinyuf2_nvs_config_t (*C++ struct*), 355
tinyuf2_nvs_config_t::modified_cb (*C++ member*), 356
tinyuf2_nvs_config_t::namespace_name (*C++ member*), 356
tinyuf2_nvs_config_t::part_name (*C++ member*), 356
tinyuf2_ota_config_t (*C++ struct*), 355
tinyuf2_ota_config_t::complete_cb (*C++ member*), 355
tinyuf2_ota_config_t::if_restart (*C++ member*), 355
tinyuf2_ota_config_t::label (*C++ member*), 355
tinyuf2_ota_config_t::subtype (*C++ member*), 355
tinyuf2_state_t (*C++ enum*), 356
tinyuf2_state_t::TINYUF2_STATE_INSTALLED (*C++ enumerator*), 356
tinyuf2_state_t::TINYUF2_STATE_MOUNTED (*C++ enumerator*), 356
tinyuf2_state_t::TINYUF2_STATE_NOT_INSTALLED (*C++ enumerator*), 356
touch_button_config_t (*C++ struct*), 488
touch_button_config_t::channel_gold_value (*C++ member*), 489
touch_button_config_t::channel_list (*C++ member*), 489
touch_button_config_t::channel_num (*C++ member*), 489
touch_button_config_t::channel_threshold (*C++ member*), 489
touch_button_config_t::debounce_times (*C++ member*), 489
touch_button_config_t::skip_lowlevel_init (*C++ member*), 489
touch_button_handle_t (*C++ type*), 489
touch_button_sensor_create (*C++ function*), 487
touch_button_sensor_delete (*C++ function*), 487
touch_button_sensor_get_data (*C++ function*), 487
touch_button_sensor_get_state (*C++ function*), 488
touch_button_sensor_get_state_bitmap (*C++ function*), 488
touch_button_sensor_handle_events (*C++ function*), 488
touch_cb_t (*C++ type*), 489
touch_proxi_config_t (*C++ struct*), 484
touch_proxi_config_t::channel_gold_value (*C++ member*), 484
touch_proxi_config_t::channel_list (*C++ member*), 484
touch_proxi_config_t::channel_num (*C++ member*), 484
touch_proxi_config_t::channel_threshold (*C++ member*), 484
touch_proxi_config_t::debounce_times (*C++ member*), 484
touch_proxi_config_t::skip_lowlevel_init (*C++ member*), 484
touch_proximity_handle_t (*C++ type*), 484
touch_proximity_sensor_create (*C++ function*), 482
touch_proximity_sensor_delete (*C++ function*), 483
touch_proximity_sensor_get_data (*C++ function*), 483
touch_proximity_sensor_get_state (*C++ function*), 483
touch_proximity_sensor_handle_events (*C++ function*), 483
touch_slider_config_t (*C++ struct*), 494
touch_slider_config_t::calculate_window (*C++ member*), 495
touch_slider_config_t::channel_gold_value (*C++ member*), 495
touch_slider_config_t::channel_list (*C++ member*), 494
touch_slider_config_t::channel_num (*C++ member*), 494
touch_slider_config_t::channel_threshold (*C++ member*), 494
touch_slider_config_t::debounce_times (*C++ member*), 495
touch_slider_config_t::filter_reset_times

(C++ *member*), 495
touch_slider_config_t::position_range (C++ *member*), 495
touch_slider_config_t::skip_lowlevel_init (C++ *member*), 495
touch_slider_config_t::swipe_alpha (C++ *member*), 495
touch_slider_config_t::swipe_hysterisis (C++ *member*), 495
touch_slider_config_t::swipe_threshold (C++ *member*), 495
touch_slider_event_cb_t (C++ *type*), 495
touch_slider_event_t (C++ *enum*), 495
touch_slider_event_t::TOUCH_SLIDER_EVENT_LEFT_SWIPE (C++ *enumerator*), 496
touch_slider_event_t::TOUCH_SLIDER_EVENT_NONE (C++ *enumerator*), 496
touch_slider_event_t::TOUCH_SLIDER_EVENT_POSITION (C++ *enumerator*), 496
touch_slider_event_t::TOUCH_SLIDER_EVENT_RELEASE (C++ *enumerator*), 496
touch_slider_event_t::TOUCH_SLIDER_EVENT_RIGHT_SWIPE (C++ *enumerator*), 496
touch_slider_handle_t (C++ *type*), 495
touch_slider_sensor_create (C++ *function*), 493
touch_slider_sensor_delete (C++ *function*), 493
touch_slider_sensor_get_data (C++ *function*), 494
touch_slider_sensor_get_state (C++ *function*), 494
touch_slider_sensor_handle_events (C++ *function*), 494
touch_state_t (C++ *enum*), 489
touch_state_t::TOUCH_STATE_ACTIVE (C++ *enumerator*), 489
touch_state_t::TOUCH_STATE_INACTIVE (C++ *enumerator*), 489

