ESP-IoT-Solution

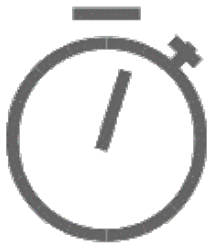
2023 年 06 月 28 日

Contents

[English]

这里是乐鑫 ESP-IoT-Solution 开发框架的文档中心。

ESP-IoT-Solution 包含物联网系统开发中常用的外设驱动和代码框架，可作为 ESP-IDF 的补充组件，方便用户实现更简单的开发。

		
入门指南	显示设备	USB 主机 & 设备
		
图形界面	输入设备	传感器集
		
音频设备	安全 & 加密	贡献代码

CHAPTER 1

快速入门

[English]

本文档旨在指导用户搭建 ESP-IoT-Solution (Espressif IoT Solution) 开发环境，通过一个简单的示例展示如何使用 ESP-IoT-Solution 搭建环境、创建工程、编译和下载固件至 ESP 系列开发板等步骤。

1.1 ESP-IoT-Solution 简介

ESP-IoT-Solution 包含物联网系统开发中常用的外设驱动和代码框架，可提供 ESP-IDF 的补充组件，方便用户实现更简单的开发，其中包含的内容如下：

- 传感器、显示屏、音频设备、输入设备、执行机构等设备驱动；
- 低功耗、安全加密、存储方案等代码框架或说明文档；
- 从实际应用的角度出发，为乐鑫开源解决方案提供了入口指引。

1.1.1 ESP-IoT-Solution 版本

不同版本的 ESP-IoT-Solution 说明如下：

ESP-IoT-Solution 版本	对应的 ESP-IDF 版本	主要变更	支持状态
master	>=v4.4	支持组件管理器，增加支持新的芯片支持	新功能开发分支
release/v1.1	v4.0.1	IDF 版本更新，删除已经移动到其它仓库的代码	停止维护
release/v1.0	v3.2.2	历史版本	停止维护

`master` 分支使用 `ESP` 组件管理器来管理组件，因此每个组件都是一个单独的软件包，每个包可能支持不同版本的 ESP-idf，这些版本将在组件的 `idf_component.yml` 文件中声明。

1.2 ESP-IDF 简介

ESP-IDF 是乐鑫为 ESP 系列芯片提供的物联网开发框架：

- ESP-IDF 包含一系列库及头文件，提供了基于 ESP SoC 构建软件项目所需的核心组件;
- ESP-IDF 还提供了开发和量产过程中最常用的工具及功能，例如：构建、烧录、调试和测量等。

注解： 详情请查阅：ESP-IDF 编程指南。

1.3 ESP 系列 SoC 简介

您可以选择任意 ESP 系列开发板使用 ESP-IoT-Solution，或者选择 板级支持组件 中支持的开发板快速开始。

ESP 系列 SoC 支持以下功能：

- 2.4 GHz Wi-Fi
- 蓝牙
- 高性能单核、双核处理器，运行频率可达 240 MHz
- 超低功耗协处理器
- 多种外设，包括 GPIO、I2C、I2S、SPI、UART、SDIO、RMT、LEDC PWM、Ethernet、TWAI、Touch、USB OTG 等
- 丰富的内存资源，内部 RAM 可达 520 KB，同时支持扩展 PSRAM
- 支持硬件加密等安全功能

ESP 系列 SoC 采用 40nm 工艺制成，具有最佳的功耗性能、射频性能、稳定性、通用性和可靠性，适用于各种应用场景和不同功耗需求。

注解： 不同系列 SoC 配置不同，详情请查阅 ESP 产品选型工具。

1.4 配置开发环境

1.4.1 1. 安装 ESP-IDF

由于 ESP-IoT-Solution 依赖 ESP-IDF 的基础功能和编译工具，因此首先需要参考 ESP-IDF 详细安装步骤 完成 ESP-IDF 开发环境的搭建。请注意，不同版本的 ESP-IoT-Solution 依赖的 ESP-IDF 版本可能不同，请参考*ESP-IoT-Solution 版本* 进行选择。

1.4.2 2. 获取 ESP-IoT-Solution

若选择 `master` 版本，可使用以下指令获取代码：

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

对于 `release/v1.1` 版本，可使用以下指令获取代码：

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

对于其它版本，请将 `release/v1.1` 替换成目标分支名。

1.5 使用 ESP-IoT-Solution 组件

如果您只想使用 ESP-IoT-Solution 中的组件，我们建议您从 ESP 组件注册表 ESP Component Registry 中使用它。

ESP-IoT-Solution 中注册的组件可在 README_CN.md 中查阅，您可以在项目根目录下使用 `idf.py add-dependency` 命令直接将组件从 Component Registry 添加到项目中。例如，执行 `idf.py add-dependency "expressif/usb_stream"` 命令添加 `usb_stream`，该组件将在 `CMake` 步骤中自动下载。

请参考 IDF Component Manager 查看更多关于组件管理器的细节.

1.6 编译和下载

1.6.1 1. 设置环境变量

以上步骤安装的代码和工具尚未添加至 PATH 环境变量，无法通过终端窗口使用这些工具。添加环境变量的步骤如下：

- 添加 ESP-IDF 环境变量：

  Windows 在 CMD 窗口运行：

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

  Linux 和 macOS 在终端运行：

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

  请将以上指令中的路径，替换成实际安装路径。

- 添加 IOT_SOLUTION_PATH 环境变量：

  Windows 在 CMD 窗口运行：

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

  Linux 和 macOS 在终端运行：

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

注解： 以上方法设置的环境变量，仅对当前终端有效，如果打开新终端，请重新执行以上步骤。

1.6.2 2. 设置编译目标

ESP-IDF 同时支持 `esp32`、`esp32s2` 等多款芯片，因此需要在编译代码之前设置的编译目标芯片（默认编译目标为 `esp32`），例如设置编译目标为 `esp32s2`：

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

对于 ESP-IoT-Solution 中基于 板级支持组件 开发的 example，还可以使用 `menuconfig` 在 `Board Options -> Choose Target Board` 中选择一个目标开发板：

```
idf.py menuconfig
```

1.6.3 3. 编译、下载程序

使用 `idf.py` 工具编译、下载程序，指令为：

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

请将 PORT 替换为当前使用的端口号，Windows 系统串口号一般为 `COMx`，Linux 系统串口号一般为 `/dev/ttyUSBx`，macOS 串口号一般为 `/dev/cu.`。

1.6.4 4. 串口打印 log

使用 `idf.py` 工具查看 log，指令为：

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

请将 `PORT` 替换为当前使用的端口号，Windows 系统串口号一般为 `COMx`，Linux 系统串口号一般为 `/dev/ttyUSBx`，macOS 串口号一般为 `/dev/cu.`。

1.7 相关文档

- ESP-IDF 详细安装步骤
- ESP-IDF 编程指南
- ESP 产品选型工具

CHAPTER 2

基础组件

[English]

2.1 通信总线组件 (Bus)

[English]

通信总线组件 (Bus) 是建立在 ESP-IDF 外设驱动代码之上的一套应用层代码，包括 `i2c_bus`、`spi_bus` 等，主要用于 ESP 芯片与外置设备之间的总线通信。该组件从应用开发的角度出发，实现了以下功能：

1. 简化外设初始化步骤
2. 线程安全的设备操作
3. 简单灵活的读写操作

该组件对以下概念进行了抽象：

1. 总线 (Bus)：通信时设备之间共同拥有的资源和配置项
2. 设备 (Device)：通信时设备特有的资源和配置项

每个物理外设总线 (Bus) 在电气条件允许的情况下，均可挂载一到多个设备（Device），其中 SPI 总线根据 CS 引脚对设备进行寻址，I2C 总线则根据设备地址进行寻址，进而实现相同总线不同设备之间软件上的独立。

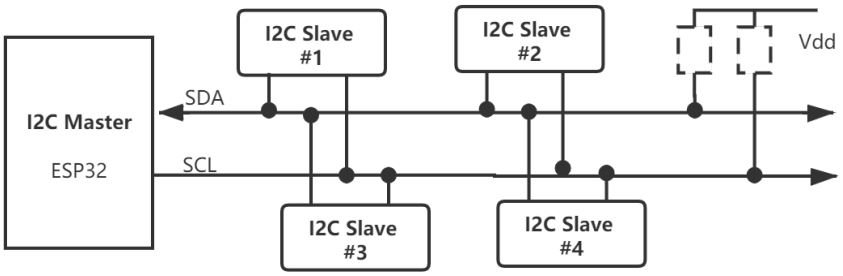


图 1: i2c_bus 连接框图

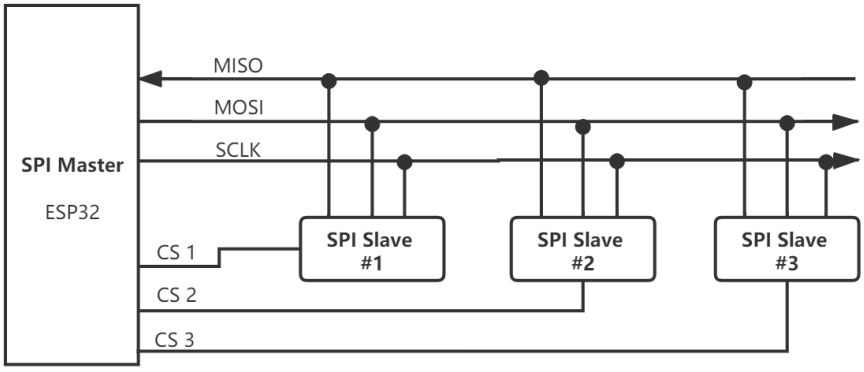


图 2: spi_bus 连接框图

2.1.1 i2c_bus 使用方法

1. 创建总线：使用*i2c_bus_create()* 创建一个总线实例。创建时需要指定 I2C 端口号，以及总线配置项 i2c_config_t。配置项包括 SDA 和 SCL 引脚号、上下拉模式，因为这些配置项在系统设计时已经确定，一般不在运行时切换。总线配置项还包括总线默认的时钟频率，在设备不指定频率时使用。
2. 创建设备：使用*i2c_bus_device_create()* 在已创建的总线实例之上创建设备，创建时需要指定总线句柄、设备的 I2C 地址、设备运行的时钟频率，I2C 传输时将根据设备的配置项动态切换频率。设备时钟速率可配置为 0，表示默认使用当前的总线频率。
3. 数据读取：使用*i2c_bus_read_byte()*、*i2c_bus_read_bytes()* 可直接进行 Byte 的读取操作；使用*i2c_bus_read_bit()*、*i2c_bus_read_bits()* 可直接进行 bit 的读取操作。只需要传入设备句柄、设备寄存器地址、用于存放读取数据的 buf 和读取长度的等。寄存器地址可设为 NULL_I2C_MEM_ADDR，用于操作没有内部寄存器的设备。
4. 数据写入：使用*i2c_bus_write_byte()*、*i2c_bus_write_bytes()* 可直接进行 Byte 的写入操作；使用*i2c_bus_write_bit()*、*i2c_bus_write_bits()* 可直接进行 bit 的写入操作。只需要传入设备句柄、设备寄存器地址、将要写入的数据位置和写入长度等。寄存器地址可设为 NULL_I2C_MEM_ADDR，用于操作没有内部寄存器的设备。
5. 删除设备和总线：如果所有的 i2c_bus 通信已经完成，可以通过删除设备和总线实例释放系统资源。可使用*i2c_bus_device_delete()* 分别将已创建的设备删除，然后使用*i2c_bus_delete()* 将总线资源删除。如果在设备未删除时删除总线，操作将不会被执行。

示例：

```
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
```

(下页继续)

(续上页)

```
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
```

注解: 对于某些特殊应用场景，例如:

1. 当寄存器地址为 16 位时，可以使用*i2c_bus_read_reg16()* 或*i2c_bus_write_reg16()* 进行读写操作;
2. 对于需要跳过地址阶段或者需要增加命令阶段的设备，可以使用*i2c_bus_cmd_begin()* 结合 I2C command link 进行操作。

2.1.2 spi_bus 使用方法

1. 创建总线：使用*spi_bus_create()* 创建一个总线实例，创建时需要指定 SPI 端口号（可选 SPI2_HOST、SPI3_HOST）以及总线配置项 spi_config_t。配置项包括 MOSI、MISO、SCLK 引脚号，因为这些引脚在系统设计时已经确定，一般不在运行时切换。总线配置项还包括 max_transfer_sz，用于配置一次传输时的最大数据量，设置为 0 将使用默认值 4096。
2. 创建设备：使用*spi_bus_device_create()* 在已创建的总线实例之上创建设备，创建时需要指定总线句柄、设备的 CS 引脚号、设备运行模式、设备运行的时钟频率，SPI 传输时将根据设备的配置项动态切换模式和频率。
3. 数据传输：使用*spi_bus_transfer_byte()*、*spi_bus_transfer_bytes()*、*spi_bus_transfer_reg16()* 以及*spi_bus_transfer_reg32()* 可直接进行数据的传输操作。由于 SPI 是全双工通信，因此每次传输发送和接收可以同时进行，只需要传入设备句柄、待发送数据、存放读取数据的 buf 和传输长度。
4. 删除设备和总线：如果所有的 spi_bus 通信已经完成，可以通过删除设备和总线实例释放系统资源。可使用*spi_bus_device_delete()* 分别将已创建的设备删除，然后使用*spi_bus_delete()* 将总线资源删除。如果在设备未删除时删除总线，操作将不会被执行。

示例:

```
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;
```

(下页继续)

(续上页)

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

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

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

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

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

注解： 对于某些特殊应用场景，可以直接使用 *spi_bus_transmit_begin()* 结合 spi_transaction_t 进行操作。

2.1.3 已适配 IDF 版本

- ESP-IDF v4.0 及以上版本。

2.1.4 已适配芯片

- ESP32
- ESP32-S2

2.1.5 API 参考

i2c_bus API 参考

Header File

- 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.

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

Parameters

- `port`: I2C port number
- `conf`: Pointer to I2C bus configuration

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

Delete and release the I2C bus resource.

Return

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `p_bus_handle`: Point to the I2C bus handle, if delete succeed handle will set to NULL.

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

Scan i2c devices attached on i2c bus.

Return uint8_t Total number of devices found on the 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, higer addresses will be discarded if num less-than the total number found on the I2C bus.

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

Get current active clock speed.

Return uint32_t current clock speed

Parameters

- `bus_handle`: I2C bus handle

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

Get created device number of the bus.

Return uint8_t created device number of the bus

Parameters

- `bus_handle`: I2C bus handle

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.

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

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.

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.

Return

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `p_dev_handle`: Point to the I2C device handle, if delete succeed handle will set to NULL.

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

Get device' s I2C address.

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

Parameters

- `dev_handle`: I2C device handle

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.

Return 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.

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

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.

Return 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.

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

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.

Return 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.

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.

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.

Return 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.

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

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.

Return 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.

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.

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.

Return 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.

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.

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.

Return 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.

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.

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.

Return 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.

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.

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

Return 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.

Parameters

- `dev_handle`: I2C device handle
- `cmd`: I2C command handler

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.

Return 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.

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.

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.

Return 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.

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

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_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

spi_bus API 参考

Header File

- 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.

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

Parameters

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

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

Deinitialize and delete the spi bus.

Return esp_err_t

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

Parameters

- `p_bus_handle`: pointer to spi bus handle, if delete succeed handle will set to NULL.

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.

Return spi_bus_device_handle_t handle for device operation, NULL if failed.

Parameters

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

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

Deinitialize and remove the device from spi bus.

Return esp_err_t

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

Parameters

- `p_dev_handle`: pointer to device handle, if delete succeed handle will set to NULL.

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.

Return esp_err_t

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

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.

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.

Return esp_err_t

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

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.

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

Return esp_err_t

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

Parameters

- `dev_handle`: handle for device operation.
- `p_trans`: Description of transaction to execute

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.

Return esp_err_t

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

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.

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.

Return esp_err_t

- ESP_ERR_INVALID_ARG if parameter is invalid

- ESP_ERR_TIMEOUT if bus is busy
- ESP_OK on success

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.

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

2.2 I2S LCD 驱动

2.2.1 API 参考

Header File

- bus/include/i2s_lcd_driver.h

Functions

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

Initilize i2s lcd driver.

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

Parameters

- `config`: configuration of i2s

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

Deinit i2s lcd driver.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG handle is invalid

Parameters

- `handle`: i2s lcd driver handle to deinitilize

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

Write a data to LCD.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG handle is invalid

Parameters

- `handle`: i2s lcd driver handle
- `data`: Data to write

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

Write a command to LCD.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG handle is invalid

Parameters

- `handle`: Handle of i2s lcd driver
- `cmd`: command to write

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

Write a command to LCD.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG handle is invalid

Parameters

- `handle`: Handle of i2s lcd driver
- `cmd`: command to write
- `length`: length of command

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

Write block data to LCD.

Return

- ESP_OK on success

- ESP_ERR_INVALID_ARG handle is invalid

Parameters

- `handle`: Handle of i2s lcd driver
- `data`: Pointer of data
- `length`: length of data

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

acquire a lock

Return Always return ESP_OK

Parameters

- `handle`: Handle of i2s lcd driver

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

release a lock

Return Always return ESP_OK

Parameters

- `handle`: Handle of i2s lcd driver

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 板级支持组件 (Boards)

[English]

本文档主要介绍板级支持组件 (Boards) 的使用方法，该组件作为示例程序的公共组件，可向应用程序提供统一的引脚宏定义和与硬件无关的初始化操作，基于板级支持开发的应用程序可以同时兼容不同的开发板，具体功能如下：

1. 提供统一的引脚资源宏定义
2. 提供默认的外设配置参数
3. 提供统一的板级初始化接口
4. 提供开发板的硬件控制接口

Boards 组件的结构如下：

- Boards 组件中包含以下内容：
  - `board_common.h`，包含了公有 API 的函数声明
  - `board_common.c`，包含了对公有 API 的函数实现（虚函数）
  - `Kconfig.projbuild`，包含了公有的配置项

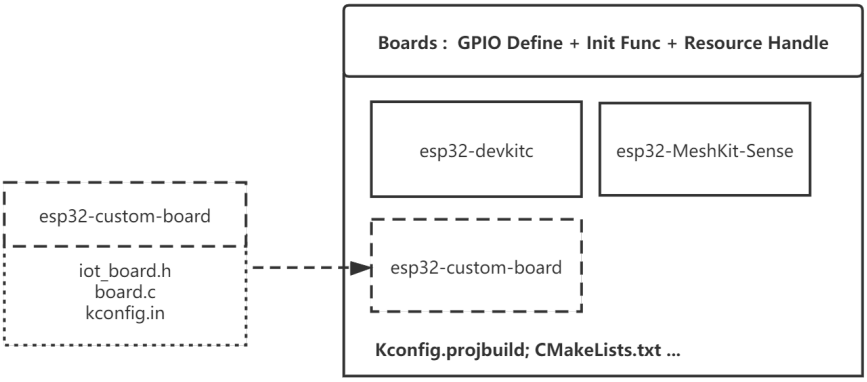


图 3: Boards 组件结构框图

- 以开发板名称命名的子文件夹包含以下内容：
  - `iot_board.h` 提供了开发板的引脚资源定义，该开发板特有的自定义 API 函数声明
  - `board.c` 提供了公有 API 的用户实现（默认虚函数），自定义 API 函数实现
  - `kconfig.in`，提供了该开发板特有的自定义配置项。

注解： Boards 在 `examples/common_components/boards` 中提供。

2.3.1 使用方法

1. 初始化开发板：在 `app_main` 使用 `iot_board_init` 初始化开发板，用户亦可在 `menuconfig` 中使用开发板切换和配置 进行初始化配置；
2. 获取外设句柄：使用 `iot_board_get_handle` 和 `board_res_id_t` 获取外设资源的句柄，如果该外设未被初始化将返回 `NULL`;
3. 使用句柄进行外设操作。

示例：

```
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,
```

(下页继续)

(续上页)

```
    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.
    */
}
```

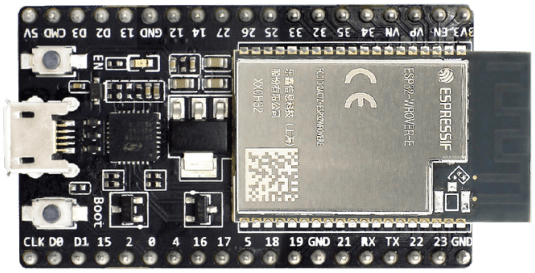
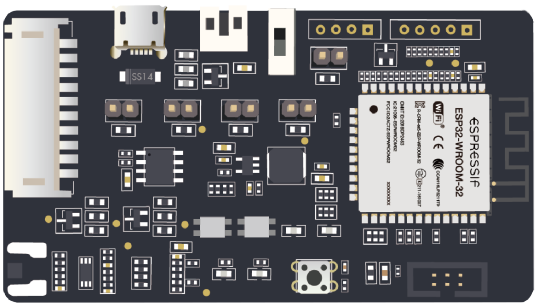
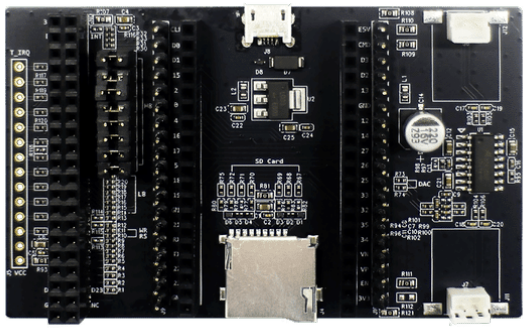
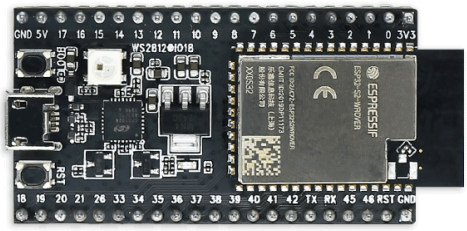
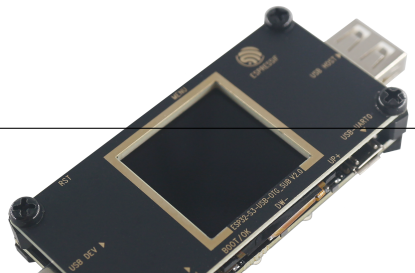
2.3.2 开发板切换和配置

基于 Boards 开发的应用程序，可以使用以下方法切换和配置开发板：

1. 选择目标开发板：在 menuconfig->Board Options->Choose Target Board 中选择一个开发板；
2. 配置开发板参数：Board Common Options 中包含公有的开发板配置项，例如配置是否在开发板初始化期间初始化 i2c_bus;XXX Board Options 中包含了该开发板特有的配置项，例如切换开发板供电状态等。
3. 使用 idf.py build flash monitor 重新编译并下载代码。

注解: 编译系统编译目标默认为 ESP32，如使用 ESP32-S2，请在编译之前使用 idf.py set-target esp32s2 配置目标。

2.3.3 已支持的开发板

ESP32 开发板	
	
esp32-devkitc	esp32-meshkit-sense
	
esp32-lcdkit	
ESP32-S2 开发板	
	
esp32s2-saola	
ESP32-S3 开发板	
	

2.3.4 添加新的开发板

通过添加新的开发板，可以快速适配基于 `Boards` 组件开发的应用程序。

添加开发板过程：

1. 按照组件文件结构 准备必要的 `iot_board.h`；
2. 按照需求在 `board_xxx.c` 添加该开发板特有的函数实现，或对公有的弱函数进行覆盖；
3. 按照需求在 `kconfig.in` 添加该开发板特有的配置项；
4. 将开发板信息添加到 `Kconfig.projbuild`，供用户选择；
5. 将开发板目录添加到 `CMakeLists.txt`，使其能被编译系统索引。如果需要支持老的 `make` 编译系统，请同时修改 `component.mk`。

注解： 可复制 `Boards` 中已添加的开发板文件夹，通过简单修改完成开发板的添加。

2.3.5 组件依赖

- 公共依赖项：bus, button, led_strip

2.3.6 已适配 IDF 版本

- ESP-IDF v4.4 及以上版本。

2.3.7 已适配芯片

- ESP32
- ESP32-S2
- ESP32-S3

CHAPTER 3

显示

[English]

3.1 显示屏

[English]

许多应用需要把信息显示给用户，因此显示屏是很重要的一个显示设备。ESP32 系列芯片为屏幕显示应用提供了丰富的外设支持，包含的接口类型有 `I2C`、`8080`、`SPI` 和 `RGB`。

一块屏幕通常有两个主要部分：*显示基板*和 *驱动 IC*。显示基板（也称显示面板）决定了显示的尺寸、分辨率、色彩等参数；驱动 IC（也称屏幕控制器）是匹配显示基板并与基板封装在一起的，它通过一定的接口对外通信并控制显示基板进行显示。

3.1.1 ESP32 系列芯片的屏幕接口

大部分屏幕控制器可以支持多种显示接口。如 ILI9486 支持 MCU 接口（即 8080 接口）、SPI 接口、RGB 接口和 MIPI 接口。

ESP32 系列芯片接口支持情况如下：

SoC	I2C 接口	SPI 接口	8080 接口	RGB 接口
ESP32	Supported	Supported	Supported	
ESP32-S2	Supported	Supported	Supported	
ESP32-S3	Supported	Supported	Supported	Supported
ESP32-C3	Supported	Supported		

