ESP-IoT-Solution User Guide

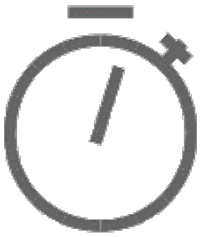
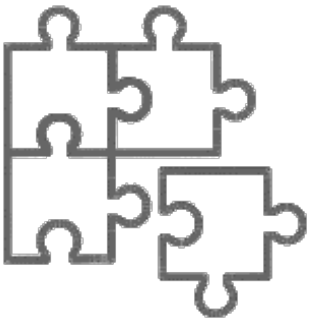
Release master
Espressif Systems
Nov 19, 2024

Table of contents

Index **399**

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 is intended to help you set up the development environment for ESP-IoT-Solution (Espressif IoT Solution). After that, a simple example will show you how to use ESP-IoT-Solution to set up environment, create a project, build and flash firmware onto an ESP series board, etc.

1.1 ESP-IoT-Solution Introduction

ESP-IoT-Solution contains peripheral drivers and code frameworks commonly used in IoT system development, which provide complementary components to ESP-IDF to facilitate simpler development, mainly including the following contents:

- Device drivers such as sensors, screens, audio devices, input devices, actuators, and etc.
- Code framework and related documents of low power management, security encryption, storage and etc.
- Entrance guideline for Espressif's open-source solutions from the perspective of practical application.

1.1.1 ESP-IoT-Solution Versions

Specifications of different ESP-IoT-Solution versions are listed in the following table:

ESP-IoT-Solution version	Corresponding ESP-IDF version	Main changes	Status
master	>=v4.4	support component manager and new chips	New features development branch
release/v1.1	v4.0.1	IDF version updated, deleted codes that have been moved to other repos	Stop maintenance
release/v1.0	v3.2.2	Legacy version	Stop maintenance

Since the `master` branch uses the `ESP Component Manager` to manager components, each of them is a separate package, and each package may support a different version of the ESP-IDF, which will be declared in the component's `idf_component.yml` file.

1.2 ESP-IDF Introduction

ESP-IDF is the IoT development framework for ESP series SoCs provided by Espressif, including:

- A series of libraries and header files, providing core components required for building software projects based on ESP SoC;

- Common tools and functions used during the development and manufacturing processes, e.g., build, flashing, debugging, measurement and etc.

Note: For detailed information, please go to ESP-IDF Programming Guide.

1.3 ESP Series SoC Introduction

You can select any development board from ESP series to get started with ESP-IoT-Solution, or select a supported board from the Boards Component directly for a quick start.

ESP series SoC support the following features:

- 2.4 GHz Wi-Fi
- Bluetooth
- High-performance single core, dual-core processor, capable of running at 240 MHz
- Ultra-low-power co-processor
- Various peripherals including GPIO, I2C, I2S, SPI ,UART, SDIO, RMT, LEDC PWM, Ethernet, TWAI®, Touch, USB OTG and etc.
- Rich memory resources, including up to 520 KB internal RAM and can support external PSRAM
- Support security functions, e.g., hardware encryption

ESP series of SoC are designed with the 40nm technology, showing the best power and RF performance, versatility and reliability in a wide variety of application and power scenarios.

Note: The configuration varies by SoC series, please refer to ESP Product Selector for details.

1.4 Setting up Development Environment

1.4.1 1. Get ESP-IDF

As ESP-IoT-Solution relies on ESP-IDF basic functions and build tools, please set up ESP-IDF development environment first following ESP-IDF Installation Step by Step. Please note that different versions of ESP-IoT-Solution may rely on different ESP-IDF versions, please refer to *ESP-IoT-Solution Versions* for specifications.

1.4.2 2. Get ESP-IoT-Solution

For `master` version, please use the following command:

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

For `release/v1.1` version, please use the following command:

```
git clone -b release/v1.1 --recursive https://github.com/espressif/esp-iot-solution
```

For other versions, please also use this command with `release/v1.1` replaced by your target branch name.

1.5 Use ESP-IoT-Solution Components

If you just want to use the components in ESP-IoT-Solution, we recommend you use it from the ESP Component Registry.

The registered components in ESP-IoT-Solution are listed in README.md , You can directly add the components from the Component Registry to your project by using the `idf.py add-dependency` command under your project's root directory. eg run `idf.py add-dependency "espressif/usb_stream"` to add the `usb_stream`, the component will be downloaded automatically during the `CMake` step.

Please refer to IDF Component Manager for details.

1.6 Build and Download

1.6.1 1. Set up the environment variables

The tools installed in above steps are not yet added to the PATH environment variables. To make the tools usable from the command line, please follow the following steps to add environment variables:

- Add ESP-IDF environment variables:
  For Windows system, please open the Command Prompt and run:

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

  For Linux and macOS, please run:

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

  Please remember to replace the paths in above commands as your actual paths.
- Add IOT_SOLUTION_PATH environment variables:
  For Windows system, please open the Command Prompt and run:

```
set IOT_SOLUTION_PATH=C:\esp\esp-iot-solution
```

  For Linux and macOS, please run:

```
export IOT_SOLUTION_PATH=~/esp/esp-iot-solution
```

Note: The environment variables set by the above method are only valid in the current terminal. Please repeat above steps if you open a new terminal.

1.6.2 2. Set build target

ESP-IDF supports multiple chips as `esp32`, `esp32s2` and others, please set your target chip before building (the default target is `esp32`). For example, you can set the build target as `esp32s2`.

```
idf.py set-target esp32s2
```

For examples in ESP-IoT-Solution developed based on Boards Component, you can go to `Board Options -> Choose Target Board` in `menuconfig` to choose a target board:

```
idf.py menuconfig
```

1.6.3 3. Build and download the program

Use the `idf.py` tool to build and download the program with:

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

Please replace PORT with your board's port name. Serial ports have the following patterns in their names: Windows is like `COMx`; Linux starting with `/dev/ttyUSBx`; macOS usually is `/dev/cu.`.

1.6.4 4. Serial print log

Use the `idf.py` tool to see logs:

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

Do not forget to replace PORT with your serial port name (`COMx` for Windows; `/dev/ttyUSBx` for Linux; `/dev/cu.` for macOS).

1.7 Related Documents

- ESP-IDF Installation Step by Step
- ESP-IDF Get Started
- ESP Product Selector

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.

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
- **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

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;
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 = {
```

(continues on next page)

(continued from previous page)

```
    .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.

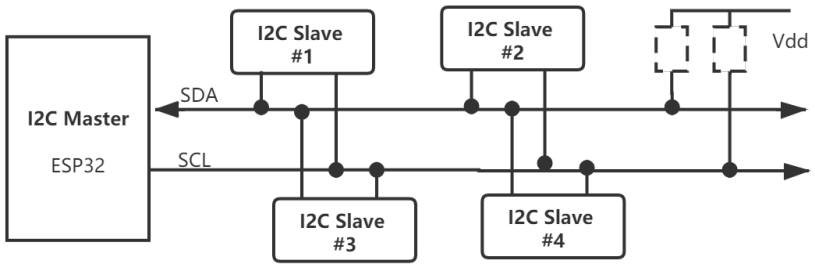


Fig. 1: i2c_bus Connection Diagram

2.2 I2S LCD 驱动

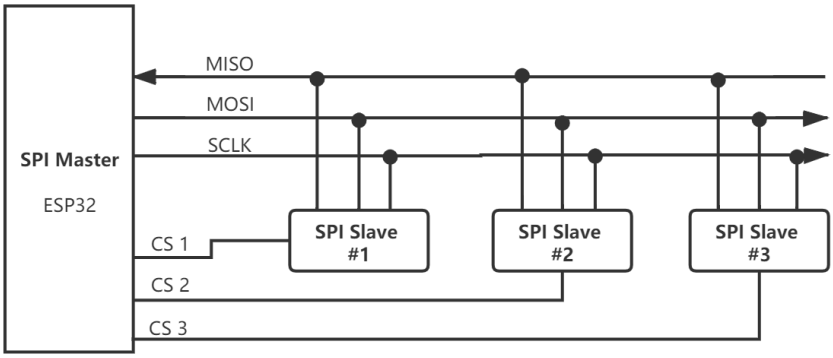


Fig. 2: spi_bus Connection Diagram

2.2.1 API 参考

Header File

- components/bus/include/i2s_lcd_driver.h

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
  - `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.

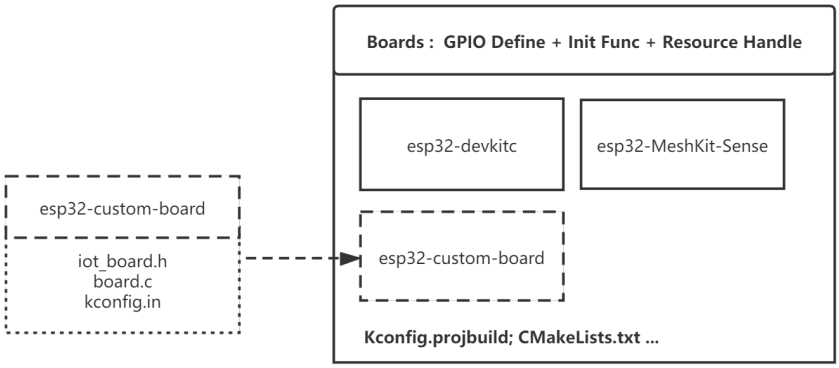


Fig. 3: Boards Component Diagram

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.
    */
}
```

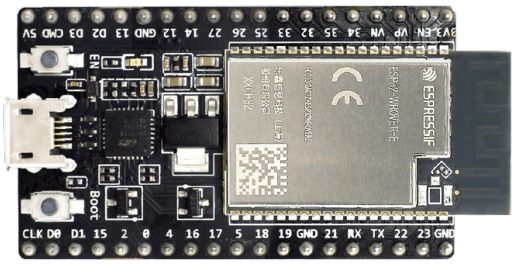
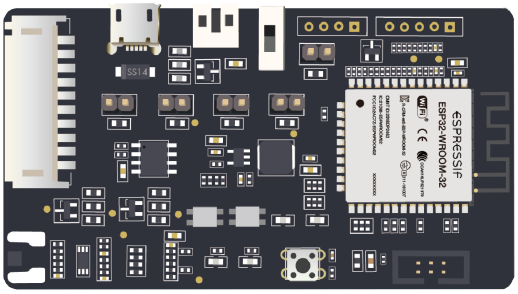
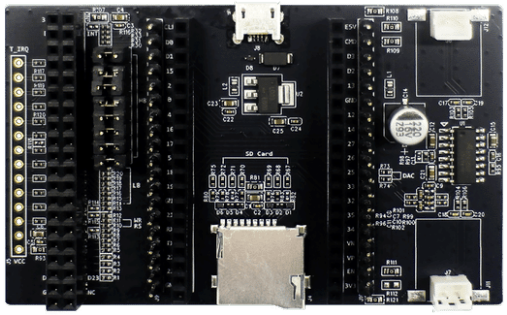
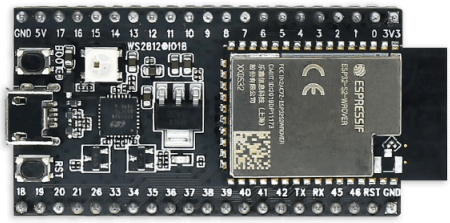
2.3.2 The Switch and Configuration of a Development Board

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

2.4.2 Use

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

```
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.

2.4.3 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 provide 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.

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_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)

This api is typically used to update maximum transmission unit value.

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 on wrong connect
- 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 on wrong disconnect
- ESP_FAIL on other error

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_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_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_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_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_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

uint8_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]

BLE device name being broadcast

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

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

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

Macros

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_UUID_CMP(type, src, dst)

BLE_UUID_TYPE(type)

MIN(a, b)

Type Definitions

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)

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
Return

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

Enumerations

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

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 Services

3.2.1 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

3.2.2 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

3.2.3 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

3.2.4 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

3.2.5 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

3.2.6 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 BLE Profiles

3.3.1 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**

3.3.2 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

3.3.3 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 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

3.4.1 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.

3.4.2 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.

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* 。

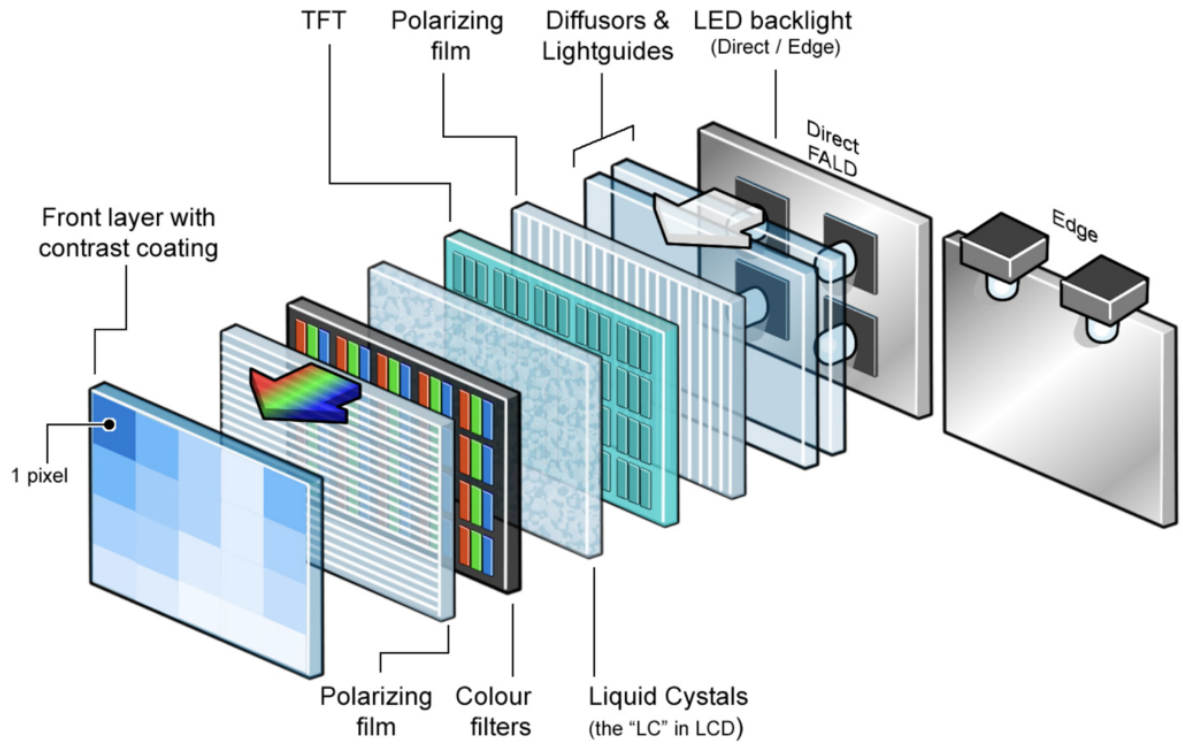


Fig. 1: TFT-LCD Hardware Diagram

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.

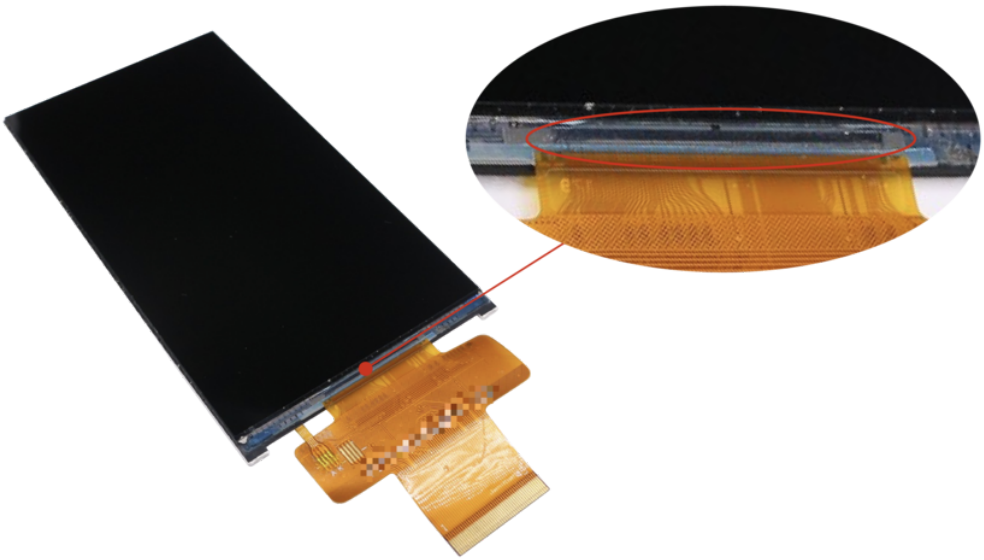


Fig. 2: LCD Module’ s Driver IC

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.

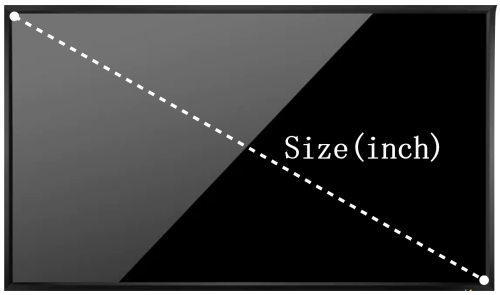


Fig. 5: Screen Size

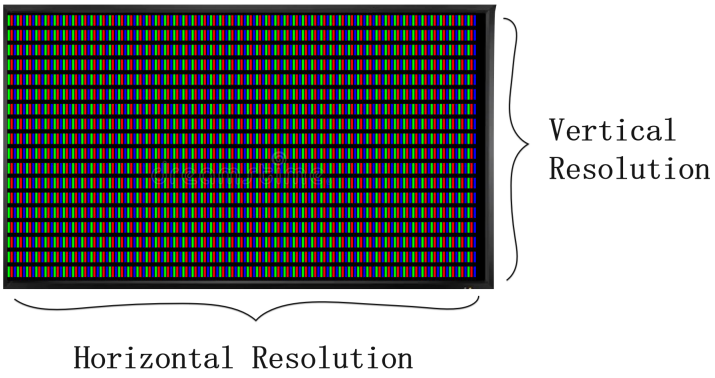


Fig. 6: Screen Resolution

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:

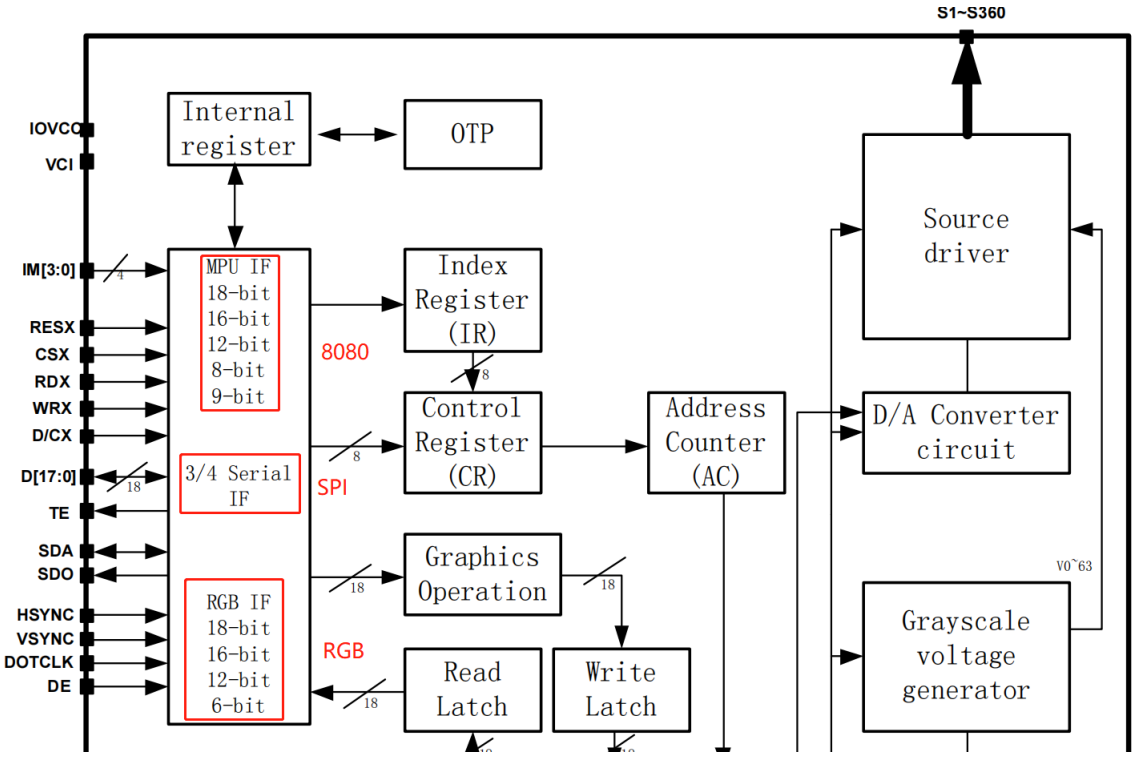


Fig. 7: Hardware Block Diagram of GC9A01

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 `MOSI` 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

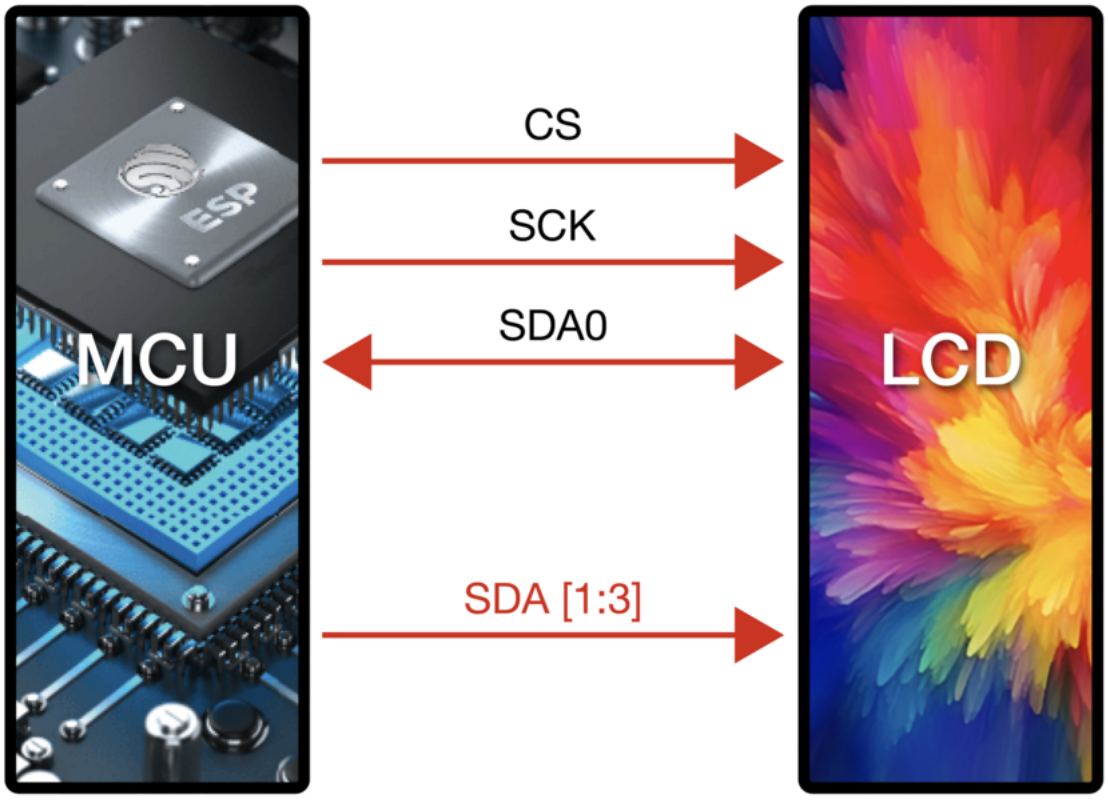


Fig. 10: Typical Connection Diagram for QSPI Interface

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:

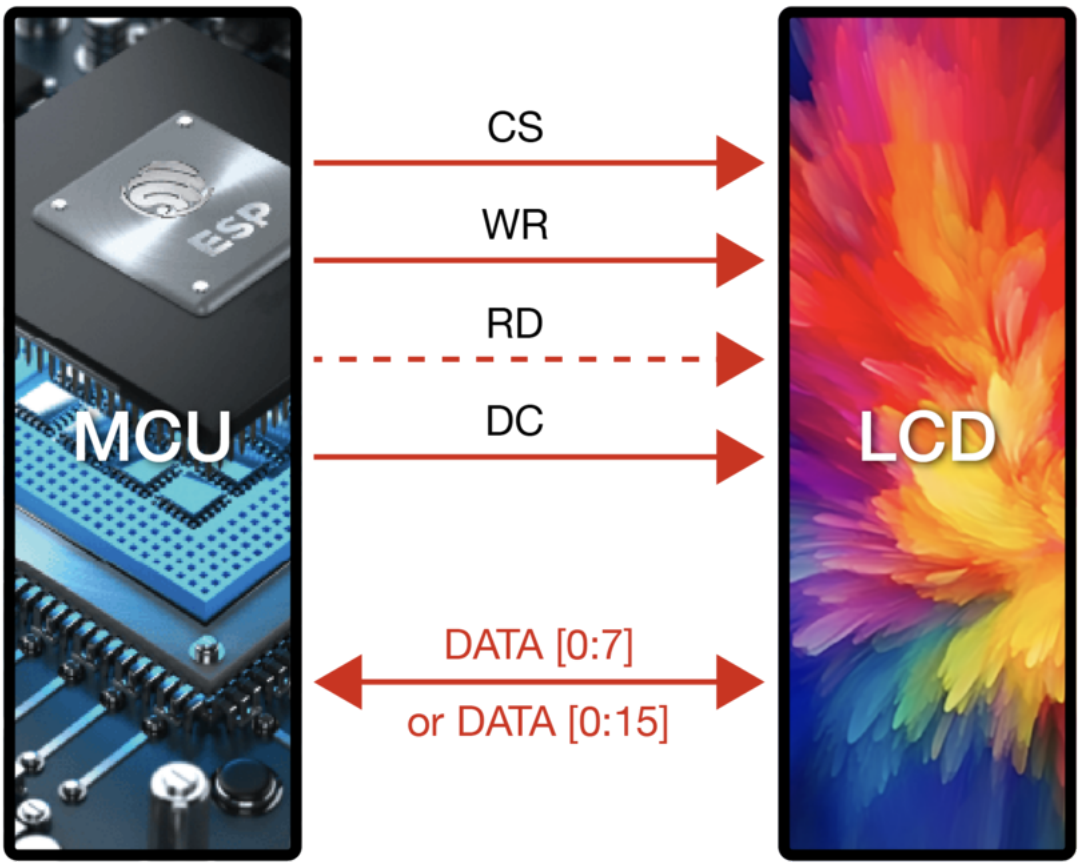


Fig. 11: Typical Connection Diagram for I80 Interface

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.

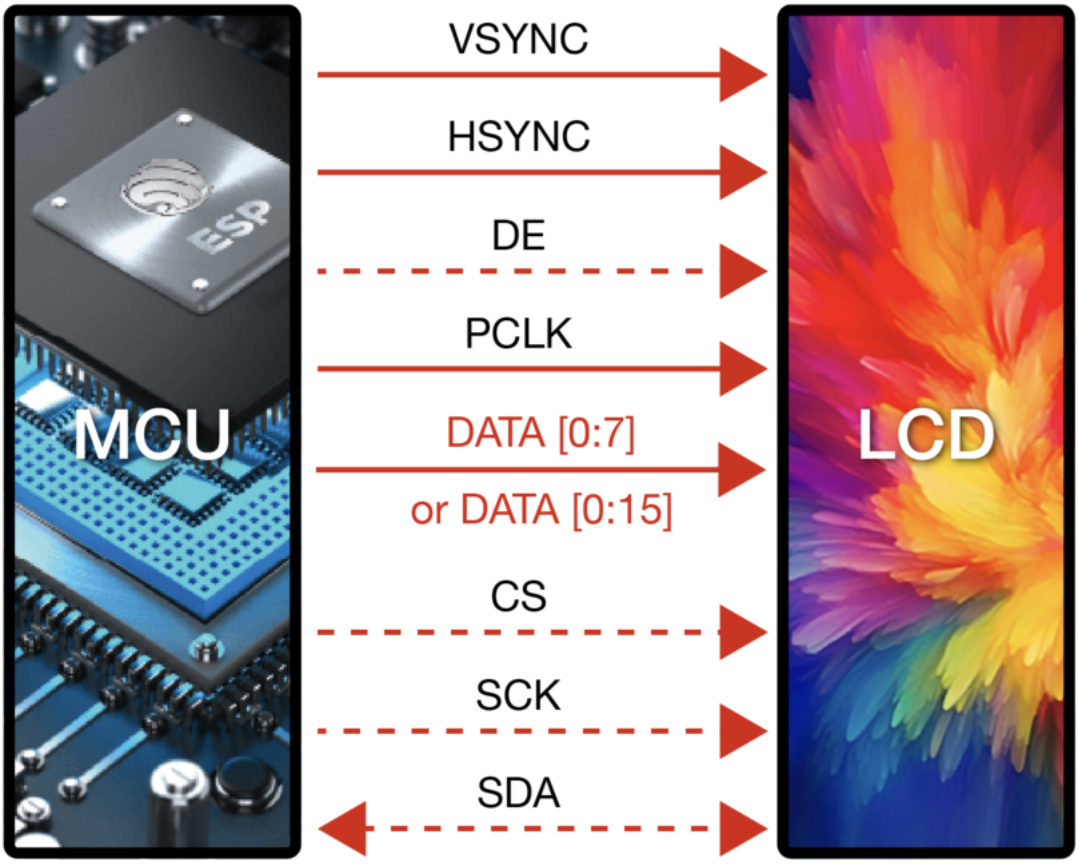


Fig. 12: Typical Connection Diagram for RGB Interface

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:

$$Interface\ Frame\ Rate = \frac{Clock\ Frequency \times Number\ of\ Data\ Lines}{Color\ Depth \times Horizontal\ Period \times Vertical\ Period}$$

$$Horizontal\ Period = Horizontal\ Pulse\ Width + Horizontal$$
$$Horizontal\ Resolution + Horizontal\ F$$
$$Vertical\ Period = Vertical\ Pulse\ Width + Vertical\ Back$$
$$Vertical\ Resolution + Vertical\ Front$$

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, ili9341, ssd1681, st7796
QSPI	axs15231b, gc9b71, sh8601, spd2010, st77903, st77916, st77922
I80	axs15231b, st7789, nt35510, ra8875, st7796
MIPI-DSI	ek79007, jd9165, jd9365, st7701, st7703, st77922, ili9881c, hx8399
3-wire SPI + RGB	st7701, st77903_rgb, st77922, gc9503

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:

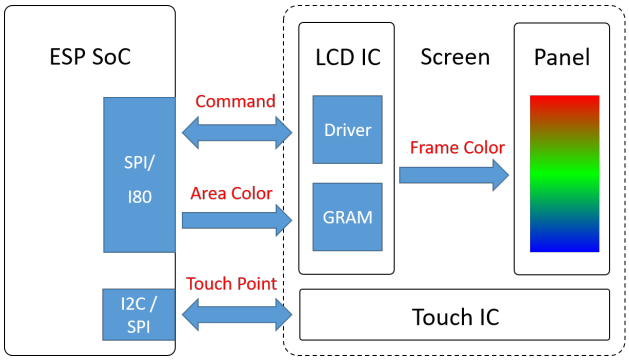


Fig. 13: Schematic diagram of hardware driver framework - SPI/I80 LCD

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:

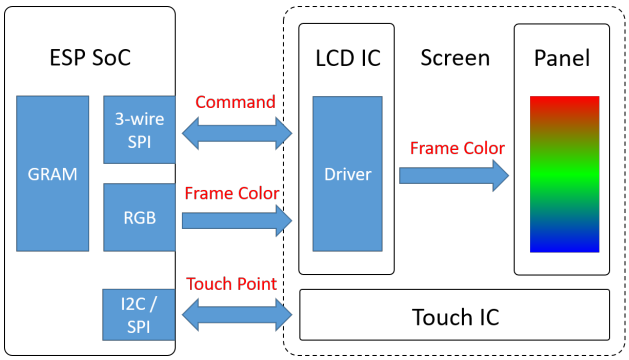


Fig. 14: Schematic diagram of hardware driver framework - RGB LCD

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:

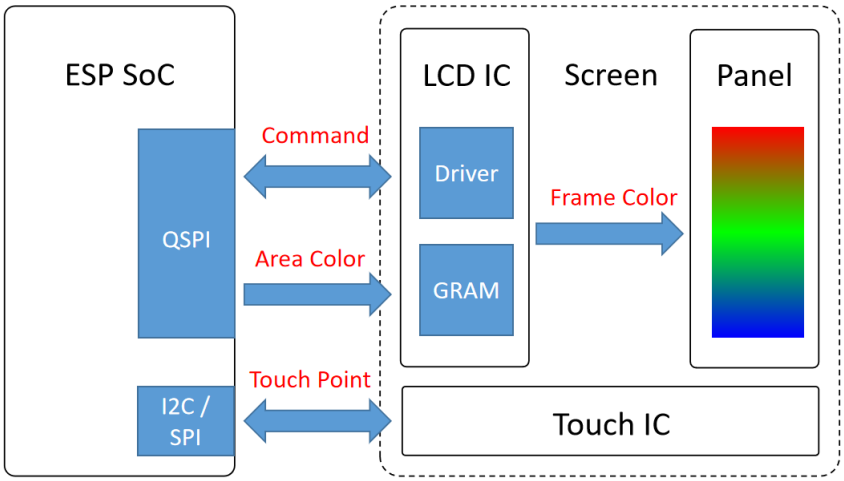


Fig. 15: Schematic diagram of hardware driver framework - QSPI LCD (with GRAM)

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.

