# CONTENTS

1 Get Started  
  1.1 ESP-IoT-Solution Introduction ................................................. 3  
  1.2 ESP-IDF Introduction ............................................................... 4  
  1.3 ESP32/ESP32-S Introduction ...................................................... 4  
  1.4 Setting up Development Environment .......................................... 4  
  1.5 Use ESP-IoT-Solution Components ............................................ 5  
  1.6 Build and Download ................................................................. 5  
  1.7 Related Documents ...................................................................... 7  

2 Basic Component  
  2.1 Communication Bus .................................................................... 9  
  2.2 I2S LCD ...................................................................................... 23  
  2.3 Boards Component ..................................................................... 26  

3 Display  
  3.1 Screen ....................................................................................... 31  
  3.2 Digital Tube ............................................................................... 46  
  3.3 LED Indicator ........................................................................... 50  

4 Audio  
  4.1 PWM Audio .............................................................................. 55  
  4.2 DAC Audio ................................................................................. 61  

5 GUI  
  5.1 LVGL Graphics Library ............................................................... 65  

6 Input Device  
  6.1 ................................................................................................ 69  
  6.2 Touch Panel ............................................................................... 73  

7 Sensors  
  7.1 Sensor Hub ............................................................................... 81  
  7.2 Humidity and Temperature Sensor .............................................. 91  
  7.3 Inertial Measurement Unit (IMU) ............................................... 94  
  7.4 Ambient Light Sensor ................................................................. 98  
  7.5 Pressure Sensor ......................................................................... 101  
  7.6 Gesture Sensor .......................................................................... 101  

8 Storage  
  8.1 Storage Media ........................................................................... 103  
  8.2 File System ................................................................................ 105
This is the documentation for ESP-IoT-Solution Development Framework.

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

<table>
<thead>
<tr>
<th>Get Started</th>
<th>Display</th>
<th>Audio</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI</td>
<td>Input</td>
<td>Sensors</td>
</tr>
<tr>
<td>Security&amp;Encryption</td>
<td>Contribute</td>
<td></td>
</tr>
</tbody>
</table>
This document is intended to help you set up the development environment for ESP-IoT-Solution (Espressif IoT Solution). After that, a simple example will show you how to use ESP-IoT-Solution to set up environment, create a project, build and flash firmware onto an ESP32/ESP32-S series board, etc.

1.1 ESP-IoT-Solution Introduction

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

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

1.1.1 ESP-IoT-Solution Versions

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

<table>
<thead>
<tr>
<th>ESP-IoT-Solution version</th>
<th>Corresponding ESP-IDF version</th>
<th>Main changes</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>master</td>
<td>release/v4.3</td>
<td>Code structure changed, added ESP32-S2 support</td>
<td>New features development branch</td>
</tr>
<tr>
<td>release/v1.1</td>
<td>v4.0.1</td>
<td>IDF version updated, deleted codes that have been moved to other repos</td>
<td>Limited maintenance</td>
</tr>
<tr>
<td>release/v1.0</td>
<td>v3.2.2</td>
<td>Legacy version</td>
<td>Stop maintenance</td>
</tr>
</tbody>
</table>
1.2 ESP-IDF Introduction

ESP-IDF is the IoT development framework for ESP32/ESP32-S2 series SoCs provided by Espressif, including:

- A series of libraries and header files, providing core components required for building software projects based on ESP32/ESP32-S2;
- Common tools and functions used during the development and manufacturing processes, e.g., build, flashing, debugging, measurement and etc.

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

1.3 ESP32/ESP32-S Introduction

You can select any development board from ESP32/ESP32-S series to get started with ESP-IoT-Solution, or select a supported board from the Boards Component directly for a quick start.

ESP32/ESP32-S series SoCs support the following features:

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

ESP32/ESP32-S series of SoCs are designed with the 40 nm technology, showing the best power and RF performance, versatility and reliability in a wide variety of application and power scenarios.