注解： ESP32 系列芯片的 I2C、SPI、8080 和 RGB 接口均可以通过 GPIO 交换矩阵将信号映射到任意管脚。

SPI 接口

ESP32 系列芯片的 SPI 接口特点：

- SPI 时钟频率最高可达 80MHz
- 支持 QSPI 模式

8080 接口

ESP32 系列芯片的 8080 接口分别由不同的外设实现

SoC	实现 8080 接口的外设	特点
ESP32	I2S	8/16 位模式下时钟频率最高可达 20MHz
ESP32-S2	I2S	8 位模式下时钟频率最高可达 40MHz，16 位模式下 20MHz
ESP32-S3	LCD_CAM	8 位模式下时钟频率最高可达 40MHz，16 位模式下 20MHz

RGB 接口

ESP32-S3 芯片拥有一个 RGB 接口，支持并行 RGB565 模式和串行 RGB888 模式。RGB 接口与 8080 接口都是并行接口，一个重要区别在于 RGB 接口的屏幕一般不带显存，这就需要每一个刷新周期都对屏幕上的所有像素数据进行更新。由于 ESP32-S3 芯片内部没有足够的 RAM 用于显存，所以需要外接 PSRAM，并将显存置于 PSRAM 上。由于显存要求的带宽较大，所以为达到良好的显示效果 **需要使用 8 线 PSRAM，并且将 PSRAM 时钟频率至少设置为 80MHz**。

注解： RGB 接口的驱动在 esp-idf 的 `components/esp_lcd` 组件中而不在 esp-iot-solution。例程请参考 `esp-idf/examples/peripherals/lcd/rgb_panel`

3.1.2 屏幕控制器

已适配的屏幕控制器型号如下表所示：

控制器型号	最大分辨率	类型
NT35510	480 x 865	Color
ILI9806	480 x 865	Color
RM68120	480 x 865	Color
ILI9486	320 x 480	Color
ILI9488	320 x 480	Color
ILI9341	240 x 320	Color
ILI9342	240 x 320	Color
ST7789	240 x 320	Color
ST7796	320 x 480	Color
SSD1351	128 x 128	Color
SSD1306	128 x 64	Mono
SSD1307	128 x 39	Mono
SSD1322	480 x 128	Gray
SSD1963	480 x 864	Color

其中 `SSD1963` 可将 8080 接口转换到 RGB 接口。

3.1.3 驱动结构

为了更加符合一个屏幕控制芯片拥有多个接口的实际情况，将屏幕驱动程序划分为 *接口驱动* 和 *屏幕控制器驱动* 两部分。

- 接口驱动：完成基本的命令和数据的读写
- 屏幕控制器驱动：通过接口驱动来完成屏幕的显示

在程序设计上，一个屏幕控制器驱动可以通过调用不同的接口驱动以实现切换硬件上不同的接口。

3.1.4 屏幕的分类

对屏幕进行分类的讨论将有助于我们对驱动的理解，这里将按照屏幕可显示的色彩来分类，而非 OLED、LCD 等屏幕的面板材料。一般情况下，屏幕显示的色彩决定了 BPP (Bits Per Pixel)，而 BPP 的不同导致程序的处理方式有一些不同，下面将更直观地列举几种 GRAM 映射到像素点的方式：

从以上图中可以看出，映射方式大概可以分为两类：

- BBP >= 8，通常是支持 RGB888、RGB666、RGB565 等编码的彩色屏幕。
- BPP < 8，通常是单色的屏幕，可能是黑白的，也可能是灰阶的。

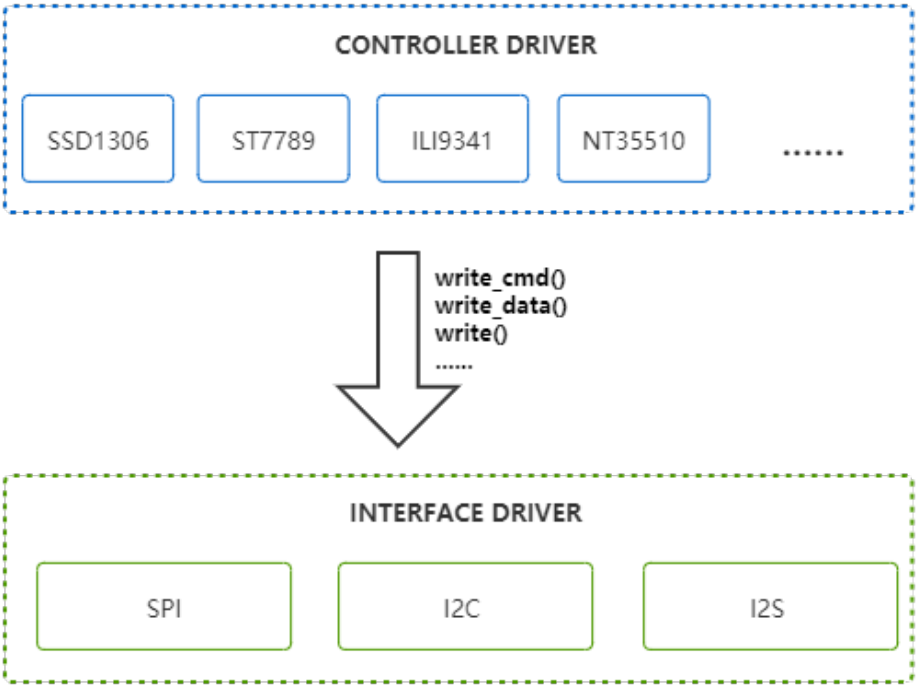


图 1: 屏幕驱动结构框图

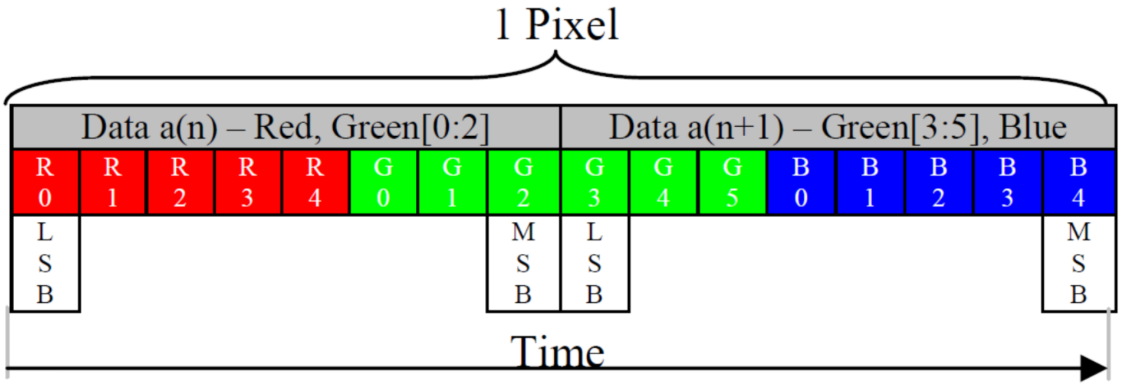


图 2: BPP = 16 GRAM 结构

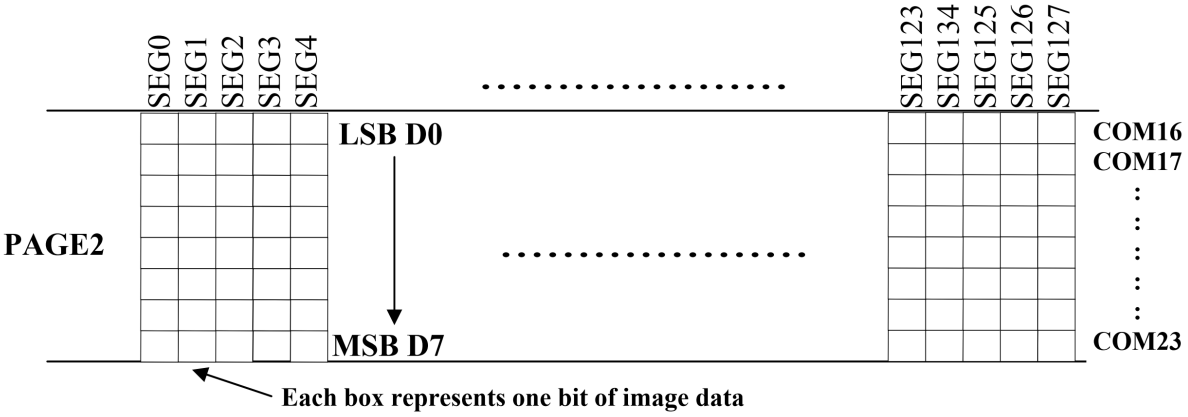


图 3: BPP = 1 GRAM 结构

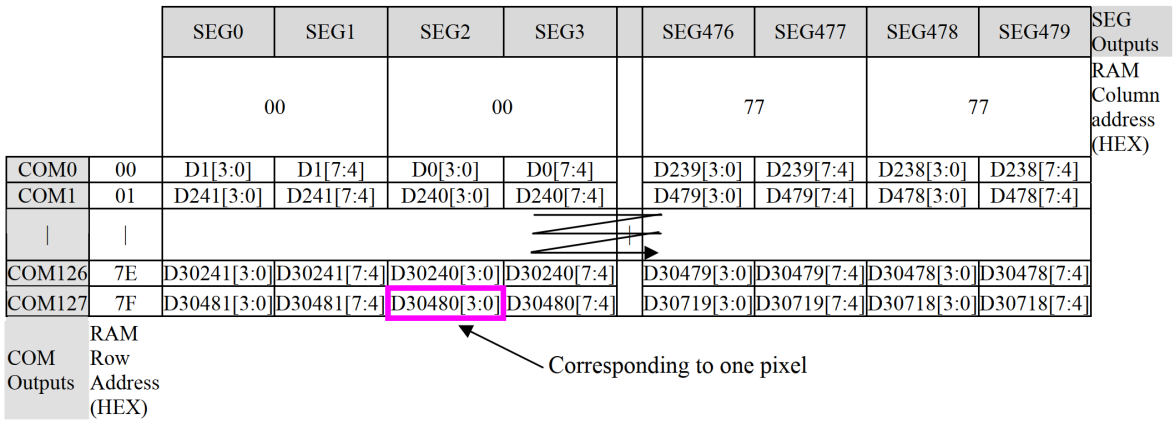


图 4: BPP = 4 GRAM 结构

BPP < 8 时，一个字节映射到了多个像素，因此无法直接地控制单个像素。这时，驱动不支持 `draw_pixel()` 函数，且 `set_window()` 的参数也将受到限制。BPP >= 8 时，则可以轻松地访问单个像素。

> **注意：** 对于彩色屏幕，驱动仅支持 RGB565 颜色编码。

3.1.5 接口驱动

一个屏幕控制器通常有多种接口，在 ESP32 上通常使用 `8080 并口`、`SPI` 和 `I2C` 这三种接口与屏幕连接，可以在调用 `scr_interface_create()` 创建接口驱动时选用其中一种接口。

注解： 使用 `scr_interface_create()` 创建不同类型的接口时需要注意传入与之对应的配置参数类型，例如 8080 接口是*`i2s_lcd_config_t`* 而 SPI 接口是*`scr_interface_spi_config_t`*。

为了方便在屏幕控制器驱动中统一使用这些接口，display/screen/screen_utility/interface_drv_def.h 中定义了所有接口，可以使用简单的参数调用接口驱动。

注解： 大部分屏幕是大端模式，而 ESP32 是小端模式，因此可在使用的接口驱动中根据 `swap_data` 配置可选择进行大小端转换。**请注意：** 当使用 SPI 接口时，由于 IDF 的 SPI 驱动内部没有提供该功能，接口驱动将会对传入数据进行转换，这要求传入的数据是可写的，因此数据 **必须**存放在 RAM 中。

3.1.6 屏幕控制器驱动

该部分将根据不同的屏幕控制器分别实现显示等功能，为了方便地移植到不同 GUI 库，对屏幕的一部分通用函数用*`scr_driver_t`* 进行了抽象。对于一些屏幕的非通用功能，需要自行调用其特定的函数完成。

对于这些通用函数，由于屏幕控制器本身的功能不尽相同，并不是所有的屏幕都全部实现了，例如：对于 BPP < 8 的屏幕不支持 `draw_pixel()` 函数。调用不支持的函数将返回 `ESP_ERR_NOT_SUPPORTED`。

显示方向

这里设置的屏幕显示方向是完全由屏幕硬件实现的，这个功能在不同的屏幕控制器上会有差异。一共有 8 种可能的显示方向，显示器可以旋转 0°、90°、180° 或 270°，也可以从顶部或底部查看，默认方向为 0° 和顶部查看。这 8 (4 × 2) 个不同的显示方向也可以表示为 3 个二进制开关的组合：X-mirroring、Y-mirroring 和 X/Y swapping。

下表将列出全部 8 种组合显示的方向。如果显示方向不正常，请查看下表中的配置开关使其正常工作。

	0 [SCR_DIR_LRTB]		SCR_MIRROR_Y [SCR_DIR_LRBT]
	SCR_MIRROR_X [SCR_DIR_RLTB]		SCR_MIRROR_X| SCR_MIRROR_Y [SCR_DIR_RLBT]
	SCR_SWAP_XY [SCR_DIR_TBLR]		SCR_SWAP_XY| SCR_MIRROR_Y [SCR_DIR_BTLR]
	SCR_SWAP_XY| SCR_MIRROR_X [SCR_DIR_TBRL]		SCR_SWAP_XY| SCR_MIRROR_X| SCR_MIRROR_Y [SCR_DIR_BTRL]

对于不同屏幕控制器，屏幕显示方向的实现并不完全相同，通常分为以下的情况：

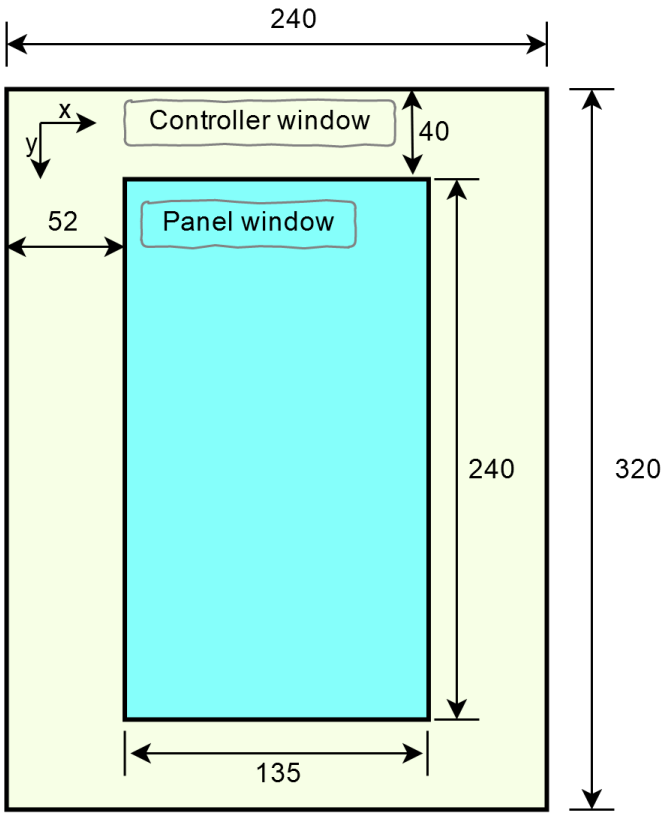
- 对于彩色屏幕，支持 8 个方向的旋转。
- 对于单色屏幕，如 SSD1306 等屏幕来说，只支持`scr_dir_t` 中定义的前 4 个方向，即不支持交换 XY 轴。

注解： 显示方向还和使用的屏幕面板有关系，你可能会发现两种异常的情况：

- 显示方向设置为`SCR_DIR_LRTB`，屏幕却不是按照上表中对应的显示方向。这可能是因为屏幕面板上的走线对 X/Y 方向上进行了镜像，这时应该根据实际情况调整旋转以得到期望的显示方向。
- 旋转了屏幕后，显示内容不见了。这可能是因为屏幕面板的分辨率小于屏幕控制器的分辨率，导致旋转后显示区域没有完全落在屏幕面板上，这时应考虑设置正确的显示区域偏移。

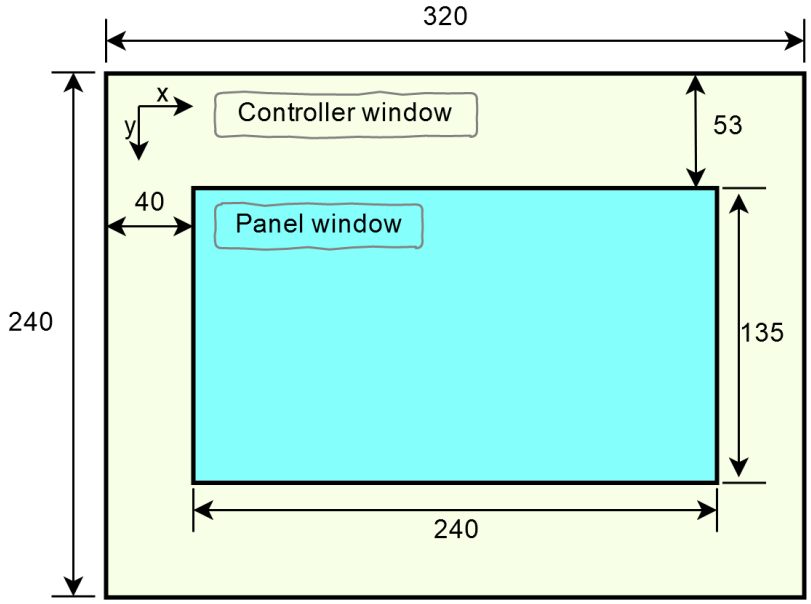
显示区域的偏移

在一些小尺寸的屏幕上，通常其可视区域分辨率小于所用屏幕控制器的分辨率。请参考以下示例图：



图中 `Controller window` 是屏幕控制器的显示窗口，分辨率为 240 × 320，`Panel window` 是屏幕面板窗口，分辨率为 135 × 240，可视的区域为屏幕面板区域。可以看出显示区域在水平方向上偏移了 52 个像素，在垂直方向上偏移了 40 个像素。

当屏幕逆时针旋转 90° 后，变成水平方向上偏移了 40 个像素，在垂直方向上偏移了 53 个像素，如下所示：



屏幕控制器驱动会帮助你自动根据屏幕的旋转来改变偏移的值，以保持正确的显示区域。你需要做的是在*scr_controller_config_t* 中正确配置屏幕在 SCR_DIR_LRTB 方向时的偏移和屏幕面板尺寸。

注解:

- 显示偏移仅支持 BPP >= 8 的屏幕。
- 当屏幕控制器是可选分辨率的时候，发现偏移不对可能是因为选择的分辨率与实际不符，此时应该修改程序，如：ILI9806 可尝试修改 ili9806.c 中的 ILI9806_RESOLUTION_VER 为实际的分辨率。

3.1.7 应用示例

注解: 以下示例不再维护，LCD 以及 LVGL 示例请参考：i80_controller、rgb_panel 和 spi_lcd_touch

初始化屏幕

```
scr_driver_t g_lcd; // A screen driver
esp_err_t ret = ESP_OK;

/** Initialize 16bit 8080 interface */
i2s_lcd_config_t i2s_lcd_cfg = {
    .data_width  = 16,
    .pin_data_num = {
        1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
    },
    .pin_num_cs = 45,
    .pin_num_wr = 34,
    .pin_num_rs = 33,
    .clk_freq = 20000000,
    .i2s_port = I2S_NUM_0,
    .buffer_size = 32000,
    .swap_data = false,
};
scr_interface_driver_t *iface_drv;
scr_interface_create(SCREEN_IFACE_8080, &i2s_lcd_cfg, &iface_drv);

/** Find screen driver for ILI9806 */
ret = scr_find_driver(SCREEN_CONTROLLER_ILI9806, &g_lcd);
if (ESP_OK != ret) {
    ESP_LOGE(TAG, "screen find failed");
```

(下页继续)

(续上页)

```
    return;
}

/** Configure screen controller */
scr_controller_config_t lcd_cfg = {
    .interface_drv = iface_drv,
    .pin_num_rst = -1,       // The reset pin is not connected
    .pin_num_bckl = -1,      // The backlight pin is not connected
    .rst_active_level = 0,
    .bckl_active_level = 1,
    .offset_hor = 0,
    .offset_ver = 0,
    .width = 480,
    .height = 854,
    .rotate = SCR_DIR_LRBT,
};

/** Initialize ILI9806 screen */
g_lcd.init(&lcd_cfg);
```

注解: 默认情况下只打开了 ILI9341 屏幕的驱动，如果要使用其他的驱动，需要在 `menuconfig -> Component config -> LCD Drivers -> Select Screen Controller` 中使能对应屏幕驱动。

显示图像

```
/** Draw a red point at position (10, 20) */
lcd.draw_pixel(10, 20, COLOR_RED);

/** Draw a bitmap */
lcd.draw_bitmap(0, 0, width_of_pic, height_of_pic, pic_data);
```

获取屏幕信息

```
scr_info_t lcd_info;
lcd.get_info(&lcd_info);
ESP_LOGI(TAG, "Screen name:%s | width:%d | height:%d", lcd_info.name, lcd_info.width,␣
↪lcd_info.height);
```

3.1.8 API 参考

Header File

- display/screen/screen_driver.h

Functions

esp_err_t **scr_find_driver** (*scr_controller_t controller*, *scr_driver_t* **out_screen*)

Find a screen driver.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG Arguments is NULL.
- ESP_ERR_NOT_FOUND Screen controller was not found.

Parameters

- `controller`: Screen controller to initialize
- `out_screen`: Pointer to a screen driver

Structures

struct scr_controller_config_t

configuration of screen controller

Public Members

scr_interface_driver_t ***interface_drv**

Interface driver for screen

int8_t **pin_num_rst**

Pin to hardreset LCD

int8_t **pin_num_bckl**

Pin for control backlight

uint8_t **rst_active_level**

Reset pin active level

uint8_t **bckl_active_level**

Backlight active level

uint16_t **width**

Screen width

uint16_t **height**

Screen height

uint16_t **offset_hor**

Offset of horizontal

uint16_t **offset_ver**

Offset of vertical

scr_dir_t **rotate**

Screen rotate direction

struct scr_info_t

Information of screen.

Public Members

uint16_t **width**

Current screen width, it may change when apply to rotate

uint16_t **height**

Current screen height, it may change when apply to rotate

scr_dir_t **dir**

Current screen direction

scr_color_type_t **color_type**

Color type of the screen, See scr_color_type_t struct

uint8_t **bpp**

Bits per pixel

const char ***name**

Name of the screen

struct scr_driver_t

Define a screen common function.

Public Members

esp_err_t (*`init`) (`const` *scr_controller_config_t* *lcd_conf)

Initialize screen.

Return

- ESP_OK on success
- ESP_FAIL Driver not installed

Parameters

- `lcd_conf`: Pointer to a structure with lcd config arguments. see struct *scr_controller_config_t*

esp_err_t (*`deinit`) (void)

Deinitialize screen.

Return

- ESP_OK on success
- ESP_FAIL Deinitialize failed
- ESP_ERR_NOT_SUPPORTED unsupported

esp_err_t (*`set_direction`) (*scr_dir_t* dir)

Set screen direction of rotation.

Note Not all screens support eight directions, it depends on the screen controller.

Return

- ESP_OK on success
- ESP_FAIL Failed

Parameters

- `dir`: Pointer to a scr_dir_t structure. You can set the direction in two ways, for example, set it to "SCR_DIR_LRBT" or "SCR_MIRROR_Y" , They are the same, depending on which expression you want to use

esp_err_t (*`set_window`) (uint16_t x0, uint16_t y0, uint16_t x1, uint16_t y1)

Set screen window.

Note When the BPP of the screen controller is less than 8, the coordinate value is limited to a multiple of some number

Return

- ESP_OK on success

- ESP_FAIL Failed

Parameters

- `x0`: Starting point in X direction
- `y0`: Starting point in Y direction
- `x1`: End point in X direction
- `y1`: End point in Y direction

esp_err_t (*`write_ram_data`) (uint16_t color)

Write a RAM data.

Return

- ESP_OK on success
- ESP_FAIL Failed

Parameters

- `color`: New color of a pixel

esp_err_t (*`draw_pixel`) (uint16_t x, uint16_t y, uint16_t color)

Draw one pixel in screen with color.

Return

- ESP_OK on success
- ESP_FAIL Failed

Parameters

- `x`: X co-ordinate of set orientation
- `y`: Y co-ordinate of set orientation
- `color`: New color of the pixel

esp_err_t (*`draw_bitmap`) (uint16_t x, uint16_t y, uint16_t w, uint16_t h, uint16_t *bitmap)

Fill the pixels on LCD screen with bitmap.

Return

- ESP_OK on success
- ESP_FAIL Failed

Parameters

- `x`: Starting point in X direction

- `y`: Starting point in Y direction
- `w`: width of image in bitmap array
- `h`: height of image in bitmap array
- `bitmap`: pointer to bitmap array

esp_err_t (*`get_info`) (*scr_info_t* *info)

Get screen information.

Return

- ESP_OK on success
- ESP_FAIL Failed

Parameters

- `info`: Pointer to a *scr_info_t* structure.