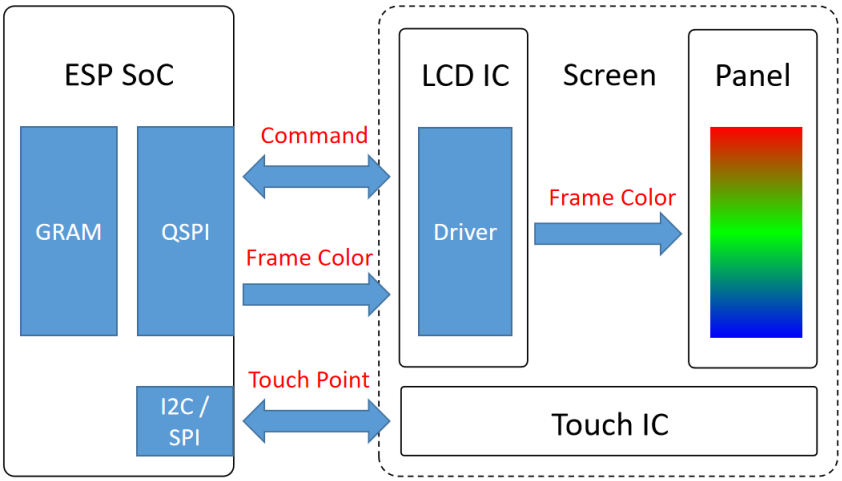


Fig. 16: Schematic diagram of hardware driver framework - QSPI LCD (without GRAM)

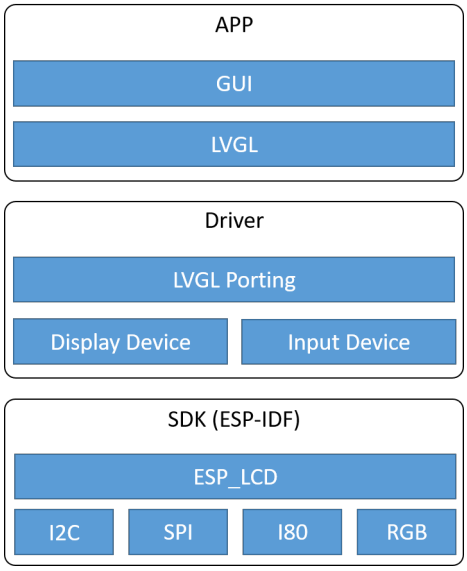


Fig. 17: Schematic diagram of software development framework

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

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

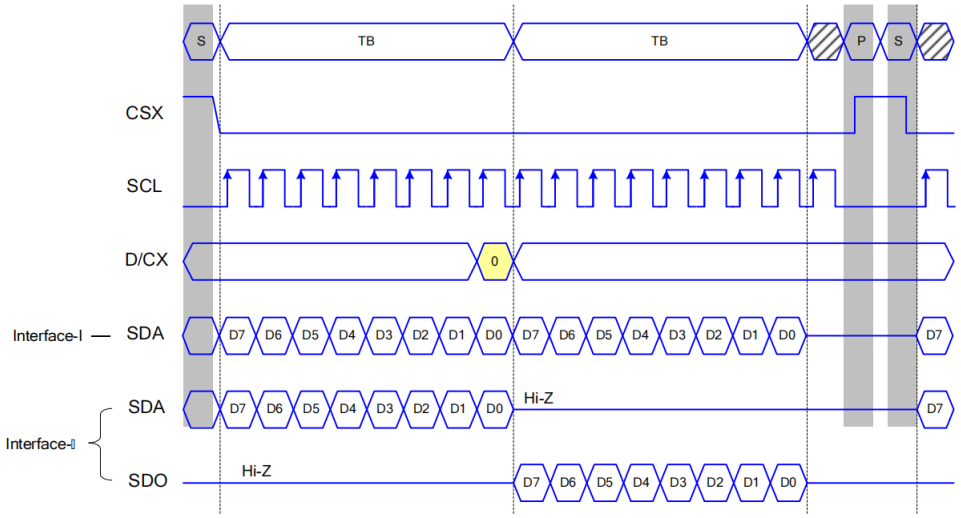


Fig. 20: Timing Diagram Comparison for Interface I/II Mode (4-line)

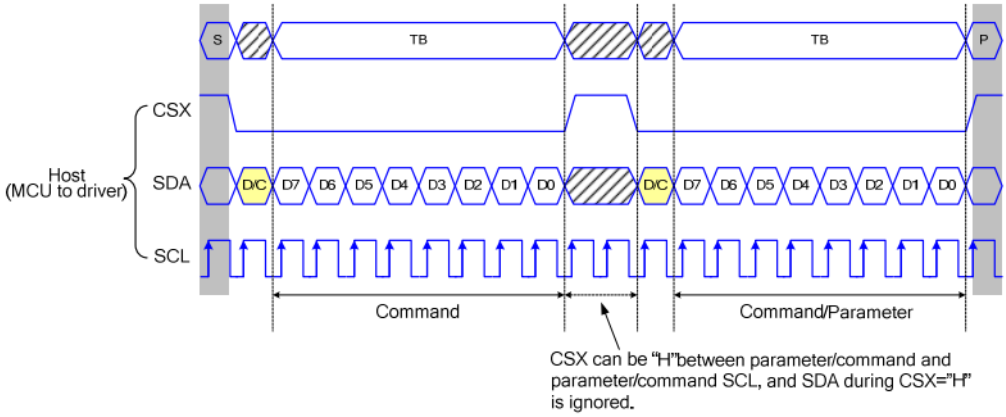


Figure 13 3-line serial interface write protocol (write to register with control bit in transmission)

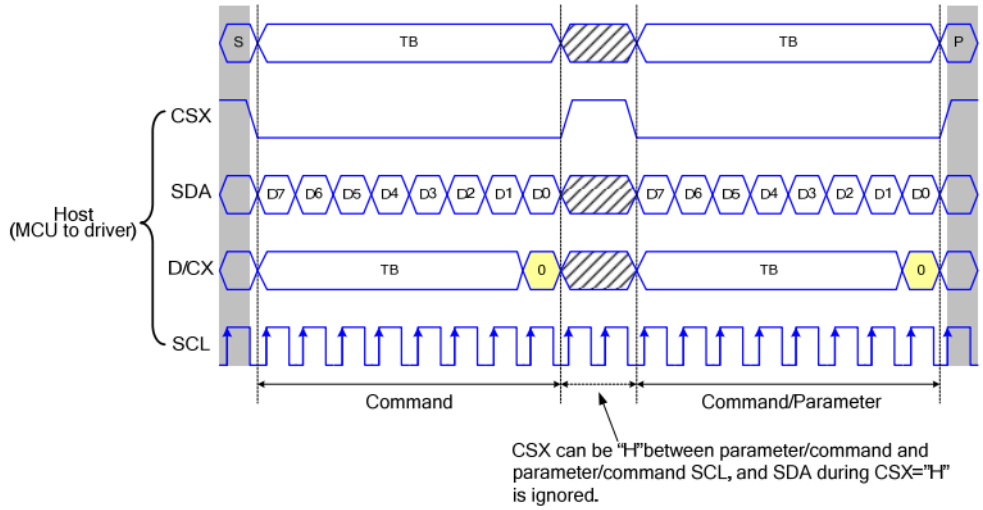


Figure 14 4-line serial interface write protocol (write to register with control bit in transmission)

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

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:

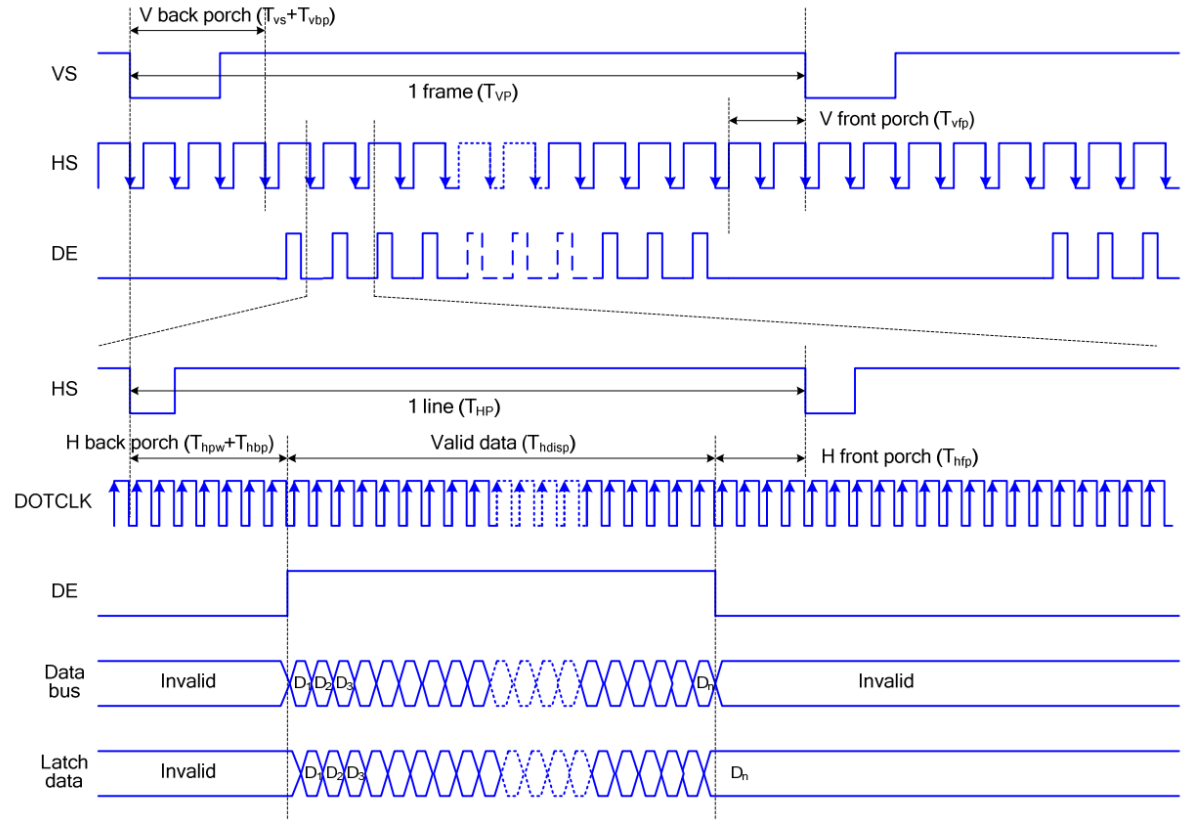
Mode	Use of DE Signal Line	Configuration of Blanking Porch Register	ESP Support
DE Mode	Yes	No	Yes
SYNC Mode	No	Yes	Yes

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

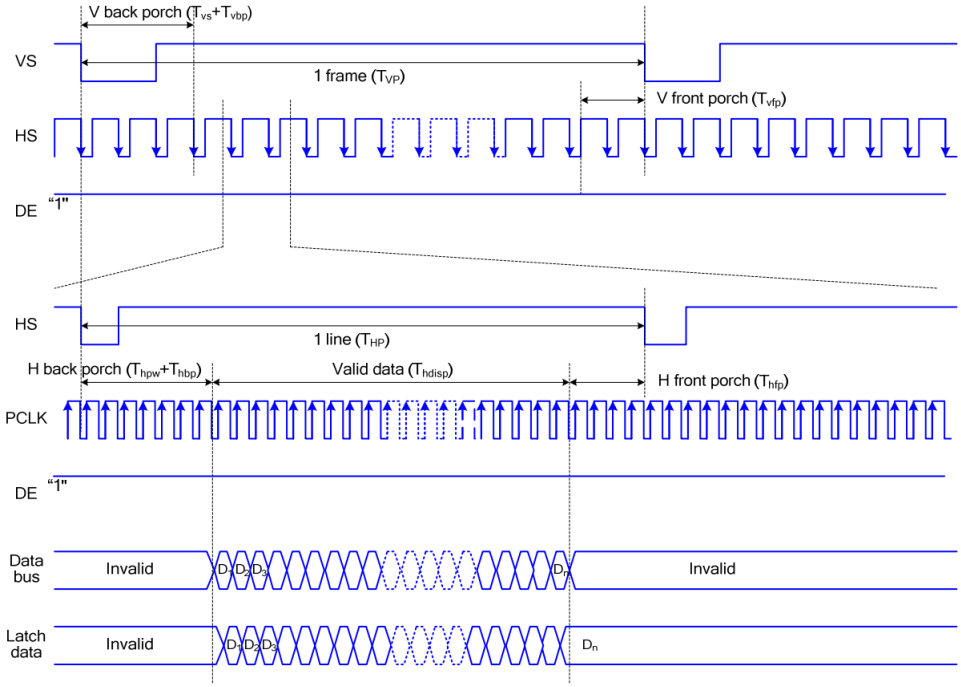


Figure 24 Timing chart of RGB interface HV mod

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
```

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},
```

(continues on next page)

(continued from previous page)

```
//     {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,
        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
```

(continues on next page)

(continued from previous page)

```
        .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
    .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));
```

(continues on next page)

(continued from previous page)

```
// 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;
}

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
```

(continues on next page)

(continued from previous page)

```
};
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 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.

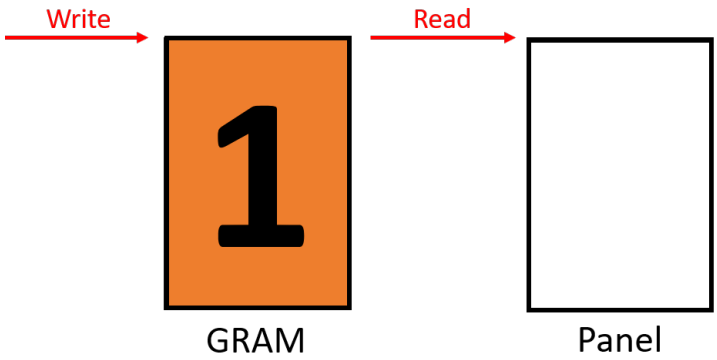


Fig. 31: Simplified screen refresh schematic diagram

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.

4.1.7 LCD Application Solution

Introduction to LCD Solutions

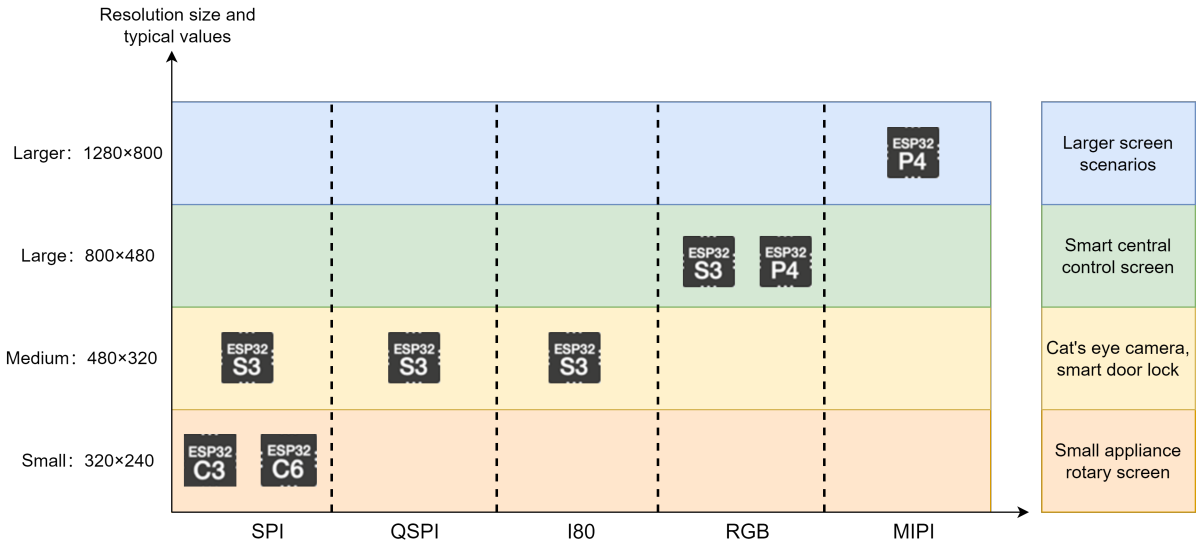
Espressif's HMI Smart Screen (LCD) solution features outstanding performance and scalability, and can be paired with different ESP main control chips. This solution performs well in various application scenarios such as smart home control, household appliance screens, medical equipment, industrial control, 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:

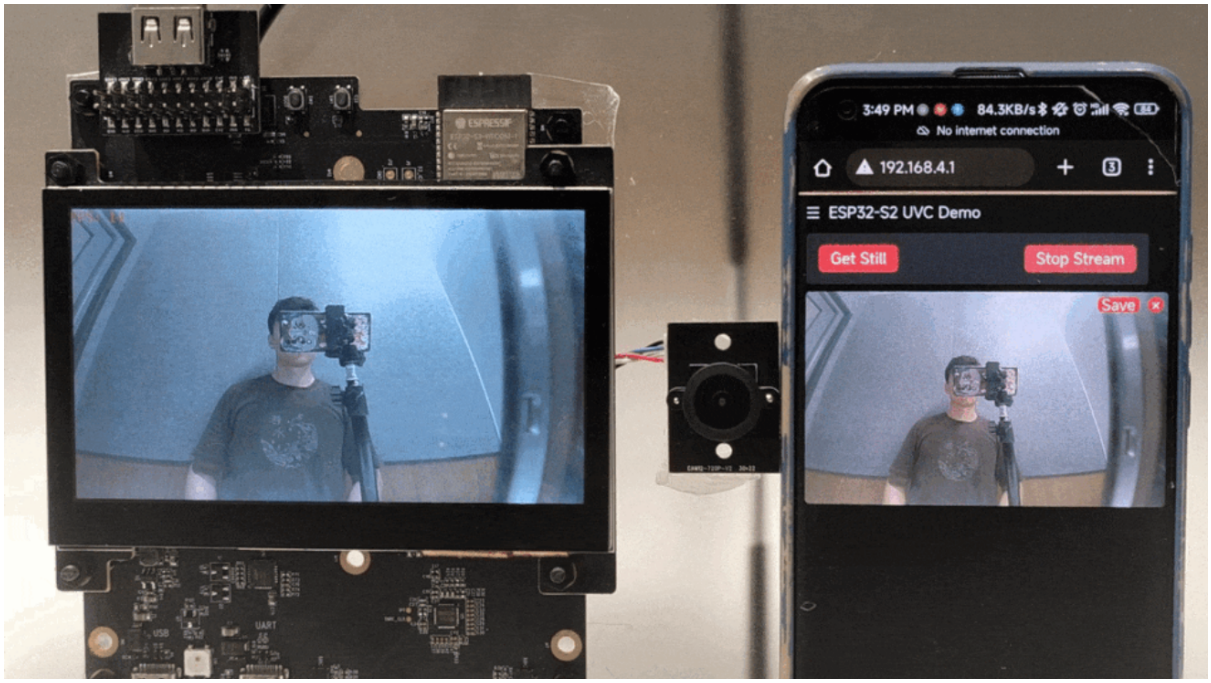
Espressif's LCD solutions are widely used in various fields, including but not limited to:

- **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

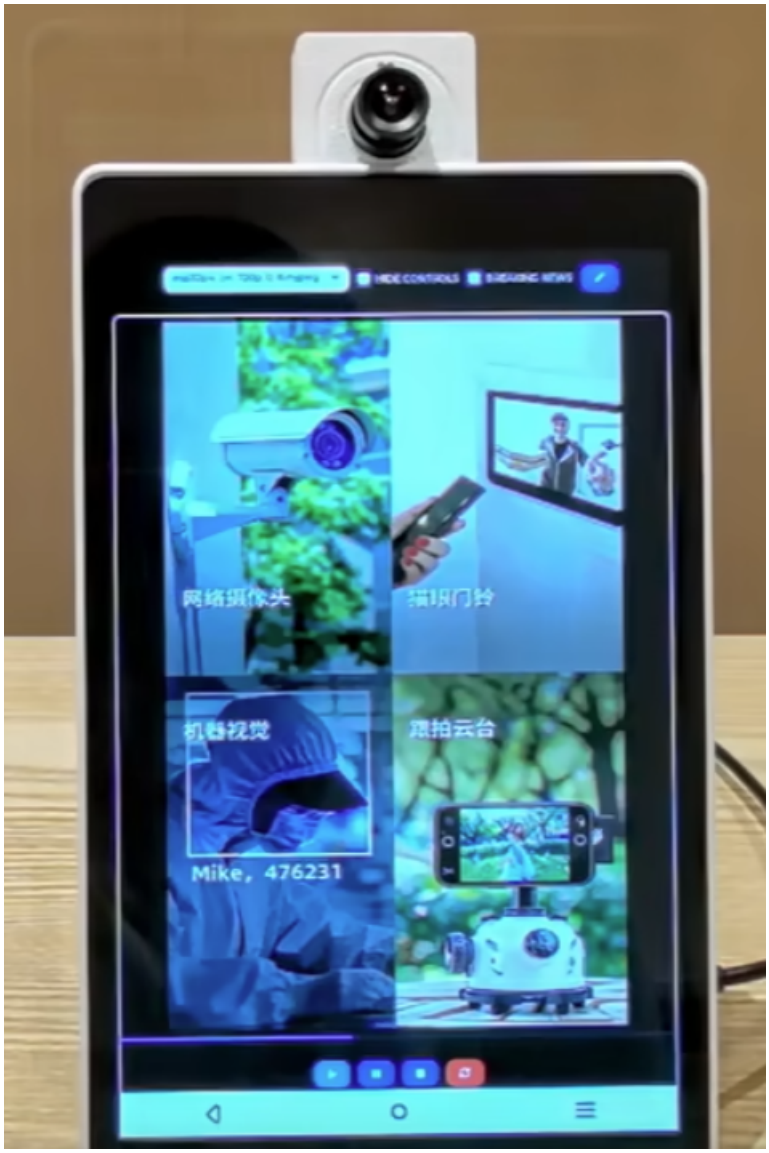
interfaces for functions such as serial communication. A typical application scenario is the rotary knob screen in small home appliance applications.

- **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 1: 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

4.2 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	Inter-face	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.

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;
```

(continues on next page)

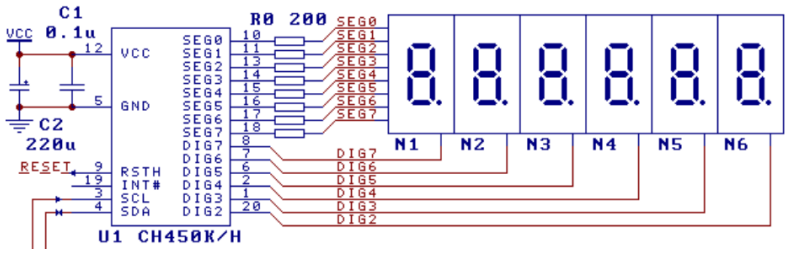


Fig. 32: CH450 Typical Application Circuit

(continued from previous page)

```
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.

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);
```

(continues on next page)

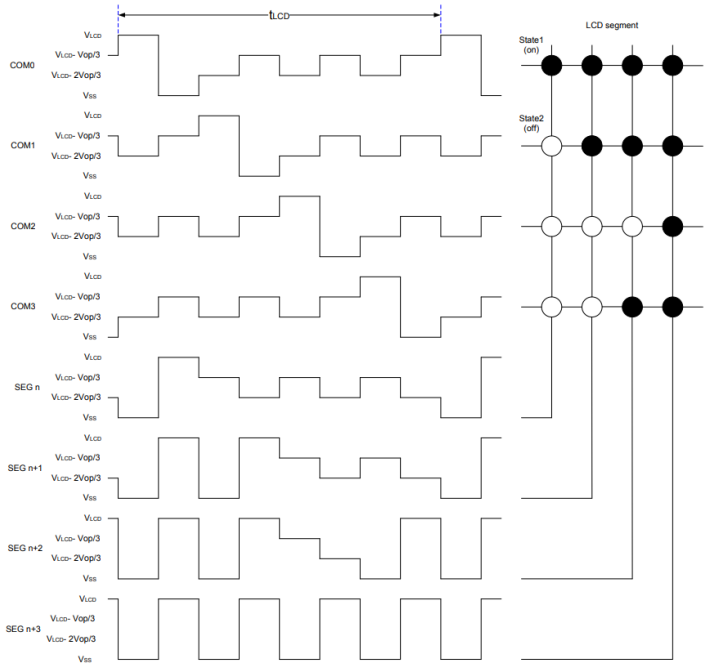


Fig. 33: HT16C21 Typical Drive Model

(continued from previous page)

```
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;
};
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.

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 = {
```

(continues on next page)

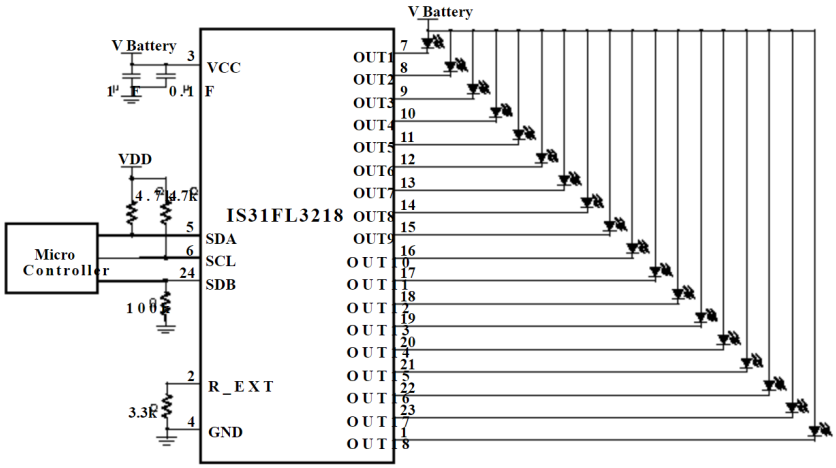


Fig. 34: IS31FL3218 Typical Application Circuit

(continued from previous page)

```
    .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

– *Enumerations*

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,
```

(continues on next page)

(continued from previous page)

```
    [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},                               // step4: 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
```

(continues on next page)

(continued from previous page)

```
    {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.

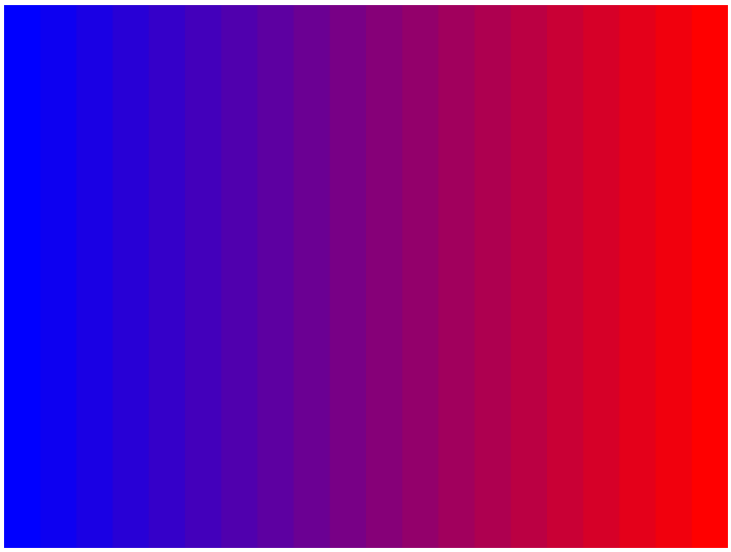


Fig. 35: RGB Gradient

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.

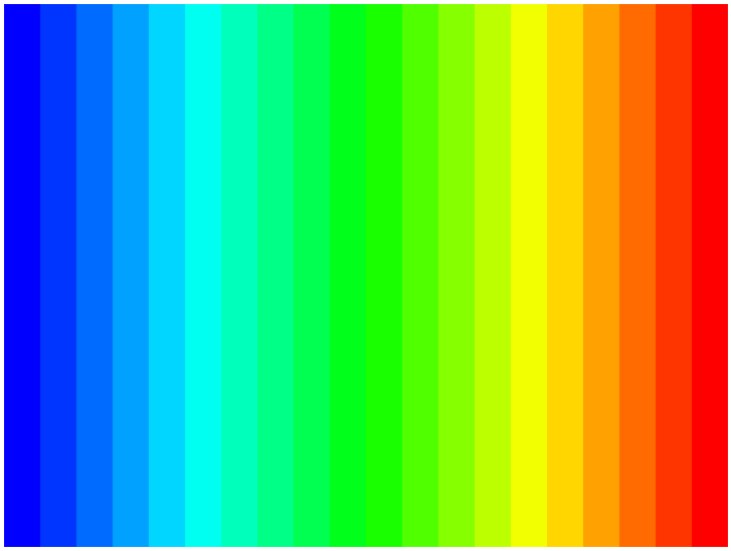
```
const blink_step_t test_blink_hsv_red_blue[] = {
    {LED_BLINK_HSV, SET_HSV(0, 255, 255), 0},               // step1: set to red␣
↪color 0 ms
```

(continues on next page)

(continued from previous page)

```
    {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
    {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_config_t config = {
    .mode = LED_GPIO_MODE,
    .led_gpio_config = {
        .active_level = 1,
        .gpio_num = 1,
    },
    .blink_lists = led_indicator_get_sample_lists(),
    .blink_list_num = led_indicator_get_sample_lists_num(),
};
led_indicator_handle_t led_handle = led_indicator_create(8, &config); // attach to␣
↪gpio 8
```

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

/*
*......
*/

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_config_t config = {
    .mode = LED_GPIO_MODE,
    .led_gpio_config = {
        .active_level = 1,
        .gpio_num = 1,
    },
    .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.

```
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

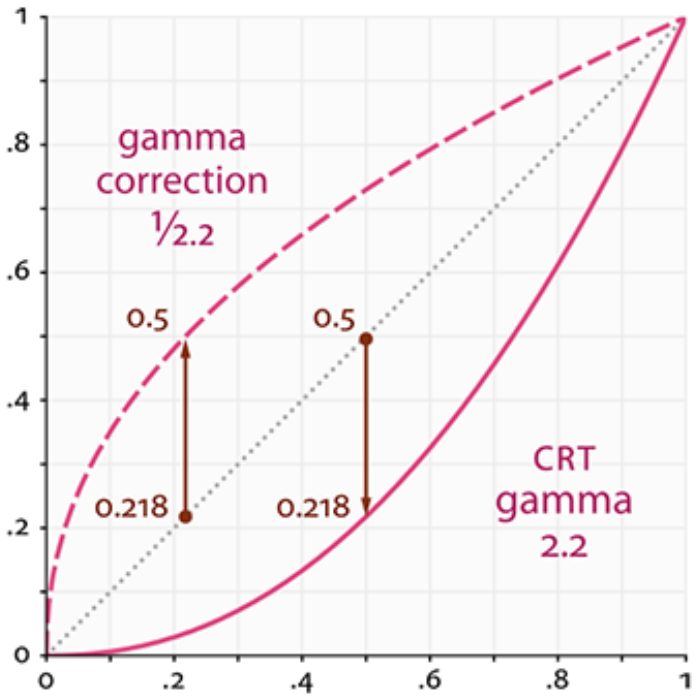


Fig. 36: Gamma Curve

Functions

led_indicator_handle_t **led_indicator_create** (const *led_indicator_config_t* *config)

create a LED indicator instance with GPIO number and configuration

Parameters **config** –configuration of the LED, eg. GPIO level when LED off
Returns led_indicator_handle_t handle of the LED indicator, NULL if create failed.

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

Structures

struct **blink_step_t**

one blink step, a meaningful signal consists of a group of steps

Public Members

blink_step_type_t **type**

action type in this step

uint32_t **value**

hold on or off, set 0 if LED_BLINK_STOP() or LED_BLINK_LOOP

uint32_t **hold_time_ms**

hold time(ms), set 0 if not LED_BLINK_HOLD

struct **led_indicator_config_t**

LED indicator specified configurations, as a arg when create a new indicator.

Public Members

led_indicator_mode_t **mode**

LED work mode, eg. GPIO or pwm mode

led_indicator_gpio_config_t ***led_indicator_gpio_config**

LED GPIO configuration

led_indicator_ledc_config_t ***led_indicator_ledc_config**

LED LEDC configuration

led_indicator_rgb_config_t ***led_indicator_rgb_config**

LED RGB configuration

led_indicator_strips_config_t ***led_indicator_strips_config**
LED LEDC rgb configuration

led_indicator_custom_config_t ***led_indicator_custom_config**
LED custom configuration

union *led_indicator_config_t*::[anonymous] **[anonymous]**
LED configuration

const *blink_step_t* ****blink_lists**
user defined LED blink lists

uint16_t **blink_list_num**
number of blink lists

Type Definitions

typedef void ***led_indicator_handle_t**
LED indicator operation handle

Enumerations

enum **[anonymous]**
LED state: 0-100, only hardware that supports to set brightness can adjust brightness.

Values:

enumerator **LED_STATE_OFF**
turn off the LED

enumerator **LED_STATE_25_PERCENT**
25% brightness, must support to set brightness

enumerator **LED_STATE_50_PERCENT**
50% brightness, must support to set brightness

enumerator **LED_STATE_75_PERCENT**
75% brightness, must support to set brightness

enumerator **LED_STATE_ON**
turn on the LED

enum **blink_step_type_t**
actions in this type

Values:

enumerator **LED_BLINK_STOP**
stop the blink

enumerator **LED_BLINK_HOLD**

hold the on-off state

enumerator **LED_BLINK_BREATHE**

breathe state

enumerator **LED_BLINK_BRIGHTNESS**

set the brightness, it will transition from the old brightness to the new brightness

enumerator **LED_BLINK_RGB**

color change with R(0-255) G(0-255) B(0-255)

enumerator **LED_BLINK_RGB_RING**

Gradual color transition from old color to new color in a color ring

enumerator **LED_BLINK_HSV**

color change with H(0-360) S(0-255) V(0-255)

enumerator **LED_BLINK_HSV_RING**

Gradual color transition from old color to new color in a color ring

enumerator **LED_BLINK_LOOP**

loop from first step

enum **led_indicator_mode_t**

LED indicator blink mode, as a member of *led_indicator_config_t*.