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

1.4 Setting up Development Environment

1.4.1 1. Get ESP-IDF

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

For master version, please use the following command:

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

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

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

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

1.5 Use ESP-IoT-Solution Components

The following ways can be used to add ESP-IoT-Solution components:

1. Add all components from ESP-IoT-Solution to the project directory by inserting the following code to `CMakeLists.txt` under the project:

   ```cmake
   cmake_minimum_required(VERSION 3.5)
   include($ENV{IOT_SOLUTION_PATH}/component.cmake)
   include($ENV{IDF_PATH}/tools/cmake/project.cmake)
   project(empty-project)
   ```

2. Add specific components from ESP-IoT-Solution to the project directory by inserting the following code to `CMakeLists.txt` under the project:

   ```cmake
   set(EXTRA_COMPONENT_DIRS "$ENV{IOT_SOLUTION_PATH}/components/{component_you_choose}"
   #Please replace {component_you_choose} with the component name you choose.
   #This command can be repeated if you need to add multiple components.
   ```

3. Copy specific components from ESP-IoT-Solution to the project directory by directly copying and pasting the components and their dependencies into the `components` folder under the project.

**Note:** It is recommended to use the build system based on CMake (the default build system for ESP-IDF v4.0 and later versions) for ESP-IoT-Solution. If you need to use the GNU Make system, please refer to Build System (Legacy GNU Make).

1.6 Build and Download

1.6.1 1. Set up the environment variables

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

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

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

Please remember to replace the paths in above commands as your actual paths.

- Add IOT_SOLUTION_PATH environment variables:

  For Windows system, please open the Command Prompt and run:

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

  For Linux and macOS, please run:

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

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

### 1.6.2 2. Set build target

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

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

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

```
idf.py menuconfig
```

### 1.6.3 3. Build and download the program

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

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

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

Use the idf.py tool to see logs:

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

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

1.7 Related Documents

- ESP-IDF Installation Step by Step
- ESP-IDF Get Started
- ESP Product Selector
2.1 Communication Bus

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

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

This component abstracts the following concepts:

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

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

![i2c_bus Connection Diagram](image-url)

Fig. 1: i2c_bus Connection Diagram
2.1.1 How to Use i2c_bus

1. Create a bus: create a bus object using `i2c_bus_create()`. During the process, you need to specify the I2C port number and the bus configuration option `i2c_config_t`, which includes SDA and SCL pin numbers, pull-up and pull-down modes, as these are determined when the system is designed and normally will not be changed at runtime. The bus configuration option also includes the default clock frequency of the bus, which is used when the device does not specify a frequency.

2. Create a device: use `i2c_bus_device_create()` to create a device on the bus object created in the first step. During the process, you need to specify bus handle, the I2C address of the device, and the clock frequency when the device is running. The frequency will be dynamically changed during I2C transmission based on device configuration options. The device clock frequency can be configured as 0, indicating the current bus frequency is used by default.

3. Data reading: use `i2c_bus_read_byte()` or `i2c_bus_read_bytes()` to read Byte data; use `i2c_bus_read_bit()` or `i2c_bus_read_bits()` to read bit data. During the process, you only need to pass in the device handle, the device register address, a buffer to hold the read data, the read length, etc. The register address can be configured as `NULL_I2C_MEM_ADDR` for devices without internal registers.

4. Data writing: use `i2c_bus_write_byte()` or `i2c_bus_write_bytes()` to write Byte data; use `i2c_bus_write_bit()` or `i2c_bus_write_bits()` to write bit data. During the process, you only need to pass in the device handle, the device register address, the data location to be written, the write length, etc. The register address can be configured as `NULL_I2C_MEM_ADDR` for devices without internal registers.

5. Delete device and bus: if all i2c_bus communication has been completed, you can free your system resources by deleting devices and bus objects. Use `i2c_bus_device_delete()` to delete created devices respectively, then use `i2c_bus_delete()` to delete bus resources. If the bus is deleted with the device not being deleted yet, this operation will not take effect.

Example:

```c
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
```
Note: For some special application scenarios:

1. When the address of a register is 16-bit, you can use `i2c_bus_read_reg16()` or `i2c_bus_write_reg16()` to read or write its data;

2. For devices that need to skip the address phase or need to add a command phase, you can operate using `i2c_bus_cmd_begin()` combined with `I2C command link`.