Macros

COLOR_BLACK

COLOR_NAVY

COLOR_DARKGREEN

COLOR_DARKCYAN

COLOR_MAROON

COLOR_PURPLE

COLOR_OLIVE

COLOR_LIGHTGREY

COLOR_DARKGREY

COLOR_BLUE

COLOR_GREEN

COLOR_CYAN

COLOR_RED

COLOR_MAGENTA

COLOR_YELLOW

COLOR_WHITE

COLOR_ORANGE

COLOR_GREENYELLOW

COLOR_PINK

COLOR_SILVER

COLOR_GRAY

COLOR_LIME

COLOR_TEAL

COLOR_FUCHSIA

COLOR_ESP_BKGD

Enumerations

enum scr_dir_t

Define all screen direction.

Values:

SCR_DIR_LRTB

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

SCR_DIR_LRBT

From left to right then from bottom to top

SCR_DIR_RLTB

From right to left then from top to bottom

SCR_DIR_RLBT

From right to left then from bottom to top

SCR_DIR_TBLR

From top to bottom then from left to right

SCR_DIR_BTLR

From bottom to top then from left to right

SCR_DIR_TBRL

From top to bottom then from right to left

SCR_DIR_BTRL

From bottom to top then from right to left

SCR_DIR_MAX

SCR_MIRROR_X = 0x40

Mirror X-axis

SCR_MIRROR_Y = 0x20

Mirror Y-axis

SCR_SWAP_XY = 0x80

Swap XY axis

enum scr_color_type_t

The types of colors that can be displayed on the screen.

Values:

SCR_COLOR_TYPE_MONO

The screen is monochrome

SCR_COLOR_TYPE_GRAY

The screen is gray

SCR_COLOR_TYPE_RGB565

The screen is colorful

enum scr_controller_t

All supported screen controllers.

Values:

SCREEN_CONTROLLER_ILI9341

SCREEN_CONTROLLER_ILI9342

SCREEN_CONTROLLER_ILI9806

SCREEN_CONTROLLER_ILI9486

SCREEN_CONTROLLER_ILI9488

SCREEN_CONTROLLER_NT35510

SCREEN_CONTROLLER_RM68120

SCREEN_CONTROLLER_ST7789

SCREEN_CONTROLLER_ST7796

SCREEN_CONTROLLER_SSD1351

SCREEN_CONTROLLER_SSD1963

SCREEN_CONTROLLER_SSD1306

SCREEN_CONTROLLER_SSD1307

SCREEN_CONTROLLER_SSD1322

Header File

- display/screen/interface_driver/scr_interface_driver.h

Functions

esp_err_t **scr_interface_create**(*scr_interface_type_t type*, void **config*, *scr_interface_driver_t **out_driver*)

Create screen interface driver.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG Arguments is NULL.
- ESP_FAIL Initialize failed
- ESP_ERR_NO_MEM: Cannot allocate memory.

Parameters

- `type`: Type of screen interface
- `config`: configuration of interface driver
- `out_driver`: Pointer to a screen interface driver

esp_err_t **scr_interface_delete**(**const** *scr_interface_driver_t *driver*)

Delete screen interface driver.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG Arguments is NULL.

Parameters

- `driver`: screen interface driver to delete

Structures

struct scr_interface_spi_config_t

SPI interface configuration.

Public Members

spi_bus_handle_t **spi_bus**

Handle of spi bus

int8_t **pin_num_cs**

SPI Chip Select Pin

int8_t **pin_num_dc**

Pin to select Data or Command for LCD

int **clk_freq**

SPI clock frequency

bool **swap_data**

Whether to swap data

struct scr_interface_i2c_config_t

I2C interface configuration.

Public Members

i2c_bus_handle_t **i2c_bus**

Handle of i2c bus

uint32_t **clk_speed**

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

uint16_t **slave_addr**

I2C slave address

struct scr_interface_driver_t

Define common function for screen interface driver.

Public Members

scr_interface_type_t **type**

Interface bus type, see scr_interface_type_t struct

esp_err_t (***write_command**) (void *handle, **const** uint8_t *cmd, uint32_t length)

Function to write command

esp_err_t (***write_data**) (void *handle, uint16_t data)

Function to write a data

esp_err_t (***write**) (void *handle, **const** uint8_t *data, uint32_t length)

Function to write a block data

esp_err_t (*`read`) (void *handle, uint8_t *data, uint32_t length)

Function to read a block data

esp_err_t (*`bus_acquire`) (void *handle)

Function to acquire interface bus

esp_err_t (*`bus_release`) (void *handle)

Function to release interface bus

Enumerations

`enum scr_interface_type_t`

Type of screen interface.

Values:

`SCREEN_IFACE_I2C`

I2C interface

`SCREEN_IFACE_8080`

8080 parallel interface

`SCREEN_IFACE_SPI`

SPI interface

3.2 数码管驱动

[English]

数码管/LED 点阵是嵌入式系统中常见的显示方案，该方案比 LCD 显示屏占用更少的引脚和内存资源，实现也更加简单，比较适合计时、计数、状态显示等具有单一显示需求的应用场景。

ESP-IoT-Solution 已经适配的数码管/LED 显示驱动器如下：

名称	功能	接口	驱动	数据手册
CH450	数码管显示驱动芯片，支持 6 位数码管	I2C	ch450	CH450
HT16C21	20×4/16×8 LCD 控制器，支持 RAM 映射	I2C	ht16c21	HT16C21
IS31FL3XXX	LED 点阵控制器	I2C	is31fl3xxx	IS31FL3XXX

3.2.1 CH450 驱动

CH450 是一款数码管显示驱动芯片，可以用于驱动 6 位数码管或 48 点 LED 矩阵，可通过 `I2C` 接口与 ESP32 进行通信。

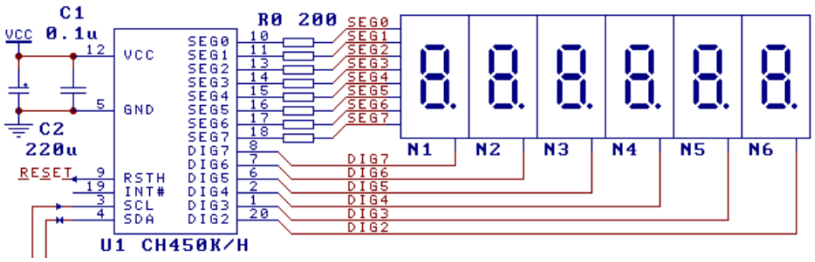


图 5: CH450 典型应用电路图

该驱动对 CH450 的基本操作进行了封装，用户可以直接调用 `ch450_write()` 或 `ch450_write_num()` 接口在数码管上进行数字显示。

示例

```
i2c_bus_handle_t i2c_bus = NULL;
ch450_handle_t seg = NULL;
i2c_config_t conf = {
    .mode = I2C_MODE_MASTER,
    .sda_io_num = I2C_MASTER_SDA_IO,
    .sda_pullup_en = GPIO_PULLUP_ENABLE,
    .scl_io_num = I2C_MASTER_SCL_IO,
    .scl_pullup_en = GPIO_PULLUP_ENABLE,
    .master.clk_speed = I2C_MASTER_FREQ_HZ,
};
i2c_bus = i2c_bus_create(I2C_MASTER_NUM, &conf);
seg = ch450_create(i2c_bus);

for (size_t i = 0; i < 10; i++) {
    for (size_t index = 0; index < 6; index++) {
        ch450_write_num(seg, index, i);
    }
    vTaskDelay(1000 / portTICK_PERIOD_MS);
}

ch450_delete(seg);
i2c_bus_delete(&i2c_bus);
```

3.2.2 HT16C21 驱动

HT16C21 是一款支持 RAM 映射的 LCD 控制/驱动芯片，可用于驱动 `20 x 4` 或 `16 x 8` 段码式液晶屏，该芯片通过 `I2C` 接口与 ESP32 进行通信。

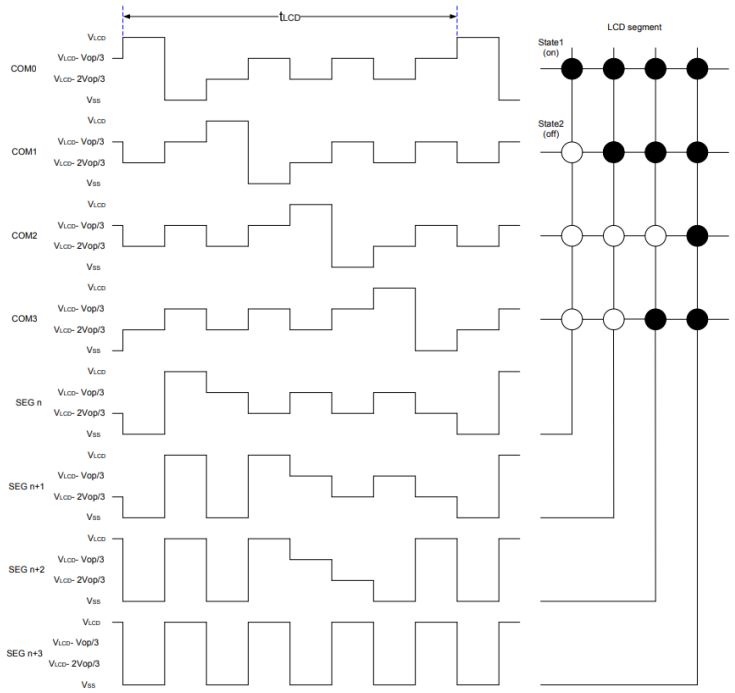


图 6: HT16C21 典型驱动模型

该驱动对 HT16C21 的基本操作进行了封装，用户使用 `ht16c21_create` 创建实例之后，通过 `ht16c21_param_config` 对驱动器参数进行配置，之后即可直接调用 `ht16c21_ram_write` 进行写入操作。

示例

```
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,
};
```

(下页继续)

(续上页)

```
i2c_bus = i2c_bus_create(I2C_MASTER_NUM, &conf);
seg = ht16c21_create(i2c_bus, HT16C21_I2C_ADDRESS_DEFAULT);

ht16c21_config_t ht16c21_conf = {
    .duty_bias = HT16C21_4DUTY_3BIAS;
    .oscillator_display = HT16C21_OSCILLATOR_ON_DISPLAY_ON;
    .frame_frequency = HT16C21_FRAME_160HZ;
    .blinking_frequency = HT16C21_BLINKING_OFF;
    .pin_and_voltage = HT16C21_VLCD_PIN_VOL_ADJ_ON;
    .adjustment_voltage = 0;
};
ht16c21_param_config(seg, &ht16c21_conf);
ht16c21_ram_write(seg, 0x00, lcd_data, 8);

ht16c21_delete(seg);
i2c_bus_delete(&i2c_bus);
```

3.2.3 IS31FL3XXX 驱动

IS31FL3XXX 系列芯片可用于驱动不同规模的 LED 点阵屏幕。其中 IS31FL3218 支持 18 个恒流通道，每个通道由独立的 PWM 控制，最大输出电流 38 mA，可直接驱动 LED 进行显示。IS31FL3736 支持更多的通道，最大可组成 `12 x 8` LED 矩阵，每个通道由一个 8 位 PWM 驱动，最大支持 256 级渐变。

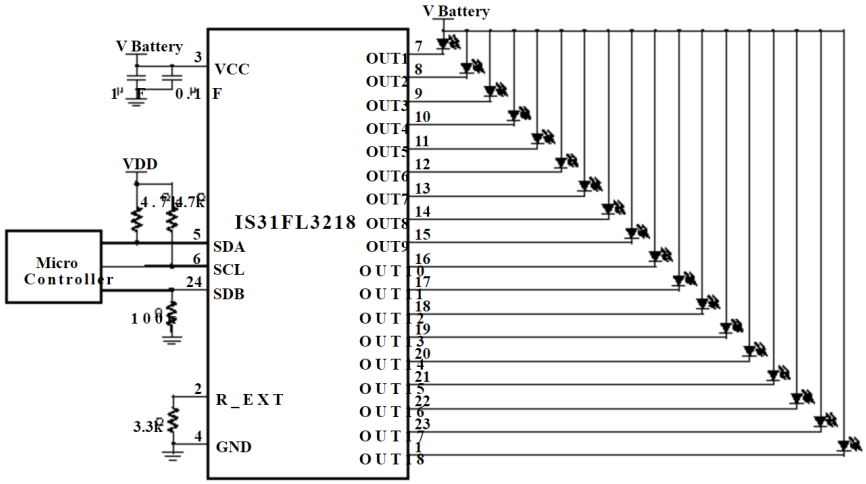


图 7: IS31FL3218 典型应用电路图

该驱动对 IS31FL3XXX 的基本操作进行了封装，示例如下节所示。

IS31FL3218 示例

```
i2c_bus_handle_t i2c_bus = NULL;
is31fl3218_handle_t fxled = NULL;
i2c_config_t conf = {
    .mode = I2C_MODE_MASTER,
    .sda_io_num = I2C_MASTER_SDA_IO,
    .sda_pullup_en = GPIO_PULLUP_ENABLE,
    .scl_io_num = I2C_MASTER_SCL_IO,
    .scl_pullup_en = GPIO_PULLUP_ENABLE,
    .master.clk_speed = I2C_MASTER_FREQ_HZ,
};
i2c_bus = i2c_bus_create(I2C_MASTER_NUM, &conf);
fxled = is31fl3218_create(i2c_bus);
is31fl3218_channel_set(fxled, 0x00ff, 128); // set PWM 1 ~ PWM 8 duty cycle 50%
is31fl3218_delete(fxled);
i2c_bus_delete(&i2c_bus);
```

3.3 LED 指示灯

[English]

LED 指示灯是最简单的输出外设之一，可以通过不同形式的闪烁指示系统当前的工作状态。ESP-IoT-Solution 提供的 LED 指示灯组件具有以下功能：

- 支持定义多组闪烁类型
- 支持定义闪烁类型优先级
- 支持创建多个指示灯
- LEDC 等驱动支持调节亮度，渐变

3.3.1 使用方法

预定义闪烁类型

闪烁步骤结构体*blink_step_t* 定义了该步骤的类型、指示灯状态和状态持续时间。多个步骤组合成一个闪烁类型，不同的闪烁类型可用于标识不同的系统状态。闪烁类型的定义方法如下：

例 1. 定义一个循环闪烁：亮 0.05 s，灭 0.1 s，开始之后一直循环。

```
const blink_step_t test_blink_loop[] = {
    {LED_BLINK_HOLD, LED_STATE_ON, 50},              // step1: turn on LED 50 ms
```

(下页继续)

(续上页)

```
    {LED_BLINK_HOLD, LED_STATE_OFF, 100},             // step2: turn off LED 100 ms
    {LED_BLINK_LOOP, 0, 0},                           // step3: loop from step1
};
```

例 2. 定义一个循环闪烁：亮 0.05 s，灭 0.1 s，亮 0.15 s，灭 0.1 s，执行完毕灯熄灭。

```
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)
};
```

例 3. 定义一个循环闪烁：渐亮 0.05s，逐灭 0.5s，执行完毕灯熄灭。（GPIO 模式不支持）

```
const blink_step_t test_blink_breathe[] = {
    {LED_BLINK_BREATHE, LED_STATE_ON, 500},              // step1: fade from off to␣
↪on 500ms
    {LED_BLINK_BREATHE, LED_STATE_OFF, 500},             // step2: fade from on to␣
↪off 500ms
    {LED_BLINK_BRIGHTNESS, 50, 500},                     // step3: set to half␣
↪brightness 500 ms
    {LED_BLINK_STOP, 0, 0},                              // step4: stop blink (50%␣
↪brightness)
};
```

定义闪烁类型之后，需要在 `led_indicator_blink_type_t` 添加该类型对应的枚举成员，然后将其添加到闪烁类型列表 `led_indicator_blink_lists`，示例如下：

```
typedef enum {
    BLINK_TEST_BLINK_ONE_TIME, /**< test_blink_one_time */
    BLINK_TEST_BLINK_LOOP,     /**< test_blink_loop */
    BLINK_MAX,                 /**< INVALIED type */
} led_indicator_blink_type_t;

blink_step_t const * led_indicator_blink_lists[] = {
    [BLINK_TEST_BLINK_ONE_TIME] = test_blink_one_time,
    [BLINK_TEST_BLINK_LOOP] = test_blink_loop,
    [BLINK_MAX] = NULL,
};
```

预定义闪烁优先级

对于同一个指示灯，高优先级闪烁可以打断正在进行的低优先级闪烁，当高优先级闪烁结束，低优先级闪烁恢复执行。可以通过调整闪烁类型 `led_indicator_blink_type_t` 枚举成员的顺序调整闪烁的优先级，数值越小的成员执行优先级越高。

例如，在以下示例中闪烁 `test_blink_one_time` 比 `test_blink_loop` 优先级高，可优先闪烁：

```
typedef enum {
    BLINK_TEST_BLINK_ONE_TIME, /**< test_blink_one_time */
    BLINK_TEST_BLINK_LOOP,     /**< test_blink_loop */
    BLINK_MAX,                 /**< INVALIED type */
} led_indicator_blink_type_t;
```

控制指示灯闪烁

创建一个指示灯：指定一个 IO 和一组配置信息创建一个指示灯

```
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
```

开始/停止闪烁：控制指示灯开启/停止指定闪烁类型，函数调用后立刻返回，内部由定时器控制闪烁流程。同一个指示灯可以开启多种闪烁类型，将根据闪烁类型优先级依次执行。

```
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
```

删除指示灯：您也可以在不需要进一步操作时，删除指示灯以释放资源

```
led_indicator_delete(&led_handle);
```

抢占操作：您可以在任何时候直接闪烁指定的类型。

```
led_indicator_preempt_start(led_handle, BLINK_TEST_BLINK_LOOP);
```

停止抢占：您可以使用停止抢占函数，来取消正在抢占的闪烁模式。

```
led_indicator_preempt_stop(led_handle, BLINK_TEST_BLINK_LOOP);
```

注解: 该组件支持线程安全操作，您可使用全局变量共享 LED 指示灯的操作句柄 `led_indicator_handle_t`，也可以使用*led_indicator_get_handle* 在其它线程通过 LED 的 IO 号获取句柄以进行操作。

自定义指示灯闪烁

```
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,
};
```

通过定义 `led_blink_lst[]` 实现自定义指示灯。

gamma 曲线调光

人眼感知亮度的方式并非线性，而是具有一定的非线性特征。在标准情况下，人眼对较暗的区域更敏感，对较亮的区域不太敏感。然而，在数字显示设备（如显示器）上，图像的亮度值通常以线性方式编码。这就导致在将线性编码的亮度值转换为人眼感知的亮度时，图像会出现明暗失真或细节丢失的情况。为了解决这个问题，需要对图像进行 Gamma 校正。Gamma 校正是通过对亮度值进行非线性调整来纠正图像的显示。通过施加一个 Gamma 值（通常介于 2.2 至 2.4 之间），可以将线性编码的亮度值映射到更符合人眼感知的非线性亮度曲线上。这样可以提高暗部细节的可见性，并使图像在视觉上更加准确和平衡。

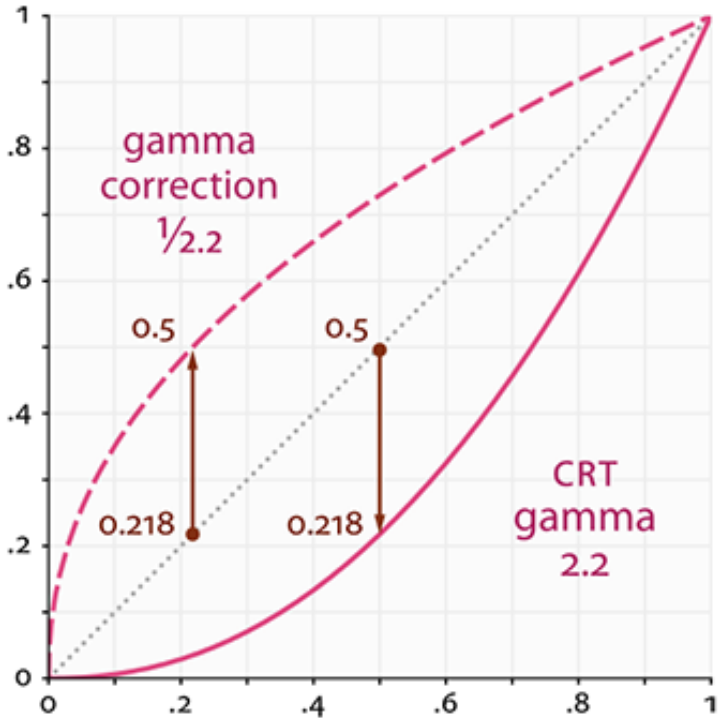


图 8: Gamma 曲线

```
float gamma = 2.3
led_indicator_new_gamma_table(gamma);
```

默认的 gamma 表是 2.3，可以通过 `led_indicator_new_gamma_table()` 生成新的 gamma 表。

3.3.2 API 参考

3.3.3 Header File

- led/led_indicator/include/led_indicator.h

3.3.4 Functions

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

create a LED indicator instance with GPIO number and configuration

Return led_indicator_handle_t handle of the LED indicator, NULL if create failed.

Parameters

- `config`: configuration of the LED, eg. GPIO level when LED off

led_indicator_handle_t **led_indicator_get_handle**(void **hardware_data*)

get the handle of created led_indicator with hardware data

Return led_indicator_handle_t handle of the created LED indicator, NULL if not created.

Parameters

- `hardware_data`: user hardware data for LED

esp_err_t **led_indicator_delete**(*led_indicator_handle_t handle*)

delete the LED indicator and release resource

Return esp_err_t

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK Success
- ESP_FAIL Delete fail

Parameters

- `handle`: pointer to LED indicator handle

esp_err_t **led_indicator_start**(*led_indicator_handle_t handle*, int *blink_type*)

start a new blink_type on the LED indicator. if mutiple blink_type started simultaneously, it will be executed according to priority.

Return esp_err_t

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NOT_FOUND no predefined blink_type found

- ESP_OK Success

Parameters

- `handle`: LED indicator handle
- `blink_type`: predefined blink type

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.

Return esp_err_t

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NOT_FOUND no predefined blink_type found
- ESP_OK Success

Parameters

- `handle`: LED indicator handle
- `blink_type`: predefined blink type

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().

Return esp_err_t

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

Parameters

- `handle`: LED indicator handle
- `blink_type`: predefined blink type

esp_err_t **`led_indicator_preempt_stop`** (*led_indicator_handle_t handle*, int *blink_type*)

Stop the current preemptive action.

Return esp_err_t

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

Parameters

- `handle`: LED indicator handle
- `blink_type`: predefined blink type

uint8_t **`led_indicator_get_current_fade_value`** (*led_indicator_handle_t handle*)

Get the current fade value of the LED indicator.

Return uint8_t Current fade value: 0-255 if handle is null return 0

Parameters

- `handle`: LED indicator handle

3.3.5 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

uint8_t **`value`**

hold on or off, set NULL if LED_BLINK_STOP or LED_BLINK_LOOP

uint32_t **`hold_time_ms`**

hold time(ms), set NULL 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_custom_config_t ***`led_indicator_custom_config`**

LED custom configuration

`union` *led_indicator_config_t*::**[anonymous] [anonymous]**

LED configuration

blink_step_t **const** ****blink_lists**

user defined LED blink lists

uint16_t **blink_list_num**

number of blink lists

3.3.6 Type Definitions

typedef void ***led_indicator_handle_t**

LED indicator operation handle

3.3.7 Enumerations

enum [anonymous]

LED state: 0-100, only hardware that supports to set brightness can adjust brightness.