Values:

enumerator **LED_GPIO_MODE**

blink with max brightness

enumerator **LED_LEDC_MODE**

blink with LEDC driver

enumerator **LED_RGB_MODE**

blink with RGB driver

enumerator **LED_STRIPS_MODE**

blink with LEDC strips driver

enumerator **LED_CUSTOM_MODE**

blink with custom driver

4.4 LCD Tools

4.4.1 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
```

(continues on next page)

(continued from previous page)

```
    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.

4.4.2 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.

4.4.3 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.

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

Optionally, users can opt to have the image automatically flashed together with the app binaries, partition tables, etc. on idf.py flash by specifying FLASH_IN_PROJECT. For example:

```
/* partitions.csv
 * ------------------------------------------------------------------------
 * | Name                | Type | SubType | Offset | Size   | Flags     |
 * ------------------------------------------------------------------------
 * | my_spiffs_partition | data | spiffs  |        | 6000K  |           |
 * ------------------------------------------------------------------------
 */
 spiffs_create_partition_assets(
     my_spiffs_partition
     my_folder
     FLASH_IN_PROJECT
     MMAP_FILE_SUPPORT_FORMAT ".png")
```

The component also supports the following options, which allow you to enable various pre-processing of the image at compile time.

```
set(options FLASH_IN_PROJECT,            // Defines storage type (flash in project)
             MMAP_SUPPORT_SJPG,          // Enable support for SJPG format
             MMAP_SUPPORT_SPNG,          // Enable support for SPNG format
             MMAP_SUPPORT_QOI,           // Enable support for QOI format
             MMAP_SUPPORT_SQOI,          // Enable support for SQOI format
             MMAP_SUPPORT_RAW,           // Enable support for RAW format (LVGL␣
↪conversion only)
             MMAP_RAW_DITHER,            // Enable dithering for RAW images (LVGL␣
↪conversion only)
             MMAP_RAW_BGR_MODE)          // Enable BGR mode for RAW images (LVGL␣
↪conversion only)

 set(one_value_args MMAP_FILE_SUPPORT_FORMAT,     // Specify supported file format␣
↪(e.g., .png, .jpg)
                MMAP_SPLIT_HEIGHT,                // Define the height for image␣
↪splitting
                MMAP_RAW_FILE_FORMAT)             // Specify the file format for␣
↪RAW images (LVGL conversion only)
```

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.

```
mmap_assets_handle_t asset_handle;

const mmap_assets_config_t config = {
    .partition_label = "my_spiffs_partition",
    .max_files = TOTAL_MMAP_FILES,
    .checksum = MMAP_CHECKSUM,
};

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.

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

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 **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

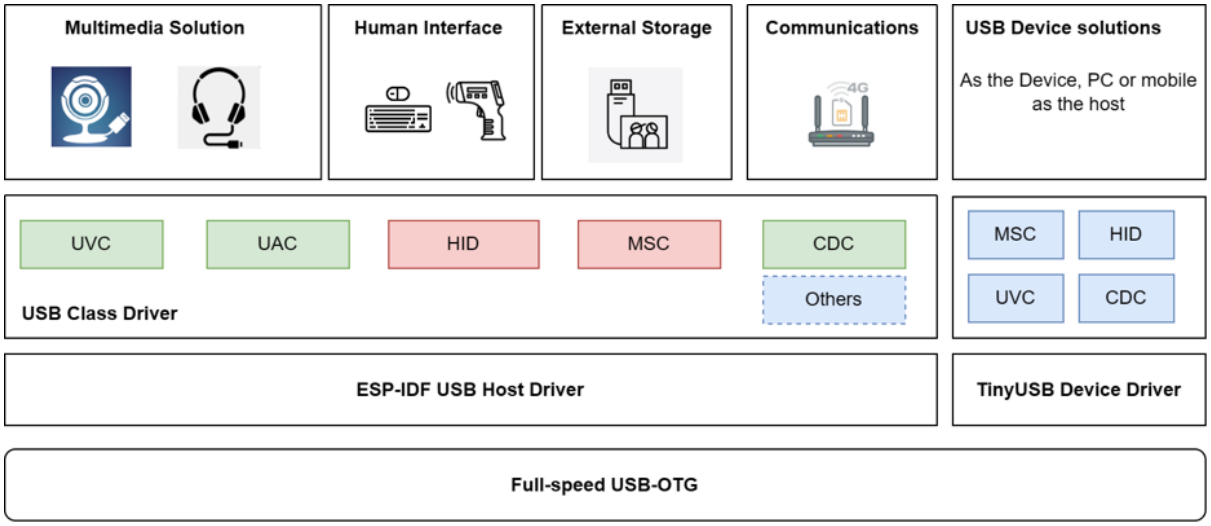
5.1.1 ESP USB Peripheral Introduction

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 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*.

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.

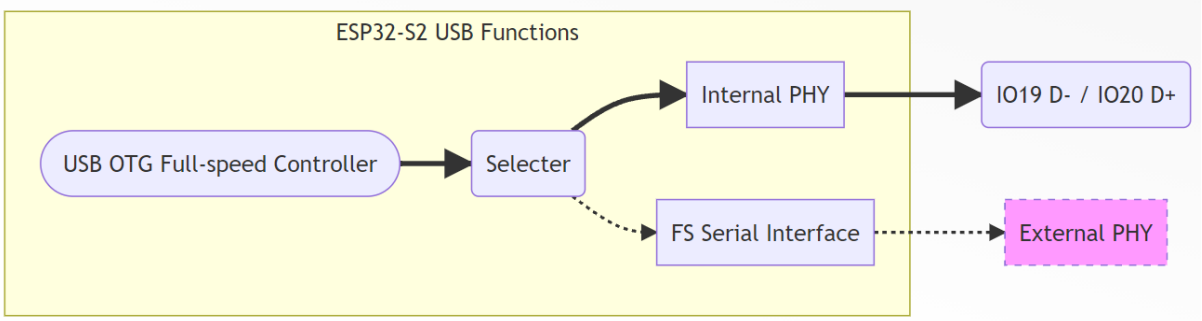
For more information, please refer to the *USB-PHY Introduction*.

ESP32-S/C Series USB Peripheral Support

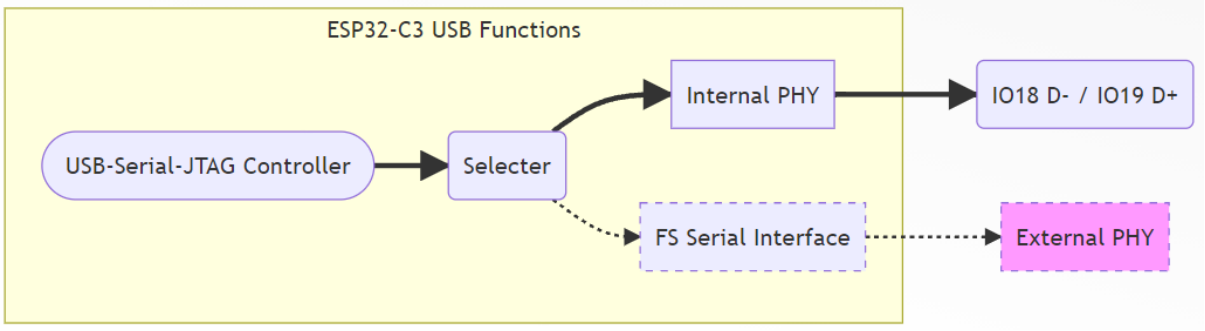
	USB OTG High-Speed	USB OTG Full-Speed	USB-Serial-JTAG	Full-Speed PHY	High-Speed PHY
ESP32-P4	√	√	√	√	√
ESP32-S3	X	√	√	√	X
ESP32-S2	X	√	X	√	X
ESP32-C6	X	X	√	√	X
ESP32-C3	X	X	√	√	X
ESP32-C2	X	X	X	X	X
ESP32	X	X	X	X	X
ESP8266	X	X	X	X	X

- √ : Supported
- X : Not Supported

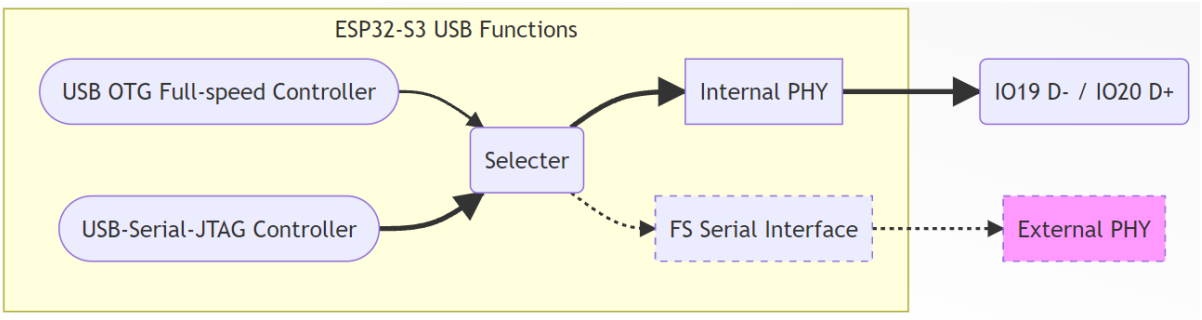
ESP32-S2 USB Function Overview The ESP32-S2 features an integrated **USB OTG Full-Speed Controller** and **USB Full-Speed PHY**. The internal architecture is as follows:



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*.



5.1.2 USB-OTG Peripheral Introduction

The ESP32-S2/S3 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, 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

- Calculation Formula for Transfer Rate: Transfer Rate (Bytes/s) = Maximum Packet Size * Number of Packets Transferred Per Millisecond * 1000
- 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 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*.

5.1.3 USB-Serial-JTAG Peripheral Introduction

The ESP32-S3/C3 chips come with a built-in USB-Serial-JTAG peripheral, which includes a USB-to-serial converter and a USB-to-JTAG converter. It supports connection to a PC via a USB cable, enabling functions such as firmware downloading, debugging, and printing system logs. The internal structure of the USB-Serial-JTAG peripheral can be referred to in the ESP32-C3 Technical Reference Manual - USB Serial/JTAG Controller.

USB-Serial-JTAG peripheral driver

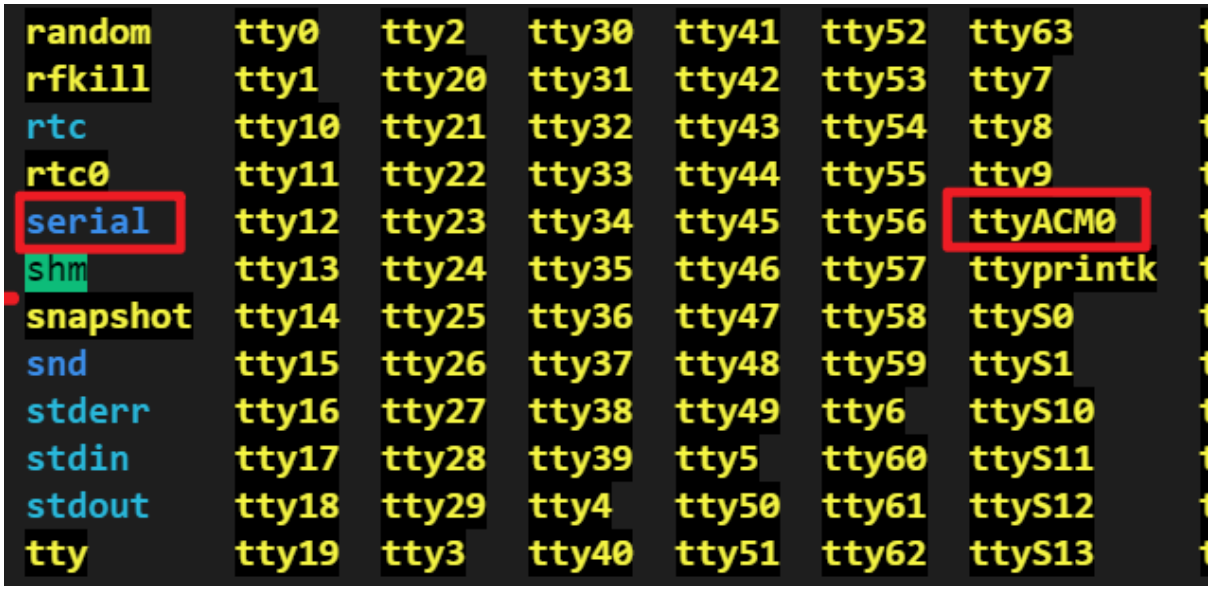
- For Linux and MacOS systems, No need to manually install drivers.
- For Windows 10 and above, drivers will be automatically installed when connected to the internet.
- For Windows 7/8 systems, manual driver installation is necessary. The driver can be downloaded from: esp32-usb-jtag-2021-07-15. Alternatively, users can use the ESP-IDF Windows Installer, selecting the USB-Serial-JTAG driver during installation.

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:



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.

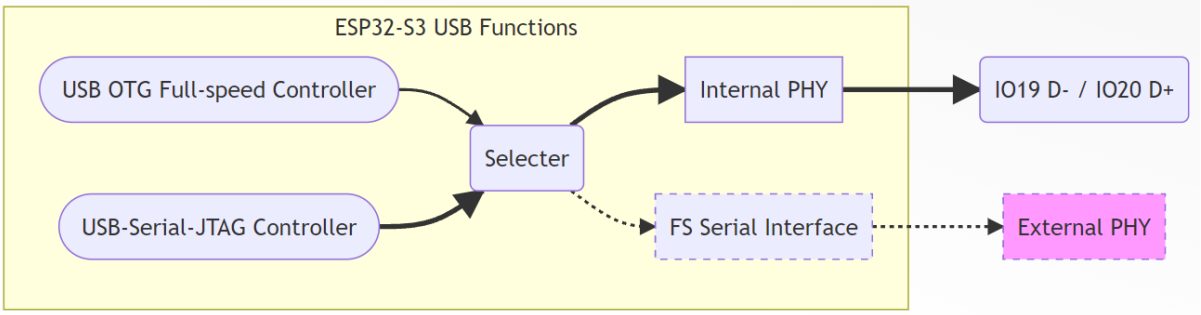
5.1.4 USB PHY/Transceiver Introduction

The function of the USB Full-speed 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 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-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.



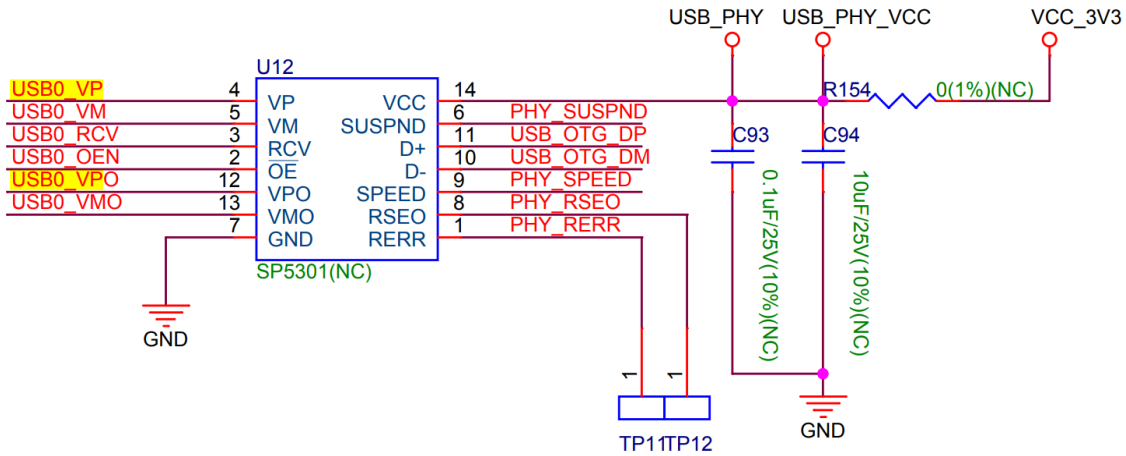
The internal USB-PHY corresponds to fixed GPIO pins, as shown in the table below:

	D+	D-
ESP32-S2	20	19
ESP32-S3	20	19
ESP32-C3	19	18
ESP32-C6	13	12

Use an external PHY

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.

5.1.6 USB Host Solution

ESP32-S2/S3 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 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. 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.

links:

- usb_stream 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
- Peripherals: USB-OTG
- USB Audio Devices: Supports PCM format

Links:

- usb_host_uac components
- 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
- Peripherals: USB-OTG
- 4G Modules: Supports Cat.1, Cat.4, and other network standard 4G modules, requiring module support for the PPP protocol.

Links:

- USB 4G Demo video
- iot_usbh_modem component
- Example Code: 4G Wi-Fi Router: usb/host/usb_cdc_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
- 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 Flash Drive OTA component
- Mount USB Flash Drive + File System Access Example

5.1.7 USB Device Solution

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
- 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
- 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
- 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
- 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](https://github.com/espressif/esp-iot-solution/tree/master/examples/usb/device/usb_extend_screen)

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 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 has no 5V tolerate␣
↪pin.
 *
 *    5V VBUS □□□□□□□□□     □□□□□□□□    GND
 *     □□□□□□□□□ 100K □□□□□□ 100K □□□□□□□□
 *             □□□□□□□□□ □  □□□□□□□□□
 *                       □             GPIOX
 *                       □□□□□□□□□□□□□□□□□□
 *        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 = {
```

(continues on next page)

(continued from previous page)

```
        .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
// Configure GPIO Pin for vbus monitoring
const gpio_config_t vbus_gpio_config = {
    .pin_bit_mask = BIT64(VBUS_MONITORING_GPIO_NUM),
    .mode = GPIO_MODE_INPUT,
    .intr_type = GPIO_INTR_DISABLE,
    .pull_up_en = false,
    .pull_down_en = false,
};
ESP_ERROR_CHECK(gpio_config(&vbus_gpio_config));
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));
```

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
VID	0x303a	0x303a	0x303a
PID	0x0002	0x1001	0x1001
Serial	0	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) and ESP32C3 (usb-serial-jtag), 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
```

Users have the option to download the script `ignore_hwserial_esp32s3c3.bat` , then run it with administrative privileges by right-clicking and selecting **Run as Administrator**.

5.1.10 TinyUSB Application Guide

This guide contains the following content:

Table of Contents

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

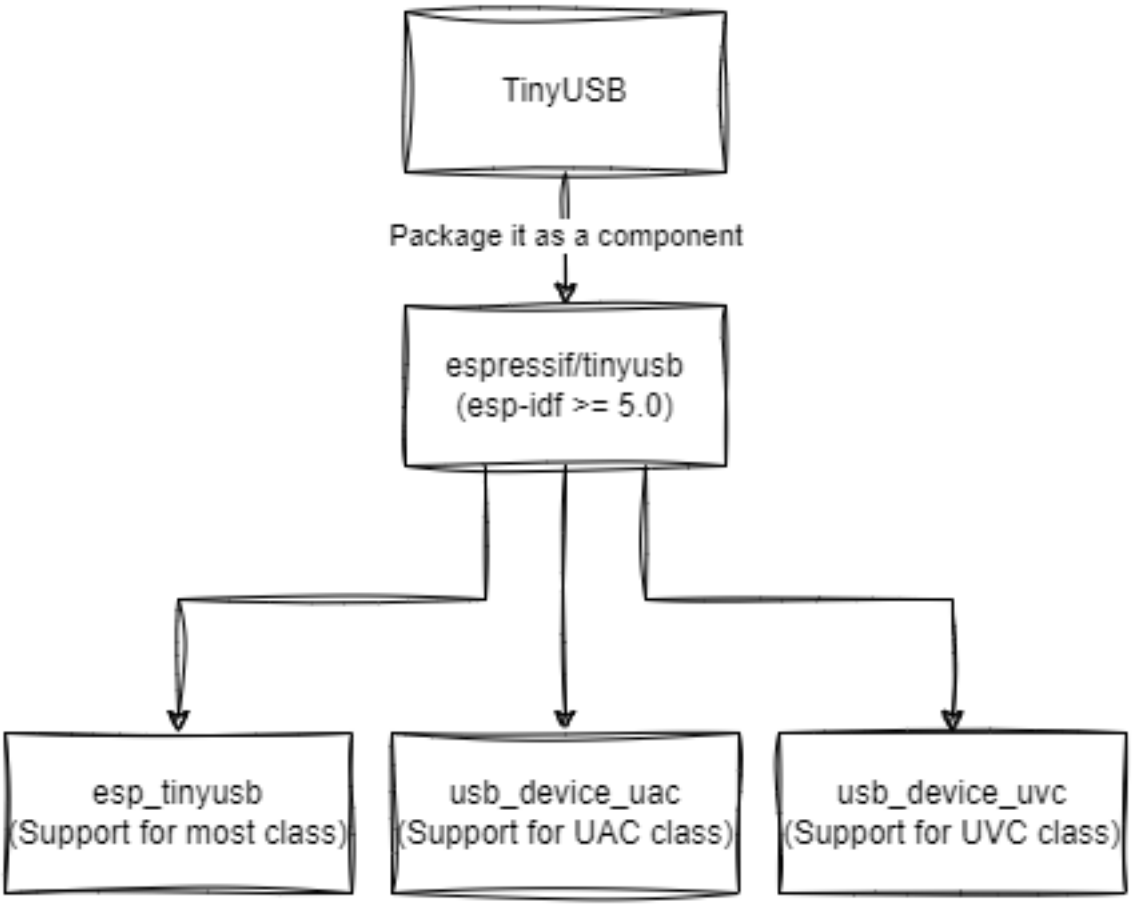


Fig. 1: TinyUSB Components

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.

- 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.

5.1.11 Developing with Native tinyusb

This guide contains the following content:

Table of Contents

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)
  ```

- **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_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_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 = {
```

(continues on next page)

(continued from previous page)

```
        .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*.

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);

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.

5.2 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 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

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 simple version of the USB host CDC driver. The API is designed similar like ESP-IDF UART driver, which can be used to replace the original UART driver to realize the update from UART to USB.

User Guide

1. Using `usbh_cdc_driver_install` to configure, user can simply configure the bulk endpoint address and the size of the internal ringbuffer, user can also configure the hot plug related callback function `conn_callback` `disconn_callback`.

```
/* install usbh cdc driver with bulk endpoint configs and size of internal␣
↪ringbuffer */
static usbh_cdc_config_t config = {
    /* use default endpoint descriptor with user address */
    .bulk_in_ep_addr = EXAMPLE_BULK_IN_EP_ADDR,
    .bulk_out_ep_addr = EXAMPLE_BULK_OUT_EP_ADDR,
    .rx_buffer_size = IN_RINGBUF_SIZE,
    .tx_buffer_size = OUT_RINGBUF_SIZE,
    .conn_callback = usb_connect_callback,
    .disconn_callback = usb_disconnect_callback,
};
/* if user want to use multiple interfaces, can configure like this */
#if (EXAMPLE_BULK_ITF_NUM > 1)
config.itf_num = 2;
config.bulk_in_ep_addrs[1] = EXAMPLE_BULK_IN1_EP_ADDR;
config.bulk_out_ep_addrs[1] = EXAMPLE_BULK_OUT1_EP_ADDR;
config.rx_buffer_sizes[1] = IN_RINGBUF_SIZE;
config.tx_buffer_sizes[1] = OUT_RINGBUF_SIZE;
#endif

/* install USB host CDC driver */
usbh_cdc_driver_install(&config);

/* Waiting for USB device connected */
usbh_cdc_wait_connect(portMAX_DELAY);
```

2. After the driver initialization, the internal state machine will automatically handle the hot plug of the USB.
3. `usbh_cdc_wait_connect` can be used to block task until USB CDC Device is connected or timeout.
4. After successfully connected, the host will automatically receive USB data from CDC device to the internal `ringbuffer`, user can poll `usbh_cdc_get_buffered_data_len` to read buffered data size or register a receive callback to get notified when data is ready. Then `usbh_cdc_read_bytes` can be used to read buffered data out.
5. `usbh_cdc_write_bytes` can be used to send data to USB Device. The data is first written to the internal transmit `ringbuffer`, then will be sent out during USB bus free.
6. `usbh_cdc_driver_delete` can uninstall the USB driver completely to release all resources.
7. If config multiple CDC interfaces, each interface contains an IN and OUT endpoint. Users can communicate with the specified interfaces using `usbh_cdc_itf_read_bytes` and `usbh_cdc_itf_write_bytes`.

Examples

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_config_t* *config)

Install USB CDC driver.

Parameters **config** –USB Host CDC configs
Returns ESP_ERR_INVALID_STATE driver has been installed ESP_ERR_INVALID_ARG args not supported ESP_FAIL driver install failed ESP_OK driver install succeed

esp_err_t **usbh_cdc_driver_delete**(void)

Uninstall USB driver.

Returns ESP_ERR_INVALID_STATE driver not installed ESP_OK start succeed

esp_err_t **usbh_cdc_wait_connect**(TickType_t ticks_to_wait)

Waiting until CDC device connected.

Parameters **ticks_to_wait** –timeout value, count in RTOS ticks
Returns ESP_ERR_INVALID_STATE driver not installed ESP_ERR_TIMEOUT wait timeout ESP_OK device connect succeed

int **usbh_cdc_write_bytes**(const uint8_t *buf, size_t length)

Send data to interface 0 of connected USB device from a given buffer and length, this function will return after copying all the data to tx ringbuffer.

Parameters
- **buf** –data buffer address
- **length** –data length to send

Returns int The number of bytes pushed to the tx buffer

int **usbh_cdc_itf_write_bytes**(uint8_t itf, const uint8_t *buf, size_t length)

Send data to specified interface of connected USB device from a given buffer and length, this function will return after copying all the data to tx buffer.

Parameters
- **itf** –the interface index
- **buf** –data buffer address
- **length** –data length to send

Returns int The number of bytes pushed to the tx buffer

esp_err_t **usbh_cdc_get_buffered_data_len**(size_t *size)

Get USB interface 0 rx buffered data length.

Parameters **size** –data length buffered
Returns ESP_ERR_INVALID_STATE cdc not configured, or not running ESP_ERR_INVALID_ARG args not supported ESP_OK start succeed

esp_err_t **usbh_cdc_itf_get_buffered_data_len**(uint8_t itf, size_t *size)

Get USB specified interface rx buffered data length.

Parameters
- **itf** –the interface index
- **size** –data length buffered

Returns ESP_ERR_INVALID_STATE cdc not configured, or not running ESP_ERR_INVALID_ARG args not supported ESP_OK start succeed

int **usbh_cdc_read_bytes**(uint8_t *buf, size_t length, TickType_t ticks_to_wait)

Read data bytes from interface 0 rx buffer.

Parameters
- **buf** –data buffer address
- **length** –data buffer size
- **ticks_to_wait** –timeout value, count in RTOS ticks

Returns int the number of bytes actually read from rx buffer

int **usbh_cdc_itf_read_bytes**(uint8_t itf, uint8_t *buf, size_t length, TickType_t ticks_to_wait)

Read data bytes from specified interface rx buffer.

Parameters
- **itf** –the interface index

- **buf** –data buffer address
- **length** –data buffer size
- **ticks_to_wait** –timeout value, count in RTOS ticks

Returns int the number of bytes actually read from rx buffer

int **usbh_cdc_get_itf_state**(uint8_t itf)

Get the connect state of given interface.

Parameters **itf** –the interface index
Returns ** int true is ready, false is not ready

esp_err_t **usbh_cdc_flush_rx_buffer**(uint8_t itf)

Flush rx buffer, discard all the data in the ring-buffer.

Parameters **itf** –the interface index
Returns ** esp_err_t ESP_ERR_INVALID_STATE cdc not configured, or not running ESP_ERR_INVALID_ARG args not supported ESP_OK start succeed

esp_err_t **usbh_cdc_flush_tx_buffer**(uint8_t itf)

Flush tx buffer, discard all the data in the ring-buffer.

Parameters **itf** –the interface index
Returns ** esp_err_t ESP_ERR_INVALID_STATE cdc not configured, or not running ESP_ERR_INVALID_ARG args not supported ESP_OK start succeed

Structures

struct **usbh_cdc_config**

USB host CDC configuration type itf_num is the total enabled interface numbers, can not exceed CDC_INTERFACE_NUM_MAX, itf_num = 0 is same as itf_num = 1 for back compatibility, if itf_num > 1, user should config params with `s` ending like `bulk_in_ep_addrs` to config each interface, callbacks should not in block state.

Public Members

int **itf_num**

interface numbers enabled

uint8_t **bulk_in_ep_addr**

USB CDC bulk in endpoint address, will be overwritten if bulk_in_ep is specified

uint8_t **bulk_in_ep_addrs**[CDC_INTERFACE_NUM_MAX]

USB CDC bulk in endpoints addresses, saved as an array, will be overwritten if bulk_in_ep is specified

uint8_t **bulk_out_ep_addr**

USB CDC bulk out endpoint address, will be overwritten if bulk_out_ep is specified

uint8_t **bulk_out_ep_addrs**[CDC_INTERFACE_NUM_MAX]

USB CDC bulk out endpoint addresses, saved as an array, will be overwritten if bulk_out_ep is specified

int **rx_buffer_size**

USB receive/in ringbuffer size

int **rx_buffer_sizes**[CDC_INTERFACE_NUM_MAX]

USB receive/in ringbuffer size of each interface

int **tx_buffer_size**

USB transmit/out ringbuffer size

int **tx_buffer_sizes**[CDC_INTERFACE_NUM_MAX]

USB transmit/out ringbuffer size of each interface

usb_ep_desc_t ***bulk_in_ep**

USB CDC bulk in endpoint descriptor, set NULL if using default param

usb_ep_desc_t ***bulk_in_eps**[CDC_INTERFACE_NUM_MAX]

USB CDC bulk in endpoint descriptors of each interface, set NULL if using default param

usb_ep_desc_t ***bulk_out_ep**

USB CDC bulk out endpoint descriptor, set NULL if using default param

usb_ep_desc_t ***bulk_out_eps**[CDC_INTERFACE_NUM_MAX]

USB CDC bulk out endpoint descriptors of each interface, set NULL if using default param

usbh_cdc_cb_t **conn_callback**

USB connect callback, set NULL if not use

usbh_cdc_cb_t **disconn_callback**

USB disconnect callback, set NULL if not use

usbh_cdc_cb_t **rx_callback**

packet receive callback, set NULL if not use

usbh_cdc_cb_t **rx_callbacks**[CDC_INTERFACE_NUM_MAX]

packet receive callbacks of each interface, set NULL if not use

void ***conn_callback_arg**

USB connect callback arg, set NULL if not use

void ***disconn_callback_arg**

USB disconnect callback arg, set NULL if not use

void ***rx_callback_arg**

packet receive callback arg, set NULL if not use

void ***rx_callback_args**[CDC_INTERFACE_NUM_MAX]

packet receive callback arg of each interface, set NULL if not use

Macros

CDC_INTERFACE_NUM_MAX

Type Definitions

typedef void (*`usbh_cdc_cb_t`)(void *arg)

USB receive callback type.

typedef struct *usbh_cdc_config* `usbh_cdc_config_t`

USB host CDC configuration type itf_num is the total enabled interface numbers, can not exceed CDC_INTERFACE_NUM_MAX, itf_num = 0 is same as itf_num = 1 for back compatibility, if itf_num > 1, user should config params with `s` ending like `bulk_in_ep_addrs` to config each interface, callbacks should not in block state.