U

UAC_BITS_ANY (C *macro*), 314
UAC_CH_ANY (C *macro*), 314
uac_config_t (C++ *struct*), 312
uac_config_t::ac_interface (C++ *member*), 314
uac_config_t::flags (C++ *member*), 314
uac_config_t::mic_bit_resolution (C++ *member*), 313
uac_config_t::mic_buf_size (C++ *member*), 313
uac_config_t::mic_cb (C++ *member*), 313
uac_config_t::mic_cb_arg (C++ *member*), 313
uac_config_t::mic_ch_num (C++ *member*), 313
uac_config_t::mic_ep_addr (C++ *member*), 313
uac_config_t::mic_ep_mps (C++ *member*), 313
uac_config_t::mic_fu_id (C++ *member*), 314
uac_config_t::mic_interface (C++ *member*), 313
uac_config_t::mic_samples_frequence (C++ *member*), 313
uac_config_t::spk_bit_resolution (C++ *member*), 313
uac_config_t::spk_buf_size (C++ *member*), 313
uac_config_t::spk_ch_num (C++ *member*), 313
uac_config_t::spk_ep_addr (C++ *member*), 313
uac_config_t::spk_ep_mps (C++ *member*), 314
uac_config_t::spk_fu_id (C++ *member*), 314
uac_config_t::spk_interface (C++ *member*), 313
uac_config_t::spk_samples_frequence (C++ *member*), 313
uac_device_config_t (C++ *struct*), 351
uac_device_config_t::cb_ctx (C++ *member*), 351
uac_device_config_t::input_cb (C++ *member*), 351
uac_device_config_t::output_cb (C++ *member*), 351
uac_device_config_t::set_mute_cb (C++ *member*), 351
uac_device_config_t::set_volume_cb (C++ *member*), 351
uac_device_config_t::skip_tinyusb_init (C++ *member*), 351
uac_device_init (C++ *function*), 350
uac_frame_size_list_get (C++ *function*), 309
uac_frame_size_reset (C++ *function*), 309
uac_frame_size_t (C++ *struct*), 312
uac_frame_size_t::bit_resolution (C++ *member*), 312
uac_frame_size_t::ch_num (C++ *member*), 312
uac_frame_size_t::samples_frequence (C++ *member*), 312
uac_frame_size_t::samples_frequence_max (C++ *member*), 312
uac_frame_size_t::samples_frequence_min (C++ *member*), 312
UAC_FREQUENCY_ANY (C *macro*), 314
uac_input_cb_t (C++ *type*), 351
uac_mic_streaming_read (C++ *function*), 309
uac_output_cb_t (C++ *type*), 351
uac_set_mute_cb_t (C++ *type*), 351
uac_set_volume_cb_t (C++ *type*), 351
uac_spk_streaming_write (C++ *function*), 309
uac_streaming_config (C++ *function*), 308
UF2_RESET_REASON_VALUE (C *macro*), 356