Values:

LED_STATE_OFF = 0

turn off the LED

LED_STATE_25_PERCENT = 64

25% brightness, must support to set brightness

LED_STATE_50_PERCENT = 128

50% brightness, must support to set brightness

LED_STATE_75_PERCENT = 191

75% brightness, must support to set brightness

LED_STATE_ON = UINT8_MAX

turn on the LED

enum blink_step_type_t

actions in this type

Values:

LED_BLINK_STOP = -1

stop the blink

LED_BLINK_HOLD

hold the on-off state

LED_BLINK_BREATHE

breathe state

LED_BLINK_BRIGHTNESS

set the brightness

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:

LED_GPIO_MODE

blink with max brightness

LED_LEDC_MODE

blink with LEDC driver

LED_CUSTOM_MODE

blink with custom driver

CHAPTER 4

USB 主机 & 设备

[English]

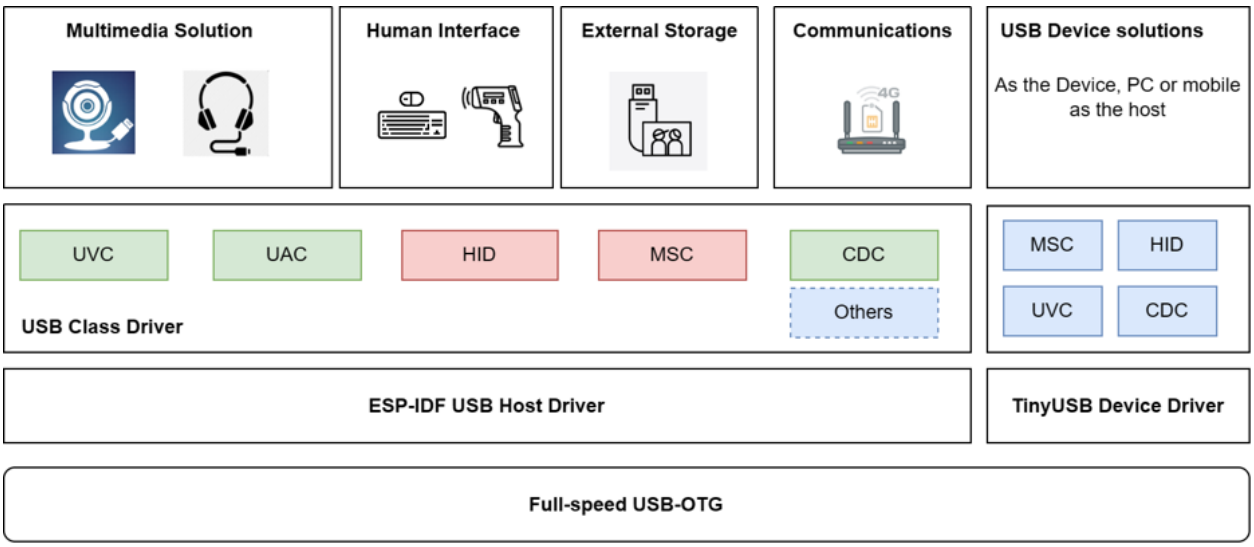
4.1 ESP USB 外设介绍

4.1.1 USB 简介

USB（Universal Serial Bus）是一种通用的总线标准，用于连接主机和外设设备。USB 主机可以通过 USB 接口与 USB 设备连接，实现数据传输、电源供给等功能。

USB IF（USB Implementers Forum）是 USB 标准的制定者，它制定了 USB 标准，包括 USB 1.1、USB 2.0、USB 3.0 等，定义了 USB 接口的物理层、数据链路层、传输层、会话层、表示层等协议，以及 USB 设备类（Device Class）标准，常见的设备类包括 HID（Human Interface Device，人机接口设备）、MSC（Mass Storage Class，大容量存储设备）、CDC（Communication Device Class，通信设备）、Audio、Video 等。

乐鑫 ESP32-S2/S3/C3 等芯片均已内置 USB-OTG 或 USB-Serial-JTAG 外设，支持各种各样的 USB 应用，包括 USB 多媒体类应用，USB 通信类应用，USB 存储类应用，USB 人机交互类应用等。



4.1.2 USB 电气属性

Type-A 接口的 USB 电气属性如下：

Pin	Name	Cable color	Description
1	VBUS	**Red**	+5V
2	D-	**White**	Data-（0 或 3.3V）
3	D+	**Green**	Data+（0 或 3.3V）
4	GND	**Black**	Ground

- 对于自供电设备，需要使用 1 个额外 IO 检测 VBUS 电压，用于检测设备是否拔出
- D- D+ 接反不会损坏硬件，但是主机将无法识别

4.1.3 USB-OTG Full-speed 控制器简介

USB OTG Full-speed 控制器是指同时具有 USB-OTG，USB Host 和 USB Device 模式的控制器，支持模式的协商和切换。支持 Full-speed (12Mbps) 和 Low-speed (1.5Mbps) 两种速率，支持 USB 1.1 和 USB 2.0 协议。

ESP-IDF 从 v4.4 开始已经包含 USB Host 和 USB Device 协议栈和各种设备类驱动，支持用户二次开发。

更多介绍，请参考USB-OTG 控制器介绍。

4.1.4 USB-Serial-JTAG 控制器简介

USB-Serial-JTAG Controller：同时具有 USB Serial 和 USB JTAG 功能的专用 USB 控制器，支持通过 USB 接口下载固件、打印 log、CDC 传输和 JTAG 调试，不支持修改 USB 功能、修改描述符等二次开发。

更多介绍，请参考USB-Serial-JTAG 控制器介绍。

4.1.5 USB Full-speed PHY 简介

USB Full-speed PHY：也称 USB Full-speed Transceiver，用于 USB Controller 数字信号到 USB 总线信号电平转换，提供总线驱动能力等。内部 USB Full-speed PHY 连接到外部固定 IO 引脚。

更多介绍，请参考USB-PHY 介绍。

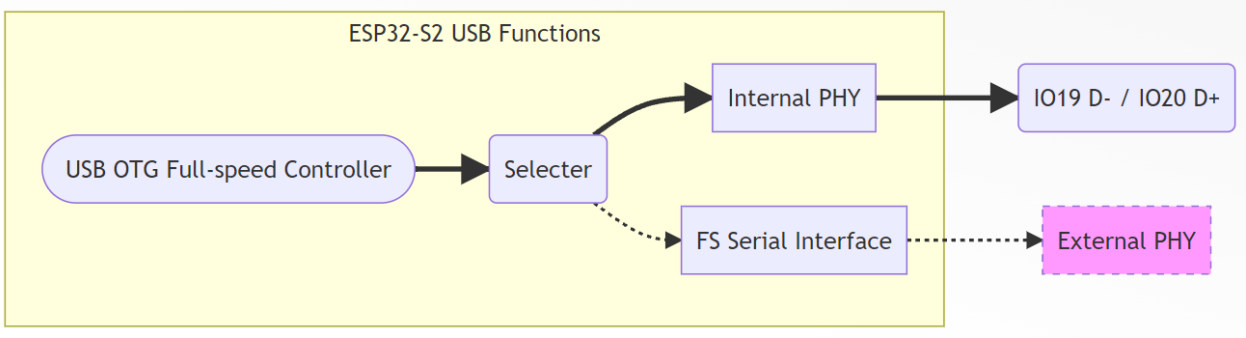
4.1.6 ESP32-S/C 系列 USB 外设支持情况

	USB OTG High-speed	USB OTG Full-speed	USB-Serial-JTAG	Fulls-peed 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

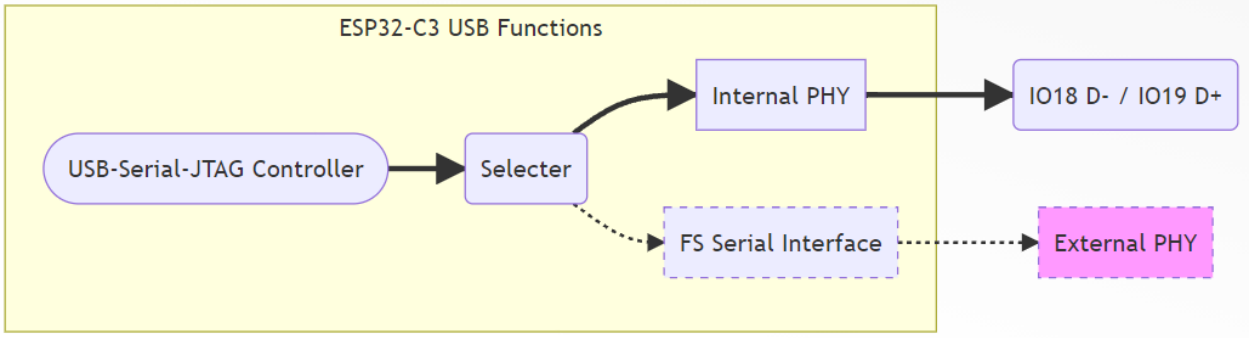
4.1.7 ESP32-S2 USB 功能简介

ESP32-S2 内置 **USB OTG Full-speed Controller** 和 **USB Full-speed PHY**，内部结构如下：



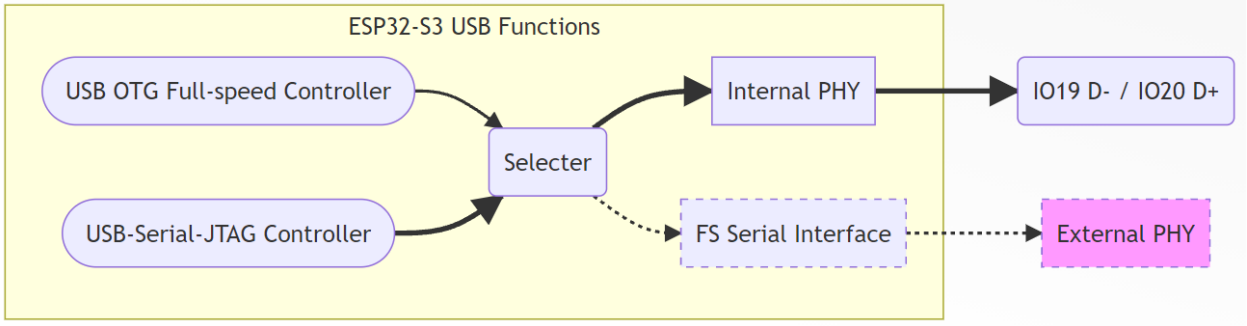
4.1.8 ESP32-C3 USB 功能简介

ESP32-C3 内置 **USB-Serial-JTAG Controller** 和 **USB Full-speed PHY**，内部结构如下:



4.1.9 ESP32-S3 USB 功能简介

ESP32-S3 内置两个 USB 控制器，分别是 **USB OTG Full-speed Controller** 和 **USB-Serial-JTAG Controller**，内置一个 **USB Full-speed PHY**。内部 USB PHY 默认连接到 **USB-Serial-JTAG** 控制器，可通过烧写 eFuse 修改默认，或配置寄存器动态切换，也可通过增加外部 PHY，同时启用两个控制器。内部 USB PHY 的切换详情，参考 USB PHY 切换。



4.2 USB-OTG 外设介绍

ESP32-S2/S3 等芯片内置 USB-OTG 外设，它包含了 USB 控制器和 USB PHY，支持通过 USB 线连接到 PC，实现 USB Host 和 USB Device 功能。

4.3 USB-OTG 传输速率

ESP32-S2/S3 USB-OTG Full Speed 总线传输速率为 12 Mbps，但由于 USB 传输存在一些校验和同步机制，实际的有效传输速率将低于 12 Mbps。具体数值和传输类型相关，如下表所示：

传输类型	**控制**	**中断**	**批量**	**同步**
适用场合	设备初始化和管理	鼠标和键盘	打印机和批量存储	流式音频和视频
支持低速	有	有	无	无
校验重传	有	有	有	无
保证传输速度	无	无	无	有
使用固定带宽	有（10%）	有（90%）	无	有（90%）
减少延迟时间	无	有	无	有
传输的最大尺寸	64 字节	64 字节	64 字节	~512 字节
每毫秒传输包数量	•	1	19	1
理论有效速率	•	64000 Bytes/s	1216000 Bytes/s	512000 Bytes/s

- 传输速率的计算公式为：传输速率 (Bytes/s) = 传输的最大尺寸 * 每毫秒传输包数量 * 1000
- 控制传输用于传输设备控制信息，包含多个阶段，有效传输速率需要按照协议栈的实现来计算。

4.4 USB-OTG 外设内置功能

4.4.1 使用 USB OTG Console 下载固件和打印 LOG

ESP32-S2/S3 等内置 USB-OTG 外设的芯片，ROM Code 中内置了 USB 通信设备类 (CDC) 的功能，该功能可用于替代 UART 接口，实现 Log、Console 和固件下载功能。

1. 由于 USB OTG Console 默认为关闭状态，如需使用它下载固件，需要通过以下方法完成初次下载：
    1. 在 `menuconfig` 中先使能 USB OTG Console 功能，然后编译固件
    2. 手动芯片的 Boot 控制引脚拉低，然后将芯片通过 USB 线连接到 PC，进入下载模式。PC 会发现新的串口设备，Windows 为 `COM*`，Linux 为 `/dev/ttyACM*`, MacOS 为 `/dev/cu*`。
    3. 使用 esptool 工具（或直接使用 idf.py flash）配置设备对应的串口号下载固件。

2. 初次下载完成以后，USB OTG Console 功能将自动使能，即可通过 USB 线连接到 PC，PC 会发现新的串口设备，Windows 为 `COM*`，Linux 为 `/dev/ttyACM*`, MacOS 为 `/dev/cu*`，LOG 数据将从该虚拟串口打印。
3. 用户无需再手动拉低 Boot 控制引脚，使用 esptool 工具（或直接使用 idf.py flash）配置设备对应的串口号即可下载固件，下载期间，esptool 通过 USB 控制协议自动将设备 Reset 并切换到下载模式。

   更多详细信息，请参考：USB OTG Console

4.4.2 使用 USB OTG DFU 下载固件

ESP32-S2/S3 等内置 USB-OTG 外设的芯片，ROM Code 中内置了 USB DFU（Device Firmware Upgrade）功能，可用于实现标准的 DFU 下载模式。

1. 使用 DFU 下载固件，用户每次都需要手动进入下载模式，将芯片的 Boot 控制引脚拉低，然后通过 USB 线连接到 PC。
2. 在工程目录下运行指令 `idf.py dfu` 生成 DFU 固件，然后使用 `idf.py dfu-flash` 下载固件。
3. 如果存在多个 DFU 设备，用户可以使用 `idf.py dfu-list` 查看 DFU 设备列表，然后使用 `idf.py dfu-flash --path <path>` 指定下载端口。

   更多详细信息，请参考：Device Firmware Upgrade via USB

4.5 使用 USB-OTG 外设进行 USB Host 开发

USB-OTG 外设支持 USB Host 功能，用户可以通过 USB 接口直接连接到外部 USB 设备。ESP-IDF 从 v4.4 版本开始，已经支持 USB Host Driver，用户可以参考 ESP-IDF USB Host，开发 USB Class Driver。

此外乐鑫也已经官方支持 USB Host HID，USB Host MSC，USB Host CDC，USB Host UVC 等设备类驱动，用户可以直接使用这些驱动进行应用开发。

USB Host 方案详情，请参考 USB Host Solution。

4.6 使用 USB-OTG 外设进行 USB Device 开发

USB-OTG 外设支持 USB Device 功能，乐鑫已经官方适配了 TinyUSB 协议栈，用户可以直接使用基于 TinyUSB 开源协议栈开发的 USB 标准设备或自定义设备，例如 HID，MSC，CDC，ECM, UAC 等。

USB Device 方案详情，请参考 USB Device Solution。

4.7 USB-Serial-JTAG 外设介绍

ESP32-S3/C3 等芯片内置 USB-Serial-JTAG 外设，它包含了 USB-to-serial 转换器和 USB-to-JTAG 转换器，支持通过 USB 线连接到 PC，实现固件下载、调试和打印系统 LOG 等功能。USB-Serial-JTAG 外设的内部结构可参考 ESP32-C3 技术参考手册-USB Serial/JTAG Controller。

4.8 USB-Serial-JTAG 外设驱动

- Linux 和 MacOS 系统下，无需手动安装驱动
- Windows 10 及以上系统，联网将自动安装驱动
- Windows 7/8 系统，需要手动安装驱动，驱动下载地址：esp32-usb-jtag-2021-07-15。用户也可以使用 ESP-IDF Windows Installer，勾选 USB-Serial-JTAG 驱动进行安装，

4.9 USB-Serial-JTAG 外设内置功能

USB-Serial-JTAG 外设接入 PC 后，设备管理器将新增两个设备：

Windows 如下图所示：

Linux 如下图所示：

```
random    tty0   tty2   tty30  tty41  tty52  tty63
rfkill    tty1   tty20  tty31  tty42  tty53  tty7
rtc       tty10  tty21  tty32  tty43  tty54  tty8
rtc0      tty11  tty22  tty33  tty44  tty55  tty9
serial    tty12  tty23  tty34  tty45  tty56  ttyACM0
shm       tty13  tty24  tty35  tty46  tty57  ttyprintk
snapshot  tty14  tty25  tty36  tty47  tty58  ttyS0
snd       tty15  tty26  tty37  tty48  tty59  ttyS1
stderr    tty16  tty27  tty38  tty49  tty6   ttyS10
stdin     tty17  tty28  tty39  tty5   tty60  ttyS11
stdout    tty18  tty29  tty4   tty50  tty61  ttyS12
tty       tty19  tty3   tty40  tty51  tty62  ttyS13
```

4.9.1 使用 USB-Serial-JTAG 下载固件

- 默认情况下，USB-Serial-JTAG 下载功能处于使能状态，可以直接使用 USB 线连接到 PC，然后使用 esptool 工具（或直接使用 idf.py flash）配置 USB-Serial-JTAG 设备对应的串口号（Windows 为 `COM*`，Linux 为 `/dev/ttyACM*`, MacOS 为 `/dev/cu*`）下载固件。下载期间，esptool 通过 USB 控制协议自动将设备 Reset 并切换到下载模式。
- 如果在应用程序中将 USB-Serial-JTAG 对应的 USB 引脚用作了其它功能，例如用作普通 GPIO 或其它外设 IO，USB-Serial-JTAG 将无法与 USB 主机建立连接，因此无法通过 USB 将设备切换到下载模式，用户必须通过 Boot 控制引脚，将设备手动切换到下载模式，然后再使用 esptool 下载固件。
- 为了避免在应用程序中将 USB-Serial-JTAG 对应的 USB 引脚用作其它功能，导致无法通过 USB 自动进入下载模式，用户需要在硬件设计时，引出 Boot 控制引脚。
- 默认情况下，通过 USB 接口下载不同的芯片，COM 号将递增，可能对量产造成不便，用户可参考 阻止 Windows 依据 USB 设备序列号递增 COM 编号。

4.9.2 使用 USB-Serial-JTAG 调试代码

`USB-Serial-JTAG` 支持通过 `JTAG` 接口调试代码，用户仅需使用 USB 线连接到 PC，然后使用 `OpenOCD` 工具即可调试代码。配置方法请参考 配置 ESP32-C3 内置 JTAG 接口。

4.9.3 使用 USB-Serial-JTAG 打印系统 LOG

- 用户可通过 `menuconfig-> Component config → ESP System Settings → Channel for console secondary output` 配置 USB-Serial-JTAG LOG 功能的使能状态。
- LOG 功能使能以后，可以直接使用 USB 线连接到 PC，然后使用 `idf.py monitor` 或其它串口工具打开 USB-Serial-JTAG 设备对应的串口号（Windows 为 `COM*`，Linux 为 `/dev/ttyACM*`, MacOS 为 `/dev/cu*`），即可打印系统 LOG。
- `USB-Serial-JTAG` 仅在主机接入后才会打印 LOG，如果主机未接入，`USB-Serial-JTAG` 不会被初始化，也不会打印 LOG。
- `USB-Serial-JTAG` LOG 功能无法在睡眠模式下使用（包括 deep sleep 和 light sleep 模式），如果需要在睡眠模式下打印 LOG，可以使用 `UART` 接口。

4.9.4 使用 USB-Serial-JTAG 引脚作为普通 GPIO

如果用户需要在应用程序中将 USB-Serial-JTAG 对应的 USB 引脚用作其它功能，例如用作普通 GPIO。需要注意 USB D+ 接口默认上拉 USB 电阻，该电阻会导致 USB D+ 引脚常为高电平，因此需要在用作 GPIO 时将其禁用。

- ESP-IDF v4.4 及以后版本 GPIO 驱动默认会禁用 USB D+ 上拉电阻，用户使用 GPIO 驱动时无需额外配置。
- 用户也可以修改寄存器值 `USB_SERIAL_JTAG.conf0.dp_pullup = 0;` 将 USB D+ 上拉电阻禁用。

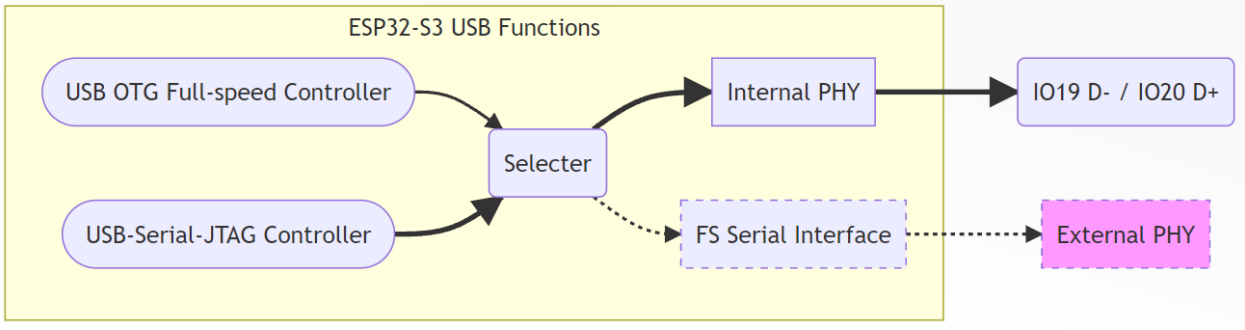
需要特别注意的是 USB D+ 引脚的上拉电阻在上电时刻即存在，用户在调用软件禁用上拉电阻之前，USB D+ 引脚已经被拉高，导致 D+ 引脚做为 GPIO 时，初始阶段为高电平，如果用户需要在上电后 USB D+ 引脚立即为低电平，需要在硬件设计时，将 USB D+ 引脚通过外部电路拉低。

4.10 USB PHY/Transceiver 介绍

USB Full-speed PHY/Transceiver 的功能是将 USB 控制器的数字信号转换为 USB 总线信号电平，提供总线驱动能力，检测接收错误等。ESP32-S2/S3 等芯片已内置一个 USB Full-speed PHY，用户可直接使用芯片指定的 USB D+ D- 与外部 USB 系统通信。此外，ESP32-S2/S3 还保留了外部 PHY 的扩展接口，用户可在需要时连接外部 PHY。

4.11 使用内部 PHY

ESP32-S2/S3/C3 内部集成了 USB PHY，因此无需外接 PHY 芯片，可以直接与外部 USB 主机或设备通过 USB D+/D- 连接。但对于集成两个 USB 控制器的芯片，例如 ESP32-S3 内置 USB-OTG 和 USB-Serial-JTAG，两者共用一个内部 PHY，同一时间只能有一个工作。



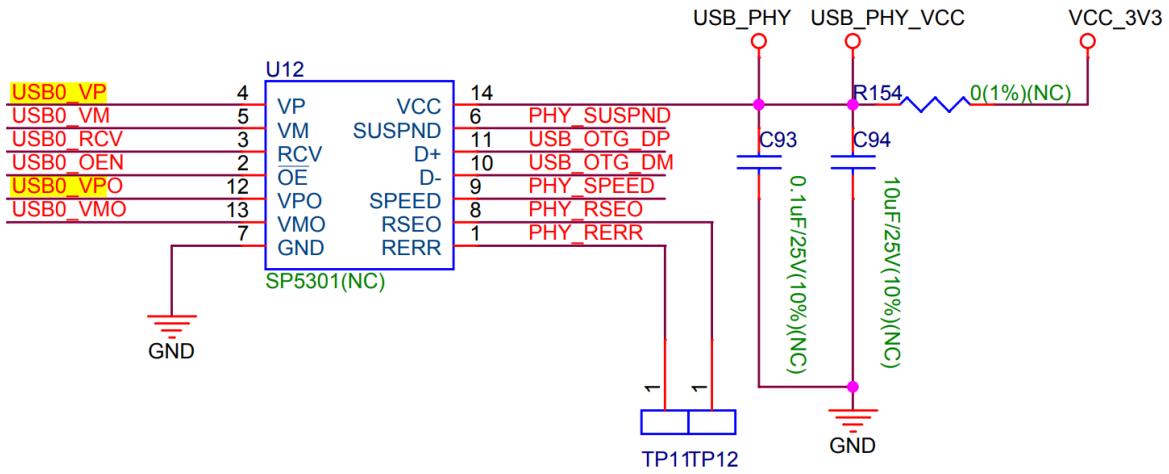
内部 USB-PHY 对应固定的 GPIO，如下表所示：

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

4.12 使用外部 PHY

通过增加一个外部 PHY，可以实现 USB-OTG 和 USB-Serial-JTAG 两个外设同时工作。

ESP32S2/S3 支持 SP5301 或同等功能的 USB PHY。外部 PHY 的典型电路图如下：



使用外部 USB PHY，将占用至少 6 个 GPIO

4.13 USB PHY 默认配置

1. 对于同时具有 USB-OTG 和 USB-Serial-JTAG 两个外设的芯片，默认情况下 USB-Serial-JTAG 与内部 USB-PHY 连接。用户可以直接通过 USB 接口进行下载或调试，无需额外配置。
2. 如需使用 USB Host Driver 或 TinyUSB 协议栈开发 USB-OTG 应用，在协议栈初始化时，USB-PHY 连接会自动切换为 USB-OTG，用户不必再进行配置。在 USB-OTG 模式下，如果用户需要使用 USB-Serial-JTAG 的下载功能，需要手动 Boot 到下载模式。

4.14 修改 USB PHY 默认配置

方法 1：通过配置寄存器，将 USB-PHY 连接切换为 USB-OTG。

- USB Host Driver 或 TinyUSB 协议栈内部通过配置 USB PHY 寄存器，将内部 USB-PHY 连接切换为 USB-OTG，如需了解更多信息，请参考 USB PHY 配置 API。

方法 2：通过烧写 efuse usb_phy_sel 位为 1，将 USB-PHY 默认连接切换为 USB-OTG：

- 仅当用户需要在 Boot 模式下使用 USB-OTG 功能时，才需要烧写此 efuse 位。烧写完成后，芯片进入 Boot 模式后，将使用 USB-OTG 提供的下载功能，例如 USB DFU。
- 注意，烧写 efuse 为 1 后不可恢复为 0。当 USB-PHY 默认连接切换为 USB-OTG，芯片进入 Boot 模式后，将使用 USB-OTG 功能，USB-Serial-JTAG 功能将无法使用。
- 注意：对于在 DateCode 2219 生产的 ESP32-S3 模组和开发板 (PW No. 早于 PW-2022-06-XXXX)，由于 EFUSE_DIS_USB_OTG_DOWNLOAD_MODE (BLK0 B19[7]) 已经被烧录为 1（USB OTG 下载被禁用），