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, 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
    return ESP_OK;
}

static void camera_stop_cb(void *cb_ctx)
{
    //user code
    return;
```

(continues on next page)

(continued from previous page)

```
}

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

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 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,
```

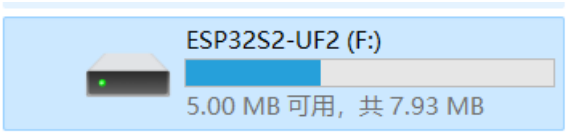
3. Using `idf.py menuconfig` to config the component's behavior in `(Top) → Component config → TinyUF2 Config`

- `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



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.

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

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()

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

Supported Socs	ESP32	ESP32-S2	ESP32-S3	ESP32-C3

The PWM audio function uses the internal LEDC peripheral in ESP32 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

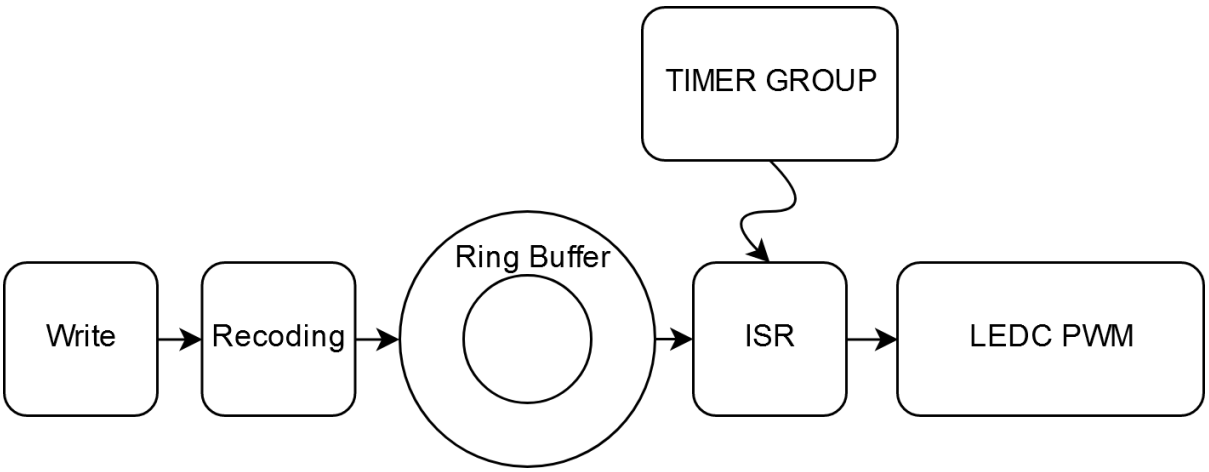


Fig. 1: Structure

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

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**(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
- **avi_data** –Pointer to the AVI file data in memory.
- **avi_size** –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**(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 **filename** –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**(void **buffer, size_t *buffer_size, *video_frame_info_t* *info, TickType_t ticks_to_wait)

Get one video frame from AVI stream.

Parameters
- **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**(void **buffer, size_t *buffer_size, *audio_frame_info_t* *info, TickType_t ticks_to_wait)

Get the audio buffer from AVI file.

Parameters
- **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**(void)

Stop AVI player.

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)

Initialize the AVI player.

Parameters **config** –**[in]** Configuration of AVI player

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**(void)

Deinitializes the AVI player.

This function deinitializes and cleans up resources used by the AVI player.

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

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)

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

GUI

8.1 LVGL Graphics Library

LVGL is an open-source free graphics library in C language providing everything you need to create embedded GUI with easy-to-use graphical elements, beautiful visual effects and low memory footprint.

8.1.1 Features

LVGL has the following features:

- More than 30 powerful, fully customizable widgets, e.g., button, slider, text area, keyboard and so on
- Supports various screens with any resolution
- Simple interface and low memory usage
- Supports multiple input device for the same screen
- Provides various drawing features, e.g., anti-aliasing, polygon, shadow, etc.
- Supports UTF-8 coding, multi language and multi font text
- Supports various image formats, can read images from flash or SD card
- provides online image converter
- Supports Micropython

8.1.2 Requirements

The minimum requirements for running LVGL are listed as follows:

- 16, 32 or 64 bit micro-controller or processor
- Clock frequency: > 16 MHz
- Flash/ROM: > 64 kB (180 kB is recommended)
- RAM: 8 kB (24 kB is recommended)
- 1 frame buffer
- Graphics buffer: > “horizontal resolution” pixels
- C99 or newer compiler

8.1.3 Online Tools

LVGL provides online Font Converter and Image Converter.

8.1.4 Demo Examples

Note: The following examples are no longer maintained. For LCD and LVGL examples, please refer to: i80_controller、rgb_panel And spi_lcd_touch

Official Demo

LVGL provides a demo project of using LVGL on ESP32 in LVGL project for ESP32.

On top of that, ESP-IoT-Solution also provides some application examples of using LVGL:

Thermostat

A thermostat control interface designed using LVGL:

Please find details of this example in hmi/lvgl_thermostat.

Coffee

An interactive interface of a coffee machine designed using LVGL:

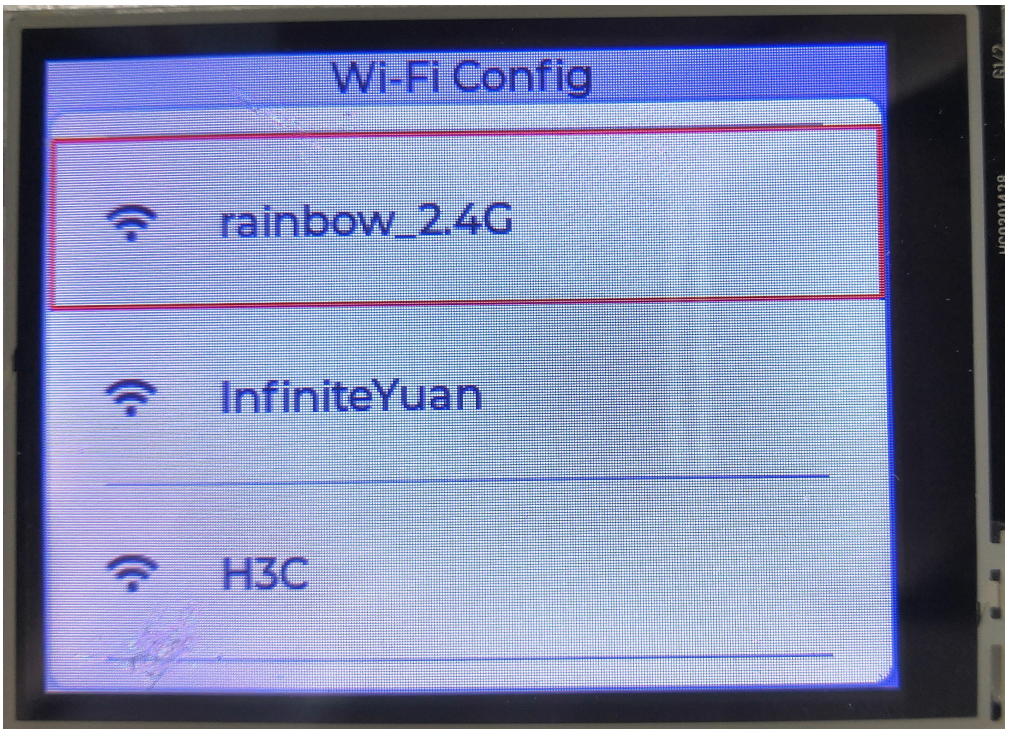
Please find details of this example in hmi/lvgl_coffee.

Wificonfig

When connecting Wi-Fi with ESP32, a Wi-Fi connection interface designed using LVGL can show information of the neighboring Wi-Fi, and you can type in Wi-Fi password on this interface.

Please find details of this example in hmi/lvgl_wificonfig.


RISTRETTO
WEAK
STRONG


Wi-Fi Config
rainbow_2.4G
InfiniteYuan
H3C


InfiniteYuan Password:
Canael
Connect
1# q w e r t y u i o p
ABC a s d f g h j k l
_ - z x c v b n m . , :

Chapter 9

AI

9.1 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 provides extensive support for the majority of OpenAI API functionalities, except for *files* and *fine-tunes*. For more details, please refer to OpenAI API REFERENCE.

9.1.1 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. **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.

9.1.2 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*

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. Save it with the first response if selected.

Param createSpeech[in] the point of OpenAI_SpeechResponse_t
Param p[in] the message for audio generation
Return *

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 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**

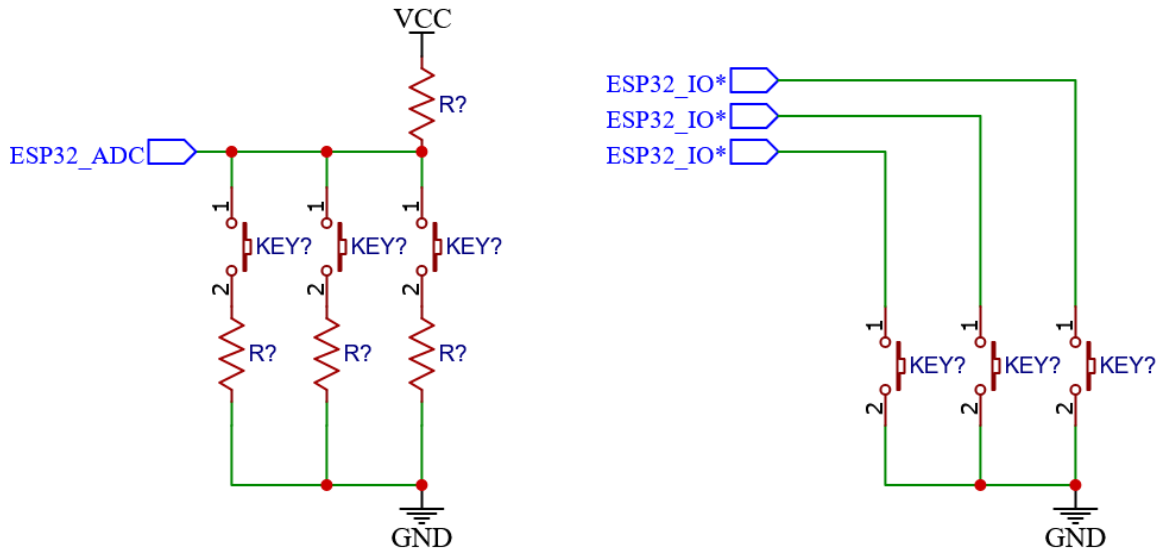
enumerator **OPENAI_AUDIO_OUTPUT_FORMAT_FLAC**

Chapter 10

Input Device

10.1 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:



ADC Button GPIO 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.

10.1.1 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.

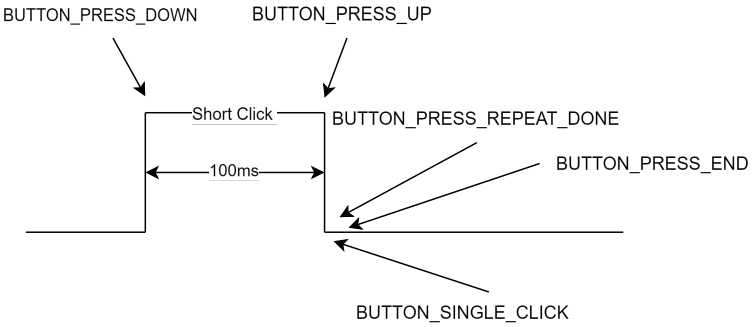
Each button supports **call-back** and **pooling** mode.

- 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

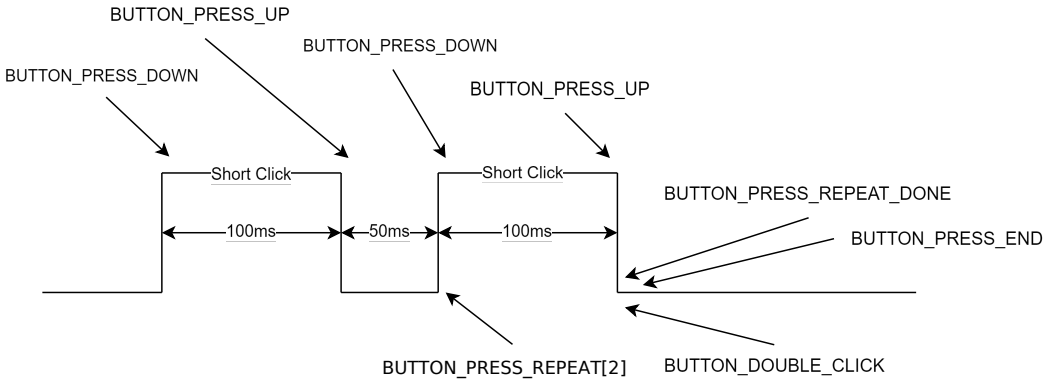
Note: you can also combine the above two methods.

Attention: No blocking operations such as **TaskDelay** are allowed in the call-back function

Single Click

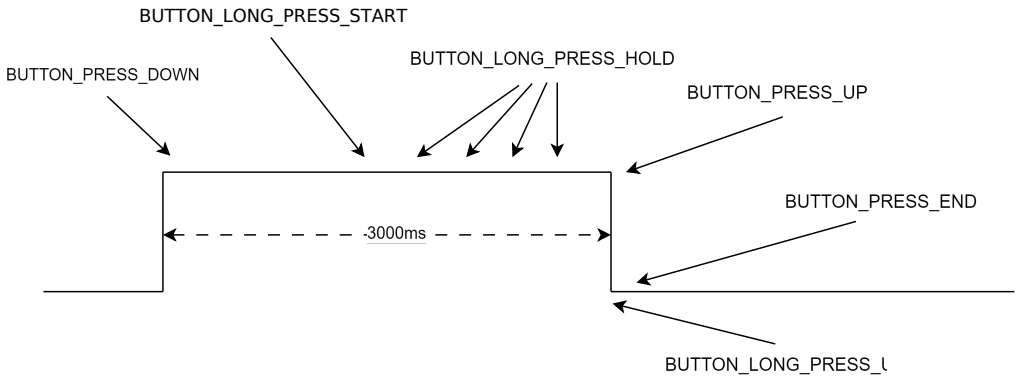


Double Click



LongPress

BUTTON_LONG_PRESS_TIME_MS: 1500ms



10.1.2 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 : ADC sample time
- BUTTON_SERIAL_TIME_MS : call-back interval triggered by long press time
- BUTTON_LONG_PRESS_TOLERANCE_MS: Used to set the tolerance time for long presses.

10.1.3 Demonstration

Create a button

```
// create gpio button
button_config_t gpio_btn_cfg = {
    .type = BUTTON_TYPE_GPIO,
    .long_press_time = CONFIG_BUTTON_LONG_PRESS_TIME_MS,
    .short_press_time = CONFIG_BUTTON_SHORT_PRESS_TIME_MS,
    .gpio_button_config = {
        .gpio_num = 0,
        .active_level = 0,
    },
};
button_handle_t gpio_btn = iot_button_create(&gpio_btn_cfg);
if(NULL == gpio_btn) {
    ESP_LOGE(TAG, "Button create failed");
}

// create adc button
button_config_t adc_btn_cfg = {
    .type = BUTTON_TYPE_ADC,
    .long_press_time = CONFIG_BUTTON_LONG_PRESS_TIME_MS,
    .short_press_time = CONFIG_BUTTON_SHORT_PRESS_TIME_MS,
    .adc_button_config = {
        .adc_channel = 0,
        .button_index = 0,
        .min = 100,
        .max = 400,
    },
};
button_handle_t adc_btn = iot_button_create(&adc_btn_cfg);
if(NULL == adc_btn) {
    ESP_LOGE(TAG, "Button create failed");
}

// create matrix keypad button
button_config_t matrix_button_cfg = {
    .type = BUTTON_TYPE_MATRIX,
    .long_press_time = CONFIG_BUTTON_LONG_PRESS_TIME_MS,
    .short_press_time = CONFIG_BUTTON_SHORT_PRESS_TIME_MS,
    .matrix_button_config = {
        .row_gpio_num = 0,
        .col_gpio_num = 1,
    }
};
button_handle_t matrix_button = iot_button_create(&matrix_button_cfg);
if(NULL == matrix_button) {
    ESP_LOGE(TAG, "Button create failed");
}
```

Note: When the IDF version is greater than or equal to release/5.0, the ADC button uses ADC1. If ADC1 is used elsewhere in the project, please provide the *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, 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_config_t cfg = {
    .event = BUTTON_LONG_PRESS_START,
    .event_data.long_press.press_time = 2000,
};

iot_button_register_event_cb(gpio_btn, cfg, button_long_press_1_cb,␣
↪NULL);

cfg.event_data.long_press.press_time = 5000;
iot_button_register_event_cb(gpio_btn, cfg, 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.

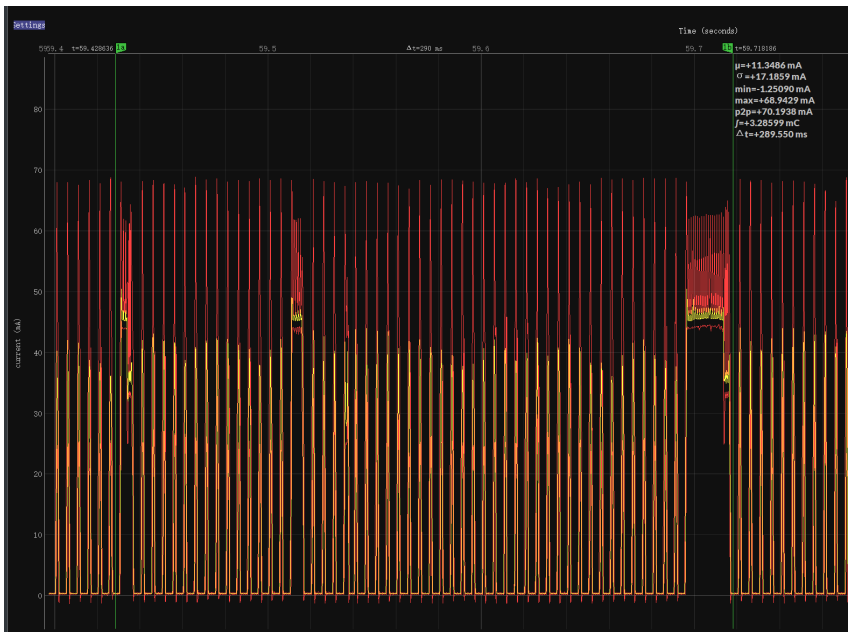
Configuration Required:

- Enable the *CONFIG_GPIO_BUTTON_SUPPORT_POWER_SAVE* option to include low-power-related code in the component.
- 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.

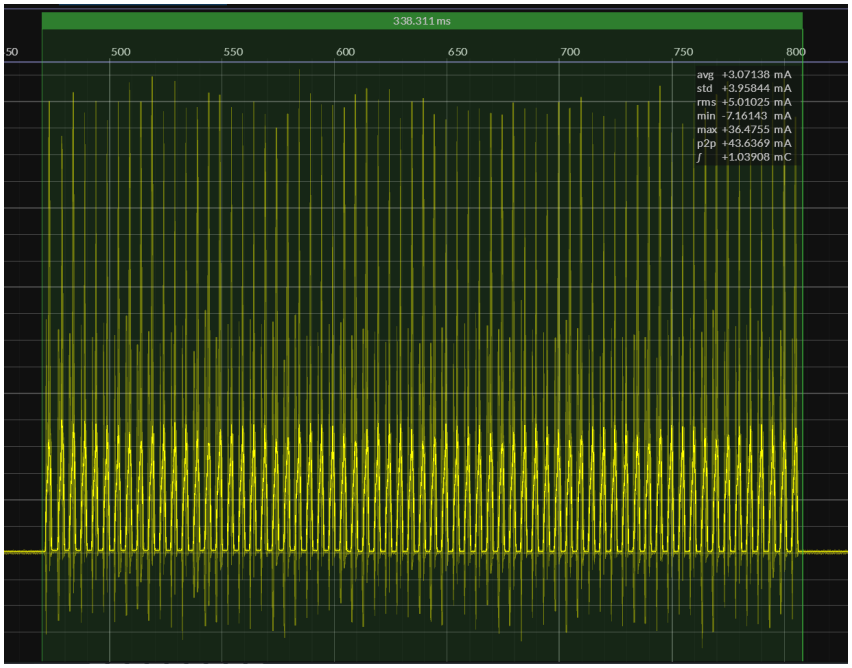
Note: This feature ensures that the Button component only wakes up the CPU when in use, but does not guarantee the CPU will always enter low-power mode.

Power Consumption Comparison:

- Without enabling low-power mode, pressing the button once:

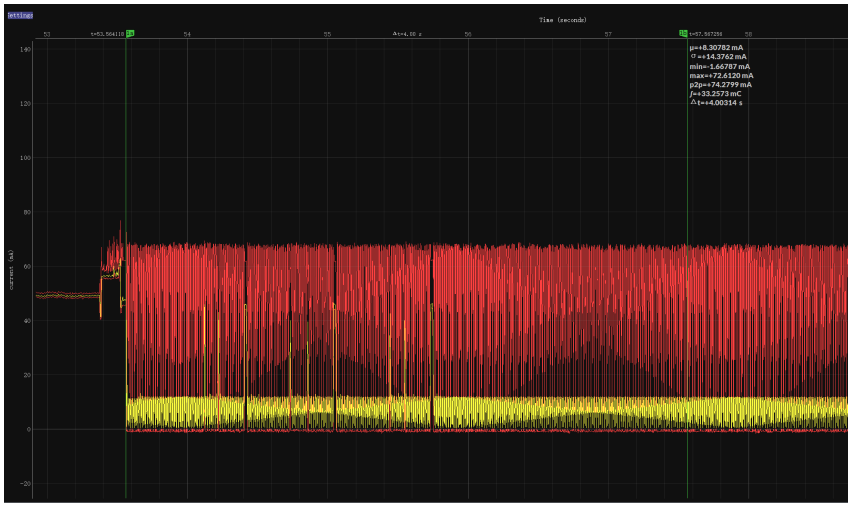


- With low-power mode enabled, pressing the button once:

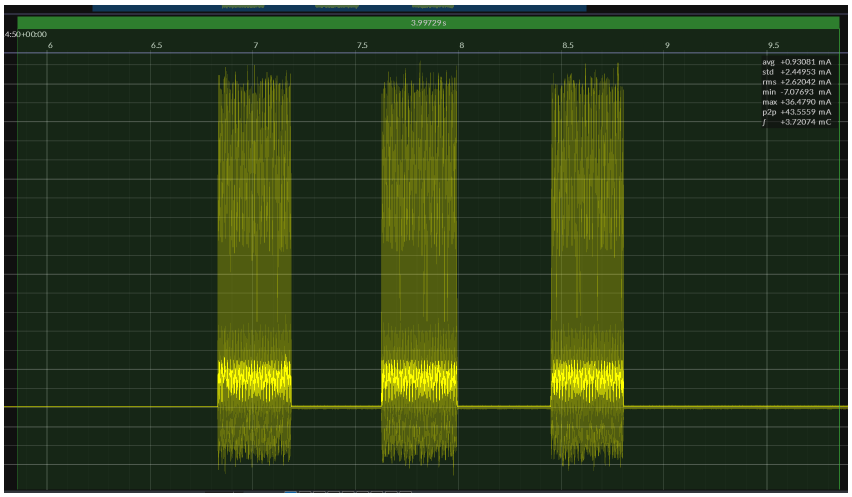


Because GPIO wakes up the CPU, supporting only level triggering, the CPU is awakened only when the button is at its operating level. Therefore, in low-power mode, the average current during a single press is higher than when low-power mode is not enabled, depending on the duration of the button press. 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 = {
    .type = BUTTON_TYPE_GPIO,
    .gpio_button_config = {
        .gpio_num = button_num,
        .active_level = BUTTON_ACTIVE_LEVEL,
        .enable_power_save = true,
    },
};
button_handle_t btn = iot_button_create(&btn_cfg);
```

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();
```

10.1.4 API Reference

Header File

- components/button/include/iot_button.h

Functions

button_handle_t **iot_button_create**(const *button_config_t* *config)

Create a button.

Parameters **config** –pointer of button configuration, must corresponding the button type
Returns A handle to the created button, or NULL in case of error.

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_cb_t* cb, void *usr_data)

Register the button event callback function.

Parameters

- **btn_handle** –A button handle to register
- **event** –Button event
- **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_register_event_cb**(*button_handle_t* btn_handle, *button_event_config_t* event_cfg, *button_cb_t* cb, void *usr_data)

Register the button event callback function.

Parameters

- **btn_handle** –A button handle to register
- **event_cfg** –Button event configuration
- **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_event**(*button_handle_t* btn_handle, *button_event_config_t* event_cfg, *button_cb_t* cb)

Unregister the button event callback function. In case event_data is also passed it will unregister function for that particular event_data only.

Parameters

- **btn_handle** –A button handle to unregister
- **event_cfg** –Button event
- **cb** –callback to unregister

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG Arguments is invalid.
- ESP_ERR_INVALID_STATE The Callback was never registered with the event

esp_err_t **iot_button_unregister_cb**(*button_handle_t* btn_handle, *button_event_t* event)

Unregister all the callbacks associated with the event.

Parameters

- **btn_handle** –A button handle to unregister
- **event** –Button event

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**(*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_ticks_time**(*button_handle_t* btn_handle)

Get button ticks time.

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

Unions

union **button_event_data_t**

#include <iot_button.h> Button events data.

Public Members

struct *button_event_data_t*::*long_press_t* **long_press**

long press struct, for event BUTTON_LONG_PRESS_START and BUTTON_LONG_PRESS_UP

struct *button_event_data_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_event_config_t**

Button events configuration.

Public Members

button_event_t **event**

button event type

button_event_data_t **event_data**

event data corresponding to the event

struct **button_custom_config_t**

custom button configuration

Public Members

uint8_t **active_level**

active level when press down

esp_err_t (***button_custom_init**)(void *param)

user defined button init

uint8_t (***button_custom_get_key_value**)(void *param)

user defined button get key value

esp_err_t (***button_custom_deinit**)(void *param)

user defined button deinit

void ***priv**

private data used for custom button, MUST be allocated dynamically and will be auto freed in iot_button_delete

struct **button_config_t**

Button configuration.

Public Members

button_type_t **type**
button type, The corresponding button configuration must be filled

uint16_t **long_press_time**
Trigger time(ms) for long press, if 0 default to BUTTON_LONG_PRESS_TIME_MS

uint16_t **short_press_time**
Trigger time(ms) for short press, if 0 default to BUTTON_SHORT_PRESS_TIME_MS

button_gpio_config_t **gpio_button_config**
gpio button configuration

button_matrix_config_t **matrix_button_config**
matrix key button configuration

button_custom_config_t **custom_button_config**
custom button configuration

union *button_config_t*::[anonymous] **[anonymous]**
button configuration

Type Definitions

typedef void (***button_cb_t**)(void *button_handle, void *usr_data)

typedef void ***button_handle_t**

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_type_t**

Supported button type.

Values:

enumerator **BUTTON_TYPE_GPIO**

enumerator **BUTTON_TYPE_ADC**

enumerator **BUTTON_TYPE_MATRIX**

enumerator **BUTTON_TYPE_CUSTOM**

enum **button_param_t**

Button parameter.

Values:

enumerator **BUTTON_LONG_PRESS_TIME_MS**

enumerator **BUTTON_SHORT_PRESS_TIME_MS**

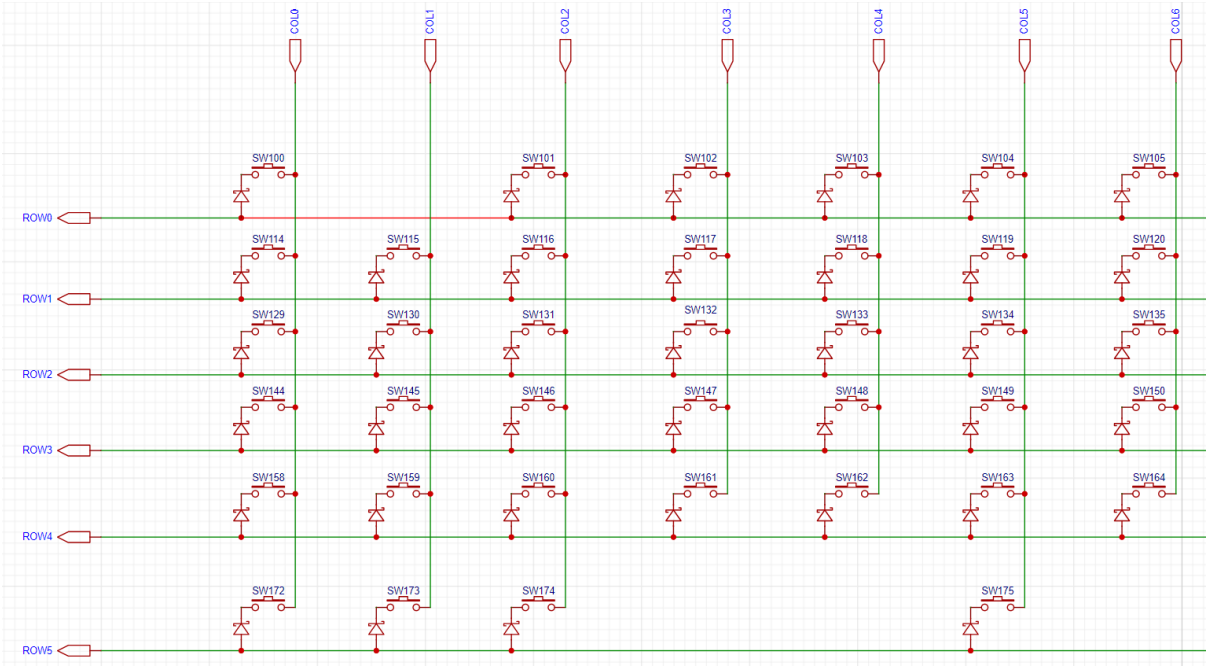
enumerator **BUTTON_PARAM_MAX**

10.2 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.

10.2.1 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.

10.2.2 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[])
```

(continues on next page)

(continued from previous page)

```
    {
        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.

10.2.3 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**

10.3 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.

10.3.1 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.

10.3.2 Hardware Design

The reference design for the rotary encoder is shown below:

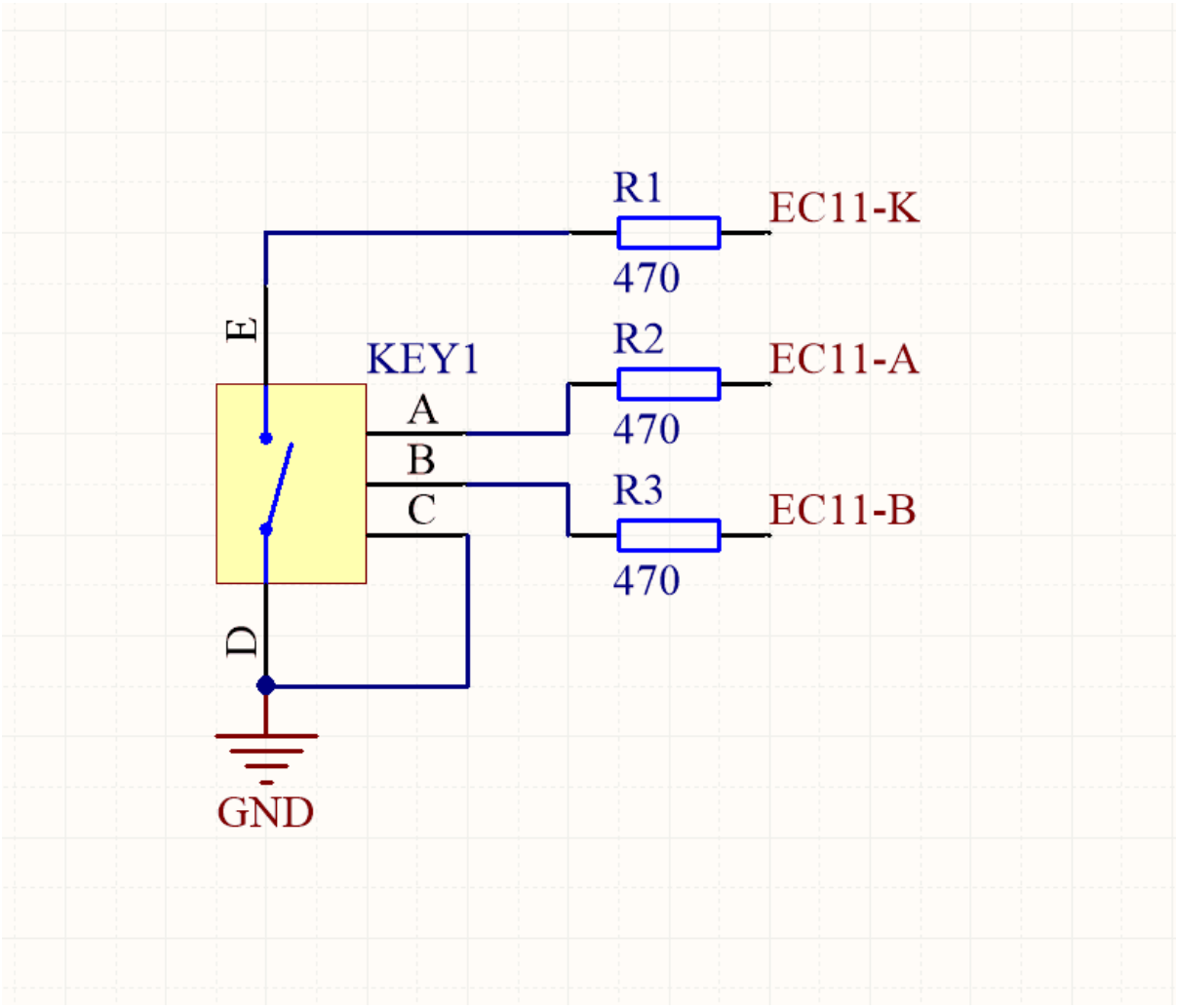
10.3.3 Knob Event

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.


R1
EC11-K
470
E
KEY1
R2
EC11-A
A
470
B
R3
C
EC11-B
470
D
GND

Attention: No blocking operations such as TaskDelay in the callback function

10.3.4 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

10.3.5 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);
```

10.3.6 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.