2.1.2 How to Use spi_bus

1. Create a bus: use `spi_bus_create()` to create a bus object. During the process, you need to specify the SPI port number (can choose between SPI2_HOST and SPI3_HOST) and the bus configuration option `spi_config_t`, which includes the pin numbers of MOSI, MISO and SCLK, as these are determined when the system is designed and normally will not be changed at runtime. The bus configuration option also includes `max_transfer_sz` to configure the maximum data size during a transmission. When `max_transfer_sz` is configured to 0, it means the maximum size will be the default value 4096.

2. Create a device: use `spi_bus_device_create()` to create a device on the bus object created in the first step. During the process, you need to specify the bus handle, the CS pin number of the device, device operation mode, the clock frequency when the device is running. The device mode and frequency will be dynamically changed during SPI transmissions based on device configuration options.

3. Data transmission: use `spi_bus_transfer_byte()`, `spi_bus_transfer_bytes()`, `spi_bus_transfer_reg16()` or `spi_bus_transfer_reg32()` to transfer data directly. Data send and receive can be operated at the same time since SPI communication is a full-duplex communication. During the process, you only need to pass in the device handle, data to be transmitted, a buffer to hold the read data, transmission length, etc.

4. Delete device and bus: if all spi_bus communication has been completed, you can free your system resources by deleting devices and bus objects. Use `spi_bus_device_delete()` to delete created devices respectively, then use `spi_bus_delete()` to delete bus resources. If the bus is deleted with the device not being deleted yet, this operation will not take effect.

Example:
```c
spi_bus_handle_t bus_handle = NULL;
spi_bus_device_handle_t device_handle = NULL;
uint8_t data8_in = 0;
uint8_t data8_out = 0xff;
uint16_t data16_in = 0;
uint32_t data32_in = 0;

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

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

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

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

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

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

### 2.1.3 Adapted IDF Versions

- ESP-IDF v4.0 and later versions.
2.1.4 Supported Chips

- ESP32
- ESP32-S2

2.1.5 API Reference

i2c_bus API Reference

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, higher 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 enabled to achieve multiple devices with different config 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

```c
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

```c
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.

```c
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_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

```c
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_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.

```c
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

2.1. Communication Bus
• 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_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

Type Definitions

- typedef void *i2c_bus_handle_t
  - i2c bus handle

- typedef void *i2c_bus_device_handle_t
  - i2c device handle

spi_bus API Reference

Header File

- bus/include/spi_bus.h
**Functions**

```c
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`

```c
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.

```c
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`

```c
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.

```c
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

2.1. Communication Bus
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.

```c
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

```c
struct spi_config_t
```

spi bus initialization parameters.

**Public Members**

```c
gpio_num_t miso_io_num
```

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

```c
gpio_num_t mosi_io_num
```

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

```c
gpio_num_t clk_io_num
```

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

```c
int max_transfer_sz
```