用户如果再烧写 efuse_usb_phy_sel 位为 1，将导致芯片进入 Boot 模式后，USB-Serial-JTAG 和 USB-OTG 下载功能均无法使用。

4.15 USB VID 和 PID

VID 和 PID 是 USB 设备的唯一标识符，用于区分不同的 USB 设备。一般 VID 和 PID 由 USB-IF 分配，USB-IF 是 USB 设备的标准制定者。

4.15.1 以下情况您可以免申请 VID 和 PID

- 如果您的产品使用的是 USB Host 模式，您不需要申请 VID 和 PID。
- 如果您的产品使用的是 USB Device 模式，准备使用乐鑫的 VID（0x303A），并且基于 TinyUSB 协议栈开发 USB 标准设备，那您不需要申请 PID，使用 TinyUSB 默认 PID 即可。

4.15.2 申请 VID 和 PID

如果您使用的产品需要使用 USB 设备模式，您可按照以下过程申请 VID (Vendor ID) 或 PID (Product ID)。

- 如果您的产品需要使用 USB-IF 分配的 VID，您需要先 注册成为 USB-IF 成员，然后按照 USB-IF 的流程申请 VID 和 PID。
- 如果您的产品考虑使用乐鑫的 VID，您可以直接申请 PID（免费），申请流程请参考 申请乐鑫 PID。

注意：使用乐鑫的 VID 和 TinyUSB 的默认 PID, 并不意味着您的产品符合 USB 规范，您仍然需要进行 USB 认证。

4.16 USB 认证

USB 认证是由 USB Implementers Forum（USB-IF）进行管理的，旨在确保产品符合 USB 规范，以保证设备之间的互操作性和兼容性。

USB 认证是一个可选的过程，但在以下情况下必须进行产品认证：

- 如果产品标榜自己符合 USB 规范并使用官方 USB 标志。
- 如果产品打算使用 USB 标志、商标或宣传资料中提及 USB 认证。

具体认证流程和要求，请查阅 https://www.usb.org 或联系 USB-IF 授权测试实验室。

4.17 USB Host 方案

ESP32-S2/S3 等芯片内置 USB-OTG 外设，支持通过 USB 接口连接多种多样的 USB 设备。以下介绍了 ESP32-S2/S3 芯片支持的 USB Host 解决方案。

4.17.1 ESP USB Camera 视频方案

支持通过 USB 接口连接摄像头模组，实现 MJPEG 格式视频流获取和传输，最高可支持 480*800@15fps。适用于猫眼门铃、智能门锁、内窥镜、倒车影像等场景。

特性:

- 快速启动
- 支持热插拔
- 支持 UVC1.1/1.5 规范的摄像头
- 支持自动解析描述符
- 支持动态配置分辨率
- 支持 MJPEG 视频流传输
- 支持批量和同步两种传输模式

硬件:

- 芯片：ESP32-S2，ESP32-S3
- 外设：USB-OTG
- USB 摄像头：支持 MJPEG 格式，批量传输模式下 800*480@15fps，同步传输模式下 480*320@15fps，摄像头限制详见 usb_stream API 说明

链接:

- usb_stream 组件
- usb_stream API 说明
- USB Camera Demo 视频
- 示例代码: USB 摄像头 + WiFi 图传 usb/host/usb_camera_mic_spk
- 示例代码: USB 摄像头 + LCD 本地刷屏 usb/host/usb_camera_lcd_display

4.17.2 ESP USB Audio 音频方案

支持通过 USB 接口连接 USB 音频设备，实现 PCM 格式音频流获取和传输，可同时支持多路 48KHz 16bit 扬声器和多路 48KHz 16bit 马克风。支持 Type-C 接口耳机，适用于音频播放器等场景。支持和 UVC 同时工作，适用于门铃对讲等场景。

特性:

- 快速启动
- 支持热插拔
- 支持自动解析描述符
- 支持 PCM 音频流传输
- 支持动态修改采样率
- 支持多通道扬声器
- 支持多通道麦克风
- 支持音量、静音控制
- 支持和 USB Camera 同时工作

硬件:

- 芯片：ESP32-S2，ESP32-S3
- 外设：USB-OTG
- USB 音频设备：支持 PCM 格式

链接:

- usb_stream 组件
- usb_stream API 说明
- USB Audio Demo 视频
- 示例代码: MP3 音乐播放器 + USB 耳机 usb/host/usb_audio_player

4.17.3 ESP USB 4G 联网方案

支持通过 USB 接口连接 4G Cat.1，Cat.4 模组，实现 PPP 拨号上网。支持通过 Wi-Fi SoftAP 热点共享互联网给其它设备。适用于物联网网关、MiFi 移动热点、智慧储能、广告灯箱等场景。

特性:

- 快速启动
- 支持热插拔
- 支持 Modem+AT 双接口（需要模组支持）
- 支持 PPP 标准协议（大部分 4G 模组均支持）
- 支持 4G 转 Wi-Fi 热点
- 支持 NAPT 网络地址转换
- 支持电源管理
- 支持网络自动恢复
- 支持卡检测、信号质量检测
- 支持网页配置界面

硬件:

- 芯片：ESP32-S2，ESP32-S3
- 外设：USB-OTG
- 4G 模组：支持 Cat.1 Cat.4 等网络制式 4G 模组，需要模组支持 PPP 协议

链接:

- USB 4G Demo 视频
- iot_usbh_modem 组件
- 示例代码: 4G Wi-Fi 路由器 usb/host/usb_cdc_4g_module

4.17.4 ESP USB 存储方案

支持通过 USB 接口连接标准 U 盘设备（兼容 USB3.1/3.0/2.0 协议 U 盘），支持将 U 盘挂载到 FatFS 文件系统，实现文件的读写。适用于户外广告灯牌、考勤机、移动音响、记录仪等应用场景。

特性:

- 兼容 USB3.1/3.0/2.0 U 盘
- 默认支持最大 32G
- 支持热插拔
- 支持 Fat32/exFat 格式
- 支持文件系统读写
- 支持 U 盘 OTA

硬件:

- 芯片：ESP32-S2，ESP32-S3
- 外设：USB-OTG
- U 盘：格式化为 Fat32 格式，默认支持 32GB 以内 U 盘。大于 32GB 需要在文件系统开启 exFat

链接:

- usb_host_msc 组件
- 示例代码: 挂载 U 盘 + 文件系统访问

4.18 USB Device 方案

4.19 自供电 USB 设备解决方案

按照 USB 协议要求，USB 自供电设备必须通过检测 5V VBUS 电压来判断设备是否拔出，进而实现热插拔。对于主机供电设备，由于主机 VBUS 断电之后，设备直接掉电关机，无需实现该逻辑。

USB 设备 VBUS 检测方法一般有两种方法：由 USB PHY 硬件检测，或**借助 ADC/GPIO 由软件检测**。

由于 ESP32S2/S3 内部 USB PHY 不支持硬件检测逻辑，该功能需要借助 ADC/GPIO 由软件实现，其中使用 GPIO 检测方法最为简便，实现方法如下：

对于 ESP-IDF 4.4 及更早版本:

1. 硬件上，需要额外占用一个 IO（任意指定，特殊引脚除外），通过两个电阻分压（例如两个 100KΩ ）与 ESP32S2/S3 相连 (ESP32S2/S3 IO 最大可输入电压为 3.3v)；
2. 在 `tinyusb_driver_install` 之后，需要调用 `usbd_vbus_detect_gpio_enable` 函数使能 VBUS 检测，该函数实现代码如下，请直接复制到需要调用的位置：

```
/**
 *
 * @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 = {
        .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;
}
```

对于 ESP-IDF 5.0 及以上版本:

1. 同上，硬件上需要额外占用一个 IO（任意指定，特殊引脚除外），通过两个电阻分压（例如两个 100KΩ ）与 ESP32S2/S3 相连；
2. 将用于检测 VBUS 的 IO 初始化为 GPIO 输入模式;

3. 直接将 IO 配置到 `tinyusb_config_t` 中（详情可参考）：

```
#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));
```

4.20 阻止 Windows 依据 USB 设备序列号递增 COM 编号

由于任何连接到 Windows PC 的设备都通过其 VID、PID 和 Serial 号进行识别。如果这 3 个参数中的任何一个发生了变化，那么 PC 将检测到新的硬件，并与该设备关联一个不同的 COM 端口，详情请参考 Windows USB device registry entries。

ESP ROM Code 中对 USB 描述符配置如下：

	ESP32S2	ESP32S3	ESP32C3
VID	0x303a	0x303a	0x303a
PID	0x0002	0x1001	0x1001
Serial	0	MAC 地址字符串	MAC 地址字符串

- ESP32S2（usb-otg）Serial 为常量 0，每个设备相同，COM 号一致
- ESP32S3（usb-serial-jtag）、ESP32C3（usb-serial-jtag）Serial 为设备 MAC 地址，每个设备均不同，COM 号默认递增

递增 COM 编号，为量产烧录带来额外工作。对于需要使用 USB 进行固件下载的客户，建议修改 Windows 的递增 COM 编号的规则，阻止依据 Serial 号递增编号。

4.21 解决方案

管理员方式打开 `Windows CMD`，执行以下指令。该指令将添加注册表项，阻止依据 Serial 号递增编号，**设置完成后请重启电脑使能修改**：

```
REG ADD HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\usbflags\303A10010101 /V␣
↪IgnoreHWSerNum /t REG_BINARY /d 01
```

用户也可下载 `ignore_hwserial_esp32s3c3.bat` 脚本，右键选择管理员方式运行。

4.22 USB Stream 组件说明

[English]

`usb_stream` 是基于 ESP32-S2/ESP32-S3 的 USB UVC + UAC 主机驱动程序，支持从 USB 设备读取/写入/控制多媒体流。例如最多同时支持 1 路摄像头 + 1 路麦克风 + 1 路播放器数据流。

特性：

1. 支持通过 UVC Stream 接口获取视频流，支持批量和同步两种传输模式
2. 支持通过 UAC Stream 接口获取麦克风数据流，发送播放器数据流
3. 支持通过 UAC Control 接口控制麦克风音量、静音等特性
4. 支持自动解析设备配置描述符
5. 支持对数据流暂停和恢复

4.22.1 USB Stream 用户指南

- 开发板
  1. 可以使用任何带有 USB 接口的 ESP32-S2/ESP32-S3 开发板，注意该 USB 接口需要能够向外供电
- USB UVC 功能
  1. 摄像头必须兼容 `USB1.1` 全速（Fullspeed）模式
  2. 摄像头需要自带 `MJPEG` 压缩
  3. 用户可通过*`uvc_streaming_config()`* 函数手动指定相机接口、传输模式和图像帧参数
  4. 如使用同步传输摄像头，需要支持设置接口 MPS（Max Packet Size）为 512
  5. 同步传输模式，图像数据流 USB 传输总带宽应小于 4 Mbps （500 KB/s）
  6. 批量传输模式，图像数据流 USB 传输总带宽应小于 8.8 Mbps （1100 KB/s）
  7. 其它特殊相机要求，请参考示例程序 README.md

- USB UAC 功能
  1. 音频功能必须兼容 `UAC 1.0` 协议
  2. 用户需要通过*`uac_streaming_config()`* 函数手动指定 spk/mic 采样率，位宽参数
- USB UVC + UAC 功能
  1. UVC 和 UAC 功能可以单独启用，例如仅配置 UAC 来驱动一个 USB 耳机，或者仅配置 UVC 来驱动一个 USB 摄像头
  2. 如需同时启用 UVC + UAC，该驱动程序目前仅支持具有摄像头和音频接口的复合设备（`Composite Device`），不支持同时连接两个单独的设备

4.22.2 USB Stream API 参考

1. 用户可通过*`uvc_config_t`* 配置摄像头分辨率、帧率参数。通过*`uac_config_t`* 配置音频采样率、位宽等参数。参数说明如下:

```
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 frequence, Hz
    .spk_bit_resolution = 16, //speaker resolution, bits
    .spk_samples_frequence = 16000, //speaker frequence, 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. 使用*uvc_streaming_config()* 配置 UVC 驱动，如果设备同时支持音频，可使用*uac_streaming_config()* 配置 UAC 驱动
3. 使用*usb_streaming_start()* 打开数据流，之后驱动将响应设备连接和协议协商
4. 之后，主机将根据用户参数配置，匹配已连接设备的描述符，如果设备无法满足配置要求，驱动将以警告提示
5. 如果设备满足用户配置要求，主机将持续接收 IN 数据流（UVC 和 UAC 麦克风），并在新帧准备就绪后调用用户的回调
    1. 接收到新的 MJPEG 图像后，将触发 UVC 回调函数。用户可在回调函数中阻塞，因为它在独立的任务上下文中工作
    2. 接收到 `mic_min_bytes` 字节数据后，将触发 mic 回调。但是这里的回调一定不能以任何方式阻塞，否则会影响下一帧的接收。如果需要对 mic 进行阻塞操作，可以通过 `uac_mic_streaming_read` 轮询方式代替回调方式
6. 发送扬声器数据时，用户可通过*uac_spk_streaming_write()* 将数据写入内部 ringbuffer，主机将在 USB 空闲时从中获取数据并发送 OUT 数据
7. 使用*usb_streaming_control()* 控制流挂起/恢复。如果 UAC 支持特性单元，可通过其分别控制麦克风和播放器的音量和静音
8. 使用*usb_streaming_stop()* 停止 USB 流，USB 资源将被完全释放。

4.22.3 Bug 报告

ESP32-S2 ECO0 芯片 SPI 屏幕和 USB 同时启用，可能导致屏幕抖动

1. 在最早版本的 ESP32-S2 ECO0 芯片上, USB 可能污染 SPI 数据 (ESP32-S2 新版本 >=ECO1 和 ESP32-S3 均不存在该 Bug)
2. 软件解决方案

- `spi_ll.h` 添加以下接口

```
//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;
}
```

- 如下在 `spi_new_trans` 中添加额外检查

```
// 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)
```

(下页继续)

(续上页)

```
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);
}
```

4.22.4 Examples

usb/host/usb_camera_mic_spk

4.22.5 API Reference

Header File

- 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.

Return 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

Parameters

- `config`: parameters defined in uvc_config_t

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.

Return 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

Parameters

- `config`: parameters defined in uvc_config_t

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.

Return 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 resourse will be free.

Return 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.

Return esp_err_t ESP_ERR_INVALID_STATE: usb streaming not started ESP_ERR_TIMEOUT: timeout ESP_OK: device connected

Parameters

- `timeout_ms`: timeout in ms

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.

Return esp_err_t

- ESP_OK Success
- ESP_ERR_INVALID_STATE USB streaming is running, callback need register before start

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.

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.

Return 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

Parameters

- `stream`: stream type defined in usb_stream_t
- `ctrl_type`: control type defined in stream_ctrl_t
- `ctrl_value`: control value

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.

Return 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

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.

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.

Return 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

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.

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 frequence. 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.

Return esp_err_t

- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE USB device not active
- ESP_OK Success

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

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 frequence, please reset when the streaming in suspend state. The new configs will be effective after streaming resume.

Return 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

Parameters

- `stream`: stream type
- `ch_num`: audio channel numbers
- `bit_resolution`: audio bit resolution
- `samples_frequence`: audio samples frequence

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.

Return esp_err_t ESP_ERR_INVALID_ARG parameter error ESP_ERR_INVALID_STATE uvc stream not config or not active ESP_OK succeed

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

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.

Return esp_err_t

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

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 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 of MJPEG

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

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 frequence(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 frequence(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:

UVC_XFER_ISOC = 0

Isochronous Transfer Mode

UVC_XFER_BULK

Bulk Transfer Mode

UVC_XFER_UNKNOWN

Unknown Mode

enum usb_stream_t

Stream id, used for control.

Values:

STREAM_UVC = 0

usb video stream

STREAM_UAC_SPK

usb audio speaker stream

STREAM_UAC_MIC

usb audio microphone stream

STREAM_MAX

max stream id

enum usb_stream_state_t

USB device connection status.

Values:

STREAM_CONNECTED = 0

STREAM_DISCONNECTED

enum stream_ctrl_t

Stream control type, which also depends on if device support.

Values:

CTRL_NONE = 0

None

CTRL_SUSPEND

streaming suspend control. ctrl_data NULL

CTRL_RESUME

streaming resume control. ctrl_data NULL

CTRL_UAC_MUTE

mute control. ctrl_data (false/true)

CTRL_UAC_VOLUME

volume control. ctrl_data (0~100)

CTRL_MAX

max type value

CHAPTER 5

音频

[English]

5.1 PWM 音频

[English]

支持的芯片	ESP32	ESP32-S2	ESP32-S3	ESP32-C3

PWM 音频功能使用 ESP32 内部的 LEDC 外设产生 PWM 播放音频，无需使用外部的音频 Codec 芯片，适用于对音质要求不高而对成本敏感的应用。

5.1.1 特性

- 允许使用任意具有输出功能的 GPIO 作为音频输出管脚
- 支持 8 ~ 10 位的 PWM 分辨率
- 支持立体声
- 支持 8 ~ 48 KHz 采样率

5.1.2 结构

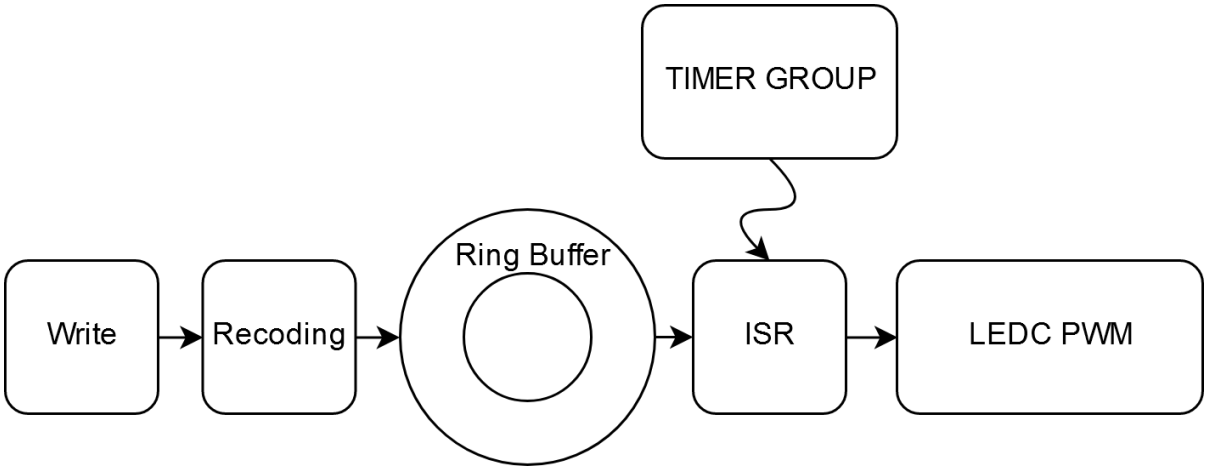


图 1: 结构

1. 首先对写入的数据进行编码调整以满足 PWM 的输入要求，包括数据的移位和偏移；
2. 通过一个 Ring Buffer 将数据发送到 Timer Group 的 ISR (Interrupt Service Routines) 函数；
3. Timer Group 按照设定的采样率从 Ring Buffer 读出数据并写入到 LEDC。

注解： 由于输出的是 PWM 信号，需要将输出进行低通滤波后才能得到音频波形。

5.1.3 PWM 频率

输出 PWM 的频率不支持直接配置，需通过配置 PWM 分辨率位数来计算，计算公式如下：

$$f_{pwm} = \frac{f_{APB\_CLK}}{2^{res\_bits}} - \left(\frac{f_{APB\_CLK}}{2^{res\_bits}} MOD 1000\right)$$

其中 f_{APB_CLK} 为 80 MHz，res_bits 为 PWM 分辨率位数，当分辨率为 `LEDC_TIMER_10_BIT` 时，PWM 频率为 78 KHz。更高的 PWM 频率和分辨率可以更好地还原音频信号，但是从公式可以看出，增大 PWM 频率意味着分辨率降低，提高分辨率意味着频率减小，可根据实际应用调整达到良好的平衡。

5.1.4 应用示例

```
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;
```

(下页继续)

(续上页)

```
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␣
↪numner */
pwm_audio_start();                  /**< Start to run */