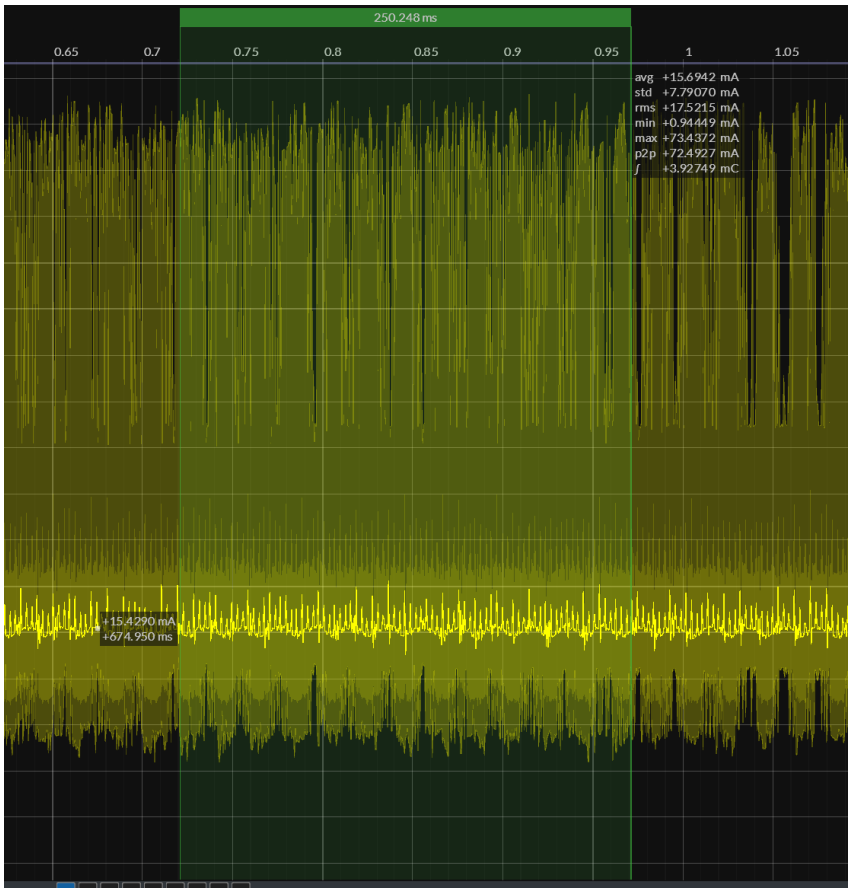
Required Configuration:

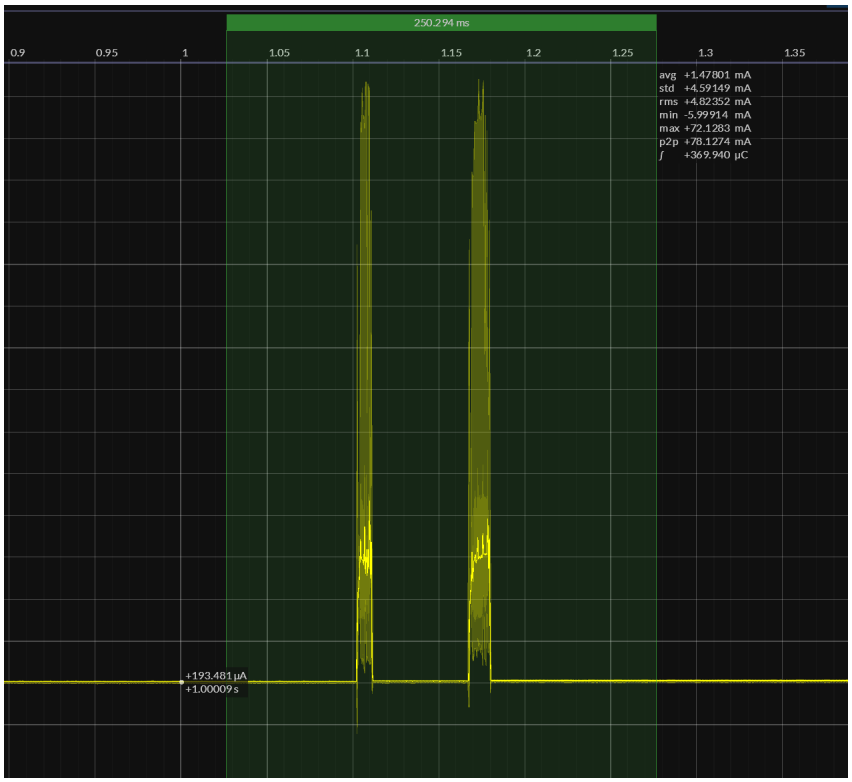
- Enable the *enable_power_save* option in *knob_config_t*.

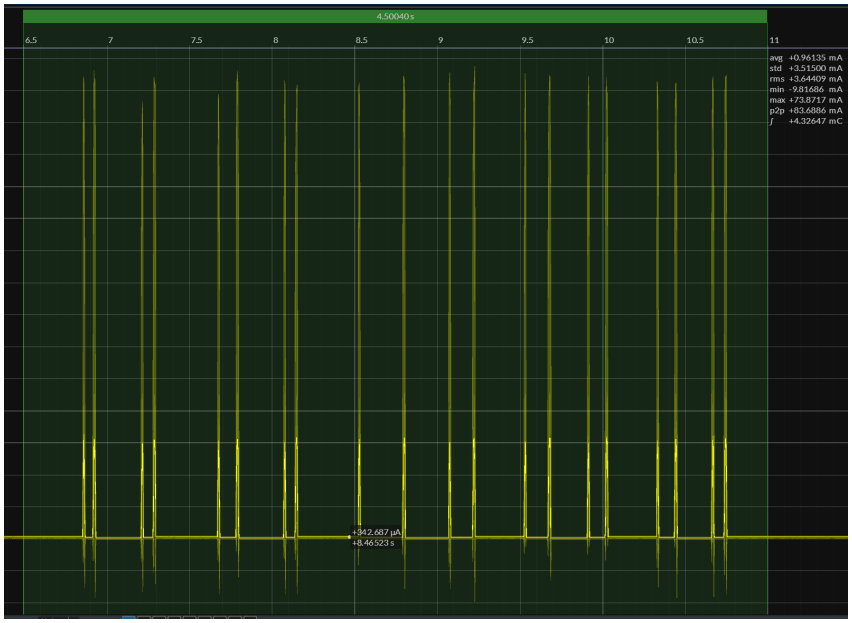
Power Consumption Comparison:

- 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.


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



10.3.7 Enable and Disable

The component supports enabling and disabling at any time.

```
// Stop knob
iot_knob_stop();
// Resume knob
iot_knob_resume();
```

10.3.8 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

10.4 Touch Panel

Touch panels are now standard components in display applications. ESP-IoT-Solution provides drivers for common types of touch panels and currently supports the following controllers:

Resistive Touch Panel	Capacitive Touch Panel
XPT2046	FT5216
NS2016	FT5436
	FT6336
	FT5316

The capacitive touch panel controllers listed above can usually be driven by `FT5x06`.

Similar to the screen driver, some common functions are encapsulated in the *touch_panel_driver_t* structure, in order to port them to different GUI libraries easily. After initializing the touch panel, users can conduct operations by calling functions inside the structure, without paying attention to specific touch panel models.

10.4.1 Touch Panel Calibration

In actual applications, resistive touch panels must be calibrated before use, while capacitive touch panels are usually calibrated by controllers and do not require extra calibration steps. A calibration algorithm is integrated in the resistive touch panel driver. During the process, three points are used to calibrate, in which one point is used for verification. If the verified error exceeds a certain threshold value, it means the calibration has failed and a new round of calibration is started automatically until it succeeds.

The calibration process will be started by calling `calibration_run()`. After finished, the parameters are stored in NVS for next initialization to avoid repetitive work.

10.4.2 Press Touch Panel

Generally, there is an interrupt pin inside the touch panel controller (both resistive and capacitive) to signal touch events. However, this is not used in the touch panel driver, because IOs should be saved for other peripherals in screen applications as much as possible; on the other hand the information in this signal is not as accurate as data in registers.

For resistive touch panels, when the pressure in the Z direction exceeds the configured threshold, it is considered as pressed; for capacitive touch panels, the detection of over one touch point will be considered as pressed.

10.4.3 Touch Panel Rotation

A touch panel has eight directions, like the screen, defined in *touch_panel_dir_t*. The rotation of a touch panel is achieved by software, which usually sets the direction of a touch panel and a screen as the same. But this should not be fixed, for example, when using a capacitive touch panel, the inherent direction of the touch panel may not fit with the original display direction of the screen. Simply setting these two directions as the same may not show the desired contents. Therefore, please adjust the directions according to the actual situation.

On top of that, the configuration of its resolution is also important since the converted display after a touch panel being rotated relies on the resolution of its width and height. An incorrect configuration of the resolution may give you a distorted display.

Note: If you are using a resistive touch panel, the touch position can become inaccurate after it being rotated, since the resistance value in each direction may not be distributed uniformly. It is recommended to not rotate a resistive touch panel after it being calibrated.

10.4.4 Application Example

Initialize a Touch Panel

```
touch_panel_driver_t touch; // a touch panel driver

i2c_config_t i2c_conf = {
    .mode = I2C_MODE_MASTER,
    .sda_io_num = 35,
    .sda_pullup_en = GPIO_PULLUP_ENABLE,
    .scl_io_num = 36,
    .scl_pullup_en = GPIO_PULLUP_ENABLE,
    .master.clk_speed = 100000,
};
i2c_bus_handle_t i2c_bus = i2c_bus_create(I2C_NUM_0, &i2c_conf);

touch_panel_config_t touch_cfg = {
    .interface_i2c = {
        .i2c_bus = i2c_bus,
        .clk_freq = 100000,
        .i2c_addr = 0x38,
    },
    .interface_type = TOUCH_PANEL_IFACE_I2C,
    .pin_num_int = -1,
    .direction = TOUCH_DIR_LRTB,
    .width = 800,
    .height = 480,
};

/* Initialize touch panel controller FT5x06 */
touch_panel_find_driver(TOUCH_PANEL_CONTROLLER_FT5X06, &touch);
touch.init(&touch_cfg);

/* start to run calibration */
touch.calibration_run(&lcd, false);
```

Note:

- When using a capacitive touch panel, the call to the calibration function will return `ESP_OK` directly.
- By default, only FT5x06 touch panel driver is enabled, please go to `menuconfig -> Component config -> Touch Screen Driver -> Choose Touch Screen Driver` to do configurations if you need to enable other drivers.

To Know If a Touch Panel is Pressed and Its Corresponding Position

```
touch_panel_points_t points;
touch.read_point_data(&points);
int32_t x = points.curx[0];
int32_t y = points.cury[0];
if(TOUCH_EVT_PRESS == points.event) {
    ESP_LOGI(TAG, "Pressed, Touch point at (%d, %d)", x, y);
}
```

10.4.5 API Reference

Header File

- components/display/touch_panel/touch_panel.h

Functions

esp_err_t **touch_panel_find_driver**(*touch_panel_controller_t* controller, *touch_panel_driver_t* *out_driver)

Find a touch panel controller driver.

Parameters
- **controller** –Touch panel controller to initialize
- **out_driver** –Pointer to a touch driver

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG Arguments is NULL.
- ESP_ERR_NOT_FOUND: Touch panel controller was not found.

Structures

struct **touch_panel_points_t**

Information of touch panel.

Public Members

touch_panel_event_t **event**

Event of touch

uint8_t **point_num**

Touch point number

uint16_t **curx**[TOUCH_MAX_POINT_NUMBER]

Current x coordinate

uint16_t **cury**[TOUCH_MAX_POINT_NUMBER]

Current y coordinate

struct **touch_panel_config_t**

Configuration of touch panel.

Public Members

i2c_bus_handle_t **i2c_bus**

Handle of i2c bus

int **clk_freq**

i2c clock frequency

spi clock frequency

uint8_t **i2c_addr**

screen i2c slave address

struct *touch_panel_config_t*::[anonymous]::[anonymous] **interface_i2c**

I2c interface

spi_bus_handle_t **spi_bus**

Handle of spi bus

int8_t **pin_num_cs**

SPI Chip Select Pin

struct *touch_panel_config_t*::[anonymous]::[anonymous] **interface_spi**

SPI interface

union *touch_panel_config_t*::[anonymous] **[anonymous]**

Interface configuration

touch_panel_interface_type_t **interface_type**

Interface bus type, see touch_interface_type_t struct

int8_t **pin_num_int**

Interrupt pin of touch panel. NOTE: You can set to -1 for no connection with hardware. If PENIRQ is connected, set this to pin number.

touch_panel_dir_t **direction**

Rotate direction

uint16_t **width**

touch panel width

uint16_t **height**

touch panel height

struct **touch_panel_driver_t**

Define screen common function.

Public Members

esp_err_t (***init**)(const *touch_panel_config_t* *config)

Initial touch panel.

Attention If you have been called function touch_panel_init() that will call this function automatically, and should not be called it again.

Param config Pointer to a structure with touch config arguments.

Return

- ESP_OK Success
- ESP_FAIL Fail

esp_err_t (*`deinit`)(void)

Deinitial touch panel.

Return
- ESP_OK Success
- ESP_FAIL Fail

esp_err_t (*`calibration_run`)(const scr_driver_t *screen, bool recalibrate)

Start run touch panel calibration.

Param screen Screen driver for display prompts
Param recalibrate Is mandatory, set true to force calibrate
Return
- ESP_OK Success
- ESP_FAIL Fail

esp_err_t (*`set_direction`)(*touch_panel_dir_t* dir)

Set touch rotate rotation.

Param dir rotate direction
Return
- ESP_OK Success
- ESP_FAIL Fail

esp_err_t (*`read_point_data`)(*touch_panel_points_t* *point)

Get current touch information, see struct *touch_panel_points_t*.

Param point a pointer of *touch_panel_points_t* contained touch information.
Return
- ESP_OK Success
- ESP_FAIL Fail

Macros

`TOUCH_MAX_POINT_NUMBER`

max point number on touch panel

Enumerations

enum `touch_panel_event_t`

Touch events.

Values:

enumerator `TOUCH_EVT_RELEASE`

Release event

enumerator `TOUCH_EVT_PRESS`

Press event

enum `touch_panel_dir_t`

Define all screen direction.

Values:

enumerator **TOUCH_DIR_LRTB**

From left to right then from top to bottom, this consider as the original direction of the touch panel

enumerator **TOUCH_DIR_LRBT**

From left to right then from bottom to top

enumerator **TOUCH_DIR_RLTB**

From right to left then from top to bottom

enumerator **TOUCH_DIR_RLBT**

From right to left then from bottom to top

enumerator **TOUCH_DIR_TBLR**

From top to bottom then from left to right

enumerator **TOUCH_DIR_BTLR**

From bottom to top then from left to right

enumerator **TOUCH_DIR_TBRL**

From top to bottom then from right to left

enumerator **TOUCH_DIR_BTRL**

From bottom to top then from right to left

enumerator **TOUCH_DIR_MAX**

enumerator **TOUCH_MIRROR_X**

Mirror X-axis

enumerator **TOUCH_MIRROR_Y**

Mirror Y-axis

enumerator **TOUCH_SWAP_XY**

Swap XY axis

enum **touch_panel_interface_type_t**

Values:

enumerator **TOUCH_PANEL_IFACE_I2C**

I2C interface

enumerator **TOUCH_PANEL_IFACE_SPI**

SPI interface

enum **touch_panel_controller_t**

All supported touch panel controllers.

Values:

enumerator **TOUCH_PANEL_CONTROLLER_FT5X06**

enumerator **TOUCH_PANEL_CONTROLLER_XPT2046**

enumerator **TOUCH_PANEL_CONTROLLER_NS2016**

Chapter 11

IR

11.1 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.

11.1.1 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);
}
```

11.1.2 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 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.

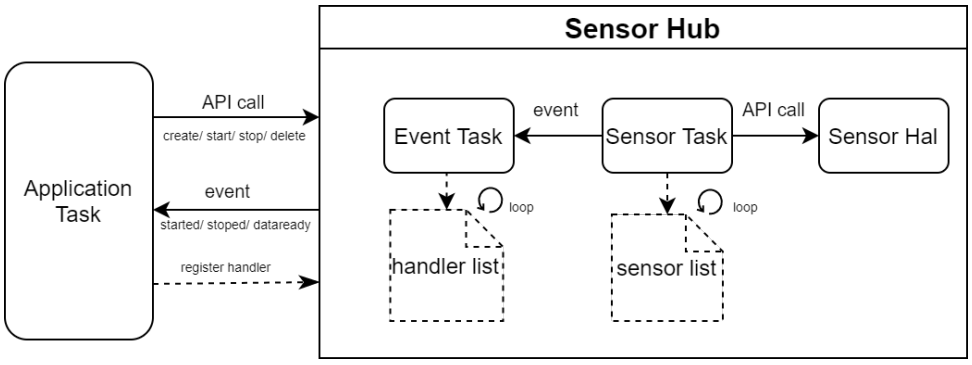


Fig. 1: Sensor Hub Programming Model

The Sensor Hub provides hardware abstraction for common sensor categories, based on which users can switch sensor models without modifying the upper layer of the application. And it allows adding new sensors to the Sensor Hub by implementing their corresponding sensor interface at hardware abstraction layer. This component can be used as a basic component for sensor applications in various intelligent scenarios such as environment monitoring, motion detection, health management and etc. as it simplifies operation and improves operating efficiency by centralized management of sensors.

12.1.1 Instructions

1. Create a sensor instance: use *iot_sensor_create()* to create a sensor instance. The related parameters include the sensor ID defined in `sensor_id_t`, configuration options for the sensor and its handler pointer. The sensor ID is used to find and load the corresponding driver, and each ID can only be used for one sensor instance. In configuration options, *bus* is used to specify the bus location on which the sensor is mounted; *mode* is used to specify the operating mode of the sensor; *min_delay* is used to specify the acquisition interval of the

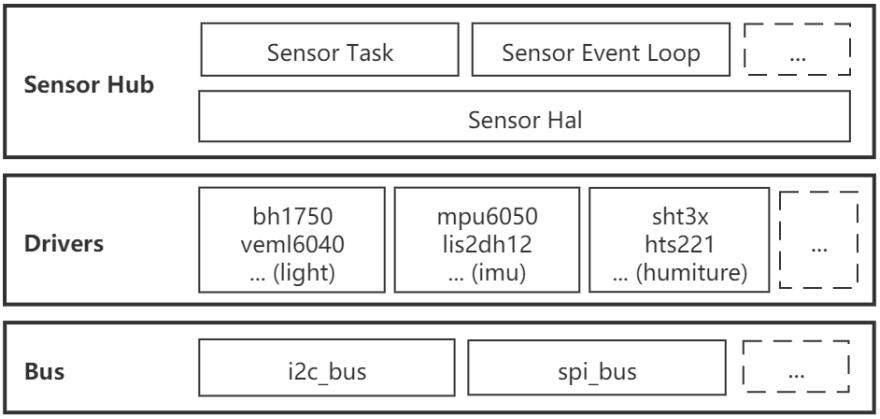


Fig. 2: Sensor Hub Driver

sensor, while other items inside are all non-required options. After the instance is created, the sensor handler is obtained;

2. 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
3. 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 `SENSOR_XXXX_DATA_READY` event. The event callback function can obtain the specific data of each event via the `event_data` parameter;
4. 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;
5. 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
6. 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

uint8_t **sensor_id**

sensor id

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_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**(*sensor_id_t* sensor_id, const *sensor_config_t* *config, *sensor_handle_t* *p_sensor_handle)

Create a sensor instance with specified sensor_id and desired configurations.

Parameters
- **sensor_id** –sensor’ s id detailed in sensor_id_t.
- **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** –point to sensor handle, will set to NULL if delete succeed.
Returns esp_err_t
- ESP_OK Success

- ESP_FAIL Fail

uint8_t **iot_sensor_scan**(*bus_handle_t* bus, *sensor_info_t* *buf[], uint8_t num)

Scan for valid sensors attached on bus.

Parameters
- **bus** –bus handle
- **buf** –Pointer to a buffer to save sensors' information, if NULL no information will be saved.
- **num** –Maximum number of sensor information to save, invalid if buf set to NULL, latter sensors will be discarded if num less-than the total number found on the bus.

Returns uint8_t total number of valid sensors found on the bus

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 type

Public Members

const char ***name**

sensor name

const char ***desc**

sensor descriptive message

sensor_id_t **sensor_id**

sensor id

const uint8_t ***addrs**

sensor address list

struct **sensor_config_t**

sensor initialization parameter

Public Members

bus_handle_t **bus**

i2c/spi bus handle

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

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

Enumerations

enum **sensor_id_t**

sensor id, used for iot_sensor_create

Values:

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

Functions

sensor_humiture_handle_t **humiture_create** (*bus_handle_t* bus, int id)

Create a humiture/temperature/humidity sensor instance. Same series' sensor or sensor with same address can only be created once.

Parameters
- **bus** –i2c bus handle the sensor attached to
- **id** –id declared in humiture_id_t

Returns sensor_humiture_handle_t return humiture sensor handle if succeed, return NULL if create failed.

esp_err_t **humiture_delete** (*sensor_humiture_handle_t* *sensor)

Delete and release the sensor resource.

Parameters **sensor** –point to humiture sensor handle, will set to NULL if delete succeed.

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **humiture_test** (*sensor_humiture_handle_t* sensor)

Test if sensor is active.

Parameters **sensor** –humiture sensor handle to operate

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **humiture_acquire_humidity** (*sensor_humiture_handle_t* sensor, float *humidity)

Acquire humiture sensor relative humidity result one time.

Parameters
- **sensor** –humiture sensor handle to operate.
- **humidity** –result data (unit:percentage)

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail
- ESP_ERR_NOT_SUPPORTED Function not supported on this sensor

esp_err_t **humiture_acquire_temperature** (*sensor_humiture_handle_t* sensor, float *sensor_data)

Acquire humiture sensor temperature result one time.

Parameters
- **sensor** –humiture sensor handle to operate.
- **sensor_data** –result data (unit:dCelsius)

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail
- ESP_ERR_NOT_SUPPORTED Function not supported on this sensor

esp_err_t **humiture_sleep** (*sensor_humiture_handle_t* sensor)

Set sensor to sleep mode.

Parameters **sensor** –humiture sensor handle to operate

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail

- ESP_ERR_NOT_SUPPORTED Function not supported on this sensor

esp_err_t **humiture_wakeup**(*sensor_humiture_handle_t* sensor)

Wakeup sensor from sleep mode.

Parameters **sensor** –humiture sensor handle to operate

Returns esp_err_t

- ESP_OK Success
- ESP_FAIL Fail
- ESP_ERR_NOT_SUPPORTED Function not supported on this sensor

esp_err_t **humiture_acquire**(*sensor_humiture_handle_t* sensor, *sensor_data_group_t* *data_group)

acquire a group of sensor data

Parameters

- **sensor** –humiture sensor handle to operate
- **data_group** –acquired data

Returns esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **humiture_control**(*sensor_humiture_handle_t* sensor, *sensor_command_t* cmd, void *args)

control sensor mode with control commands and args

Parameters

- **sensor** –humiture sensor handle to operate
- **cmd** –control commands detailed in sensor_command_t
- **args** –control commands args
  - ESP_OK Success
  - ESP_FAIL Fail
  - ESP_ERR_NOT_SUPPORTED Function not supported on this sensor

Type Definitions

typedef void ***sensor_humiture_handle_t**

humiture sensor handle

Enumerations

enum **humiture_id_t**

humiture sensor id, used for humiture_create

Values:

enumerator **SHT3X_ID**

sht3x humiture sensor id

enumerator **HTS221_ID**

hts221 humiture sensor id

enumerator **HUMITURE_MAX_ID**

max humiture sensor id

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

Functions

sensor_imu_handle_t **imu_create** (*bus_handle_t* bus, int imu_id)

Create a Inertial Measurement Unit sensor instance. Same series' sensor or sensor with same address can only be created once.

Parameters
- **bus** –i2c bus handle the sensor attached to
- **imu_id** –id declared in imu_id_t

Returns sensor_imu_handle_t return imu sensor handle if succeed, NULL is failed.

esp_err_t **imu_delete** (*sensor_imu_handle_t* *sensor)

Delete and release the sensor resource.

Parameters **sensor** –point to imu sensor handle, will set to NULL if delete succeed.

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **imu_test** (*sensor_imu_handle_t* sensor)

Test if sensor is active.

Parameters **sensor** –imu sensor handle to operate

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **imu_acquire_acce** (*sensor_imu_handle_t* sensor, axis3_t *acce)

Acquire imu sensor accelerometer result one time.

Parameters
- **sensor** –imu sensor handle to operate
- **acce** –result data (unit:g)

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail
- ESP_ERR_NOT_SUPPORTED Function not supported on this sensor

esp_err_t **imu_acquire_gyro**(*sensor_imu_handle_t* sensor, axis3_t *gyro)

Acquire imu sensor gyroscope result one time.

Parameters
- **sensor** –imu sensor handle to operate
- **gyro** –result data (unit:dps)

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail
- ESP_ERR_NOT_SUPPORTED Function not supported on this sensor

esp_err_t **imu_sleep**(*sensor_imu_handle_t* sensor)

Set sensor to sleep mode.

Parameters **sensor** –imu sensor handle to operate

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail
- ESP_ERR_NOT_SUPPORTED Function not supported on this sensor

esp_err_t **imu_wakeup**(*sensor_imu_handle_t* sensor)

Wakeup sensor from sleep mode.

Parameters **sensor** –imu sensor handle to operate

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail
- ESP_ERR_NOT_SUPPORTED Function not supported on this sensor

esp_err_t **imu_acquire**(*sensor_imu_handle_t* sensor, *sensor_data_group_t* *data_group)

acquire a group of sensor data

Parameters
- **sensor** –imu sensor handle to operate
- **data_group** –acquired data

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **imu_control**(*sensor_imu_handle_t* sensor, *sensor_command_t* cmd, void *args)

control sensor mode with control commands and args

Parameters
- **sensor** –imu sensor handle to operate
- **cmd** –control commands detailed in sensor_command_t
- **args** –control commands args
  - ESP_OK Success
  - ESP_FAIL Fail
  - ESP_ERR_NOT_SUPPORTED Function not supported on this sensor

Type Definitions

typedef void ***sensor_imu_handle_t**

imu sensor handle

Enumerations

enum **imu_id_t**

imu sensor id, used for imu_create

Values:

enumerator **MPU6050_ID**

MPU6050 imu sensor id

enumerator **LIS2DH12_ID**

LIS2DH12 imu sensor id

enumerator **IMU_MAX_ID**

max imu sensor id

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

Functions

sensor_light_handle_t **light_sensor_create**(*bus_handle_t* bus, int id)

Create a light sensor instance. same series' sensor or sensor with same address can only be created once.

Parameters

- **bus** –i2c bus handle the sensor attached to
- **id** –id declared in light_sensor_id_t

Returns sensor_light_handle_t return light sensor handle if succeed, return NULL if failed.

esp_err_t **light_sensor_delete**(*sensor_light_handle_t* *sensor)

Delete and release the sensor resource.

Parameters **sensor** –point to light sensor handle, will set to NULL if delete succeed.
Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **light_sensor_test**(*sensor_light_handle_t* sensor)

Test if sensor is active.

Parameters **sensor** –light sensor handle to operate.
Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **light_sensor_acquire_light**(*sensor_light_handle_t* sensor, float *lux)

Acquire light sensor illuminance result one time.

Parameters
- **sensor** –light sensor handle to operate.
- **lux** –result data (unit:lux)

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail
- ESP_ERR_NOT_SUPPORTED Function not supported on this sensor

esp_err_t **light_sensor_acquire_rgbw**(*sensor_light_handle_t* sensor, rgbw_t *rgbw)

Acquire light sensor color result one time. light color includes red green blue and white.

Parameters
- **sensor** –light sensor handle to operate.
- **rgbw** –result data (unit:lux)

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail
- ESP_ERR_NOT_SUPPORTED Function not supported on this sensor

esp_err_t **light_sensor_acquire_uv**(*sensor_light_handle_t* sensor, uv_t *uv)

Acquire light sensor ultra violet result one time. light Ultraviolet includes UVA UVB and UV.

Parameters
- **sensor** –light sensor handle to operate.
- **uv** –result data (unit:lux)

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail
- ESP_ERR_NOT_SUPPORTED Function not supported on this sensor

esp_err_t **light_sensor_sleep**(*sensor_light_handle_t* sensor)

Set sensor to sleep mode.

Parameters **sensor** –light sensor handle to operate.
Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail
- ESP_ERR_NOT_SUPPORTED Function not supported on this sensor

esp_err_t **light_sensor_wakeup**(*sensor_light_handle_t* sensor)

Wakeup sensor from sleep mode.

Parameters **sensor** –light sensor handle to operate.
Returns esp_err_t

- ESP_OK Success
- ESP_FAIL Fail
- ESP_ERR_NOT_SUPPORTED Function not supported on this sensor

esp_err_t **light_sensor_acquire**(*sensor_light_handle_t* sensor, *sensor_data_group_t* *data_group)

acquire a group of sensor data

Parameters
- **sensor** –light sensor handle to operate
- **data_group** –acquired data

Returns esp_err_t
- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **light_sensor_control**(*sensor_light_handle_t* sensor, *sensor_command_t* cmd, void *args)

control sensor mode with control commands and args

Parameters
- **sensor** –light sensor handle to operate
- **cmd** –control commands detailed in sensor_command_t
- **args** –control commands args
  - ESP_OK Success
  - ESP_FAIL Fail
  - ESP_ERR_NOT_SUPPORTED Function not supported on this sensor

Type Definitions

typedef void ***sensor_light_handle_t**

light sensor handle

Enumerations

enum **light_sensor_id_t**

light sensor id, used for light_sensor_create

Values:

enumerator **BH1750_ID**

BH1750 light sensor id

enumerator **VEML6040_ID**

VEML6040 light sensor id

enumerator **VEML6075_ID**

VEML6075 light sensor id

enumerator **LIGHT_MAX_ID**

max light sensor id

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	Datesheet	

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	Datesheet	

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℃ 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

Chapter 13

Touch Sensor

13.1 Touch Proximity Sensor

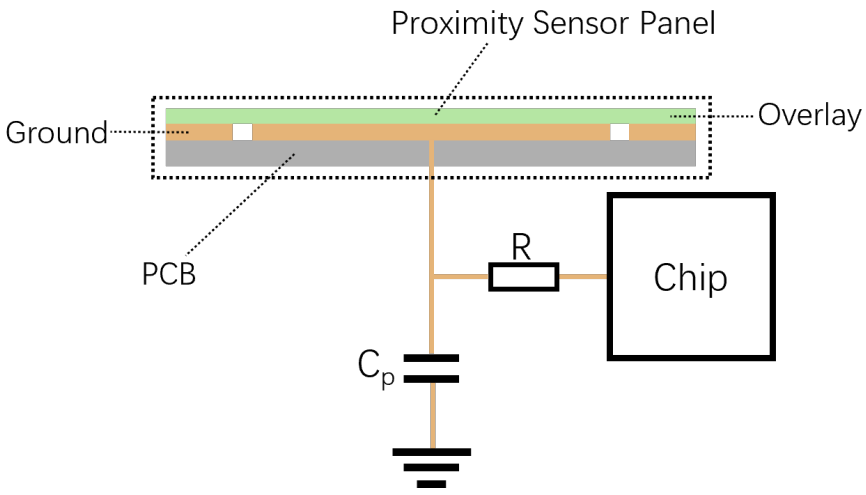
The `touch_proximity_sensor` component is developed based on the built-in touch sensors of ESP32-S3. Using this component makes it easy to implement touch proximity sensing functionality.

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 *`proxi_config_t`* structure.

```
// Configuration structure for touch proximity sensor
typedef struct {
    uint32_t channel_num;
    uint32_t channel_list[TOUCH_PROXIMITY_NUM_MAX];
    uint32_t meas_count;
    float smooth_coef;
    float baseline_coef;
    float max_p;
    float min_n;
    float threshold_p[TOUCH_PROXIMITY_NUM_MAX];
    float threshold_n[TOUCH_PROXIMITY_NUM_MAX];
    float hysteresis_p;
    float noise_p;
    float noise_n;
    uint32_t debounce_p;
    uint32_t debounce_n;
    uint32_t reset_p;
    uint32_t reset_n;
    uint32_t gold_value[TOUCH_PROXIMITY_NUM_MAX];
} proxi_config_t;
```

Specific parameter descriptions are as follows:

Configuration Parameter	Description
channel_num	Number of proximity sensing channels, up to 3
channel_list	List of proximity sensing channels, i.e., touch channels
meas_count	Cumulative measurement count for proximity sensing channels
smooth_coef	Smoothing coefficient for data
baseline_coef	Baseline coefficient determining the rate of baseline adjustment
max_p	Maximum effective positive change rate
min_n	Minimum effective negative change rate
threshold_p	Positive trigger threshold
threshold_n	Negative trigger threshold
hysteresis_p	Positive threshold hysteresis coefficient, providing a buffer zone between trigger and release triggers to prevent continuous false triggers
noise_p	Positive noise threshold, related to baseline update
noise_n	Negative noise threshold, related to baseline update
debounce_p	Debounce count for positive threshold to reduce false triggering
debounce_n	Debounce count for negative threshold to reduce false release triggering
reset_p	Positive threshold for trigger baseline reset
reset_n	Negative threshold for trigger baseline reset
gold_value	Gold standard value, used to restore normal values under special circumstances

Then use *touch_proximity_sensor_create()* to configure and create the proximity sensing sensor object.

```
proxi_config_t config = (proxi_config_t)DEFAULTS_PROX_CONFIGS();
esp_err_t ret = touch_proximity_sensor_create(&config, &s_touch_proximity_sensor, &
↪example_proxi_callback, NULL);
if (ret != ESP_OK) {
    ESP_LOGE(TAG, "touch proximity sense create failed");
}
```

Here, *s_touch_proximity_sensor* is the handle of the touch proximity sensor, and *example_proxi_callback* is the proximity sensing sensor event callback function.

Start and Stop the Proximity Sensing Sensor

Use *touch_proximity_sensor_start()* to start the proximity sensing sensor:

```
// Start the touch proximity sensor
touch_proximity_sensor_start(s_touch_proximity_sensor);
```

Use *touch_proximity_sensor_stop()* to stop the proximity sensing sensor:

```
// Stop the touch proximity sensor
touch_proximity_sensor_stop(s_touch_proximity_sensor);
```

Note: Starting and stopping the proximity sensing sensor process takes some time to complete. Therefore, it is necessary to add a waiting time after calling the start and stop APIs. Typically, the startup time is 300 ms, and the stop process requires 200 ms. Please refer to the example program for details.

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(s_touch_proximity_sensor);
```

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.
- Increasing *meas_count* slows down the update rate of new data from the touch sensor.
- *smooth_coef* 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* 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.
- When the difference between *raw* and *baseline* exceeds *baseline * max_p* , the *raw* value is considered an outlier and ignored.
- 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.
- 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.