uint24_t (*C++ struct*), 68
uint24_t::u24 (*C++ member*), 68
update_complete_cb_t (*C++ type*), 356
usb_dev_match_flags_t (*C++ enum*), 334
usb_dev_match_flags_t::USB_DEVICE_ID_MATCH_ALL (*C++ enumerator*), 334
usb_dev_match_flags_t::USB_DEVICE_ID_MATCH_DEV (*C++ enumerator*), 334
usb_dev_match_flags_t::USB_DEVICE_ID_MATCH_DEV_BCD (*C++ enumerator*), 334
usb_dev_match_flags_t::USB_DEVICE_ID_MATCH_DEV_CLASS (*C++ enumerator*), 334
usb_dev_match_flags_t::USB_DEVICE_ID_MATCH_DEV_PROTOCOL (*C++ enumerator*), 334
usb_dev_match_flags_t::USB_DEVICE_ID_MATCH_DEV_SUBCLASS (*C++ enumerator*), 334
usb_dev_match_flags_t::USB_DEVICE_ID_MATCH_INT_CLASS (*C++ enumerator*), 334
usb_dev_match_flags_t::USB_DEVICE_ID_MATCH_INT_NUMBER (*C++ enumerator*), 334
usb_dev_match_flags_t::USB_DEVICE_ID_MATCH_INT_PROTOCOL (*C++ enumerator*), 334
usb_dev_match_flags_t::USB_DEVICE_ID_MATCH_INT_SUBCLASS (*C++ enumerator*), 334
usb_dev_match_flags_t::USB_DEVICE_ID_MATCH_PRODUCT (*C++ enumerator*), 334
usb_dev_match_flags_t::USB_DEVICE_ID_MATCH_VENDOR (*C++ enumerator*), 334
usb_dev_match_flags_t::USB_DEVICE_ID_MATCH_VID_PID (*C++ enumerator*), 335
usb_device_match_id_t (*C++ struct*), 333
usb_device_match_id_t::bcdDevice (*C++ member*), 333
usb_device_match_id_t::bDeviceClass (*C++ member*), 333
usb_device_match_id_t::bDeviceProtocol (*C++ member*), 333
usb_device_match_id_t::bDeviceSubClass (*C++ member*), 333
usb_device_match_id_t::bInterfaceClass (*C++ member*), 333
usb_device_match_id_t::bInterfaceNumber (*C++ member*), 334
usb_device_match_id_t::bInterfaceProtocol (*C++ member*), 334
usb_device_match_id_t::bInterfaceSubClass (*C++ member*), 333
usb_device_match_id_t::idProduct (*C++ member*), 333
usb_device_match_id_t::idVendor (*C++ member*), 333
usb_device_match_id_t::match_flags (*C++ member*), 333
USB_DEVICE_PRODUCT_ANY (*C macro*), 334
USB_DEVICE_VENDOR_ANY (*C macro*), 334
usb_ecm_get_cdc_port_handle (*C++ function*), 345
usb_rndis_get_cdc_port_handle (*C++ function*), 338
usb_stream_state_t (*C++ enum*), 316
usb_stream_state_t::STREAM_CONNECTED (*C++ enumerator*), 316
usb_stream_state_t::STREAM_DISCONNECTED (*C++ enumerator*), 316
usb_stream_t (*C++ enum*), 315
usb_stream_t::STREAM_MAX (*C++ enumerator*), 316
usb_stream_t::STREAM_UAC_MIC (*C++ enumerator*), 315
usb_stream_t::STREAM_UAC_SPK (*C++ enumerator*), 315
usb_stream_t::STREAM_UVC (*C++ enumerator*), 315
usb_streaming_connect_wait (*C++ function*), 308
usb_streaming_control (*C++ function*), 308
usb_streaming_start (*C++ function*), 308
usb_streaming_state_register (*C++ function*), 308
usb_streaming_stop (*C++ function*), 308
usbh_intf_desc_print (*C++ function*), 328
usbh_cdc_device_event_callback_t (*C++ type*), 331
usbh_cdc_device_event_data_t (*C++ union*), 329
usbh_cdc_device_event_data_t::active_config_desc (*C++ member*), 329
usbh_cdc_device_event_data_t::dev_addr (*C++ member*), 329
usbh_cdc_device_event_data_t::dev_gone (*C++ member*), 329
usbh_cdc_device_event_data_t::dev_hdl (*C++ member*), 329
usbh_cdc_device_event_data_t::device_desc (*C++ member*), 329
usbh_cdc_device_event_data_t::matched_intf_num (*C++ member*), 329
usbh_cdc_device_event_data_t::new_dev (*C++ member*), 329
usbh_cdc_device_event_t (*C++ enum*), 331
usbh_cdc_device_event_t::CDC_HOST_DEVICE_EVENT_CO (*C++ enumerator*), 331
usbh_cdc_device_event_t::CDC_HOST_DEVICE_EVENT_DI (*C++ enumerator*), 331
usbh_cdc_driver_config_t (*C++ struct*), 329
usbh_cdc_driver_config_t::skip_init_usb_host_driv (*C++ member*), 330
usbh_cdc_driver_config_t::task_coreid (*C++ member*), 330
usbh_cdc_driver_config_t::task_priority (*C++ member*), 330
usbh_cdc_driver_config_t::task_stack_size (*C++ member*), 330
usbh_cdc_driver_install (*C++ function*), 325