while(1) {

    //< Perpare audio data, such as decode mp3/wav file

    /**< Write data to paly */
    pwm_audio_write(audio_data, length, &written, 1000 / portTICK_PERIOD_MS);
}
```

5.1.5 API 参考

Header File

- 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.

Return

- 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

Parameters

- `cfg`: configurations - see *pwm_audio_config_t* struct

esp_err_t **pwm_audio_deinit** (void)

Deinitialize LEDC timer_group and output gpio.

Return

- ESP_OK Success
- ESP_FAIL pwm_audio not initialized

esp_err_t **pwm_audio_start** (void)

Start to run pwm audio.

Return

- 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

Return

- 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.

Return

- ESP_OK Success
- ESP_FAIL Write encounter error
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `inbuf`: Pointer source data to write
- `len`: length of data in bytes
- `[out] bytes_written`: 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.

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.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `rate`: sample rate (ex: 8000, 44100···)
- `bits`: bit width
- `ch`: channel number

esp_err_t **pwm_audio_set_sample_rate** (int *rate*)

Set sample rate.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `rate`: sample rate (ex: 8000, 44100···)

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

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

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

esp_err_t **pwm_audio_get_volume** (int8_t *volume)

Get current volume.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `volume`: Pointer to volume

esp_err_t **pwm_audio_get_param** (int *rate, int *bits, int *ch)

Get parameter for pwm audio.

Return

- Always return ESP_OK

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

esp_err_t **pwm_audio_get_status** (*pwm_audio_status_t* *status)

get pwm audio status

Return

- ESP_OK Success

Parameters

- `status`: current pwm_audio status

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:

PWM_AUDIO_STATUS_UN_INIT = 0

pwm audio uninitialized

PWM_AUDIO_STATUS_IDLE = 1

pwm audio idle

PWM_AUDIO_STATUS_BUSY = 2

pwm audio busy

enum pwm_audio_channel_t

pwm audio channel define

Values:

PWM_AUDIO_CH_MONO = 0

1 channel (mono)

PWM_AUDIO_CH_STEREO = 1

2 channel (stereo)

PWM_AUDIO_CH_MAX

5.2 DAC 音频

[English]

ESP32 拥有两个独立的 DAC 通道，并可直接使用 I2S 通过 DMA 播放音频。这里在 ESP-IDF 的基础上简化了 API，并对数据进行了重新编码以支持更多类型的采样位宽。

5.2.1 API 参考

Header File

- audio/dac_audio/include/dac_audio.h

Functions

esp_err_t **dac_audio_init** (*dac_audio_config_t* **cfg*)

initialize i2s build-in dac to play audio with

Attention only support ESP32, because i2s of ESP32S2 not have a build-in dac

Return

- ESP_OK Success
- ESP_FAIL Encounter error
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM Out of memory

Parameters

- `cfg`: configurations - see *dac_audio_config_t* struct

esp_err_t **dac_audio_deinit** (void)

deinitialize dac

Return

- ESP_OK Success
- ESP_FAIL Encounter error

esp_err_t **dac_audio_start** (void)

Start dac to play.

Return

- ESP_OK Success

- ESP_FAIL Encounter error

esp_err_t **dac_audio_stop** (void)

Stop play.

Return

- ESP_OK Success
- ESP_FAIL Encounter error

esp_err_t **dac_audio_set_param** (int *rate*, int *bits*, int *ch*)

Configuration dac parameter.

Return

- ESP_OK Success
- ESP_FAIL Encounter error
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `rate`: sample rate (ex: 8000, 44100···)
- `bits`: bit width
- `ch`: channel number

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

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

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

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.

Return

- ESP_OK Success

- ESP_FAIL Write encounter error
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `inbuf`: Pointer source data to write
- `len`: length of data in bytes
- `[out] bytes_written`: 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.

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 6

图形界面

[English]

6.1 LVGL 图形库

[English]

LVGL 是一个 C 语言编写的免费的开源图形库，提供了用于嵌入式 GUI 的各种元素。用户可以利用丰富的图形库资源，在消耗极低内存的情况下构建视觉效果丰富多彩的 GUI 。

6.1.1 主要特性

LVGL 具有以下特性：

- 超过 30 多种丰富的用户自定义控件，如按钮，滑条，文本框，键盘等
- 支持各种分辨率的屏幕，适配性好
- 接口简单，内存占用少
- 支持多个输入设备
- 提供抗锯齿，多边形，阴影等多种绘图元素
- 采用 UTF-8 编码，支持多语言，多字体的文本
- 支持各种图片类型，可从 Flash 和 SD 卡中读取图片显示

- 提供在线图片取模工具
- 支持 Micropython

6.1.2 配置要求

运行 LVGL 的最低配置要求如下：

- 16、32、64 位的微控制器或处理器
- 时钟频率：大于 16 MHz
- Flash/ROM：大于 64 kB（推荐 180 kB）
- RAM：8 kB（推荐 24 kB）
- 需要一个帧缓存区
- 显示缓存：至少大于水平分辨率的像素
- C99 或更高版本的编译器

6.1.3 在线工具

LVGL 提供了在线的 字模提取工具 和 图片取模工具。

6.1.4 示例方案

注解： 以下示例不再维护，LCD 以及 LVGL 示例请参考：i80_controller、rgb_panel 和 spi_lcd_touch

官方例程

LVGL 官方提供了 ESP32 上使用 LVGL 的 LVGL ESP32 示例程序。

除此之外，在 ESP-IoT-Solution 中也提供了一些应用 LVGL 的实例：

thermostat

使用 LVGL 设计了一个恒温计控制的界面：

相应例程在 hmi/lvgl_thermostat

coffee

使用 LVGL 绘制了一个咖啡机的交互界面：

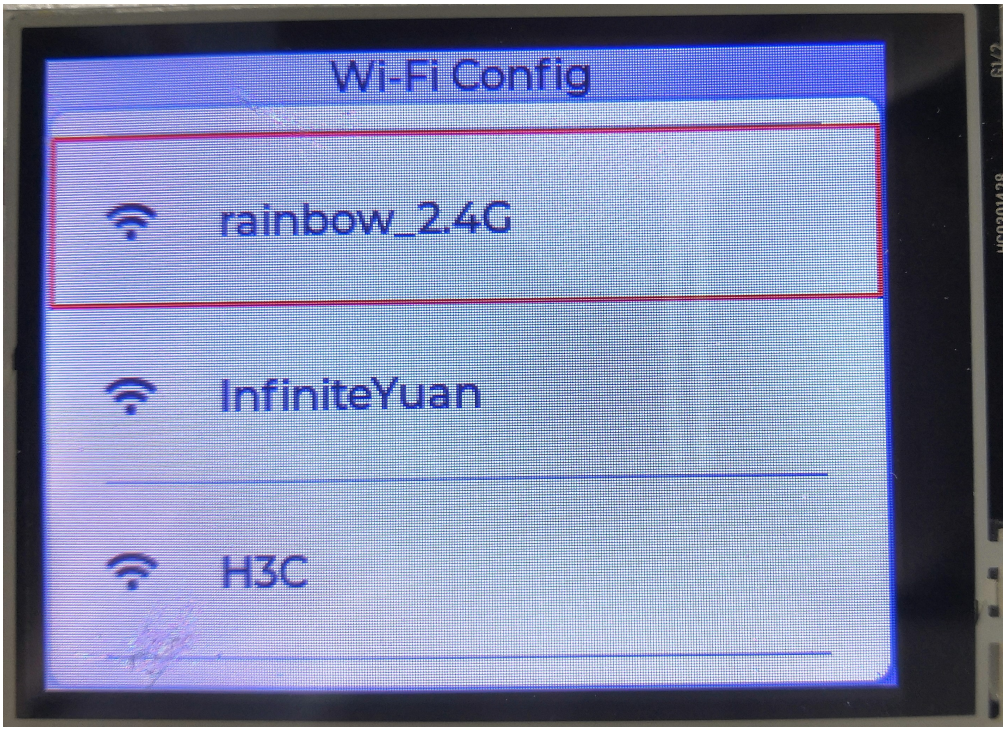
相应例程在 hmi/lvgl_coffee

wificonfig

ESP32 连接 Wi-Fi，利用 LVGL 绘制的 Wi-Fi 连接界面，可以显示附近 Wi-Fi 信息，在屏幕上输入密码等。

相应例程在 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 7

输入设备

[English]

7.1 按键

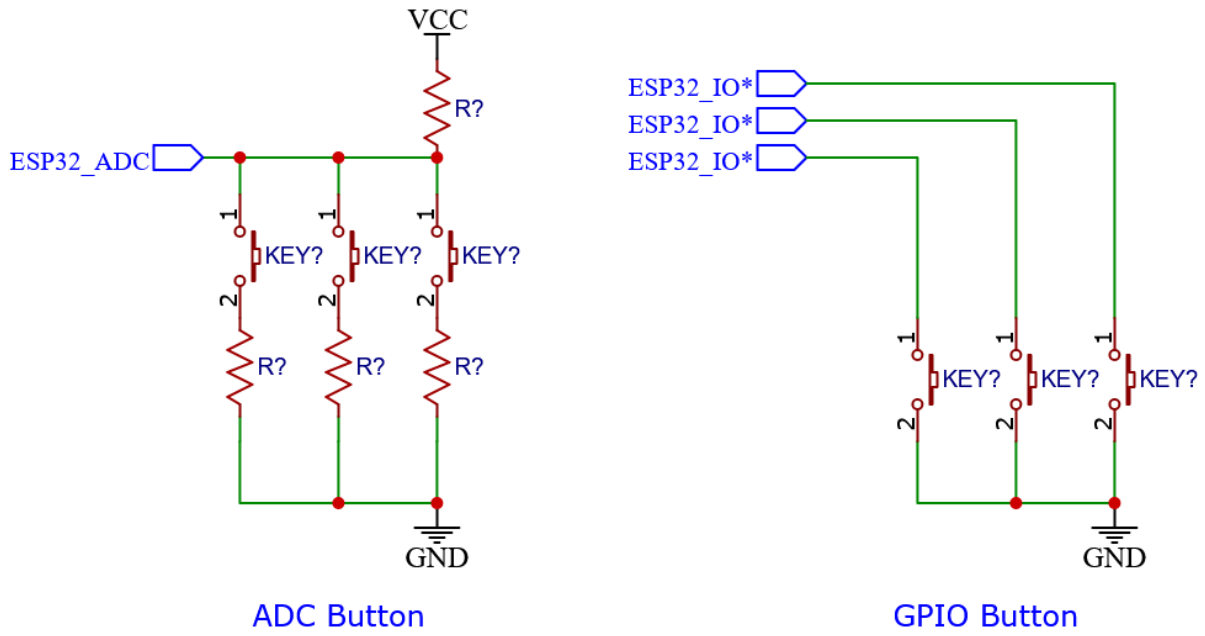
[English]

按键组件实现了 GPIO 和 ADC 两种按键，并允许同时创建两种不同的按键。下图显示了两种按键的硬件设计：

- GPIO 按键优点有：每一个按键占用独立的 IO，之间互不影响，稳定性高；缺点有：按键数量多时占用太多 IO 资源。
- ADC 按键优点有：可多个按键共用一个 ADC 通道，占用 IO 资源少；缺点有：不能同时按下多按键，当按键因氧化等因素导致闭合电阻增大时，容易误触，稳定性不高。

注解:

- GPIO 按键需注意上下拉问题，组件内部会启用芯片内部的上下拉电阻，但是在仅支持输入的 IO 内部没有电阻，**需要外部连接**。
- ADC 按键需注意电压不能超过 ADC 量程。



7.1.1 按键事件

每个按键拥有下表的 8 个事件：

事件	触发条件
BUTTON_PRESS_DOWN	按下
BUTTON_PRESS_UP	弹起
BUTTON_PRESS_REPEAT	按下弹起次数 >= 2 次
BUTTON_SINGLE_CLICK	按下弹起 1 次
BUTTON_DOUBLE_CLICK	按下弹起 2 次
BUTTON_LONG_PRESS_START	按下时间达到阈值的瞬间
BUTTON_LONG_PRESS_HOLD	长按期间一直触发
BUTTON_PRESS_REPEAT_DONE	多次按下弹起结束

每个按键可以有 **回调**和 **轮询**两种使用方式：

- 回调：一个按键的每个事件都可以为其注册一个回调函数，产生事件时回调函数将会被调用。这种方式的效率和实时性高，不会丢失事件。
- 轮询：在程序中周期性调用 `iot_button_get_event()` 查询按键当前的事件。这种方式使用简单，适合任务简单的场合

当然你也可以将以上两种方式组合使用。

注意: 回调函数中不能有 TaskDelay 等阻塞的操作

7.1.2 配置项

- BUTTON_PERIOD_TIME_MS : 扫描周期
- BUTTON_DEBOUNCE_TICKS : 消抖次数
- BUTTON_SHORT_PRESS_TIME_MS : 连续短按有效时间
- BUTTON_LONG_PRESS_TIME_MS : 长按有效时间
- ADC_BUTTON_MAX_CHANNEL : ADC 按钮的最大通道数
- ADC_BUTTON_MAX_BUTTON_PER_CHANNEL : ADC 一个通道最多的按钮数
- ADC_BUTTON_SAMPLE_TIMES : 每次扫描的样本数
- BUTTON_SERIAL_TIME_MS : 长按期间触发的 CALLBACK 间隔时间

7.1.3 应用示例

创建按键

```
// create gpio button
button_config_t gpio_btn_cfg = {
    .type = BUTTON_TYPE_GPIO,
    .long_press_ticks = CONFIG_BUTTON_LONG_PRESS_TIME_MS,
    .short_press_ticks = 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_ticks = CONFIG_BUTTON_LONG_PRESS_TIME_MS,
    .short_press_ticks = CONFIG_BUTTON_SHORT_PRESS_TIME_MS,
    .adc_button_config = {
```

(下页继续)

(续上页)

```
        .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");
}
```

注解: ADC 按钮使用的是 ADC1，当项目中还有其他地方使用到了 ADC1 时，请传入 adc_handle 和 adc_channel 来配置 ADC 按钮。

注册回调函数

```
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);
```

查询按键事件

```
button_event_t event;
event = iot_button_get_event(button_handle);
```

7.1.4 API Reference

Header File

- button/include/iot_button.h

Functions

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

Create a button.

Return A handle to the created button, or NULL in case of error.

Parameters

- `config`: pointer of button configuration, must corresponding the button type

esp_err_t **iot_button_delete**(*button_handle_t btn_handle*)

Delete a button.

Return

- ESP_OK Success
- ESP_FAIL Failure

Parameters

- `btn_handle`: A button handle to delete

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.

Return

- 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.

Parameters

- `btn_handle`: A button handle to register
- `event`: Button event
- `cb`: Callback function.
- `usr_data`: user data

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

Unregister the button event callback function.

Return

- 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.

Parameters

- `btn_handle`: A button handle to unregister
- `event`: Button event

size_t **iot_button_count_cb** (*button_handle_t btn_handle*)

how many Callbacks are still registered.

Return 0 if no callbacks registered, or 1 .. (BUTTON_EVENT_MAX-1) for the number of Registered Buttons.

Parameters

- `btn_handle`: A button handle to unregister

button_event_t **iot_button_get_event** (*button_handle_t btn_handle*)

Get button event.

Return Current button event. See button_event_t

Parameters

- `btn_handle`: Button handle

uint8_t **iot_button_get_repeat** (*button_handle_t btn_handle*)

Get button repeat times.

Return button pressed times. For example, double-click return 2, triple-click return 3, etc.

Parameters

- `btn_handle`: Button handle

uint16_t **iot_button_get_ticks_time** (*button_handle_t btn_handle*)

Get button ticks time.

Return Actual time from press down to up (ms).

Parameters

- `btn_handle`: Button handle

uint16_t **iot_button_get_long_press_hold_cnt** (*button_handle_t btn_handle*)

Get button long press hold count.

Return Count of trigger cb(BUTTON_LONG_PRESS_HOLD)

Parameters

- `btn_handle`: Button handle

Structures

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_adc_config_t **adc_button_config**

adc 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:

BUTTON_PRESS_DOWN = 0

BUTTON_PRESS_UP

BUTTON_PRESS_REPEAT

BUTTON_PRESS_REPEAT_DONE

BUTTON_SINGLE_CLICK

BUTTON_DOUBLE_CLICK

BUTTON_LONG_PRESS_START

BUTTON_LONG_PRESS_HOLD

BUTTON_EVENT_MAX

BUTTON_NONE_PRESS

enum button_type_t

Supported button type.

Values:

BUTTON_TYPE_GPIO

BUTTON_TYPE_ADC

BUTTON_TYPE_CUSTOM

7.2 旋钮

[English]

Knob 是提供软件 PCNT 的组件，可以用在没有 PCNT 硬件功能的芯片（esp32c2, esp32c3）上。使用 Knob 可以快速适配物理编码器，如 EC11 编码器。

7.2.1 适用的场景

适用于每秒脉冲数小于 30 个的低速旋钮计数场景。例如 EC11 编码器。适用于对脉冲数不要求 100% 准确的场景。

注解: 如需精确或快速的脉冲计数，请使用 硬件 PCNT 功能 。硬件 PCNT 支持的芯片 ESP32, ESP32-C6, ESP32-H2, ESP32-S2, ESP32-S3。

7.2.2 旋钮事件

每个旋钮拥有下表的 5 个事件:

事件	触发条件
KNOB_LEFT	左旋
KNOB_RIGHT	右旋
KNOB_H_LIM	计数达到最高限制
KNOB_L_LIM	计数达到最低限制
KNOB_ZERO	计数变成零

每个旋钮都可以有 **回调**的使用方式:

- 回调: 一个旋钮的每个事件都可以为其注册一个回调函数，产生事件时回调函数将会被调用。这种方式的效率和实时性高，不会丢失事件。

注意: 回调函数中不能有 TaskDelay 等阻塞的操作

7.2.3 配置项

- KNOB_PERIOD_TIME_MS : 扫描周期
- KNOB_DEBOUNCE_TICKS : 消抖次数
- KNOB_HIGH_LIMIT : 旋钮所能计数的最高数字
- KNOB_LOW_LIMIT : 旋钮所能计数的最低数字

7.2.4 应用示例

创建旋钮

```
// 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");
}
```

注册回调函数

```
static void _knob_left_cb(void *arg, void *data)
{
    ESP_LOGI(TAG, "KONB: KONB_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);
```

7.2.5 API Reference

Header File

- knob/iot_knob.h

Functions

knob_handle_t **iot_knob_create**(**const** *knob_config_t* **config*)

create a knob

Return A handle to the created knob

Parameters

- `config`: pointer of knob configuration

esp_err_t **iot_knob_delete**(*knob_handle_t knob_handle*)

Delete a knob.

Return

- ESP_OK Success
- ESP_FAIL Failure

Parameters

- `knob_handle`: A knob handle to delete

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.

Return

- ESP_OK Success
- ESP_FAIL Failure

Parameters

- `knob_handle`: A knob handle to register
- `event`: Knob event
- `cb`: Callback function
- `usr_data`: user data

esp_err_t **`iot_knob_unregister_cb`**(*knob_handle_t knob_handle*, *knob_event_t event*)

Unregister the knob event callback function.

Return

- ESP_OK Success
- ESP_FAIL Failure

Parameters

- `knob_handle`: A knob handle to register
- `event`: Knob event

knob_event_t **`iot_knob_get_event`**(*knob_handle_t knob_handle*)

Get knob event.

Return knob_event_t Knob event

Parameters

- `knob_handle`: A knob handle to register

int **iot_knob_get_count_value** (*knob_handle_t knob_handle*)

Get knob count value.

Return int count_value

Parameters

- `knob_handle`: A knob handle to register

esp_err_t **iot_knob_clear_count_value** (*knob_handle_t knob_handle*)

Clear knob cout value to zero.

Return

- ESP_OK Success
- ESP_FAIL Failure

Parameters

- `knob_handle`: A knob handle to register

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

Type Definitions

typedef void (***knob_cb_t**) (void *, void *)

typedef void ***knob_handle_t**

Enumerations

enum knob_event_t

Knob events.

Values:

KNOB_LEFT = 0

EVENT: Rotate to the left

KNOB_RIGHT

EVENT: Rotate to the right

KNOB_H_LIM

EVENT: Count reaches maximum limit

KNOB_L_LIM

EVENT: Count reaches the minimum limit

KNOB_ZERO

EVENT: Count back to 0

KNOB_EVENT_MAX

EVENT: Number of events

7.3 触摸屏驱动

[English]

触摸屏现已是显示屏应用中的标配，ESP-IoT-Solution 提供了常见类型的触摸屏驱动，已支持如下控制芯片：

电阻触摸屏	电容触摸屏
XPT2046	FT5216
NS2016	FT5436
	FT6336
	FT5316

上面列出的电容触摸屏控制芯片通常可使用 `FT5x06` 驱动。

与屏幕驱动相似，为了方便移植到不同的 GUI 库，将一部分通用的函数封装到了一个*touch_panel_driver_t* 结构体中。完成初始化后将通过调用结构体里面的函数完成对触摸屏的操作，而无需关心是具体哪一个触摸屏型号。

7.3.1 校准触摸屏

在实际应用中，电阻触摸屏必须在使用前进行校准，而电容触摸屏则一般由控制芯片完成该工作，无需额外的校准步骤。驱动中已经集成了电阻触摸屏的校准算法，校准过程使用了三个点来校准，用一个点来验证，当最后验证的误差大于某个阈值将导致校准失败，然后自动重新进行校准，直到校准成功。

调用校准函数 `calibration_run()` 将会在屏幕上开始校准的过程，校准完成后，参数将保存在 NVS 中用于下次启动，避免每次使用前的重复校准。

7.3.2 触摸屏的按下

不论是电阻还是电容触摸屏，通常的触摸屏控制芯片会有一个用于通知触摸事件的中断引脚。但是驱动中没有使用该信号，一方面是因为对于有屏幕的应用需要尽量节省出 IO 给其他外设；另一方面是触摸控制器给出的该信号不如程序通过寄存器数据判断的准确性高。

对于电阻触摸屏来说，判断按下的依据是 Z 方向的压力大于配置的阈值；对于电容触摸屏则是判断至少有一个触摸点存在。

7.3.3 触摸屏的旋转

触摸屏具有与显示屏一样的 8 个方向，定义在*touch_panel_dir_t* 中。这里的旋转是通过软件换算来实现的，通常把二者的方向设置为相同。但这并不是一成不变的，例如：在使用电容触摸屏时，有可能触摸屏固有的方向与显示屏原始显示方向不一致，如果简单的将这两个方向设置为相同后，将无法正确的点击屏幕内容，这时需要根据实际情况调整。

触摸屏的分辨率设置也是很重要的，因为触摸屏旋转后的换算依赖于触摸屏的宽和高分辨率大小，设置不当将无法得到正确的旋转效果。

注解： 在使用电阻屏时，由于电阻屏在每个方向上的电阻值不一定均匀分布，这会导致经过旋转换算后触摸位置的不准确，所以建议电阻屏校准后不再进行旋转操作。

7.3.4 应用示例

初始化触摸屏

```
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,
```

(下页继续)

(续上页)

```
    .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);
```

注解:

- 当使用的是电容触摸屏时，调用校准函数将直接返回 `ESP_OK`。
- 默认情况下只打开了 FT5x06 触摸屏的驱动，如果要使用其他的驱动，需要在 `menuconfig -> Component config -> Touch Screen Driver -> Choose Touch Screen Driver` 中使能对应驱动。

获取触摸屏是否按下及其触点坐标

```
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) {
```

(下页继续)

(续上页)