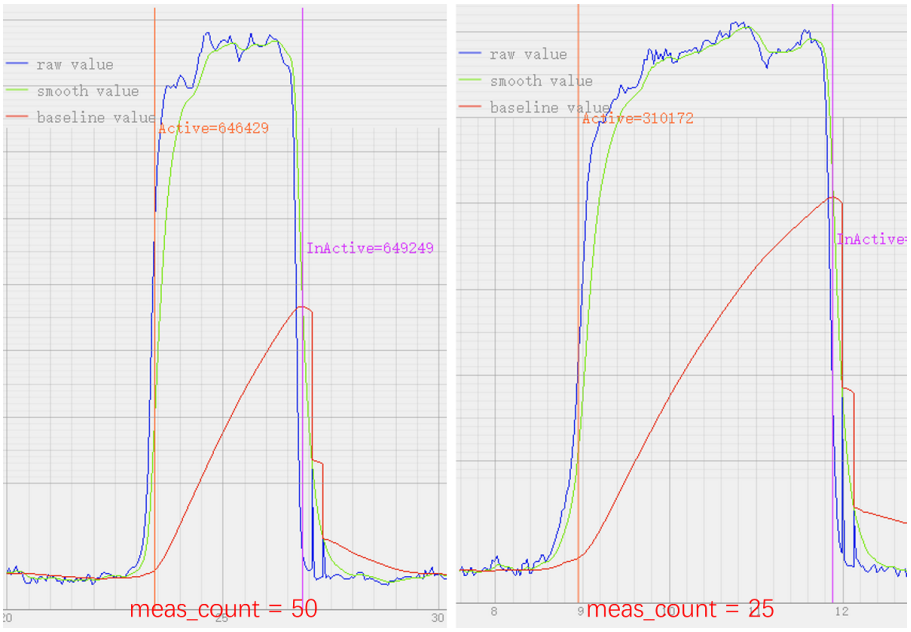
Parameter Adjustment Comparison

The default configuration parameters for the touch proximity sensor are as follows:

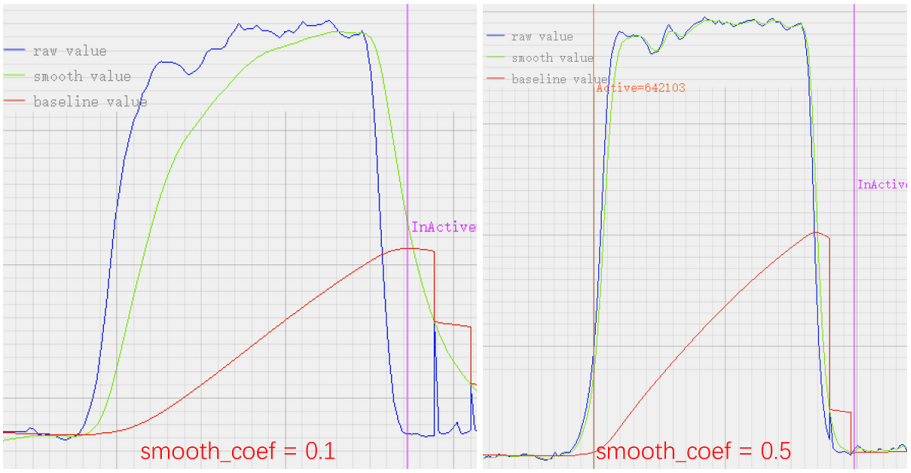
Parameter	Default Value
channel_num	1
channel_list	TOUCH_PAD_NUM8
meas_count	50
smooth_coef	0.2
baseline_coef	0.1
max_p	0.2
min_n	0.08
threshold_p	0.002
threshold_n	0.002
hysteresis_p	0.2
noise_p	0.001
noise_n	0.001
debounce_p	2
debounce_n	1
reset_p	1000
reset_n	3

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:

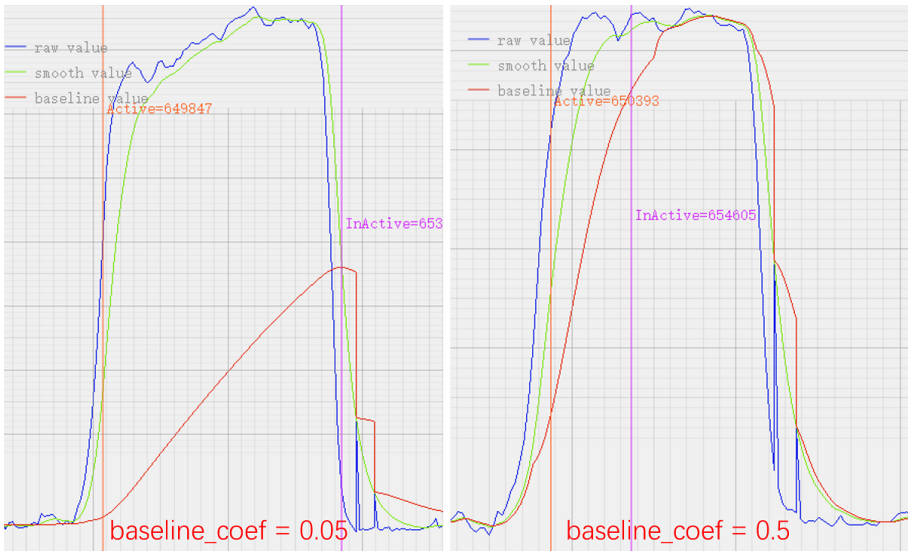


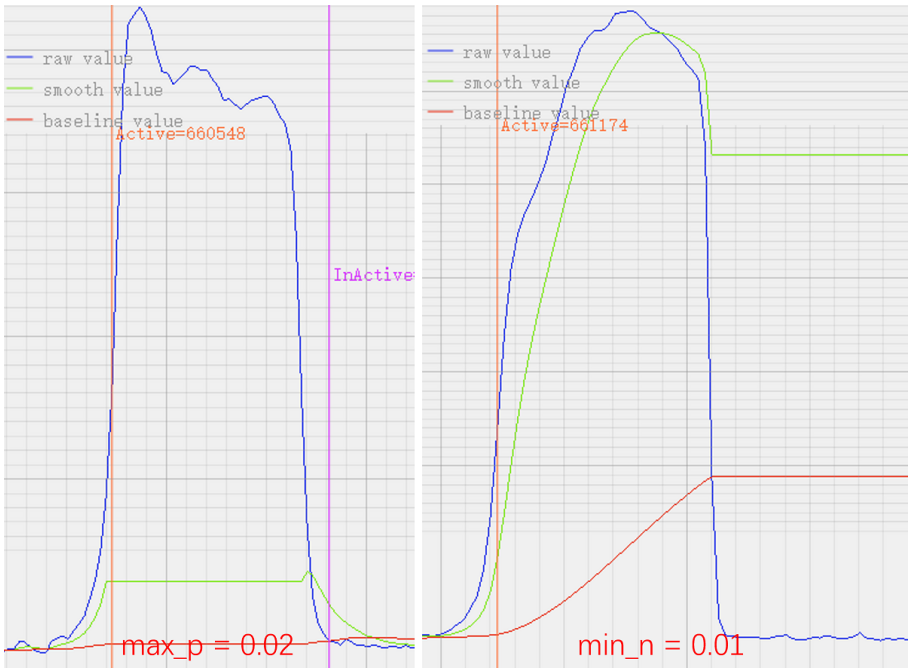
3. Modifying the value of *baseline_coef* changes the updating effect of the *baseline* . Smaller values of *baseline_coef* result in slower response time and longer trigger duration. The waveform comparison under different *baseline_coef* is shown below:

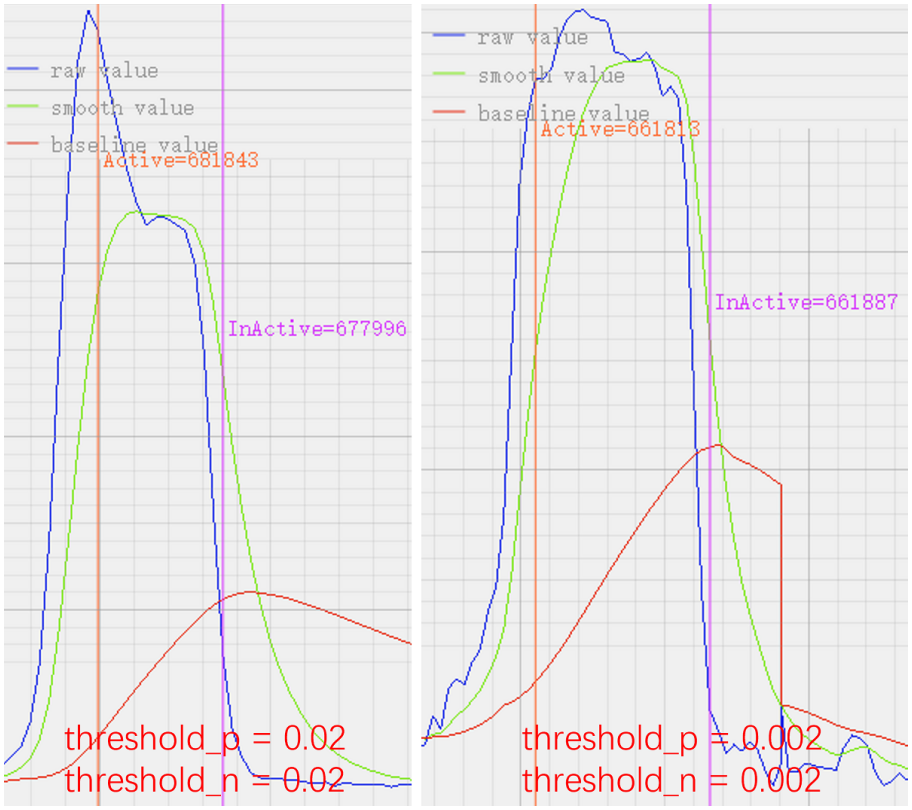
4. Modifying the values of *max_p* and *min_n* will change the update logic of *smooth* and *baseline* . A too small value of *max_p* can cause the *smooth* to be "locked" when the hand approaches the sensing panel, potentially leading to failure to trigger; a too small value of *min_n* can cause both *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:


raw value
smooth value
baseline value
Active=649847
InActive=653
baseline_coef = 0.05
raw value
smooth value
baseline value
Active=650393
InActive=654605
baseline_coef = 0.5


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



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:

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.

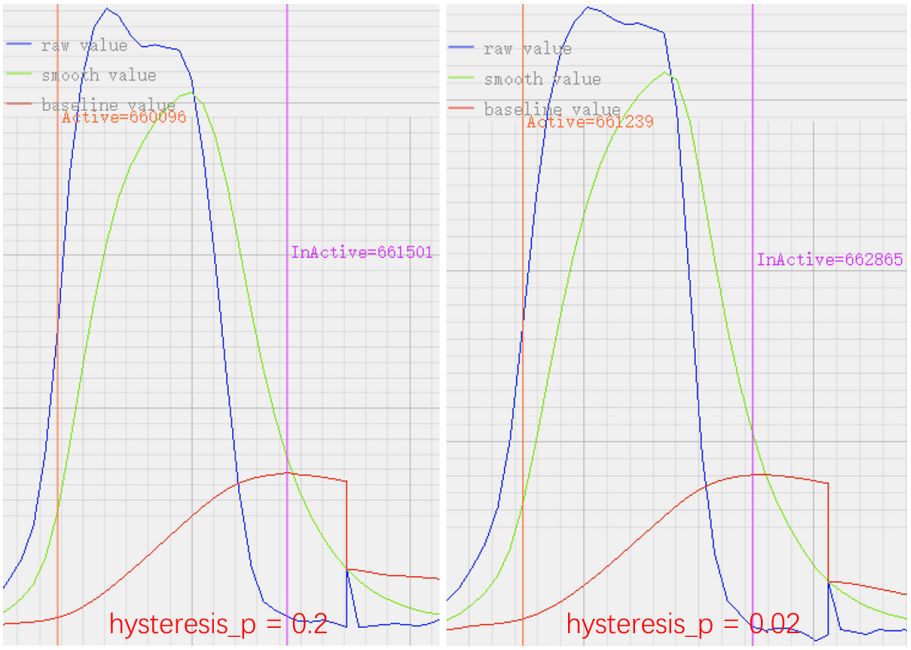
13.1.5 Examples

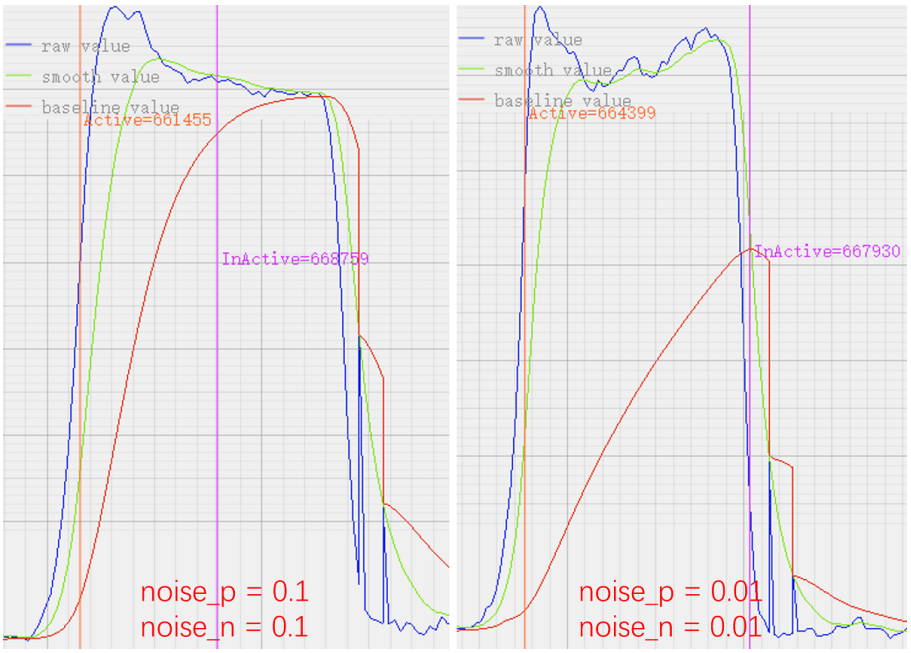
- touch/touch_proximity

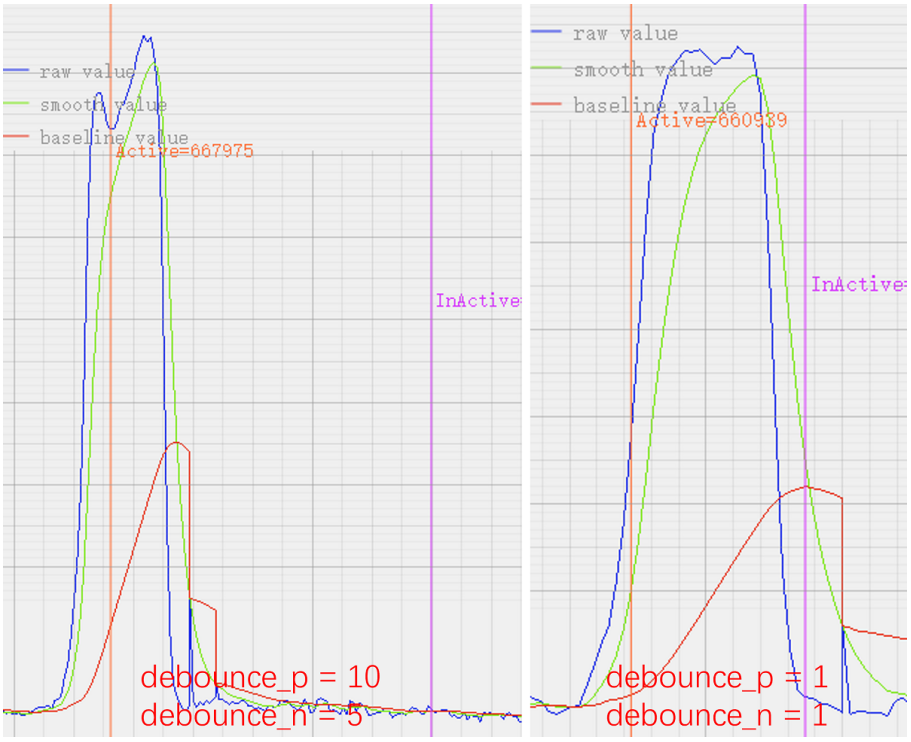
13.1.6 API Reference

Header File

- components/touch/touch_proximity_sensor/include/touch_proximity_sensor.h


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



Functions

esp_err_t **touch_proximity_sensor_create** (*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_start** (*touch_proximity_handle_t* proxi_sensor)

Start the touch proximity sensor.

This function starts the touch proximity sensor operation.

Parameters **proxi_sensor** –Pointer to the handle of the touch proximity sensor instance.

Returns

- ESP_OK: Start the touch proximity sensor successfully
- ESP_ERR_INVALID_ARG: The touch proximity sensor failed to start (touch_proximity_handle_t is NULL, or channel_num is zero).
- ESP_FAIL: The touch proximity sensor failed to start (failed to create queue for touch pad).

esp_err_t **touch_proximity_sensor_stop** (*touch_proximity_handle_t* proxi_sensor)

Stop the touch proximity sensor.

This function stops the operation of the touch proximity sensor associated with the provided sensor handle.

Parameters **proxi_sensor** –Pointer to the handle of the touch proximity sensor instance.

Returns

- ESP_OK: Stop the touch proximity sensor successfully

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

Structures

struct **proxi_config_t**

Configuration structure for touch proximity sensor.

This structure defines the configuration parameters for a touch proximity sensor.

Public Members

uint32_t **channel_num**

Number of touch proximity sensor channels

uint32_t **channel_list**[TOUCH_PROXIMITY_NUM_MAX]

Touch proximity sensor channel list

uint32_t **meas_count**

Accumulated measurement count

float **smooth_coef**

Smoothing coefficient

float **baseline_coef**

Baseline coefficient

float **max_p**

Maximum effective positive change rate

float **min_n**

Minimum effective negative change rate

float **threshold_p**[TOUCH_PROXIMITY_NUM_MAX]

Positive threshold

float **threshold_n**[TOUCH_PROXIMITY_NUM_MAX]

Negative threshold

float **hysteresis_p**

Hysteresis for positive threshold

float **noise_p**

Positive noise threshold

float **noise_n**

Negative noise threshold

uint32_t **debounce_p**

Debounce times for positive threshold

uint32_t **debounce_n**

Debounce times for negative threshold

uint32_t **reset_p**

Baseline reset positive threshold

uint32_t **reset_n**

Baseline reset negative threshold

uint32_t **gold_value**[TOUCH_PROXIMITY_NUM_MAX]

Gold value

Macros

TOUCH_PROXIMITY_NUM_MAX

DEFAULTS_PROX_CONFIGS ( )

Type Definitions

typedef struct touch_proximity_sensor_t ***touch_proximity_handle_t**

typedef void (***proxi_cb_t**)(uint32_t channel, *proxi_evt_t* event, void *cb_arg)

proximity sensor user callback type

Enumerations

enum **proxi_evt_t**

Values:

enumerator **PROXI_EVT_INACTIVE**

enumerator **PROXI_EVT_ACTIVE**

Chapter 14

Storage

14.1 Storage Media

The supported storage media is listed in the following table:

Name	Key features	Application scenario	Size	Transmission	Speed	Driver	Note
SPI Flash	Can be shared with code, no extra cost	Store parameters, text or images	MB	SPI	40/80 MHz 4-line	SPI Flash Driver	
SD Card	Large capacity, pluggable	Store audio or video files	GB	SDIO/SPI	20/40 MHz 1/4-line	SD/SDIO/MMC Driver	1
eMMC	Large capacity, high-speed read/write	Store audio or video files	GB	SDIO	20/40 MHz 1/4/8-line	SD/SDIO/MMC Driver	2
EEPROM	Can address by byte, low cost	Store parameters	MB	I2C	100 ~ 400 KHz	eeprom	

Note:

1. Only SPI mode is supported for ESP32-S2
2. Not supported for ESP32-S2

14.1.1 SPI Flash

By default, the ESP32/ESP32-S/ESP32-C series chips use NOR flash to store and access users' code and data. The flash can be integrated into the module or chip and is typically 4 MB, 8 MB or 16 MB. For ESP-IDF v4.0 and later versions, the SPI flash component not only supports read and write operations to the main flash, but can also connect to an another external flash for data storage.

Flash can be partitioned using the partition table. Based on functions of the partition table, flash can not only be used to store the binary code generated by users, but can also act as a non-volatile storage (NVS) to store application programming parameters. On top of that, specific flash areas can be mounted to a file system (e.g., FatFS) to store text, images and other files.

The flash chip supports 2-line (DOUT/DIO) and 4-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 needs

for additional memory chips, it is particularly suitable for cost-sensitive applications with small capacity requirements (MB) and high integration needs.

Related documents:

- SPI Flash API

14.1.2 SD Card

The ESP32 supports using either the SDIO or SPI interface to access SD cards. The SDIO interface supports 1/4/8-line modes and supports both the default rate of 20 MHz and the high-speed rate of 40 MHz. Please note that this interface occupies at least 6 GPIOs and only uses fixed pins. The SPI interface can assign any IO for SD cards via the GPIO matrix and supports accessing multiple SD cards via CS pins. In hardware design level, the SPI interface is more flexible for development, but with lower access rate than that of the SDIO interface.

The `SD/SDIO/MMC Driver` in ESP-IDF is wrapped at the protocol layer based on the two access modes of the SD card, and provides the initialization interface and protocol-layer APIs for the SD card. The SD card, with features as large capacity and being pluggable, is widely used in application scenarios with large storage needs such as smart speaker, electronic album and etc.

Related documents:

- SD/SDIO/MMC Driver: supports both SDIO and SPI transmission modes;
- SDMMC Host Driver: supports SDIO mode;
- SD SPI Host Driver: supports SPI mode;
- When using SPI or 1-bit modes, please pay special attention to Pull-up Requirements of Pins.

Examples:

- storage/sd_card: access the SD card which uses FAT file system.

14.1.3 eMMC

The eMMC (embedded MMC) memory chip uses the similar protocol to SD cards and can use the same driver SD/SDIO/MMC Driver as SD cards. However, please note that the eMMC chip can only use SDIO mode and does not support SPI mode. Currently, the eMMC chip supports the default rate of 20 MHz and high-speed rate of 40 MHz in 8-line mode, and supports high-speed rate of 40 MHz in 4-line DDR mode.

The eMMC is generally soldered to the main board as a chip, which is more integrated than SD cards, and is suitable for wearable devices and other scenarios with high storage needs and certain requirements for system integration in the meantime.

Related documents:

- SD/SDIO/MMC Driver;
- Supported eMMC Speed Modes.

14.1.4 EEPROM

EEPROM (e.g., AT24C0X series) is a 1024-16384 bits of serial erasable memory, which can also operate in read-only mode by configuring pin levels. Generally, its storage space is distributed by `word`, with each `word` containing `8-bit` spaces. The EEPROM supports byte addressing and is easy to read and write, making it especially suitable for saving configuration parameters and etc. On top of that, it can also be used in industrial and commercial scenarios with requirements for power consumption and reliability after being optimized.

Adapted EEPROM chips:

Name	Function	Bus	Vendor	Datasheet	Driver
AT24C01/02	1024/2048 bits EEPROM	I2C	Atmel	Datasheet	eeprom

14.2 File System

Supported file systems:

Table 1: File system features comparison

Key Features	NVS Library	FAT File System	SPIFFS File System	LittleFS File System
Features	Operates on key-value pairs, with safe interfaces	Operation system supported, strong compatibility	Developed for embedded systems, low resource occupancy	Low resource occupancy, Read, write, erase speed is fast
Application Scenarios	Stores parameters	Stores audio, video and other files	Stores audio, video and other files	Stores files
Size	KB-MB	GB	< 128 MB	< 128 MB
Directory Support	X	√	X	√
Wear Levelling	√	Optional	√	√
R/W Efficiency	0	0	0	High
Resources Occupancy	0	0	1	1
Power Failure Protection	√	X	X	√
Encryption	√	√	X	√

Note:

- 0: data not available or not for comparison.
- 1: low RAM occupancy.

14.2.1 NVS Library

Non-volatile storage (NVS) is used to read and write data stored in the flash NVS partition. NVS operated on key-value pairs. Keys are ASCII strings; values can be integers, strings and variable binary large object (BLOB). NVS supports power loss protection and data encryption, and works best for storing many small values, such as application parameters. If you need to store large blobs or strings, please consider using the facilities provided by the FAT file system on top of the wear levelling library.

Related documents:

- Non-volatile storage library.
- For mass production, you can use the NVS Partition Generator Utility.

Examples:

- Write a single integer value: storage/nvs_rw_value.
- Write a blob: storage/nvs_rw_blob.

14.2.2 FAT File System

ESP-IDF uses the FatFs library to work with FAT file system. FatFs is a file system layer independent to platform and storage media that can realize access to physical devices (e.g., flash, SD card) via a unified interface. Although the library can be used directly, many of its features can be accessed via VFS, using the C standard library and POSIX API functions.

The operating system of FAT is compatible with a wide range of mobile storage devices such as USB memory disc or SD cards. And ESP32 series chips can access these common storage devices by supporting the FAT file system.

Related documents:

- Using FatFs with VFS.
- Using FatFs with VFS and SD cards.

Examples:

- storage/sd_card: access the SD card which uses the FAT file system.
- storage/ext_flash_fatfs: access the external flash chip which uses the FAT file system.

14.2.3 SPIFFS File System

SPIFFS is a file system intended for SPI NOR flash devices on embedded targets. It supports wear levelling, file system consistency checks, and more. Users can directly use the Posix interfaces provided by SPIFFS, or use many of its features via VFS.

As a dedicated file system for SPI NOR flash devices on embedded targets, the SPIFFS occupies less RAM resources than FAT and is only used to support flash chips with capacities less than 128 MB.

Related documents:

- SPIFFS Filesystem.
- Two Tools to Generate SPIFFS Images.

Examples:

- storage/spiffs: SPIFFS examples.

14.2.4 LittleFS File System

LittleFS is a file system intended for SPI NOR flash devices on embedded targets. It supports wear levelling, file system consistency checks, power failure protection and more.

LittleFS as a high integrity embedded SPI NOR Flash file system, it supports efficient read and write speed and occupies less RAM resources.

Related documents:

- LittleFS Filesystem Component SDK .
- LittleFS Filesystem Component Usage Guide .

Examples:

- storage/littlefs : LittleFS example.

14.2.5 Virtual File System (VFS)

The Virtual File System (VFS) component from ESP-IDF provides a unified interface for different file systems (FAT, SPIFFS), and also provides a file-like interface for device drivers.

Related documents:

- Virtual Filesystem Component.

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.

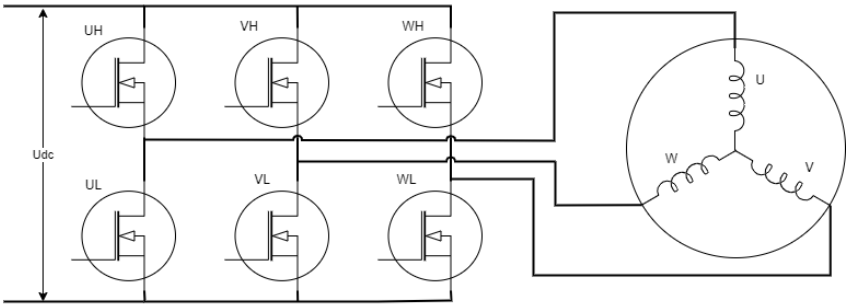


Fig. 2: BLDC Three-phase Inverter Circuit

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.

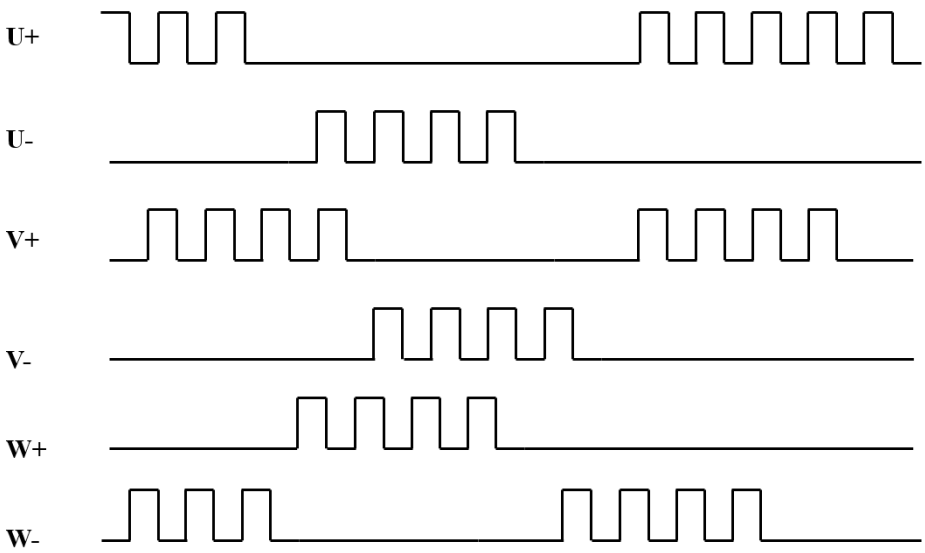


Fig. 3: BLDC PWM Speed Control

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.

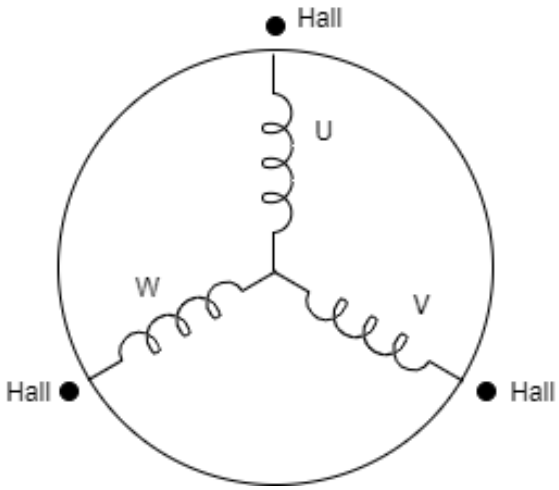


Fig. 4: BLDC Hall Sensor Installation

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.

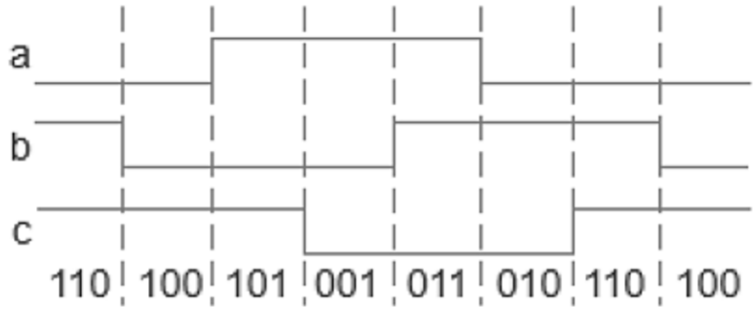


Fig. 5: BLDC Hall Sensor Waveform

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

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.

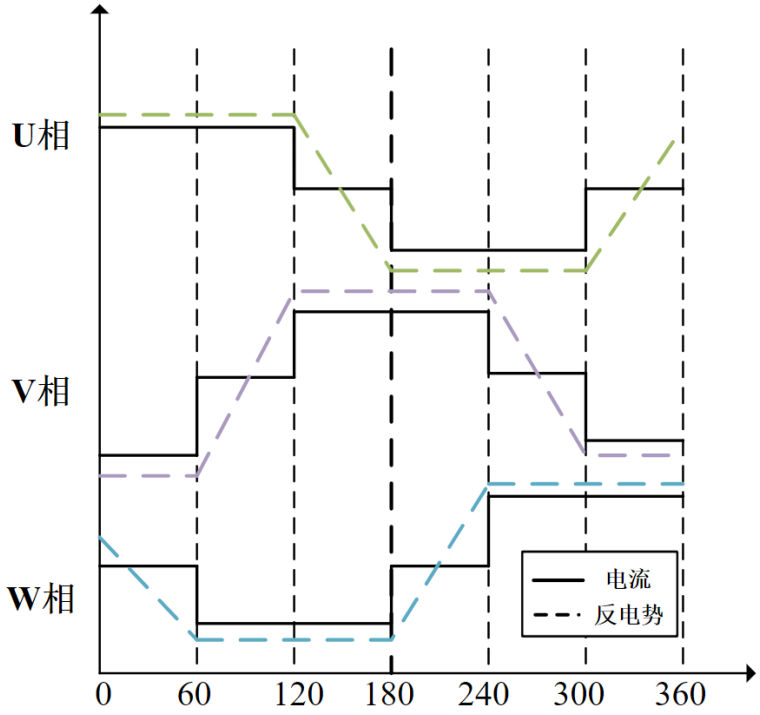


Fig. 6: Current and Back EMF Waveforms

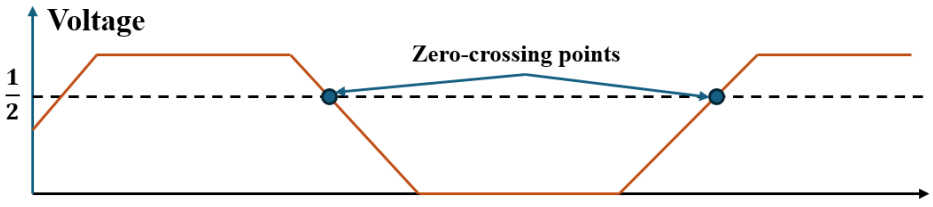


Fig. 7: Implementation of ADC Zero-crossing Detection

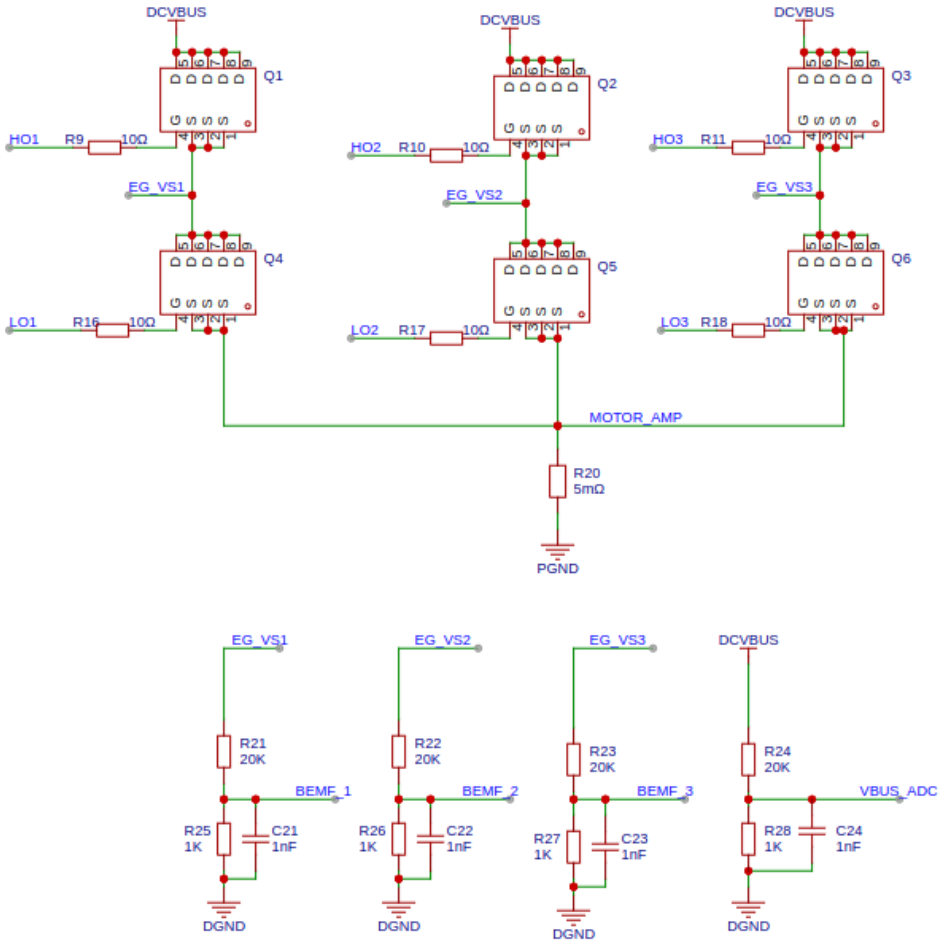


Fig. 8: ADC Zero-crossing Detection Hardware

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 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.