usbh_cdc_driver_uninstall (C++ function), 325
usbh_cdc_flush_rx_buffer (C++ function), 328
usbh_cdc_flush_tx_buffer (C++ function), 328
usbh_cdc_get_dev_handle (C++ function), 328
usbh_cdc_get_rx_buffer_size (C++ function), 328
usbh_cdc_port_close (C++ function), 326
usbh_cdc_port_config_t (C++ struct), 330
usbh_cdc_port_config_t::cbs (C++ member), 331
usbh_cdc_port_config_t::dev_addr (C++ member), 330
usbh_cdc_port_config_t::flags (C++ member), 331
usbh_cdc_port_config_t::in_ringbuf_size (C++ member), 330
usbh_cdc_port_config_t::in_transfer_buffer_size (C++ member), 331
usbh_cdc_port_config_t::itf_num (C++ member), 330
usbh_cdc_port_config_t::out_ringbuf_size (C++ member), 331
usbh_cdc_port_config_t::out_transfer_buffer_size (C++ member), 331
usbh_cdc_port_event_callbacks_t (C++ struct), 330
usbh_cdc_port_event_callbacks_t::closed (C++ member), 330
usbh_cdc_port_event_callbacks_t::notif_cb (C++ member), 330
usbh_cdc_port_event_callbacks_t::recv_data (C++ member), 330
usbh_cdc_port_event_callbacks_t::user_data (C++ member), 330
usbh_cdc_port_flags_t (C++ enum), 331
usbh_cdc_port_flags_t::USBH_CDC_FLAGS_DISABLE_NOTIFICATION (C++ enumerator), 331
usbh_cdc_port_get_intf_desc (C++ function), 329
usbh_cdc_port_handle_t (C++ type), 331
usbh_cdc_port_open (C++ function), 326
usbh_cdc_read_bytes (C++ function), 327
usbh_cdc_register_dev_event_cb (C++ function), 325
usbh_cdc_send_custom_request (C++ function), 326
usbh_cdc_unregister_dev_event_cb (C++ function), 326
usbh_cdc_write_bytes (C++ function), 327
usbh_match_device (C++ function), 332
usbh_match_device_from_id_list (C++ function), 332
usbh_match_id_from_list (C++ function), 333
usbh_match_interface (C++ function), 332
usbh_match_interface_in_configuration (C++ function), 332
usbh_modem_config_t (C++ struct), 341
usbh_modem_config_t::at_rx_buffer_size (C++ member), 341
usbh_modem_config_t::at_tx_buffer_size (C++ member), 341
usbh_modem_config_t::modem_id_list (C++ member), 341
usbh_modem_get_atparser (C++ function), 341
usbh_modem_get_netif (C++ function), 340
usbh_modem_install (C++ function), 340
usbh_modem_ppp_auto_connect (C++ function), 340
usbh_modem_ppp_start (C++ function), 341
usbh_modem_ppp_stop (C++ function), 341
usbh_modem_uninstall (C++ function), 340
uvc_config (C++ struct), 310
uvc_config::ep_addr (C++ member), 311
uvc_config::ep_mps (C++ member), 311
uvc_config::flags (C++ member), 311
uvc_config::format (C++ member), 311
uvc_config::format_index (C++ member), 311
uvc_config::frame_buffer (C++ member), 310
uvc_config::frame_buffer_size (C++ member), 310
uvc_config::frame_cb (C++ member), 310
uvc_config::frame_cb_arg (C++ member), 311
uvc_config::frame_height (C++ member), 310
uvc_config::frame_index (C++ member), 311
uvc_config::frame_interval (C++ member), 310
uvc_config::frame_width (C++ member), 310
uvc_config::interface (C++ member), 311
uvc_config::interface_alt (C++ member), 311
uvc_config::xfer_buffer_a (C++ member), 310
uvc_config::xfer_buffer_b (C++ member), 310
uvc_config::xfer_buffer_size (C++ member), 310
uvc_config::xfer_type (C++ member), 311
uvc_config_t (C++ type), 315
uvc_device_config (C++ function), 347
uvc_device_config_t (C++ struct), 348
uvc_device_config_t::cb_ctx (C++ member), 349
uvc_device_config_t::fb_get_cb (C++ member), 349
uvc_device_config_t::fb_return_cb (C++ member), 349
uvc_device_config_t::start_cb (C++ member), 348
uvc_device_config_t::stop_cb (C++ mem-