```
    ESP_LOGI(TAG, "Pressed, Touch point at (%d, %d)", x, y);
}
```

7.3.5 API 参考

Header File

- 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.

Return

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

Parameters

- `controller`: Touch panel controller to initialize
- `out_driver`: Pointer to a touch driver

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 adddress

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.

Return

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `config`: Pointer to a structure with touch config arguments.

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.

Return

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `screen`: Screen driver for display prompts
- `recalibrate`: Is mandatory, set true to force calibrate

esp_err_t (*__set_direction__)(*touch_panel_dir_t* dir)

Set touch rotate rotation.

Return

- ESP_OK Success

- ESP_FAIL Fail

Parameters

- `dir`: rotate direction

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

Get current touch information, see struct *touch_panel_points_t*.

Return

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `point`: a pointer of *touch_panel_points_t* contained touch information.

Macros

`TOUCH_MAX_POINT_NUMBER`

max point number on touch panel

Enumerations

`enum touch_panel_event_t`

Touch events.

Values:

`TOUCH_EVT_RELEASE` = 0x0

Release event

`TOUCH_EVT_PRESS` = 0x1

Press event

`enum touch_panel_dir_t`

Define all screen direction.

Values:

`TOUCH_DIR_LRTB`

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

`TOUCH_DIR_LRBT`

From left to right then from bottom to top

`TOUCH_DIR_RLTB`

From right to left then from top to bottom

TOUCH_DIR_RLBT

From right to left then from bottom to top

TOUCH_DIR_TBLR

From top to bottom then from left to right

TOUCH_DIR_BTLR

From bottom to top then from left to right

TOUCH_DIR_TBRL

From top to bottom then from right to left

TOUCH_DIR_BTRL

From bottom to top then from right to left

TOUCH_DIR_MAX

TOUCH_MIRROR_X = 0x40

Mirror X-axis

TOUCH_MIRROR_Y = 0x20

Mirror Y-axis

TOUCH_SWAP_XY = 0x80

Swap XY axis

enum touch_panel_interface_type_t

Values:

TOUCH_PANEL_IFACE_I2C

I2C interface

TOUCH_PANEL_IFACE_SPI

SPI interface

enum touch_panel_controller_t

All supported touch panel controllers.

Values:

TOUCH_PANEL_CONTROLLER_FT5X06

TOUCH_PANEL_CONTROLLER_XPT2046

TOUCH_PANEL_CONTROLLER_NS2016

CHAPTER 8

传感器集

[English]

8.1 Sensor Hub 简介

[English]

Sensor Hub 是一个传感器管理组件，可以实现对传感器设备的硬件抽象、设备管理和数据分发。基于 Sensor Hub 开发应用程序时，用户无需处理复杂的传感器实现，只需要对传感器的工作方式、采集间隔、量程等进行简单的选择，然后向关心的事件消息注册回调函数，即可在传感器状态切换或者数据采集好时收到通知。

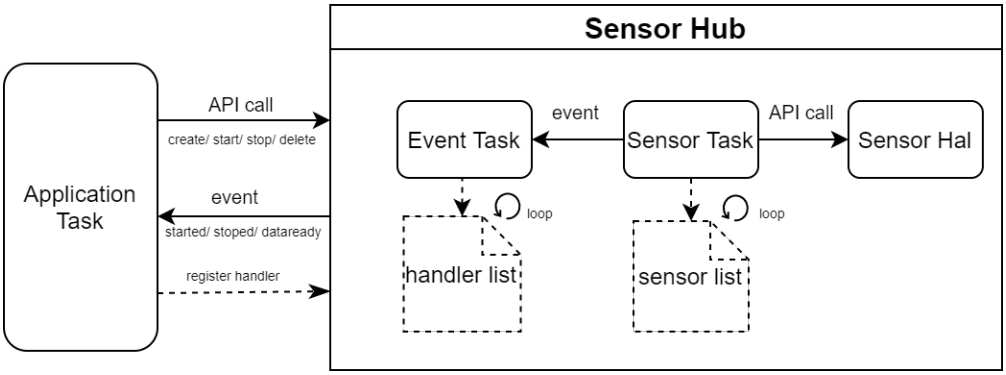


图 1: Sensor Hub 编程模型

Sensor Hub 对常见的传感器类别进行了硬件抽象，用户在切换传感器型号时，无需修改上层应用程序，也可以通过实现硬件抽象层的传感器接口添加新的传感器到 Sensor Hub 中。该组件由于实现了对传感器的集中管

理，在简化操作的同时也提高了运行效率，可作为传感器应用的基础组件，应用在环境感知、运动感知、健康管理等更多智能化场景中。

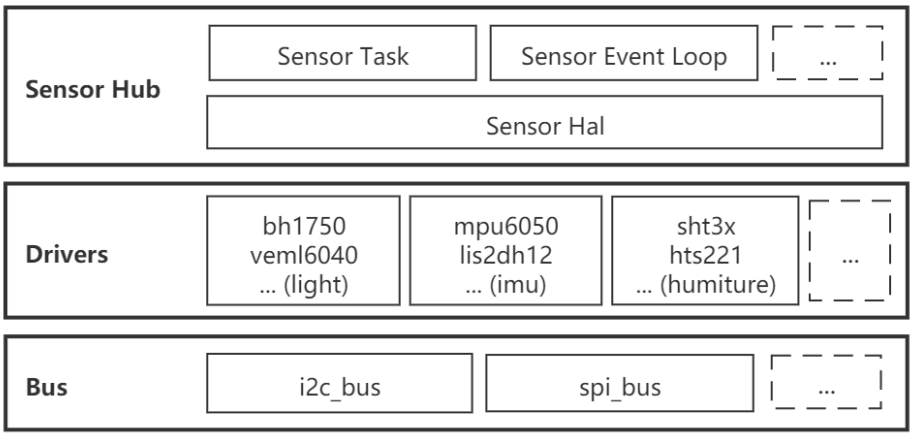


图 2: Sensor Hub 驱动

8.1.1 Sensor Hub 使用方法

1. 创建一个传感器实例：使用`iot_sensor_create()` 创建一个传感器实例，参数包括 `sensor_id_t` 中定义的传感器 ID、传感器配置项和传感器句柄指针。传感器 ID 用于查找和加载对应的驱动，一个 ID 只能对应创建一个实例。配置项中 *bus* 用于指定传感器挂载到的总线位置；*mode* 用于指定传感器的工作模式；*min_delay* 用于指定传感器的采集间隔，其它均为非必须项。创建成功之后，获得该传感器句柄；
2. 注册传感器事件回调函数：在传感器事件发生时，回调函数将会被依次调用，注册回调函数的方法有以下两种，注册成功之后将返回事件回调函数实例句柄：
   - 使用`iot_sensor_handler_register()` 通过传感器句柄注册回调函数
   - 使用`iot_sensor_handler_register_with_type()` 通过传感器类型注册回调函数
3. 启动传感器：使用`iot_sensor_start()` 启动指定的传感器，传感器启动之后将发出 `SENSOR_STARTED` 事件，之后将以设定的周期持续采集传感器数据，并发送 `SENSOR_XXXX_DATA_READY` 事件。事件回调函数可通过 `event_data` 参数获取每一个事件的具体数据；
4. 停止传感器：使用`iot_sensor_stop()` 可临时关闭指定的传感器，传感器关闭之后将发出 `SENSOR_STOPED` 事件，之后采集工作将停止。如果该传感器驱动支持电源管理，传感器将被设置为睡眠模式；
5. 取消注册传感器事件回调函数：用户程序可在任意时刻使用事件回调函数实例句柄取消对事件的注册，之后该事件发生时，该回调函数将不再被调用。取消注册的方法对应也有两种：

- 使用`iot_sensor_handler_unregister()`通过传感器句柄取消已注册的回调函数
- 使用`iot_sensor_handler_unregister_with_type()`通过传感器类型取消已经注册的回调函数

6. 删除传感器：使用`iot_sensor_delete()`可删除对应的传感器，释放已分配的内存等资源。

8.1.2 示例程序

1. 光照传感器控制 LED 开关示例：sensors/sensor_control_led。
2. 传感器监测示例：sensors/sensor_hub_monitor。

8.1.3 API 参考

Header File

- 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:

NULL_ID

NULL

HUMITURE_ID

humidity or temperature sensor

IMU_ID

gyro or acc sensor

LIGHT_SENSOR_ID

light illumination or uv or color sensor

SENSOR_TYPE_MAX

max sensor type

enum sensor_command_t

sensor operate command

Values:

COMMAND_SET_MODE

set measure mdoe

COMMAND_SET_RANGE

set measure range

COMMAND_SET_ODR

set output rate

COMMAND_SET_POWER

set power mode

COMMAND_SELF_TEST

sensor self test

COMMAND_MAX

max sensor command

enum sensor_power_mode_t

sensor power mode

Values:

POWER_MODE_WAKEUP

wakeup from sleep

POWER_MODE_SLEEP

set to sleep

POWER_MAX

max sensor power mode

enum sensor_mode_t

sensor acquire mode

Values:

MODE_DEFAULT

default work mode

MODE_POLLING

polling acquire with a interval time

MODE_INTERRUPT

interrupt mode, acquire data when interrupt comes

MODE_MAX

max sensor mode

enum sensor_range_t

sensor acquire range

Values:

RANGE_DEFAULT

default range

RANGE_MIN

minimum range for high-speed or high-precision

RANGE_MEDIUM

medium range for general use

RANGE_MAX

maximum range for full scale

enum sensor_event_id_t

sensor general events

Values:

SENSOR_STARTED

sensor started, data acquire will be started

SENSOR_STOPED

sensor stoped, data acquire will be stoped

SENSOR_EVENT_COMMON_END = 9

max common events id

enum sensor_data_event_id_t

sensor data ready events

Values:

SENSOR_ACCE_DATA_READY = 10

Accelerometer data ready

SENSOR_GYRO_DATA_READY

Gyroscope data ready

SENSOR_MAG_DATA_READY

Magnetometer data ready

SENSOR_TEMP_DATA_READY

Temperature data ready

SENSOR_HUMI_DATA_READY

Relative humidity data ready

SENSOR_BARO_DATA_READY

Pressure data ready

SENSOR_LIGHT_DATA_READY

Light data ready

SENSOR_RGBW_DATA_READY
Color data ready

SENSOR_UV_DATA_READY
ultraviolet data ready

SENSOR_PROXI_DATA_READY
Distance data ready

SENSOR_HR_DATA_READY
Heat rate data ready

SENSOR_TVOC_DATA_READY
TVOC data ready

SENSOR_NOISE_DATA_READY
Noise Loudness data ready

SENSOR_STEP_DATA_READY
Step data ready

SENSOR_FORCE_DATA_READY
Force data ready

SENSOR_CURRENT_DATA_READY
Current data ready

SENSOR_VOLTAGE_DATA_READY
Voltage data ready

SENSOR_EVENT_ID_END
max common events id

Header File

- 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.

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

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.

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.

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `sensor_handle`: sensor handle for operation

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.

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `sensor_handle`: sensor handle for operation

esp_err_t **`iot_sensor_delete`** (*sensor_handle_t *p_sensor_handle*)

delete and release the sensor resource.

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `p_sensor_handle`: point to sensor handle, will set to NULL if delete suceed.

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

Scan for valid sensors attached on bus.

Return uint8_t total number of valid sensors found on the 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.

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.

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

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.

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.

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `sensor_handle`: sensor handle for operation
- `context`: the instance object of the registration to be unregistered

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.

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `sensor_type`: sensor type decleared in sensor_type_t.
- `event_id`: sensor event decleared 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.

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.

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `sensor_type`: sensor type decleared in sensor_type_t.
- `event_id`: sensor event decleared in sensor_event_id_t and sensor_data_event_id_t
- `context`: the instance object of the registration to be unregistered

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 const char ***sensor_event_base_t**

unique pointer to a subsystem that exposes events

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:

8.2 温湿度传感器

[English]

温湿度传感器包括温度传感器、湿度传感器和兼具两种功能的传感器，可用于智慧人居、智慧农场、智能工厂等场景下的环境温湿度检测。

8.2.1 已适配列表

名称	功能	总线	供应商	规格书	硬件抽象层
HDC2010	温度、湿度	I2C	TI	规格书	
HTS221	温度、湿度	I2C	ST	规格书	√
SHT3X	温度、湿度	I2C	Sensirion	规格书	√
MVH3004D	温度、湿度	I2C	–		

8.2.2 API 参考

以下 API 实现了对温湿度传感器的硬件抽象，用户可直接调用该层代码编写传感器应用程序，也可以使用*sensor_hub* 中的传感器接口实现更简单的调用。

Header File

- 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.

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

Parameters

- `bus`: i2c bus handle the sensor attached to
- `id`: id declared in humiture_id_t

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

Delete and release the sensor resource.

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `sensor`: point to humiture sensor handle, will set to NULL if delete succeed.

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

Test if sensor is active.

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `sensor`: humiture sensor handle to operate

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

Acquire humiture sensor relative humidity result one time.

Return esp_err_t

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

Parameters

- `sensor`: humiture sensor handle to operate.
- `humidity`: result data (unit:percentage)

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

Acquire humiture sensor temperature result one time.

Return esp_err_t

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

Parameters

- `sensor`: humiture sensor handle to operate.
- `sensor_data`: result data (unit:dCelsius)

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

Set sensor to sleep mode.

Return esp_err_t

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

Parameters

- `sensor`: humiture sensor handle to operate

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

Wakeup sensor from sleep mode.

Return esp_err_t

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

Parameters

- `sensor`: humiture sensor handle to operate

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

acquire a group of sensor data

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `sensor`: humiture sensor handle to operate
- `data_group`: acquired data

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:

`SHT3X_ID` = 0x01

sht3x humiture sensor id

`HTS221_ID`

hts221 humiture sensor id

`HUMITURE_MAX_ID`

max humiture sensor id

8.3 惯性传感器 (IMU)

[English]

惯性传感器包含陀螺仪传感器、加速度传感器和兼具多种功能的传感器等。主要用于测量物体的加速度、角速度等，进而解算出物体的运动姿态。

8.3.1 已适配列表

名称	功能	总线	供应商	规格书	硬件抽象层
LIS2DH12	3-axis acceler	I2C	ST	规格书	√
MPU6050	3-axis acceler + 3-axis gyro	I2C	InvenSense	规格书	√

8.3.2 API 参考

以下 API 实现了对惯性传感器的硬件抽象，用户可直接调用该层代码编写传感器应用程序，也可以使用*sensor_hub* 中的传感器接口实现更简单的调用。

Header File

- 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.

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

Parameters

- `bus`: i2c bus handle the sensor attached to
- `imu_id`: id declared in imu_id_t

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

Delete and release the sensor resource.

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `sensor`: point to imu sensor handle, will set to NULL if delete succeed.

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

Test if sensor is active.

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `sensor`: imu sensor handle to operate

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

Acquire imu sensor accelerometer result one time.

Return esp_err_t

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

Parameters

- `sensor`: imu sensor handle to operate
- `acce`: result data (unit:g)

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

Acquire imu sensor gyroscope result one time.

Return esp_err_t

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

Parameters

- `sensor`: imu sensor handle to operate
- `gyro`: result data (unit:dps)

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

Set sensor to sleep mode.

Return esp_err_t

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

Parameters

- `sensor`: imu sensor handle to operate

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

Wakeup sensor from sleep mode.

Return esp_err_t

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

Parameters

- `sensor`: imu sensor handle to operate

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

acquire a group of sensor data

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `sensor`: imu sensor handle to operate
- `data_group`: acquired data

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:

MPU6050_ID = 0x01

MPU6050 imu sensor id

LIS2DH12_ID

LIS2DH12 imu sensor id

IMU_MAX_ID

max imu sensor id

8.4 环境光传感器

[English]

环境光传感器包含光照强度传感器、颜色传感器、紫外线传感器和兼具多种功能的传感器。

8.4.1 已适配列表

名称	功能	总线	供应商	规格书	硬件抽象层
BH1750	Light	I2C	rohm	规格书	√
VEML6040	Light RGBW	I2C	Vishay	规格书	√
VEML6075	Light UVA UVB	I2C	Vishay	规格书	√

8.4.2 API 参考

以下 API 实现了对环境光传感器的硬件抽象，用户可直接调用该层代码编写传感器应用程序，也可以使用*sensor_hub* 中的传感器接口，实现更简单的调用。

Header File

- 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.

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

Parameters

- `bus`: i2c bus handle the sensor attached to
- `id`: id declared in light_sensor_id_t

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

Delete and release the sensor resource.

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `sensor`: point to light sensor handle, will set to NULL if delete succeed.

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

Test if sensor is active.

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `sensor`: light sensor handle to operate.

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

Acquire light sensor illuminance result one time.

Return esp_err_t

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

Parameters

- `sensor`: light sensor handle to operate.
- `lux`: result data (unit:lux)

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.

Return esp_err_t

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

Parameters

- `sensor`: light sensor handle to operate.
- `rgbw`: result data (unit:lux)

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.

Return esp_err_t

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

Parameters

- `sensor`: light sensor handle to operate.
- `uv`: result data (unit:lux)

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

Set sensor to sleep mode.

Return esp_err_t

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

Parameters

- `sensor`: light sensor handle to operate.

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

Wakeup sensor from sleep mode.

Return esp_err_t

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

Parameters

- `sensor`: light sensor handle to operate.

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

acquire a group of sensor data

Return esp_err_t

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `sensor`: light sensor handle to operate
- `data_group`: acquired data

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:

BH1750_ID = 0x01

BH1750 light sensor id

VEML6040_ID

VEML6040 light sensor id

VEML6075_ID

VEML6075 light sensor id

LIGHT_MAX_ID

max light sensor id

8.5 气压传感器

[English]

气压传感器用于检测气体绝对压强，也可以用于计算海拔高度等，常用于环境监测设备，高度测量和空间定位设备等。

8.5.1 已适配列表

名称	功能	总线	供应商	规格书	硬件抽象层
BME280	Pressure	I2C/SPI	BOSCH	规格书	

8.6 手势传感器

[English]

手势传感器一般是指将反射红外光的测量结果转换为有关物理运动的传感器，可用于实现非接触式人机交互等。

8.6.1 已适配列表

名称	功能	总线	供应商	规格书	硬件抽象层
APDS9960	Light, RGB and Gesture Sensor	I2C	Avago	规格书	

CHAPTER 9

存储方案

[English]

9.1 存储媒介

[English]

已支持存储媒介列表：

名称	关键特性	应用场景	容量	传输模式	速度	驱动	备注
SPI Flash	可与代码共用，无附加成本	参数存储、文本、图像存储	MB	SPI	40/80 MHz 4 线	SPI Flash Driver	
SD Card	大容量、可插拔	声音、视频文件存储	GB	SDIO/SPI	20/40 MHz 1 线/4 线	SD/SDIO/MMC Driver	*1
eMMC	大容量、高速读写	声音、视频文件存储	GB	SDIO	20/40 MHz 1 线/4 线/8 线	SD/SDIO/MMC Driver	*2
EEPROM	可按字节寻址，低成本	参数存储	MB	I2C	100 ~ 400 KHz	eeprom	

注解：

- *1. ESP32-S2 仅支持 SPI 模式

- *2. ESP32-S2 不支持

9.1.1 SPI Flash

ESP32/ESP32-S/ESP32-C 系列芯片默认使用 NOR Flash 存取用户代码和数据，Flash 可集成在模组或芯片中，容量一般为 4 MB、8 MB 或 16 MB。对于 ESP-IDF v4.0 以上版本，SPI Flash 组件除了支持对主 Flash 进行读写操作以外，同时可支持外接第二块 Flash 芯片进行数据的存储。

用户可以使用 分区表 对 Flash 进行分区，根据分区表的功能划分，Flash 除了存放用户编译生成的二进制代码之外，也可以作为非易失性存储 (NVS) 空间存放应用程序参数，还可以将指定 Flash 空间挂载到文件系统（FatFS 等）存放文本、图像等文件。

Flash 芯片支持双线 (DOUT/DIO) 和四线 (QOUT/QIO) 操作模式，可通过配置工作在 40 MHz 或 80 MHz 模式。由于可以直接使用主 Flash 芯片进行数据存储，无需额外添加存储芯片，因此特别适合于容量需求较小 (MB)、集成度要求高、成本敏感的应用场景。

参考文档：

- SPI Flash API

9.1.2 SD Card

ESP32 支持使用 SDIO 接口或 SPI 接口访问 SD 卡。SDIO 接口支持 1 线、4 线或 8 线模式，支持默认速率 20 MHz 和高速 40 MHz 两种速率。需要注意的是该接口至少占用 6 个 GPIO，并且只能使用 固定的引脚。SPI 接口支持通过 GPIO matrix 为 SD 卡指定任意的 IO，通过 CS 引脚可以支持对多个 SD 卡的访问，SPI 接口硬件设计上更加灵活，但是相比 SDIO 接口访问速率较低。

ESP-IDF 中的 `SD/SDIO/MMC Driver` 基于 SD 卡的两种访问模式进行了协议层的封装，提供了 SD 卡的初始化接口和协议层 API。SD 卡具有大容量、可插拔的特点，广泛适用于智能音响、电子相册等具有大容量存储需求的应用场景。

参考文档：

- SD/SDIO/MMC 驱动：支持 SDIO 和 SPI 两种传输模式；
- SDMMC 主机驱动：SDIO 模式；
- SD SPI 主机驱动：SPI 模式；
- 使用 SPI 或 1-bit 模式，请注意 引脚上拉需求。

示例程序：

- storage/sd_card：访问使用 FAT 文件系统的 SD 卡。

9.1.3 eMMC

eMMC (embedded MMC) 内存芯片与 SD 卡具有相似的协议，可以使用与 SD 卡相同的驱动程序 SD/SDIO/MMC 驱动。但是需要注意的是，eMMC 芯片仅能使用 SDIO 模式，不支持 SPI 模式。eMMC 目前在 8 线模式下支持默认速率 20 MHz 和高速 40 MHz，在 4 线模式下支持高速 40 MHz DDR 模式。

eMMC 一般以芯片的形式焊接到主板上，相比 SD 卡集成度更高，适用于可穿戴设备等，具有大容量存储需求同时对系统集成度有一定要求的场景。

参考文档：

- SD/SDIO/MMC 驱动；
- 已支持的 eMMC 速度模式。

9.1.4 EEPROM

EEPROM（如 AT24C0X 系列）是 1024-16384 位的串行电可擦写存储器（通过控制引脚电平也可运行在只读模式），它的存储空间一般按照 `word` 进行分布，每个 `word` 包含 `8-bit` 空间。EEPROM 可按字节寻址，读写操作简单，特别适合于保存配置参数等，经过优化也可应用于对功耗和可靠性等有一定要求的工业和商业场景。

已适配的 EEPROM 芯片：

名称	功能	总线	供应商	规格书	驱动
AT24C01/02	1024/2048 bits EEPROM	I2C	Atmel	规格书	eeprom

9.2 文件系统

[English]

已支持的文件系统列表：

关键特性	NVS 库	FAT 文件系统	SPIFFS 文件系统
特点	键值对保存，接口安全	操作系统支持，兼容性强	针对嵌入式开发，资源占用低
应用场景	参数保存	音视频、文件保存	音视频、文件保存
容量	KB-MB	GB	< 128 MB
目录支持	X	√	X
磨损均衡	√	可选	√
读写效率	0	0	0
资源占用	0	0	*1
掉电保护	√	X	X
加密	√	√	X

注解:

- 0：暂无数据或不参与比较。
- *1：低 RAM 占用。

9.2.1 NVS 库

非易失性存储 (NVS) 库主要用于读写在 flash NVS 分区中存储的数据。NVS 中的数据以键值对的方式保存，其中键是 ASCII 字符串，值可以是整数、字符串、二进制数据 (BLOB) 类型。NVS 支持掉电保护和数据加密，适合存储一些较小的数据，如应用程序参数等。如需存储较大的 BLOB 或者字符串，请考虑使用基于磨损均衡库的 FAT 文件系统。

参考文档:

- 非易失性存储库。
- 批量生产时，可以使用 NVS 分区生成工具。

示例程序:

- 写入单个整数值：storage/nvs_rw_value。
- 写入二进制大对象：storage/nvs_rw_blob。

9.2.2 FAT 文件系统

ESP-IDF 使用 FatFs 库实现了对 FAT 文件系统的支持，FatFs 是独立于平台和存储介质的文件系统层，通过统一接口实现对物理设备（如 flash、SD 卡）的访问。用户可以直接调用 FatFs 的接口操作，也可以借助 C 标准库和 Posix API 通过 VFS（虚拟文件系统）使用 FatFs 库的大多数功能。

FAT 文件系统操作系统兼容性强，广泛应用于 USB 存储盘或 SD 卡等移动存储设备上。ESP32 系列芯片通过支持 FAT 文件系统，可以实现对这些常见存储设备的访问。

参考文档:

- FatFs 与 VFS 配合使用。
- FatFs 与 VFS 和 SD 卡配合使用。

示例程序:

- storage/sd_card：访问使用 FAT 文件系统的 SD 卡。
- storage/ext_flash_fatfs：访问使用 FAT 文件系统的外部 Flash 芯片。

9.2.3 SPIFFS 文件系统

SPIFFS 是一个专用于 SPI NOR flash 的嵌入式文件系统，原生支持磨损均衡、文件系统一致性检查等功能。用户可以直接调用 SPIFFS 提供的 Posix 样式接口，也可以通过 VFS 操作 SPIFFS 的大多数功能。

SPIFFS 作为嵌入式平台 SPI NOR Flash 的专用文件系统，相比 FAT 文件系统占用 RAM 资源更少，但是仅用于支持容量小于 128 MB 的 Flash 芯片。

参考文档：

- SPIFFS 文件系统。
- 两种生成 SPIFFS 镜像的工具。

示例程序：

- storage/spiffs：SPIFFS 使用示例。

9.2.4 虚拟文件系统 (VFS)

ESP-IDF 虚拟文件系统 (VFS) 组件可以为不同文件系统 (FAT, SPIFFS) 提供统一的接口，也可以为设备驱动程序提供类似文件读写的操作接口。

参考文档：

- 虚拟文件系统组件。

CHAPTER 10

电机

[English]

10.1 舵机

[English]

该组件使用 LEDC 外设产生 PWM 信号，可以实现对最多 16 路舵机的独立控制（ESP32 支持 16 路通道，ESP32-S2 支持 8 路通道），控制频率可选择为 50 ~ 400 Hz。使用该层 API，用户只需要指定舵机所在组、通道和目标角度，即可实现对舵机的角度操作。

舵机内部一般存在一个产生固定周期和脉宽的基准信号，通过与输入 PWM 信号进行比较，获得电压差输出，进而控制电机的转动方向和转动角度。常见的 180 度角旋转舵机一般以 20 ms (50 Hz) 为时钟周期，通过 0.5 ~ 2.5 ms 高电平脉冲，对应控制舵机在 0 ~ 180 度之间转动。

该组件可用于对控制精度要求较低的场景，例如玩具小车、遥控机器人、家庭自动化等。

10.1.1 使用方法

1. 舵机初始化：使用 `servo_init()` 对一组通道进行初始化，ESP32 包含 `LEDC_LOW_SPEED_MODE` 和 `LEDC_HIGH_SPEED_MODE` 两组通道，有些芯片可能只支持一组。初始化配置项主要包括最大角度、信号频率、最小输入脉宽和最大输入脉宽，用于计算角度和占空比的对应关系；引脚和通道用于分别指定芯片引脚和 LEDC 通道的对应关系；

2. 设置目标角度：使用 `servo_write_angle()` 通过指定舵机所在组、所在通道和目标角度，对舵机角度进行控制；
3. 读取当前角度：可使用 `servo_read_angle()` 获取舵机当前角度，需要注意的是该角度是根据输入信号进行推算的理论角度；
4. 舵机去初始化：当一组多个舵机都不再使用时，可使用 `servo_deinit()` 对一组通道进行去初始化。

10.1.2 应用示例

```
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 = {
            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;
```

(下页继续)

(续上页)

```
// 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);
```

10.1.3 API 参考

Header File

- 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.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Configure ledc failed

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

esp_err_t **iot_servo_deinit** (ledc_mode_t *speed_mode*)

Deinitialize ledc for servo.

Return

- ESP_OK Success

Parameters

- `speed_mode`: Select the LEDC channel group with specified speed mode.

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

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

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

esp_err_t **iot_servo_read_angle** (ledc_mode_t *speed_mode*, uint8_t *channel*, float **angle*)

Read current angle of one channel.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

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

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

CHAPTER 11

安全 & 加密

[English]

11.1 Flash 加密

11.1.1 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。

11.1.2 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。

11.1.3 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

11.1.4 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）

11.1.5 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。

11.1.6 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）

11.1.7 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

11.1.8 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

11.1.9 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

11.2 安全启动

11.2.1 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

11.2.2 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

11.2.3 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

11.2.4 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。

11.2.5 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。**

11.2.6 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

11.2.7 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
- 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

11.2.8 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
```