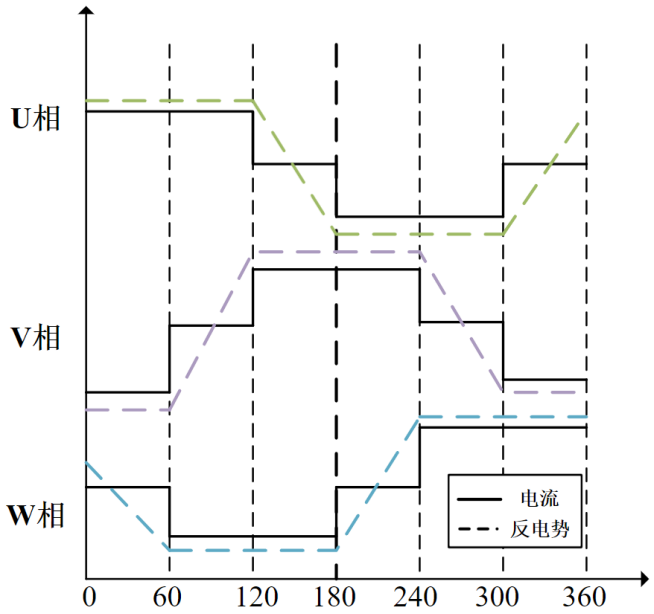


Fig. 9: Current and Back EMF Waveforms

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.

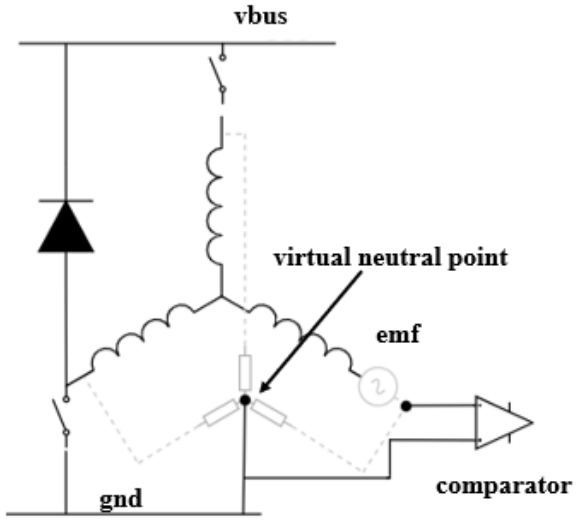


Fig. 10: Comparator Zero-crossing Principle

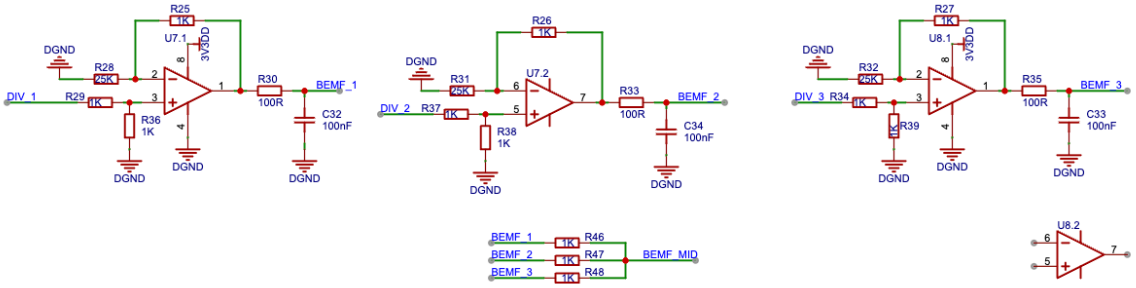


Fig. 11: Comparator Zero-crossing Hardware

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

- *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:

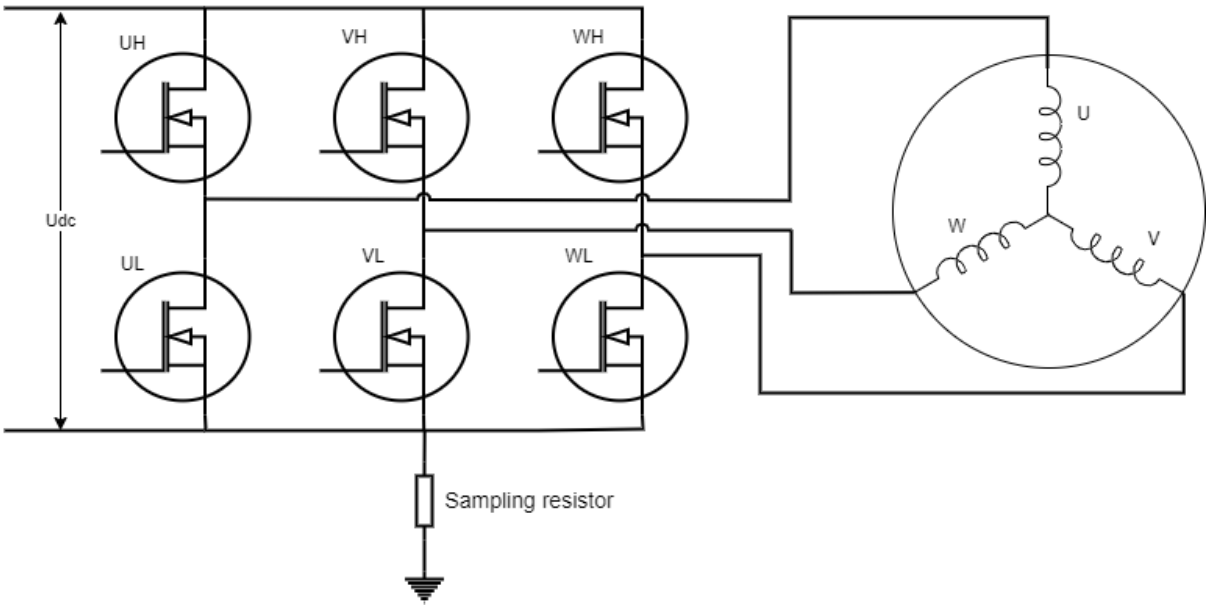


Fig. 12: BLDC Bus Current Collection

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.

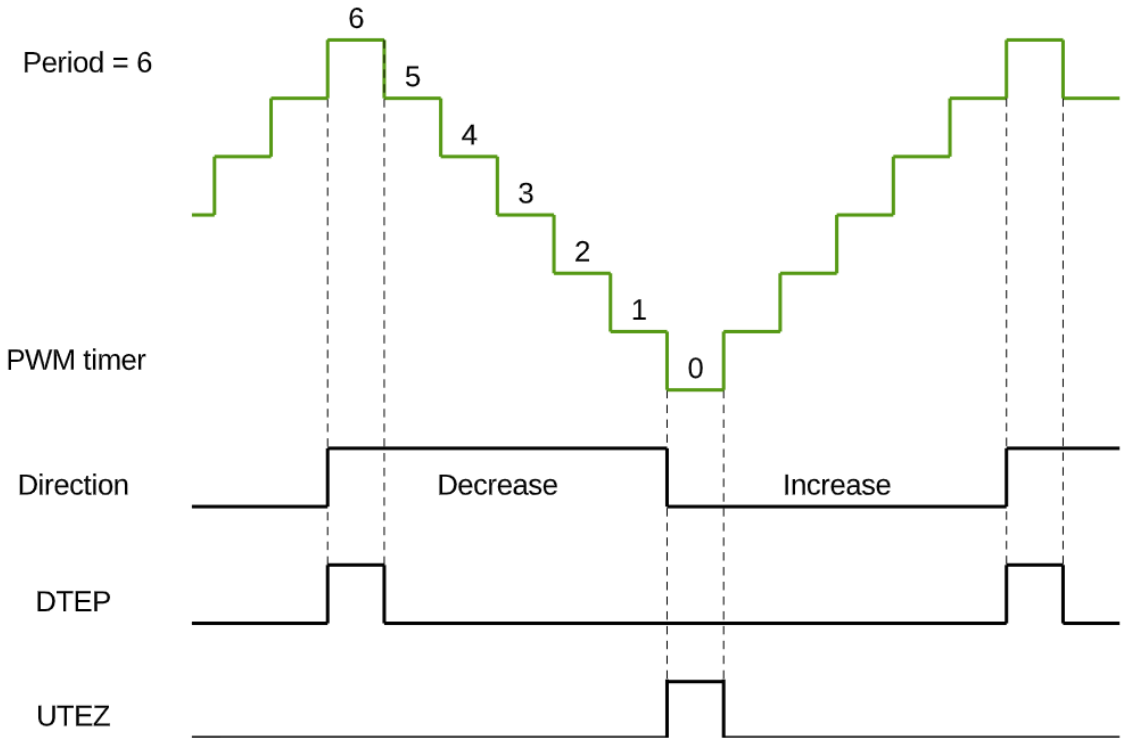


Fig. 13: MCPWM Rising and Falling Mode

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

Security & Encryption

16.1 Flash 加密

16.1.1 概述

- 使能 flash encryption 后，使用物理手段（如串口）从 SPI flash 中读取的数据都是经过加密的，大部分数据无法恢复出真实数据。
- flash encryption 使用 256-bit AES key 加密 flash 数据，key 保存在芯片的 efuse 中，生成之后变成软件读写保护。
- 用户烧写 flash 时烧写的是数据明文，第一次 boot 时，软件 bootloader 会对 flash 中的数据在原处加密。
- 一般使用情况下一共有 4 次机会通过串口烧写 flash，通过 OTA 更新 flash 数据没有次数限制。开发阶段可以在 menuconfig 中设置无烧写次数限制，但不要在产品中这么做。

16.1.2 使用步骤

1. make menuconfig 中选择“Security features”->”Enable flash encryption on boot”
2. 按通常操作编译出 bootloader, partition table 和 app image 并烧写到 flash 中
3. 第一次 boot 时 flash 中被指定加密的数据被加密（大的 partition 加密过程可能需要花费超过 1 分钟），之后就可以正常使用被加密的 flash 数据。

16.1.3 加密过程（第一次 boot 时进行）

1. bootloader 读取到 efuse 中的 FLASH_CRYPT_CNT 为 0，于是利用硬件随机数生成器产生加密用的 key，此 key 被保存在 efuse 中，对于软件是读写保护的。
2. bootloader 对所有需要被加密的 partition 在 flash 中原处加密
3. 默认情况下 efuse 中的 DISABLE_DL_ENCRYPT, DISABLE_DL_DECRYPT 和 DISABLE_DL_CACHE 会被烧写为 1，这样 UART bootloader 时就不能读取到解密后的 flash 数据
4. efuse 中的 FLASH_CRYPT_CONFIG 被烧写成 0xf，此标志用于决定加密 key 的多少位被用于计算每一个 flash 块（32 字节）对应的秘钥，设置为 0xf 时使用所有 256 位
5. efuse 中的 FLASH_CRYPT_CNT 被烧写成 0x01，此标志用于 flash 烧写次数限制以及加密控制，详见“FLASH_CRYPT_CNT”一节
6. bootloader 将自己重启，从加密的 flash 执行软件 bootloader

16.1.4 串口重烧 flash （3 次重烧机会）

- 串口重烧 flash 过程
  1. make menuconfig 中选择 “Security features” ->” Enable flash encryption on boot”
  2. 编译工程，将所有之前加密的 images （包括 bootloader）烧写到 flash 中。
  3. 在 esp-idf 的 components/esptool_py/esptool 路径下使用命令 `espefuse.py burn\_efuse FLASH\_CRYPT\_CNT` 烧写 efuse 中的 FLASH_CRYPT_CNT
  4. 重启设备，bootloader 根据 FLASH_CRYPT_CNT 的值重新加密 flash 数据。
- 若用户确定不再需要通过串口重烧 flash，可以在 esp-idf 的 `components/esptool\_py/esptool` 路径下使用命令 `espefuse.py --port PORT write\_protect\_efuse FLASH\_CRYPT\_CNT` 将 `FLASH\_CRYPT\_CNT` 设置为读写保护（注意此步骤必须在 bootloader 已经完成对 flash 加密后进行）

16.1.5 FLASH_CRYPT_CNT

- FLASH_CRYPT_CNT 是 flash 加密方案中非常重要的控制标志，它是 8-bit 的值，它的值一方面决定 flash 中的值是否马上需要加密，另一方面控制 flash 烧写次数限制。
- 当 FLASH_CRYPT_CNT 有（0,2,4,6,8）位被烧写为 1 时，bootloader 会对 flash 中的内容进行加密。
- 当 FLASH_CRYPT_CNT 有（1,3,5,7）位被烧写为 1 时，bootloader 知道 flash 的内容已经过加密，直接读取 flash 中的数据解密后使用。
- FLASH_CRYPT_CNT 的变化过程：
  1. 没有使能 flash 加密时，永远是 0
  2. 使能了 flash 加密，在第一次 boot 时 bootloader 发现它的值是 0x00，于是知道 flash 中的数据还未加密，利用硬件随机数生成器产生 key，然后加密 flash，最后将它的最低位置 1（取值为 0x01）
  3. 后续 boot 时，bootloader 发现它的值是 0x01，知道 flash 中的数据已加密，可以解密后直接使用
  4. 用户需要串口重烧 flash ，于是使用命令行手动烧写 FLASH_CRYPT_CNT，此时 2 个 bit 被置为 1（取值为 0x03）
  5. 重启设备，bootloader 发现 FLASH_CRYPT_CNT 的值是 0x03（2 bit 1），于是重新加密 flash 数据，加密完成后 bootloader 将 FLASH_CRYPT_CNT 烧写为 0x07（3 bit 1），flash 加密正常使用
  6. 用户需要串口重烧 flash ，于是使用命令行手动烧写 FLASH_CRYPT_CNT，此时 4 个 bit 被置为 1（取值为 0x0f）
  7. 重启设备，bootloader 发现 FLASH_CRYPT_CNT 的值是 0x0f（4 bit 1），于是重新加密 flash 数据，加密完成后 bootloader 将 FLASH_CRYPT_CNT 烧写为 0x1f（5 bit 1），flash 加密正常使用
  8. 用户需要串口重烧 flash ，于是使用命令行手动烧写 FLASH_CRYPT_CNT，此时 6 个 bit 被置为 1（取值为 0x3f）
  9. 重启设备，bootloader 发现 FLASH_CRYPT_CNT 的值是 0x4f（6 bit 1），于是重新加密 flash 数据，加密完成后 bootloader 将 FLASH_CRYPT_CNT 烧写为 0x7f（7 bit 1），flash 加密正常使用
  10. 注意！此时不能再使用命令行烧写 FLASH_CRYPT_CNT，bootloader 读到 FLASH_CRYPT_CNT 为 0xff（8 bit 1）时，会停止后续的 boot。

16.1.6 被加密的数据

- Bootloader
- Secure boot bootloader digest（若 Secure Boot 被使能，flash 中会多出这一项，具体查看 “Secure Boot” 中 “执行过程” 的步骤 3）
- Partition table
- Partition table 中指向的所有 Type 域标记为 “app” 的部分
- Partition table 中指向的所有 Flags 域标记为 “encrypted” 的部分（用于非易失性存储（NVS）部分的 flash 在任何情况下都不会被加密）

16.1.7 哪些方式读到解密后的数据（真实数据）

- 通过内存管理单元的 flash 缓存读取的 flash 数据都是经过解密后的数据，包括：
  - flash 中的可执行应用程序代码

  - 存储在 flash 中的只读数据
  - 任何通过 API esp_spi_flash_mmap() 读取的数据
  - 由 ROM bootloader 读取的软件 bootloader image 数据
- 如果调用 API esp_partition_read() 读取被加密区域的数据，则读取的 flash 数据是经过解密后的数据

16.1.8 哪些方式读到不解密的数据（无法使用的脏数据）

- 通过 API esp_spi_flash_read() 读取的数据
- ROM 中的函数 SPIRead() 读取的数据

16.1.9 软件写入加密数据

- 调用 API esp_partition_write() 时，只有写到被加密的 partition 的数据才会被加密
- 函数 esp_spi_flash_write() 根据参数 write_encrypted 是否被设为 true 决定是否对数据加密
- ROM 函数 esp_rom_spiflash_write_encrypted() 将加密后的数据写入 flash 中，而 SPIWrite() 将不加密的数据写入到 flash 中

16.2 安全启动

16.2.1 概述

- Secure Boot 的目的是保证芯片只运行用户指定的程序，芯片每次启动时都会验证从 flash 中加载的 partition table 和 app images 是否是用户指定的
- Secure Boot 中采用 ECDSA 签名算法对 partition table 和 app images 进行签名和验证，ECDSA 签名算法使用公钥/私钥对，秘钥用于对指定的二进制文件签名，公钥用于验证签名
- 由于 partition table 和 app images 是在软件 bootloader 中被验证的，所以为了防止攻击者篡改软件 bootloader 从而跳过签名验证，Secure Boot 过程中会在 ROM bootloader 时检查软件 bootloader image 是否被篡改，检查用到的 secure boot key 由硬件随机数生成器产生，保存的 efuse 中，对于软件是读写保护的

16.2.2 所用资源

- ECDSA 算法公钥/私钥对
  - 烧写 flash 前在 PC 端生成
  - 公钥会被编译到 bootloader image 中，软件 bootloader 在执行时会读取公钥，使用公钥验证 flash 中 partition table 和 app images 是否是经过相应的私钥签名的
  - 私钥在编译时被用于对 partition table 和 app images 签名，私钥必须被保密好，一旦泄露任何使用此私钥签名的 image 都能通过 boot 时的签名验证
- secure bootloader key
  - 这是一个 256-bit AES key，在第一次 Secure Boot 时由硬件随机数生成，保存在 efuse 中，软件无法读取
  - 使用此 key 验证软件 bootloader image 是否被修改

16.2.3 执行过程

1. 编译 bootloader image 时发现 menuconfig 中使能了 secure boot，于是根据 menuconfig 中指定的公钥/秘钥文件路径将公钥编译到 bootloader image 中，bootloader 被编译成支持 secure boot
2. 编译 partition table 和 app images 时使用秘钥计算出签名，将签名编译到相应的二进制文件中
3. 芯片第一次 boot 时，软件 bootloader 根据一下步骤使能 secure boot：
   - 硬件产生一个 secure boot key，将这个 key 保存在 efuse 中，利用这个 key、一个随机数 IV 和 bootloader image 计算出 secure digest

- secure digest 与随机数 IV 保存在 flash 的 0x0 地址，用于在后续 boot 时验证 bootloader image 是否被篡改
- 若 menuconfig 中选择了禁止 JTAG 中断和 ROM BASIC 中断，bootloader 会将 efuse 中的一些标志位设置为禁止这些中断（强烈建议禁止这些中断）
- bootloader 通过烧写 efuse 中的 ABS_DONE_0 永久使能 secure boot

4. 芯片在后面的 boot 中，ROM bootloader 发现 efuse 中的 ABS_DONE_0 被烧写，于是从 flash 的地址 0x0 读取第一次 boot 时保存的 secure digest 和随机数 IV，硬件使用 efuse 中的 secure boot key 、随机数 IV 与当前的 bootloader image 计算当前的 secure digest，若与 flash 中的 secure digest 不同，则 boot 不会继续，否则就执行软件 bootloader。
5. 软件 bootloader 使用 bootloader image 中保存的公钥对 flash 中的 partition table 和 app images 签字进行验证，验证成功之后才会 boot 到 app 代码中

16.2.4 使用步骤

1. make menuconfig 选择“enable secure boot in bootloader”
2. make menuconfig 设置保存公钥/秘钥对的文件
3. 生成公钥和秘钥，先执行“make”命令，此时由于还没有公钥/秘钥对，所以命令行中会提示生成公钥/秘钥对的命令，按提示执行命令即可。但在产品级使用中，建议使用 openssl 或者其他工业级加密程序生成公钥/秘钥对。例如使用 openssl：“openssl ecparam -name prime256v1 -genkey -noout -out my_secure_boot_signing_key.pem”（若使用现有的公钥/秘钥对文件，可以跳过此步）
4. 运行命令“make bootloader”产生一个使能 secure boot 的 bootloader image
5. 执行完 4 后命令行会提示下一步烧写 bootloader image 的命令，按提示烧写即可
6. 运行命令“make flash”编译并烧写 partition table 和 app images
7. 重启芯片，软件 bootloader 会使能 secure boot ，查看串口打印确保 secure boot 成功启用。

16.2.5 注意事项

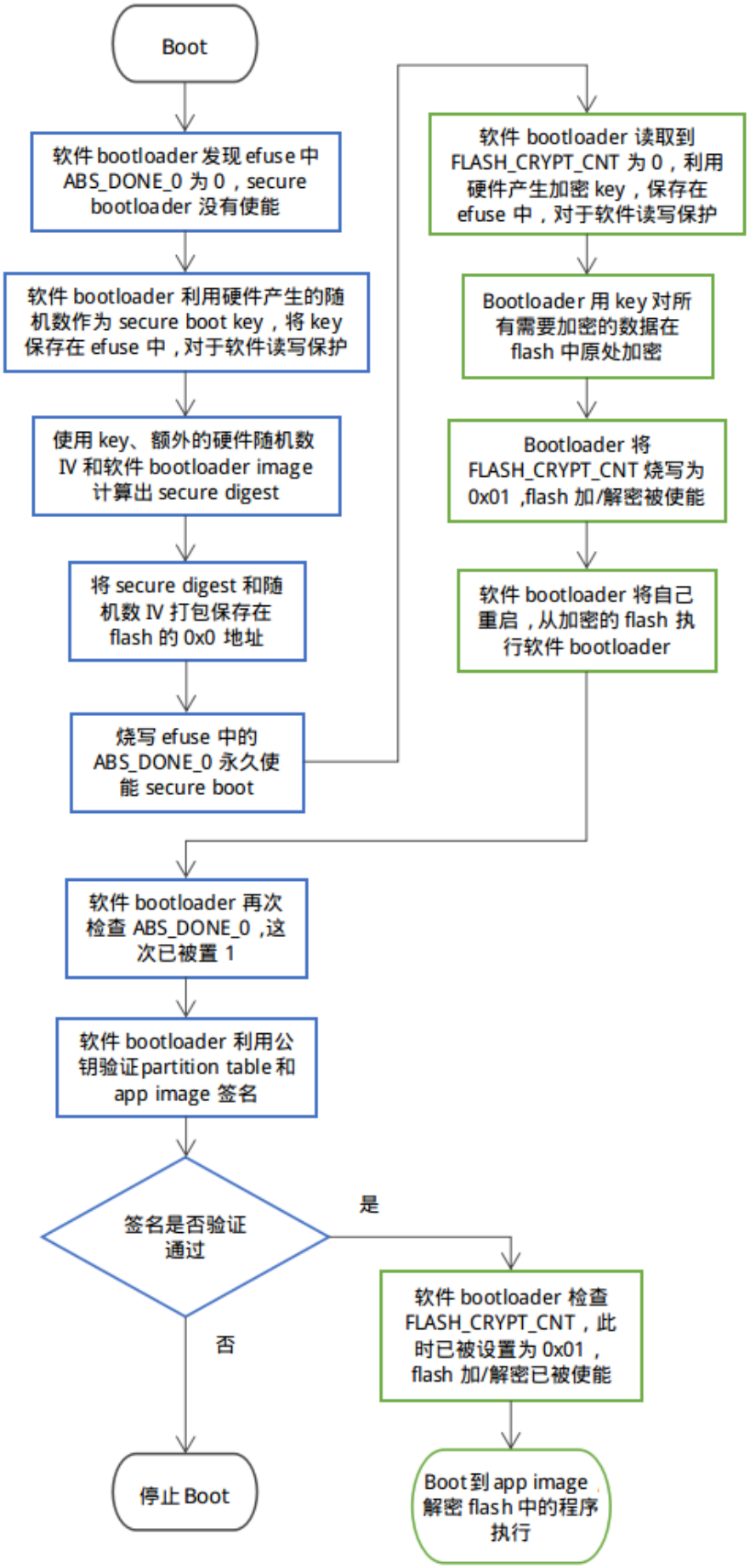
- 正常使用情况下，bootloader image 只能烧写一次，partition table 和 app images 可以重复烧写
- 秘钥必须保密，一旦泄露 secure boot 将失去作用
- 用于 OTA 的 image 必须进行秘钥签名，OTA 时会使用公钥进行验证
- **在默认设置下，bootloader 的从 0x1000 地址开始，最大长度为 28KB（bootloader）。如果发现 Secure Boot 发送错误，请先检查是否因为 Bootloader 地址过大。通过在 menuconfig 中调整 Bootloader 的 log 等级，可以有效降低编译后的 Bootloader 大小。**

16.2.6 可重复烧写 bootloader

- 默认情况下 bootloader image 只能烧写一次，在产品中强烈建议这样做，因为 bootloader image 可以重新烧写的情况下可以通过修改 bootloader 跳过后续 image 的验证过程，这样 secure boot 就失去作用
- 可重复烧写 bootloader 模式下，secure bootloader key 是在 PC 端产生的，此 key 必须保密，一旦 key 被泄露，其它使用此 key 生成 digest 的 bootloader image 也能通过硬件检查
- 使用步骤：
  1. make menuconfig 中选择“secure bootloader mode”->”Reflashable”
  2. 按“使用步骤”一节步骤 2 和 3 生成公钥与秘钥
  3. 运行指令“make bootloader”，一个 256-bit secure boot key 会根据用于签名的私钥计算出，命令行会打印两个后续步骤，按循序执行：
     - 将 PC 端生成的 secure boot key 烧入 efuse 中的命令
     - 将编译好的带有预计算出的 secure digest 的 bootloader image 烧写到 flash 中
  4. 从“使用步骤”一节的步骤 6 继续执行

16.2.7 Secure Boot 与 Flash Encryption 流程图

- 第一次 boot 时 secure boot 与 flash encrypt 的生效过程如下图所示，图中蓝色框是 secure boot 的步骤，绿色框是 flash encrypt 的步骤


Boot
软件 bootloader 发现 efuse 中 ABS_DONE_0 为 0，secure bootloader 没有使能
软件 bootloader 利用硬件产生的随机数作为 secure boot key，将 key 保存在 efuse 中，对于软件读写保护
使用 key、额外的硬件随机数 IV 和软件 bootloader image 计算出 secure digest
将 secure digest 和随机数 IV 打包保存在 flash 的 0x0 地址
烧写 efuse 中的 ABS_DONE_0 永久使能 secure boot
软件 bootloader 读取到 FLASH_CRYPT_CNT 为 0，利用硬件产生加密 key，保存在 efuse 中，对于软件读写保护
Bootloader 用 key 对所有需要加密的数据在 flash 中原处加密
Bootloader 将 FLASH_CRYPT_CNT 烧写为 0x01 ,flash 加/解密被使能
软件 bootloader 将自己重启，从加密的 flash 执行软件 bootloader
软件 bootloader 再次检查 ABS_DONE_0 ,这次已被置 1
软件 bootloader 利用公钥验证partition table 和 app image 签名
签名是否验证通过
是
否
软件 bootloader 检查 FLASH_CRYPT_CNT，此时已被设置为 0x01，flash 加/解密已被使能
停止 Boot
Boot到app image 解密 flash 中的程序执行

- 后续 boot 时流程图如下，图中绿色框中的步骤会执行解密，解密是由硬件自动完成的

16.2.8 开发阶段使用可重复烧写 flash 的 Secure Boot 与 Flash encryption

1. make menuconfig 中使能 secure boot 和 flash encrypt，“Secure bootloader mode” 选择 “Reflashable”，并设置你的公钥/私钥.pem 文件路径
2. 编译 bootloader 并生成 secure boot key：

```
make bootloader
```

3. 使用 key 和 bootloader 计算带 digest 的 bootloader

```
python $IDF_PATH/components/esptool_py/esptool/espsecure.py digest_secure_
↪bootloader --keyfile ./build/bootloader/secure_boot_key.bin -o ./build/
↪bootloader/bootloader_with_digest.bin ./build/bootloader/bootloader.bin
```

4. 编译 partition_table 与 app

```
make partition_table
make app
```

5. 加密三个 bin 文件

```
python $IDF_PATH/components/esptool_py/esptool/espsecure.py encrypt_flash_data␣
↪--keyfile flash_encrypt_key.bin --address 0x0 -o build/bootloader/bootloader_
↪digest_encrypt.bin build/bootloader/bootloader_with_digest.bi
python $IDF_PATH/components/esptool_py/esptool/espsecure.py encrypt_flash_data␣
↪--keyfile flash_encrypt_key.bin --address 0x8000 -o build/partitions_
↪singleapp_encrypt.bin build/partitions_singleapp.bin
python $IDF_PATH/components/esptool_py/esptool/espsecure.py encrypt_flash_data␣
↪--keyfile flash_encrypt_key.bin --address 0x10000 -o build/iot_encrypt.bin␣
↪build/iot.bin
```

6. 烧写三个加密后的 bin 文件

```
python $IDF_PATH/components/esptool_py/esptool/esptool.py --baud 1152000 write_
↪flash 0x0 build/bootloader/bootloader_digest_encrypt.bin
python $IDF_PATH/components/esptool_py/esptool/esptool.py --baud 1152000 write_
↪flash 0x8000 build/partitions_singleapp_encrypt.bin
python $IDF_PATH/components/esptool_py/esptool/esptool.py --baud 1152000 write_
↪flash 0x10000 build/iot_encrypt.bin
```

7. 将 flash_encryption_key 烧入 efuse (仅在第一次 boot 前烧写):

```
python $IDF_PATH/components/esptool_py/esptool/espefuse.py burn_key flash_
↪encryption flash_encrypt_key.bin
```

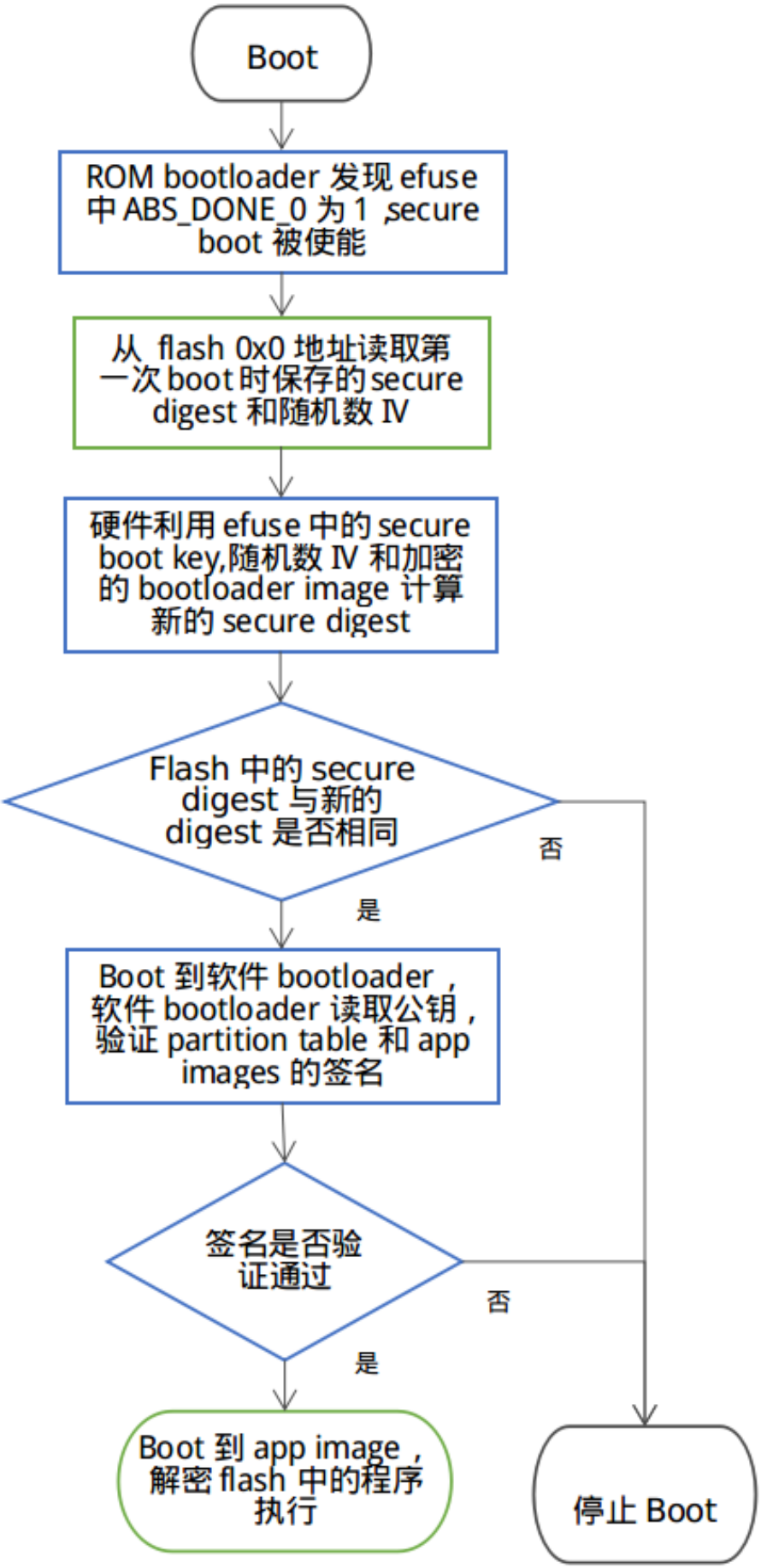
8. 将 secure boot key 烧入 efuse（仅在第一次 boot 前烧写）：

```
python $IDF_PATH/components/esptool_py/esptool/espefuse.py burn_key secure_
↪boot ./build/bootloader/secure_boot_key.bin
```