ber), 349
uvc_device_config_t::uvc_buffer (C++ member), 348
uvc_device_config_t::uvc_buffer_size (C++ member), 348
uvc_device_deinit (C++ function), 348
uvc_device_init (C++ function), 348
uvc_fb_t (C++ struct), 348
uvc_fb_t::buf (C++ member), 348
uvc_fb_t::format (C++ member), 348
uvc_fb_t::height (C++ member), 348
uvc_fb_t::len (C++ member), 348
uvc_fb_t::timestamp (C++ member), 348
uvc_fb_t::width (C++ member), 348
uvc_format_t (C++ enum), 349
uvc_format_t::UVC_FORMAT_H264 (C++ enumerator), 349
uvc_format_t::UVC_FORMAT_JPEG (C++ enumerator), 349
uvc_frame_size_list_get (C++ function), 309
uvc_frame_size_reset (C++ function), 310
uvc_frame_size_t (C++ struct), 312
uvc_frame_size_t::height (C++ member), 312
uvc_frame_size_t::interval (C++ member), 312
uvc_frame_size_t::interval_max (C++ member), 312
uvc_frame_size_t::interval_min (C++ member), 312
uvc_frame_size_t::interval_step (C++ member), 312
uvc_frame_size_t::width (C++ member), 312
uvc_input_fb_get_cb_t (C++ type), 349
uvc_input_fb_return_cb_t (C++ type), 349
uvc_input_start_cb_t (C++ type), 349
uvc_input_stop_cb_t (C++ type), 349
uvc_streaming_config (C++ function), 308
uvc_xfer_t (C++ enum), 315
uvc_xfer_t::UVC_XFER_BULK (C++ enumerator), 315
uvc_xfer_t::UVC_XFER_ISOC (C++ enumerator), 315
uvc_xfer_t::UVC_XFER_UNKNOWN (C++ enumerator), 315

V

video_frame_format (C++ enum), 373
video_frame_format::FORMAT_H264 (C++ enumerator), 373
video_frame_format::FORMAT_MJEPG (C++ enumerator), 373
video_frame_info_t (C++ struct), 371
video_frame_info_t::frame_format (C++ member), 371
video_frame_info_t::height (C++ member), 371
video_frame_info_t::width (C++ member), 371
video_write_cb (C++ type), 373