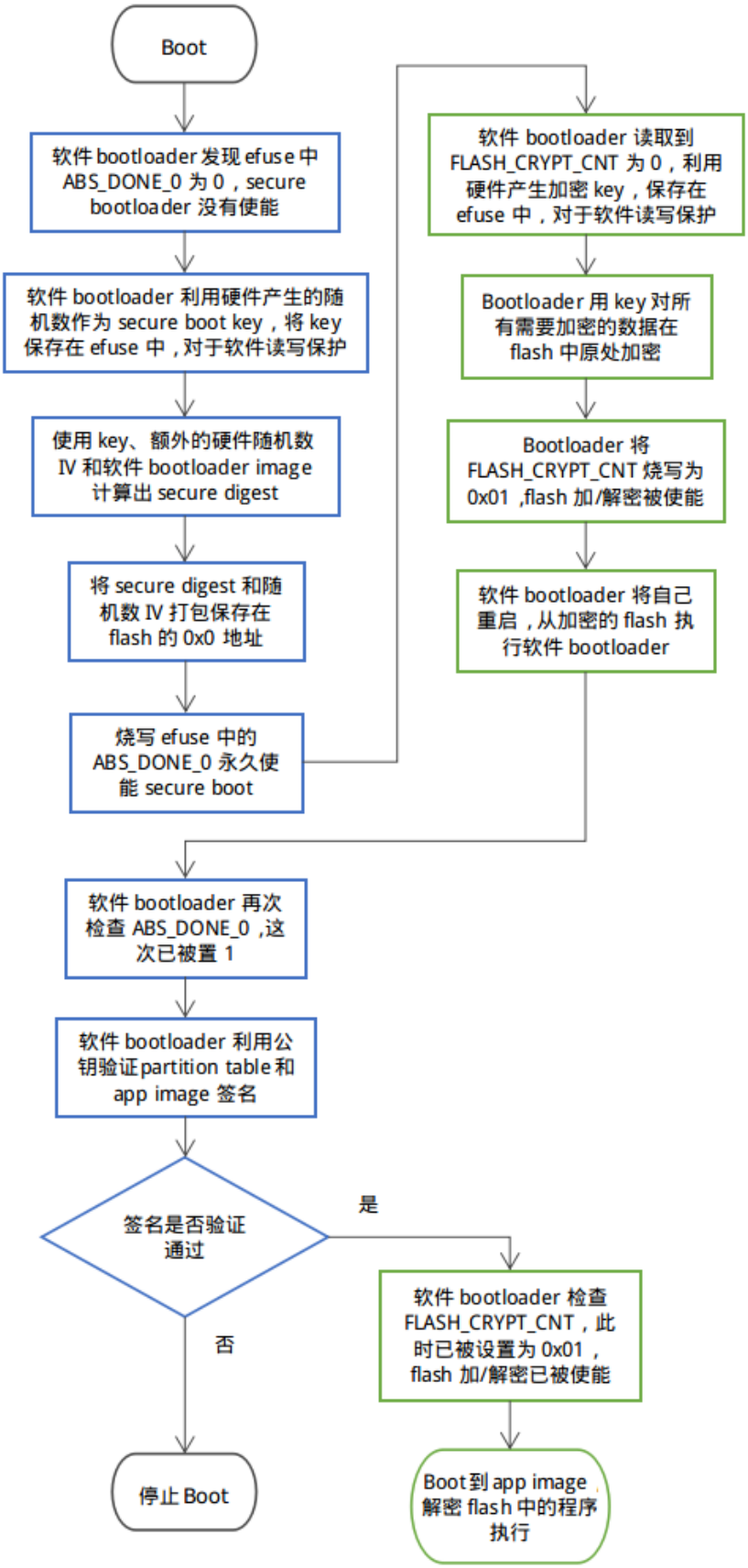
4. 编译 partition_table 与 app

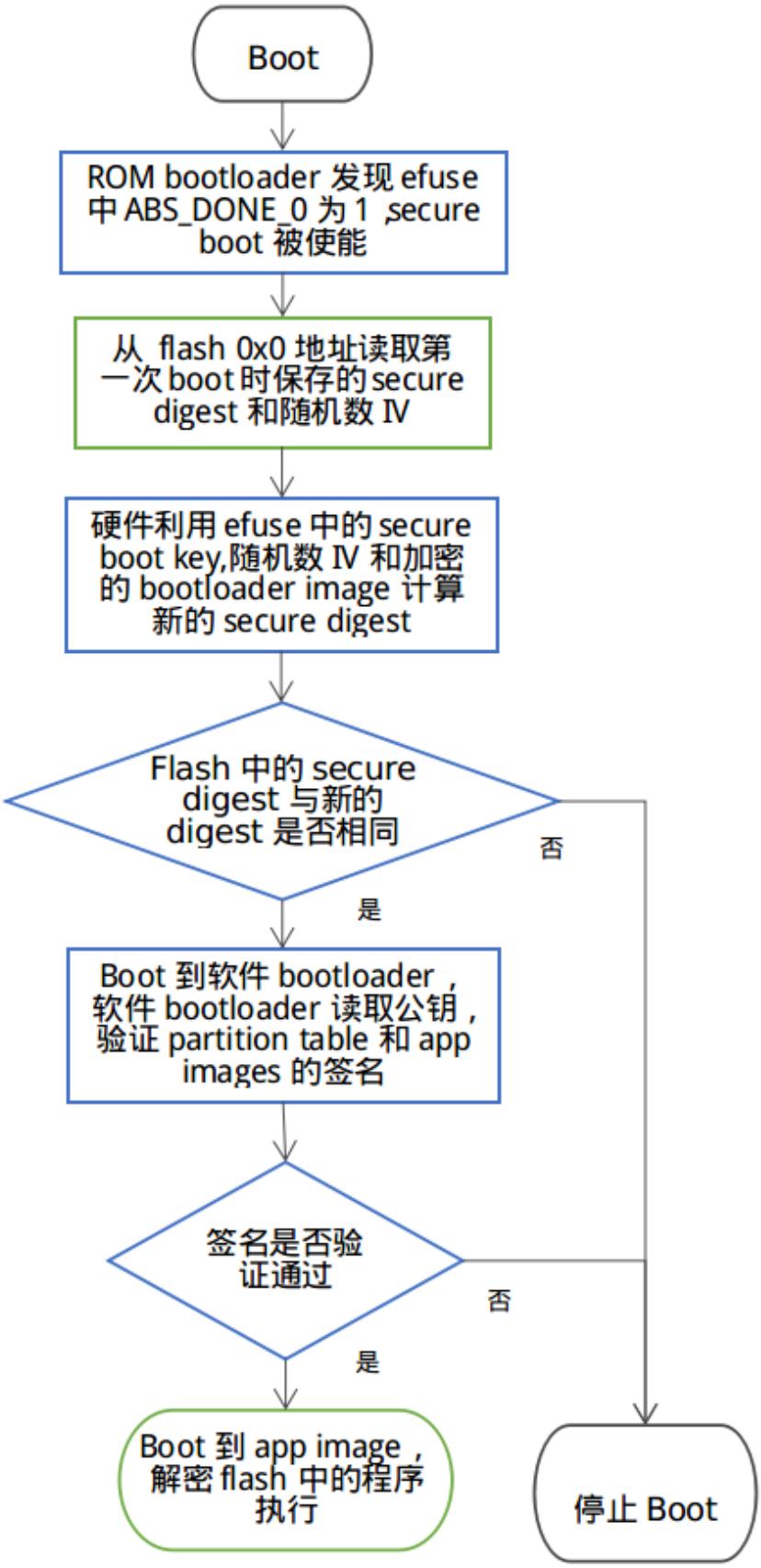
```
make partition_table
make app
```

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


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


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

```
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
```

11.3 启用安全加密的生产方案

11.3.1 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

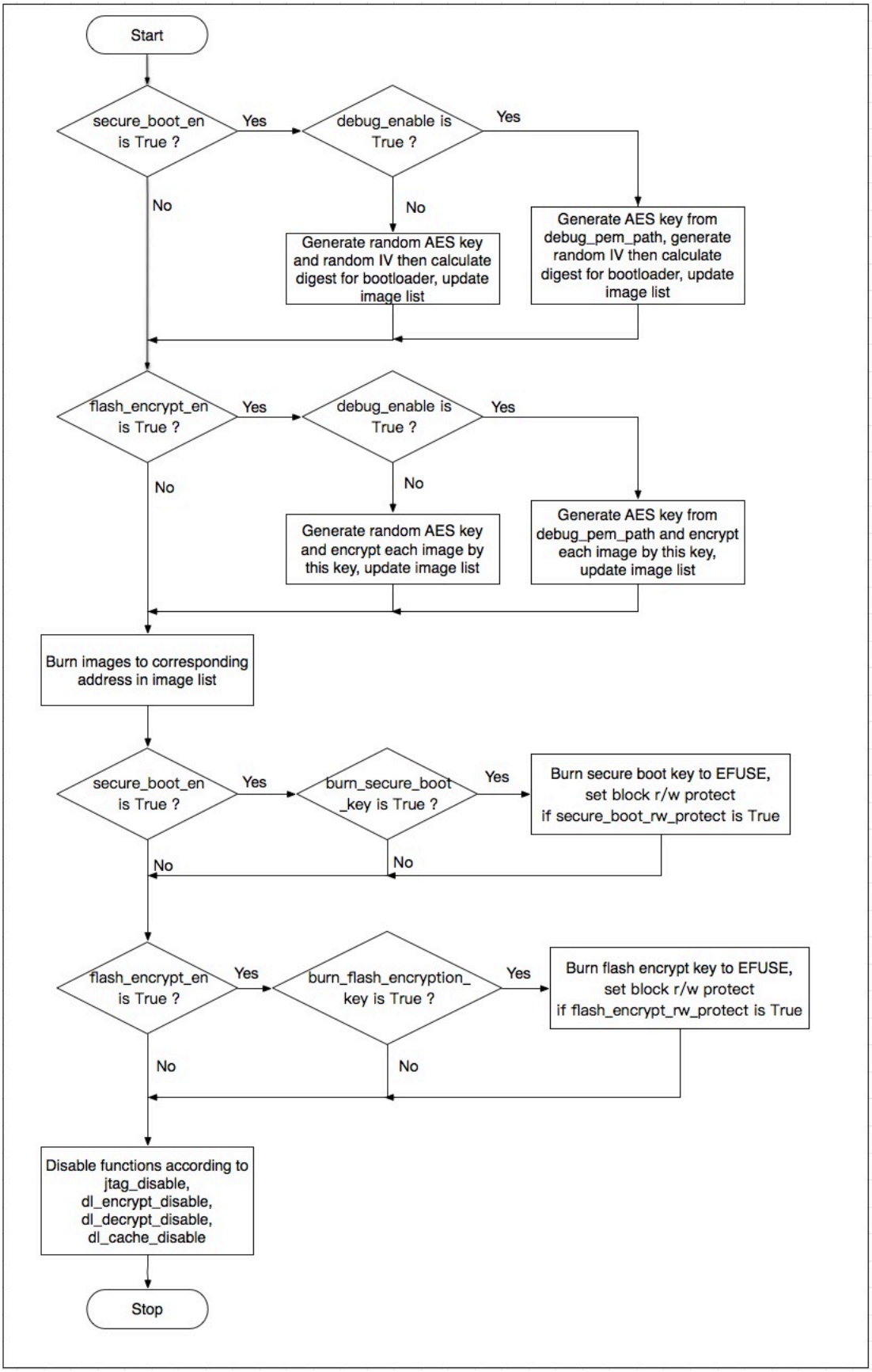
- 下载工具的内部逻辑和流程如下：

11.3.2 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
```


Start
secure_boot_en is True ?
Yes
No
debug_enable is True ?
Yes
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
No
debug_enable is True ?
Yes
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
No
burn_secure_boot _key is True ?
Yes
No
Burn secure boot key to EFUSE, set block r/w protect if secure_boot_rw_protect is True
flash_encrypt_en is True ?
Yes
No
burn_flash_encryption_ key is True ?
Yes
No
Burn flash encrypt key to EFUSE, set block r/w protect if flash_encrypt_rw_protect is True
Disable functions according to jtag_disable, dl_encrypt_disable, dl_decrypt_disable, dl_cache_disable
Stop

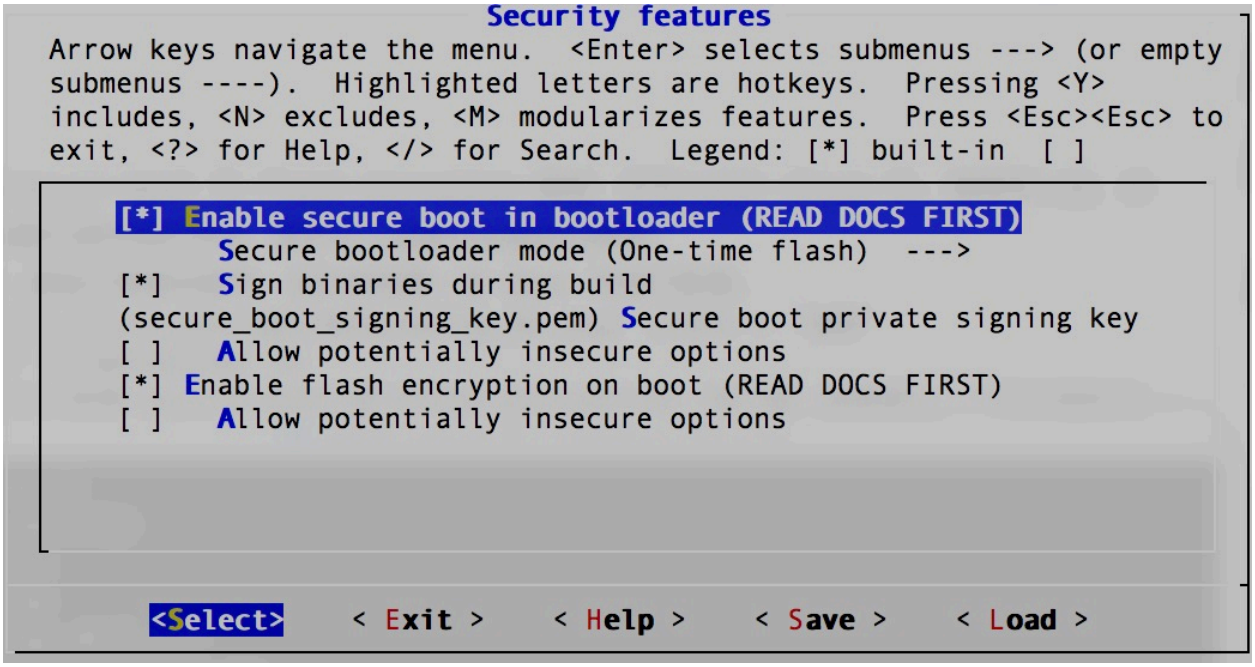
方案 1: 通过 bootloader 完成 security 特性初始化

- 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。
- 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：

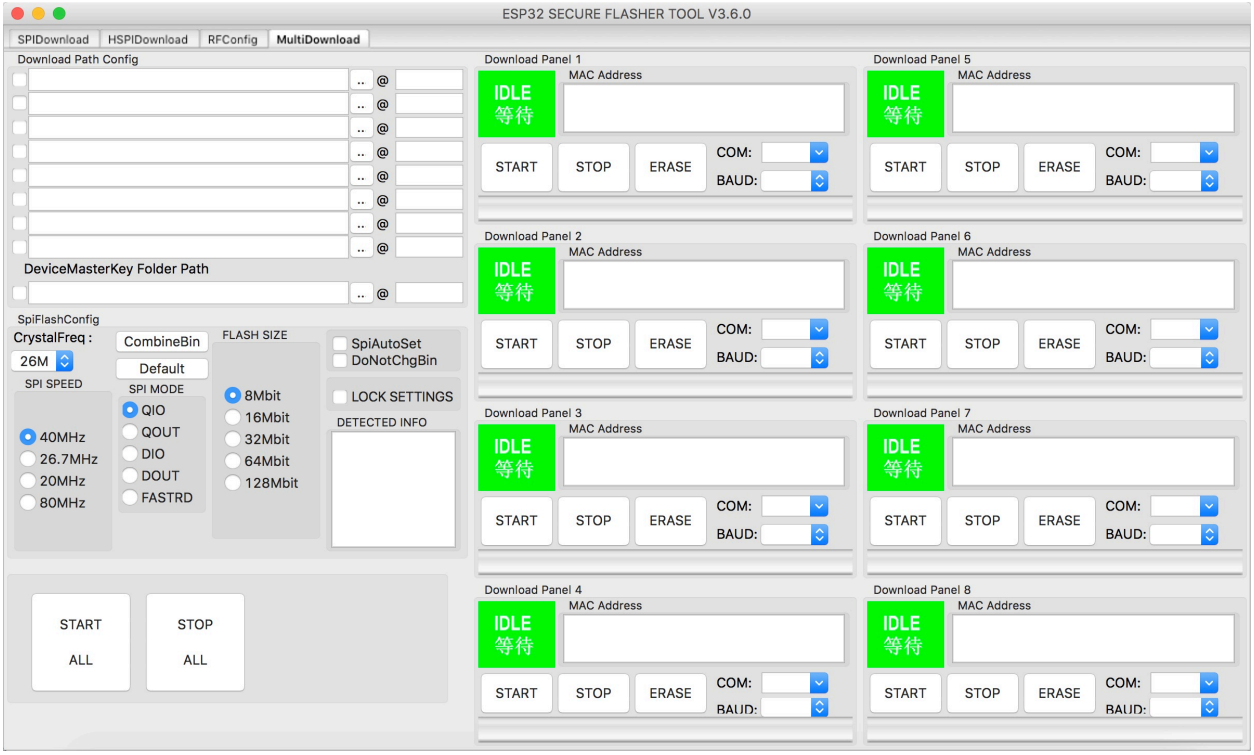
```
bootloader.bin -->   0x1000
partition.bin  -->   0x8000
app.bin        -->  0x10000
python $IDF_PATH/components/esptool_py/esptool/esptool.py --chip esp32 --port␣
↪/dev/cu.SLAB_USBtoUART --baud 1152000 --before default_reset --after no_
↪reset write_flash -z --flash_mode dio --flash_freq 40m --flash_size 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
```

注解： 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
```

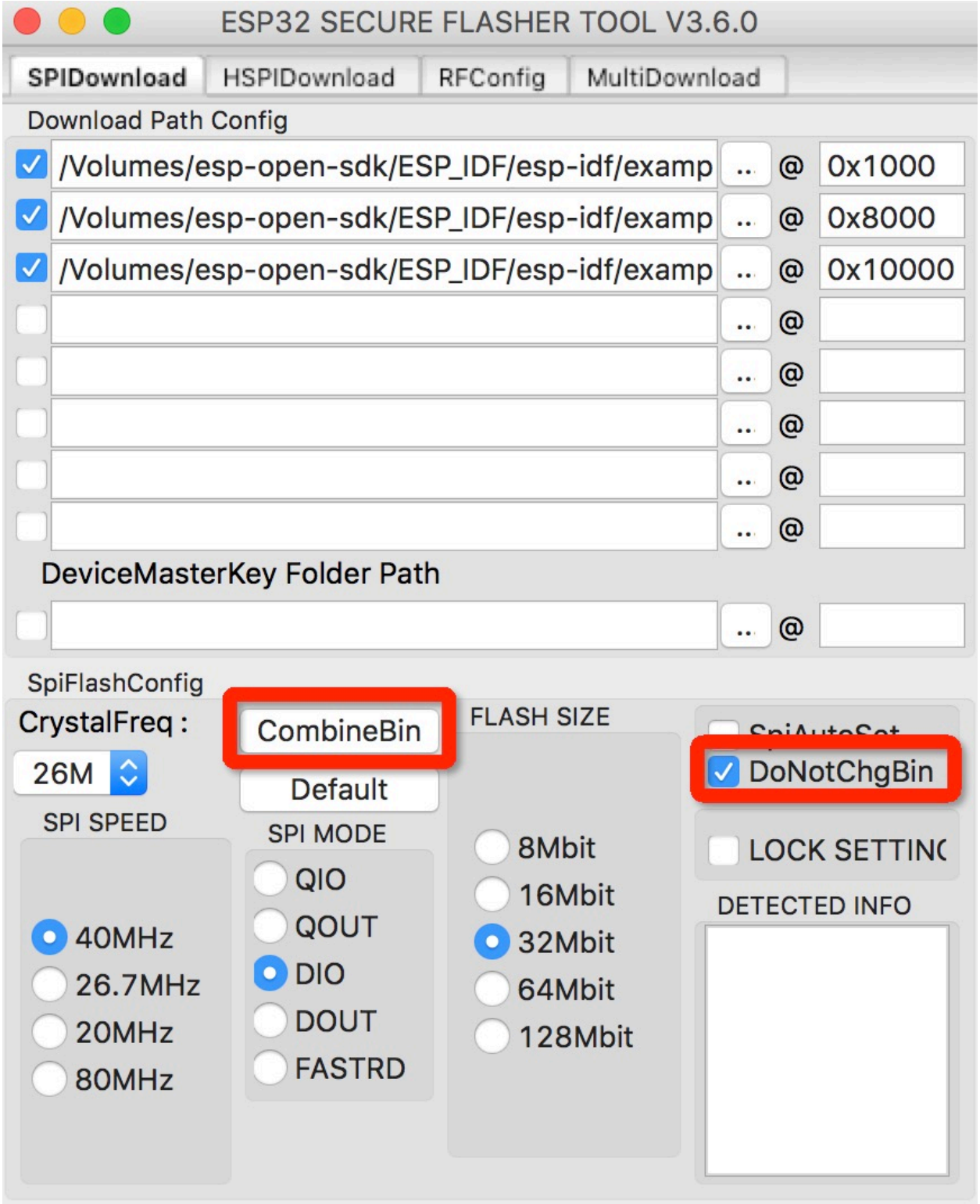
注解： 修改并保存参数前，请先关闭下载工具，配置文件修改完成并保存后，再开启运行下载工具。



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。

注解： 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


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
SPI SPEED
40MHz
26.7MHz
20MHz
80MHz
SPI MODE
QIO
QOUT
DIO
DOUT
FASTRD
8Mbit
16Mbit
32Mbit
64Mbit
128Mbit
LOCK SETTING
DETECTED INFO

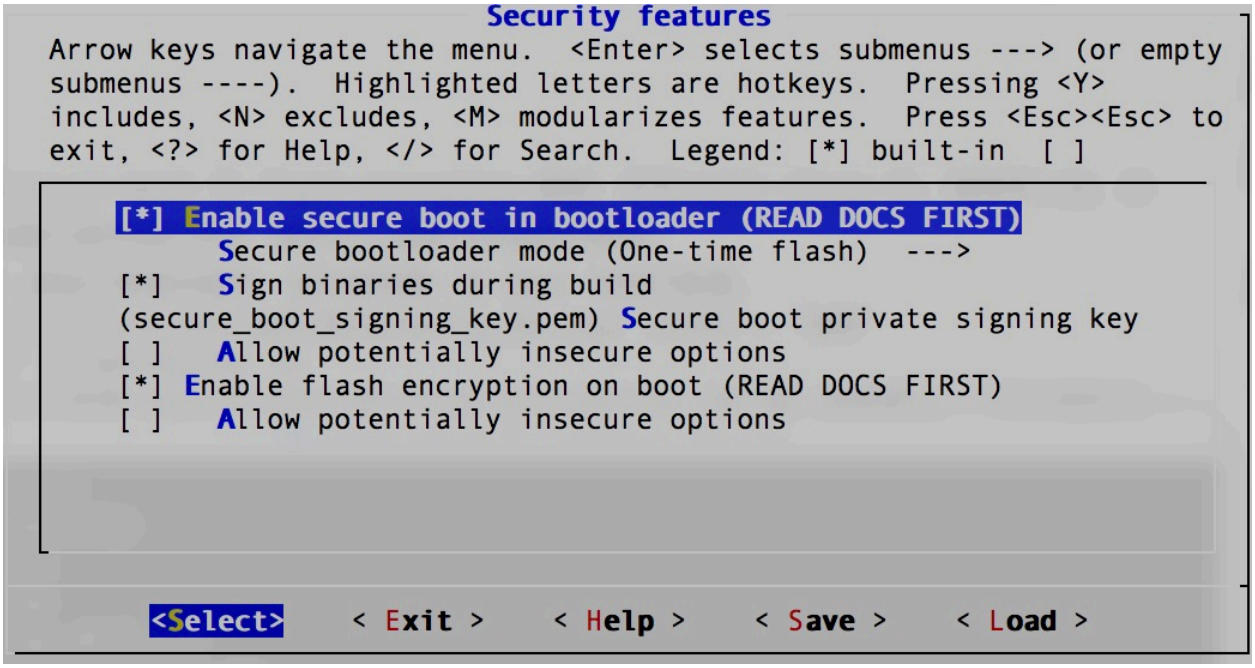
- 若第一次执行 bootloader，flash 加密进行到一半芯片掉电
  * 没有使能 secure boot 时，可重新将 images 明文烧写到 flash 中，让芯片下次 boot 时重新加密 flash
  * 使能了 secure boot 时，由于无法重新烧写 flash，芯片将永久无法 boot

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
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 12

其它资源

[English]

12.1 GPIO 扩展

[English]

随着 ESP32 系列芯片应用领域的进一步扩展，更多需求多样的应用场景正在被不断导入，包括一些对 GPIO 数量有较大要求的场景。乐鑫后续发布的 ESP32-S2 等产品包含多达 43 个 GPIO，可大大缓解 GPIO 资源紧张的问题。如果依旧无法满足需求，还可通过为 ESP32 添加 GPIO 扩展芯片解决，例如使用基于 I2C 接口控制的 GPIO 扩展模块 MCP23017，每个模块可外扩 16 个 GPIO 口，同时可挂载多达 8 个扩展模块，即可额外扩展 128 个 GPIO 口。

12.1.1 已适配列表

名称	功能	总线	供应商	规格书	驱动
MCP23017	16-bit I/O expander	I2C	Microchip	规格书	mcp23017

12.2 ADC 扩展量程方案

[English]

12.2.1 ESP32-S3 ADC 扩展量程方案

ESP32-S3 ADC 最大有效量程为 0 ~ 3100 mV，通过外部分压电路，能够满足大部分的 ADC 按键、电池电压检测等功能。但是对于 NTC 等应用场景，可能需要支持满量程（0 ~ 3300 mV）测量。ESP32-S3 可通过配置寄存器调整 ADC 的偏置电压，再结合高电压区域非线性补偿方法，我们可以实现对 ESP32-S3 量程的扩展。

实现量程扩展的过程为：

1. 使用默认偏置，测量一次电压值
2. 当测量电压值小于 2900 mV，直接输出测量电压值作为测量结果
3. 当测量电压值大于 2900 mV，提升偏置电压，进行二次测量，并进行非线性修正计算，输出修正值作为测量结果
4. 测量完成，恢复偏置电压

因此，每次 ADC 测量，期间可能读取 1~2 次实际电压值。对于大部分的应用场景，该方案引入的测量延迟可以忽略不计。

12.2.2 Patch 使用方法

当前 Patch 基于 ESP-IDF `release/v4.4` 分支开发：

1. 确认 ESP-IDF 已经 `checkout` 到 `release/v4.4` 分支
2. 下载 `esp32s3_adc_range_to_3100.patch` 文件
3. 使用指令 `git am --signoff < esp32s3_adc_range_to_3100.patch` 将 Patch 应用到 IDF 中
4. 请注意，该方案仅对 `esp_adc_cal_get_voltage` 接口有效，用户可直接调用该接口获取扩展后的读数

12.2.3 API 使用说明

1. 如果想要读取量程扩展后的电压值，用户必须使用 `esp_adc_cal_get_voltage` 直接获取 `ADC1` 或 `ADC2` 的通道电压。
2. ESP-IDF ADC 其它 API 不受影响，读取的结果和默认结果一致

CHAPTER 13

Contributions Guide

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

13.1 How to Contribute

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

13.2 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* ?
- 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.

13.3 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.

13.4 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.

13.5 Related Documents

13.5.1 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
```

(下页继续)

(续上页)

```
*/
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）

```
// Copyright 2019-2020 Espressif Systems (Shanghai) PTE LTD
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at

//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
```

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	command	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:
        // ...
}
```

(下页继续)

(续上页)

```
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
```

(下页继续)

(续上页)

```
    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。

索引

符号

B

C

D

F

G

H

I

K

L

M

N

P

R

S

T

U

V