9. 烧写 efuse 中的控制标志（仅在第一次 boot 前烧写）

```
python $IDF_PATH/components/esptool_py/esptool/espefuse.py burn_efuse ABS_DONE_
↪0
python $IDF_PATH/components/esptool_py/esptool/espefuse.py burn_efuse FLASH_
↪CRYPT_CNT
python $IDF_PATH/components/esptool_py/esptool/espefuse.py burn_efuse FLASH_
↪CRYPT_CONFIG 0xf
python $IDF_PATH/components/esptool_py/esptool/espefuse.py burn_efuse DISABLE_
↪DL_ENCRYPT
python $IDF_PATH/components/esptool_py/esptool/espefuse.py burn_efuse DISABLE_
↪DL_DECRYPT
python $IDF_PATH/components/esptool_py/esptool/espefuse.py burn_efuse DISABLE_
↪DL_CACHE
```

(continues on next page)


Boot
ROM bootloader 发现 efuse 中 ABS_DONE_0 为 1，secure boot 被使能
从 flash 0x0 地址读取第一次 boot 时保存的 secure digest 和随机数 IV
硬件利用 efuse 中的 secure boot key,随机数 IV 和加密的 bootloader image 计算新的 secure digest
Flash 中的 secure digest 与新的 digest 是否相同
否
是
Boot 到软件 bootloader，软件 bootloader 读取公钥，验证 partition table 和 app images 的签名
签名是否验证通过
否
是
Boot 到 app image，解密 flash 中的程序执行
停止 Boot

(continued from previous page)

16.3 启用安全加密的生产方案

16.3.1 Windows 平台的下载工具

- 乐鑫提供 windows 平台的下载工具，能够在工厂生产环境中批量烧写固件
- 生产下载工具的配置文件在 configure 文件夹内，涉及安全特性的配置在 security.conf 中，目前涉及的配置内容如下表：

ITEM	Function	default
debug_enable	是否开启 debug 模式，在 debug 模式下，工具会根据 pem 文件产生相同密钥，否则随机生成密钥	True
debug_pem_path	设置证书地址，用于生成可重复烧写的密钥，尽在 debug 模式下有效	
SECURE BOOT		
secure_boot_en	开启 secure boot 功能	False
burn_secure_boot_key	使能 secure boot key 烧写	False
secure_boot_force_write	是否不检查 secure boot key block，强制烧写 key	False
secure_boot_rw_protect	开启 secure boot key 区域的读写保护	False
FLASH ENCRYPTION		
flash_encryption_en	开启 flash 加密功能	False
burn_flash_encryption_key	使能 flash encrypt key 烧写	False
flash_encrypt_force_write	是否不检查 flash encrypt key block，强制烧写 key	False
flash_encrypt_rw_protect	开启 flash encrypt key 区域的读写保护	False
AES KEY	Not used yet	
DISABLE FUNC		
jtag_disable	是否关闭 JTAG 调试功能	False
dl_encrypt_disable	是否关闭下载模式下 flash 加密功能	False
dl_decrypt_disable	是否关闭下载模式下 flash 解密功能	False
dl_cache_disable	是否关闭下载模式下的 flash cache 功能	False

- 下载工具的内部逻辑和流程如下：

16.3.2 操作步骤

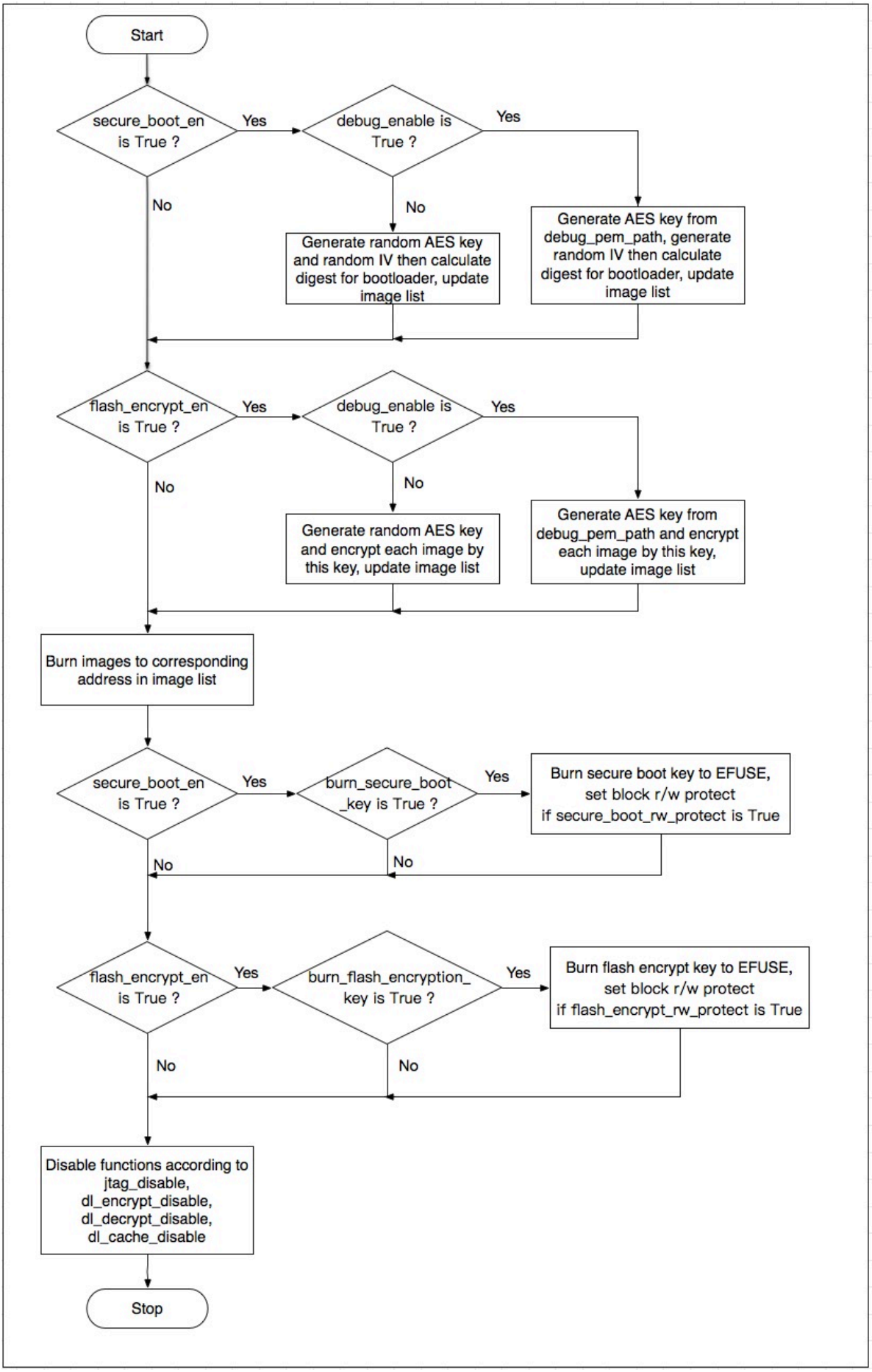
准备工作

- 安装 eptool
  - esptool 默认路径在 $IDF_PATH/components/esptool_py/esptool/
  - 也可以通过 python 安装：

```
pip install esptool
或者
pip3 install esptool
```

方案 1: 通过 bootloader 完成 security 特性初始化

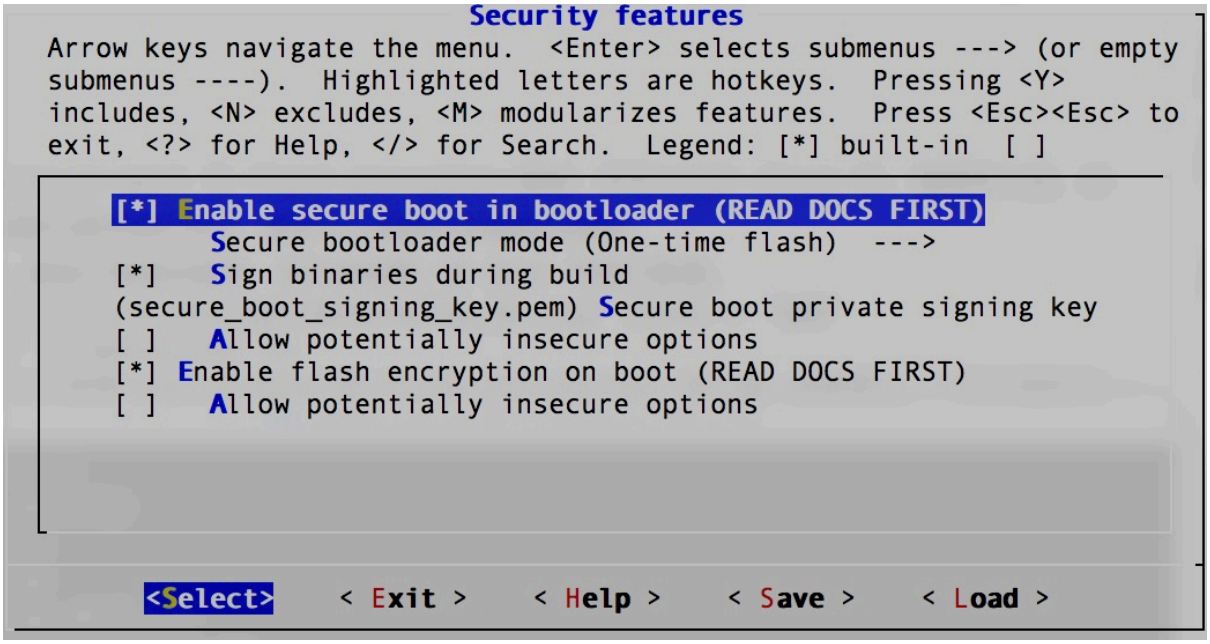
- 优势: 可以批量进行 flash 烧录，初始化的固件相同，密钥在第一次上电有在设备内随机生成。


Start
secure_boot_en is True ?
Yes
debug_enable is True ?
Yes
No
No
Generate random AES key and random IV then calculate digest for bootloader, update image list
Generate AES key from debug_pem_path, generate random IV then calculate digest for bootloader, update image list
flash_encrypt_en is True ?
Yes
debug_enable is True ?
Yes
No
No
Generate random AES key and encrypt each image by this key, update image list
Generate AES key from debug_pem_path and encrypt each image by this key, update image list
Burn images to corresponding address in image list
secure_boot_en is True ?
Yes
burn_secure_boot _key is True ?
Yes
Burn secure boot key to EFUSE, set block r/w protect if secure_boot_rw_protect is True
No
No
flash_encrypt_en is True ?
Yes
burn_flash_encryption_ key is True ?
Yes
Burn flash encrypt key to EFUSE, set block r/w protect if flash_encrypt_rw_protect is True
No
No
Disable functions according to jtag_disable, dl_encrypt_disable, dl_decrypt_disable, dl_cache_disable
Stop

- 缺陷: 设备在首次初始化过程所用时间较长，如果在首次初始化过程发生掉电等意外情况，设备可能无法正常启动。
- 由芯片端自动随机生成 secure boot 与 flash encrypton 密钥，并写入芯片 efuse 中, 密钥写入后，对应的 efuse block 会被设置为读写保护状态，软件与工具都无法读取出密钥。
- 所有编译出的 images 都按正常情况烧写，芯片会在第一次 boot 时进行配置。
- 通过 make menuconfig 配置 secure boot 和 flash encryption，按照第一、二节介绍的步骤执行即可，具体操作步骤如下，如果了解第一、二节的内容，可以跳过:
  1. 随机生成 RSA 密钥文件:

```
espsecure.py generate_signing_key secure_boot_signing_key.pem
or
openssl ecparam -name prime256v1 -genkey -noout -out secure_boot_signing_key.
↪pem
```

  2. 在 menuconfig 中，选择 Sign binaries during build, 并指定刚才生成的密钥路径, 如下图。



  3. 分别编译 bootloader 与应用代码

```
make bootloader
make
```

  4. 使用 esptool 将编译生成的 bin 文件写入 flash 对应地址, 以 example 中 hellow-world 工程为例:

```
bootloader.bin -->   0x1000
partition.bin  -->   0x8000
app.bin        -->  0x10000
python $IDF_PATH/components/esptool_py/esptool/esptool.py --chip esp32 --
↪port /dev/cu.SLAB_USBtoUART --baud 1152000 --before default_reset --
↪after no_reset write_flash -z --flash_mode dio --flash_freq 40m --flash_
↪size detect 0x1000 $IDS_PATH/esp-idf/examples/get-started/hello_world/
↪build/bootloader/bootloader.bin 0xf000 $IDF_PATH/esp-idf/examples/get-
↪started/hello_world/build/phy_init_data.bin 0x10000 $IDF_PATH/examples/
↪get-started/hello_world/build/hello-world.bin 0x8000 $IDF_PATH/examples/
↪get-started/hello_world/build/partitions_singleapp.bin
```

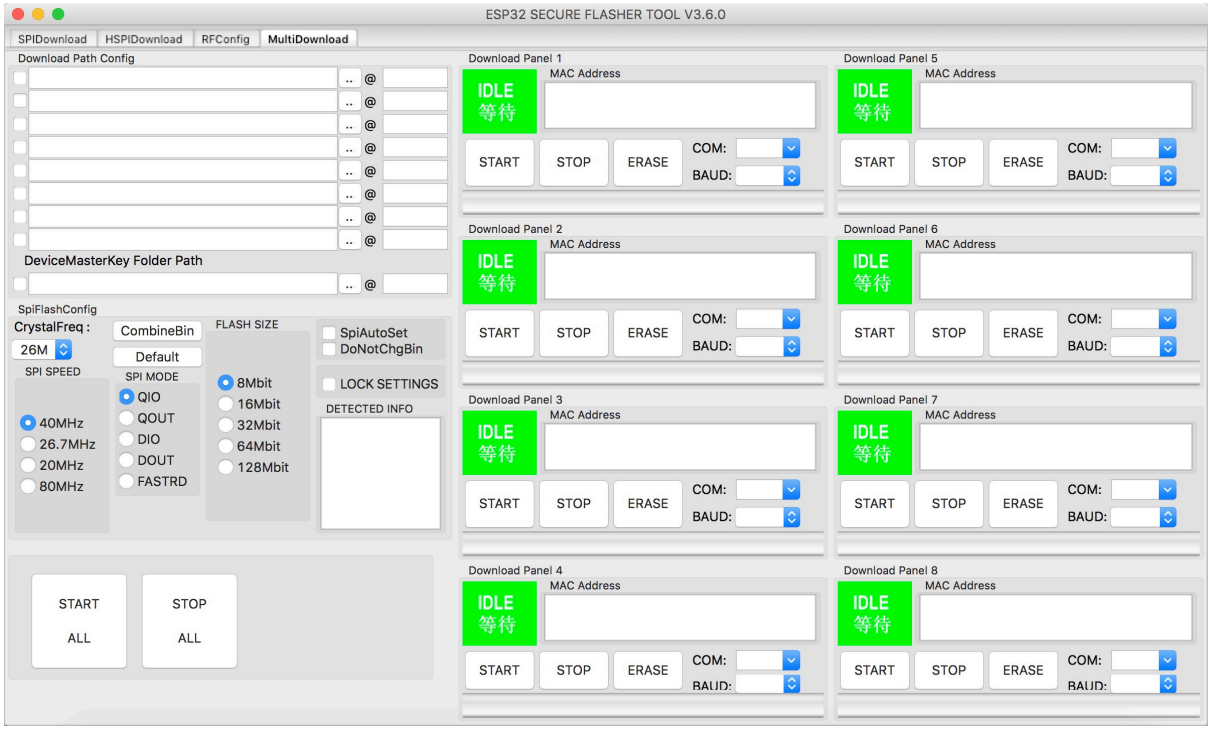
Note: 以上命令仅是示例代码，请在使用时，替换其中的文件路径以及所选参数，包括串口、波特率、SPI 模式和频率等。

  5. 我们也可以使用 window 平台的下载工具来完成工厂下载。需要在配置文件中，关闭工具的 security 功能，这样工具端就不会操作 security 相关特性，完全由硬件和 bootloader 来完成初始

化：

```
[SECURE BOOT]
secure_boot_en * False
[FLASH ENCRYPTION]
flash_encryption_en * False
```

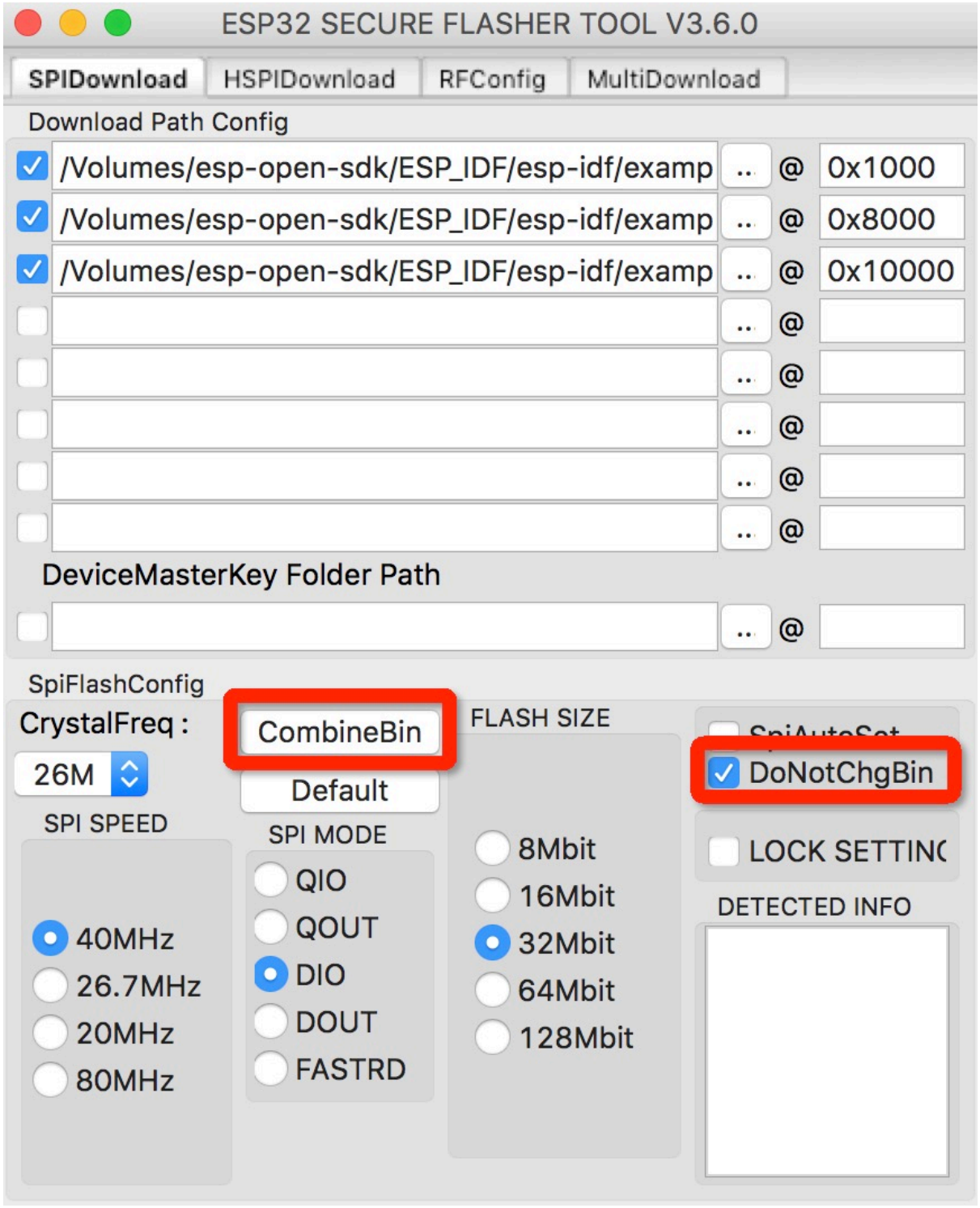
Note: 修改并保存参数前，请先关闭下载工具，配置文件修改完成并保存后，再开启运行下载工具。



6. 或者我们可以通过下载工具的 combine 功能，将多个 bin 文件打包为一个文件，再由工厂 flash 烧录器烧录进 flash 进行批量生产。
   - 选择 bin 文件并制定 flash 中的地址
   - 选中‘DoNotChgBin’选项，这样工具不会对 bin 文件的配置 (SPI 模式速率等) 进行任何修改。
   - 点击‘CombineBin’按键，生产合并后的 bin 文件。
   - 在‘combine’文件夹下，生成 target.bin，将其烧写到 Flash 的 0x0 地址即可。
   - 工具只会对填写的最大地址范围内的空白区域填充 0xff。并将文件按地址组合。
7. 下载完成后，需要运行一次程序，使 bootloader 完成 security 相关特性的初始化，包括 AES 密钥的随机生成并写入 EFUSE，以及对明文的 flash 进行首次加密。

Note: 请误在首次启动完成前，将芯片断电，以免造成芯片无法启动的情况。

- 注意事项：
  - 用于签名的私钥需要保密，如果泄漏，app.bin 有被伪造的可能性。
  - 使用者不能遗失私钥，必须使用私钥用于对 OTA app 签名 (如果有 OTA 功能)。
  - 芯片通过软件 bootloader 对 flash 加密是一个比较缓慢的过程，对于较大的 partition 可能需要花费一分钟左右
  - 若第一次执行 bootloader，flash 加密进行到一半芯片掉电
    - 没有使能 secure boot 时，可重新将 images 明文烧写到 flash 中，让芯片下次 boot 时重新加密 flash
    - 使能了 secure boot 时，由于无法重新烧写 flash，芯片将永久无法 boot

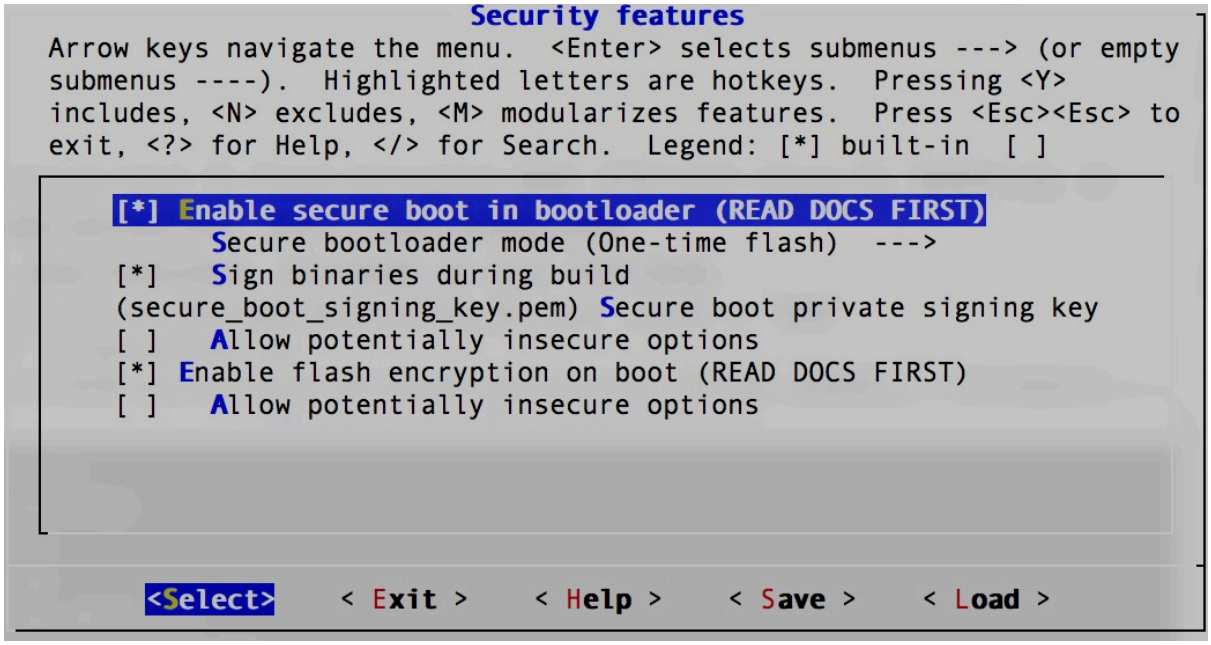

ESP32 SECURE FLASHER TOOL V3.6.0
SPIDownload
HSPIDownload
RFConfig
MultiDownload
Download Path Config
/Volumes/esp-open-sdk/ESP_IDF/esp-idf/examp
0x1000
/Volumes/esp-open-sdk/ESP_IDF/esp-idf/examp
0x8000
/Volumes/esp-open-sdk/ESP_IDF/esp-idf/examp
0x10000
DeviceMasterKey Folder Path
SpiFlashConfig
CrystalFreq :
26M
CombineBin
Default
FLASH SIZE
DoNotChgBin
LOCK SETTINC
DETECTED INFO
SPI SPEED
40MHz
26.7MHz
20MHz
80MHz
SPI MODE
QIO
QOUT
DIO
DOUT
FASTRD
8Mbit
16Mbit
32Mbit
64Mbit
128Mbit

方案 2: 通过下载工具初始化 security 特性

- 优势: 工具进行密钥的随机生成，直接将 image 密文烧写进 flash，然后配置 efuse. 避免过程中掉电造成无法启动的情况。
- 缺陷: 每个设备必须通过下载工具进行烧写，因为密钥不同，无法预先烧写相同的固件到 flash 中。
- 使用下载工具应用 secure boot 和 flash encryption，这时用户只需要的在 make menuconfig 中选择“enable secure boot in bootloader”并设置公钥/秘钥路径即可
- 下载工具在运行时，会随机产生 secure boot 与 flash encryption 密钥, 并烧写到对应的 EFUSE 位置中。
- 操作步骤:
  1. 随机生成 RSA 密钥文件，用于签名固件:

```
espsecure.py generate_signing_key secure_boot_signing_key.pem
or
openssl ecparam -name prime256v1 -genkey -noout -out secure_boot_signing_
↪key.pem
```

  2. 在 menuconfig 中，选择 Sign binaries during build, 并指定刚才生成的密钥路径, 如下图。



  3. 分别编译 bootloader 与应用代码

```
make bootloader
make
```

  4. 设置下载工具的安全配置文件

```
[DEBUG MODE]
debug_enable * False                    # 关闭 debug 模式，工具随机生成密钥。否则根据
pem 文件产生相同密钥
debug_pem_path *                        #debug 模式下，设置证书地址，用于生成可重复烧
写的密钥
[SECURE BOOT]
secure_boot_en * True                   # 开启 secure boot 功能
burn_secure_boot_key * True             # 使能 secure boot key 烧写
secure_boot_force_write * False         # 是否不检查 secure boot key block，强制烧
写 key
secure_boot_rw_protect * True           # 开启 secure boot key 区域的读写保护
[FLASH ENCRYPTION]
flash_encryption_en * True              # 开启 flash 加密功能
burn_flash_encryption_key * True        # 使能 flash encrypt key 烧写
flash_encrypt_force_write * False       # 是否不检查 flash encrypt key block，强制
烧写 key
```

(continues on next page)

(continued from previous page)

```
flash_encrypt_rw_protect * True      # 开启 flash encrypt key 区域的读写保护
[AES KEY]
aes_key_en * False                   # 目前未实现，仅保留该选项
burn_aes_key * False                 # 目前未实现，仅保留该选项
[DISABLE FUNC]
jtag_disable * True                  # 是否关闭 JTAG 调试功能
dl_encrypt_disable * True            # 是否关闭下载模式下 flash 加密功能
dl_decrypt_disable * True            # 是否关闭下载模式下 flash 解密功能
dl_cache_disable * True              # 是否关闭下载模式下的 flash cache 功能

注意：
修改并保存参数前，请先关闭下载工具，配置文件修改完成并保存后，再开启运行下载工具。
```

5. 使用下载工具进行下载，若不希望工具修改任何配置参数 (比如 flash 频率和模式)，请勾选‘DoNotChgBin’选项。下载工具会更具配置文件的设置，在下载过程中完成固件加密下载和密钥随机生成与烧写。

- 注意事项:
  - 用于签名的私钥需要保密，如果泄漏，app.bin 有被伪造的可能性。
  - 使用者不能遗失私钥，必须使用私钥用于对 OTA app 签名 (如果有 OTA 功能)。
  - 用户可以选择不启用 app image 的签名校验，只需要关闭 menuconfig 中的 secure boot 功能即可。下载工具会更具配置文件，通过 efuse 启用 secure boot。禁用 app image 的签名校验会存在安全隐患。

Chapter 17

Electrical&Lighting Solution

17.1 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

17.1.1 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);
```

17.1.2 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.

17.1.3 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.

17.1.4 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.

17.1.5 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.

17.1.6 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_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 **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.

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_iic_out_pin_t **red**

Port of the IIC dimming chip for red output

lightbulb_iic_out_pin_t **green**

Port of the IIC dimming chip for green output

lightbulb_iic_out_pin_t **blue**

Port of the IIC dimming chip for blue output

lightbulb_iic_out_pin_t **cold_white**

Port of the IIC dimming chip for cold or white output

lightbulb_iic_out_pin_t **warm_yellow**

Port of the IIC dimming chip for warm or yellow output

struct *lightbulb_config_t*::[anonymous]::[anonymous] **iic_io**

Configuration for IIC 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

Enumerations

enum **lightbulb_driver_t**

Supported drivers.

Values:

enumerator **DRIVER_SELECT_INVALID**

enumerator **DRIVER_ESP_PWM**

enumerator **DRIVER_SM2135E**

enumerator **DRIVER_SM2135EH**

enumerator **DRIVER_SM2x35EGH**

enumerator **DRIVER_BP57x8D**

enumerator **DRIVER_BP1658CJ**

enumerator **DRIVER_KP18058**

enumerator **DRIVER_WS2812**

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_iic_out_pin_t**

Port enumeration names for IIC chips.

Values:

enumerator **OUT1**

IIC output port 1.

enumerator **OUT2**

IIC output port 2.

enumerator **OUT3**

IIC output port 3.

enumerator **OUT4**

IIC output port 4.

enumerator **OUT5**

IIC output port 5.

enumerator **OUT_MAX**

The maximum value for the IIC output port enumeration, this is invalid value.

Chapter 18

Other Resources

18.1 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.

18.1.1 Adapted Products

Name	Function	Bus	Vendor	Datasheet	Driver
MCP23017	16-bit I/O expander	I2C	Microchip	Datasheet	mcp23017

18.2 ADC Range Extension Solution

18.2.1 ESP32-S3 ADC Range Extension

The maximum effective range of the ESP32-S3 ADC is 0 ~ 3100 mV. Through the external voltage divider circuit, it can meet most of the functions such as the ADC button or battery voltage detection. However, for applications such as NTC (Negative Temperature Coefficient) based temperature measurement, it may need to support full-scale (0 ~ 3300 mV) measurement. ESP32-S3 can adjust the ADC offset through registers, and combined with the nonlinear compensation method of the high voltage area, the expansion of the ADC range can be implemented.

The process is as follows:

1. Measure the first voltage value using the default offset
2. If the measured voltage is less than 2900 mV, the first voltage is directly output as the measurement result
3. Else if the measured voltage is greater than 2900 mV, increase the offset value to take the secondary measurement. Then carried out the nonlinear correction calculation on the secondary value, will be output as the final measurement 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.

18.2.2 Patch Use Guide

How to Apply a Patch Based on ESP-IDF `v4.4.8`

1. Please make sure ESP-IDF has been `checked out` to the `v4.4.8`
2. Please download file `esp32s3_adc_range_to_3100.patch` to anywhere you want
3. Using command `git am --signoff < esp32s3_adc_range_to_3100.patch` to apply the patch to ESP-IDF

How to Apply a Patch Based on ESP-IDF `v5.3.1`

1. Please make sure ESP-IDF has been `checked out` to the `v5.3.1`
2. Please download file `esp32s3_adc_range_to_3100_v531.patch` to anywhere you want
3. Using command `git am --signoff < esp32s3_adc_range_to_3100_v531.patch` to apply the patch to ESP-IDF

18.2.3 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

18.3 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.

18.3.1 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

18.3.2 Configuration

- USE_GPTIMER : Decide whether to use the GPTimer driver; default is to use the ESPTimer

18.3.3 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");
        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) {
```

(continues on next page)

(continued from previous page)

```
    ESP_LOGE(TAG, "Zero Detection create failed");
}
zero_detect_register_cb(g_zcds, zero_detection_event_cb, NULL);
```

18.3.4 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

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

Chapter 19

Contributions Guide

We welcome contributions to the esp-iot-solution project!

19.1 How to Contribute

Contributions to esp-iot-solution - fixing bugs, adding features, adding documentation - are welcome. We accept contributions via Github Pull Requests.

19.2 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.

19.3 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.

19.4 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.

19.5 Related Documents

19.5.1 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

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

P

R

S

T

U

V

Z