Z

ZERO_CROSS_ADVANCE (C macro), 522
zero_cross_cb_t (C++ type), 577
ZERO_CROSS_DETECTION_ACCURACY (C macro), 522
zero_detect_cb_param_t (C++ union), 574
zero_detect_cb_param_t::signal_falling_edge_event (C++ member), 575
zero_detect_cb_param_t::signal_falling_edge_event (C++ struct), 575
zero_detect_cb_param_t::signal_falling_edge_event (C++ member), 575
zero_detect_cb_param_t::signal_falling_edge_event (C++ member), 575
zero_detect_cb_param_t::signal_falling_edge_event (C++ member), 575
zero_detect_cb_param_t::signal_freq_event_data (C++ member), 574
zero_detect_cb_param_t::signal_freq_event_data_t (C++ struct), 575
zero_detect_cb_param_t::signal_freq_event_data_t: (C++ member), 575
zero_detect_cb_param_t::signal_freq_event_data_t: (C++ member), 575
zero_detect_cb_param_t::signal_invalid_event_data (C++ member), 574
zero_detect_cb_param_t::signal_invalid_event_data (C++ struct), 575
zero_detect_cb_param_t::signal_invalid_event_data (C++ member), 575
zero_detect_cb_param_t::signal_invalid_event_data (C++ member), 575
zero_detect_cb_param_t::signal_invalid_event_data (C++ member), 575
zero_detect_cb_param_t::signal_rising_edge_event_ (C++ member), 574
zero_detect_cb_param_t::signal_rising_edge_event_ (C++ struct), 575
zero_detect_cb_param_t::signal_rising_edge_event_ (C++ member), 576
zero_detect_cb_param_t::signal_rising_edge_event_ (C++ member), 576
zero_detect_cb_param_t::signal_rising_edge_event_ (C++ member), 576
zero_detect_cb_param_t::signal_valid_event_data (C++ member), 574
zero_detect_cb_param_t::signal_valid_event_data_t (C++ struct), 576
zero_detect_cb_param_t::signal_valid_event_data_t (C++ member), 576
zero_detect_cb_param_t::signal_valid_event_data_t (C++ member), 576
zero_detect_cb_param_t::signal_valid_event_data_t (C++ member), 576

zero_detect_config_t (*C++ struct*), 576
zero_detect_config_t::capture_pin (*C++ member*), 576
zero_detect_config_t::freq_range_max_hz (*C++ member*), 577
zero_detect_config_t::freq_range_min_hz (*C++ member*), 577
zero_detect_config_t::invalid_times (*C++ member*), 576
zero_detect_config_t::signal_lost_time_us (*C++ member*), 576
zero_detect_config_t::valid_times (*C++ member*), 576
zero_detect_config_t::zero_driver_type (*C++ member*), 576
zero_detect_config_t::zero_signal_type (*C++ member*), 576
zero_detect_create (*C++ function*), 573
zero_detect_delete (*C++ function*), 573
zero_detect_event_t (*C++ enum*), 577
zero_detect_event_t::SIGNAL_FALLING_EDGE (*C++ enumerator*), 577
zero_detect_event_t::SIGNAL_FREQ_OUT_OF_RANGE (*C++ enumerator*), 577
zero_detect_event_t::SIGNAL_INVALID (*C++ enumerator*), 577
zero_detect_event_t::SIGNAL_LOST (*C++ enumerator*), 577
zero_detect_event_t::SIGNAL_RISING_EDGE (*C++ enumerator*), 577
zero_detect_event_t::SIGNAL_VALID (*C++ enumerator*), 577
zero_detect_get_power_status (*C++ function*), 574
zero_detect_get_signal_type (*C++ function*), 574
zero_detect_handle_t (*C++ type*), 577
zero_detect_pause (*C++ function*), 573
zero_detect_register_cb (*C++ function*), 574
zero_detect_resume (*C++ function*), 573
zero_detect_singal_invaild_status (*C++ function*), 574
ZERO_DETECTION_INIT_CONFIG_DEFAULT (*C macro*), 577
zero_driver_type_t (*C++ enum*), 578
zero_driver_type_t::GPIO_TYPE (*C++ enumerator*), 578
ZERO_REPEAT_TIME (*C macro*), 522
zero_show_data (*C++ function*), 573
zero_signal_edge_t (*C++ enum*), 578
zero_signal_edge_t::CAP_EDGE_NEG (*C++ enumerator*), 578
zero_signal_edge_t::CAP_EDGE_POS (*C++ enumerator*), 578
zero_signal_type_t (*C++ enum*), 577
zero_signal_type_t::PULSE_WAVE (*C++ enumerator*), 577
zero_signal_type_t::SQUARE_WAVE (*C++ enumerator*), 577
ZERO_STABLE_FLAG_CNT (*C macro*), 522