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

```c
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

2.2.1 API

Header File

• bus/include/i2s_lcd_driver.h

Functions

i2s_lcd_handle_t i2s_lcd_driver_init (const i2s_lcd_config_t *config)
    Initialize i2s lcd driver.

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 deinitialize

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(\textit{i2s_lcd_handle_t} handle) 
acquire a lock

\textbf{Return} Always return ESP_OK

\textbf{Parameters}
- handle: Handle of i2s lcd driver

esp_err_t i2s_lcd_release(\textit{i2s_lcd_handle_t} handle) 
release a lock

\textbf{Return} Always return ESP_OK

\textbf{Parameters}
- handle: Handle of i2s lcd driver

\section*{Structures}

\textbf{struct} \texttt{i2s_lcd_config_t} 
Configuration of i2s lcd mode.

Handle of i2s lcd driver

\textbf{Public Members}

\texttt{int8_t} \texttt{data_width} 
Parallel data width, 16bit or 8bit available

\texttt{int8_t} \texttt{pin_data_num}[16] 
Parallel data output IO

\texttt{int8_t} \texttt{pin_num_cs} 
CS io num

\texttt{int8_t} \texttt{pin_num_wr} 
Write clk io

\texttt{int8_t} \texttt{pin_num_rs} 
RS io num

\texttt{int} \texttt{clk_freq} 
I2s clock frequency

\texttt{i2s_port_t} \texttt{i2s_port} 
I2S port number

\texttt{bool} \texttt{swap_data} 
Swap the 2 bytes of RGB565 color

\texttt{uint32_t} \texttt{buffer_size} 
DMA buffer size
Macros

LCD_CMD_LEV
LCD_DATA_LEV

Type Definitions

typedef void *i2s_lcd_handle_t

2.3 Boards Component

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

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

The following figure shows the structure of the Boards component:

![Boards Component Diagram](image)

- The Boards component contains several folders named after the development board’s name, as well as its CMakeLists.txt and Kconfig.projbuild files, therefore, its configuration option will be included in menuconfig;
- The board.h and board.c files must be included in the board file, while kconfig.in is one of the non-required file, which provides configuration options specific to a board.

**Note:** The Boards component is provided in examples/common_components/boards.
2.3.1 Instructions

1. Initialize development board: use `iot_board_init` in `app_main` to initialize the development board. you can also do some configurations regarding this process using *The Switch and Configuration of a Development Board* in `menuconfig`;

2. Get the handle of a peripheral: use `iot_board_get_handle` and `board_res_id_t` to get peripheral resources. `NULL` will be returned if this peripheral is not initialized;

3. Operate on peripherals with handles directly.

Example:

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

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

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

2.3.2 The Switch and Configuration of a Development Board

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

1. Choose a target board: choose a development board in `menuconfig`->`Board Options`->`Choose Target Board`;

2. Configure the parameter of your board: the configuration options provided by the current development board are included in `xxxx Board Options`, including whether `i2c_bus` should be initialized during the initialization stage of the board, the power status of sensor peripherals when the board is started, etc. The configurable options will be determined by the maintainer of the development board;

3. Use `idf.py build flash monitor` to re-build and download code.

**Note:** The default target of this build system is `ESP32`, please set the target before building via `idf.py set-target esp32` if you need to use ESP32-S2.
### 2.3.3 Supported Development Boards

<table>
<thead>
<tr>
<th>ESP32 Development Boards</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp32-devkitc</td>
</tr>
<tr>
<td>esp32-meshkit-sense</td>
</tr>
<tr>
<td>esp32-lcdkit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ESP32-S2 Development Boards</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp32s2-saola</td>
</tr>
</tbody>
</table>
2.3.4 Add a New Development Board

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

The main process is as follows:

1. Prepare the necessary board.h and board.c files according to component file structure. You can refer to Mandatory APIs;
2. Add configuration options specific to this board in kconfig.in according to your needs;
3. Add the information of this board to Kconfig.projbuild for users;
4. Add the directory of this board to CMakeLists.txt so that it can be indexed by the build system. Please also update component.mk if you need to support the old make system.

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

Mandatory APIs

```c
/**
 * @brief Board level init.
 * Peripherals can be chosen through menuconfig, which will be initialized with default configurations during iot_board_init.
 * After board init, initialized peripherals can be referenced by handles.
 *
 * @return esp_err_t
 */
esp_err_t iot_board_init(void);

/**
 * @brief Board level deinit.
 * After board deinit, initialized peripherals will be deinit and related handles will be set to NULL.
 *
 * @return esp_err_t
 */
esp_err_t iot_board_deinit(void);

/**
 * @brief Check if board is initialized
 *
 * @return true if board is initialized
 * @return false if board is not initialized
 */
bool iot_board_is_init(void);

/**
 * @brief Using resource's ID declared in board_res_id_t to get board level resource's handle
 *
 * @param id Resource's ID declared in board_res_id_t
 * @return board_res_handle_t Resource's handle
 * if no related handle, NULL will be returned
 */
```
2.3.5 Component Dependencies

- Common dependencies: the bus component.

2.3.6 Adapted IDF Versions

- ESP-IDF v4.0 and later versions.

2.3.7 Supported Chips

- ESP32
- ESP32-S2
3.1 Screen

Screen is a very important display device as many information from various applications needs to be displayed to users. Both ESP32 and ESP32-S2 chips support screens driven by I2C interface, 8080 parallel interface, SPI interface and etc. The supported types of screen controllers are listed in the following table:

<table>
<thead>
<tr>
<th>Controller</th>
<th>Max Resolution</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NT35510</td>
<td>480 x 865</td>
<td>Color</td>
</tr>
<tr>
<td>ILI9806</td>
<td>480 x 865</td>
<td>Color</td>
</tr>
<tr>
<td>RM68120</td>
<td>480 x 865</td>
<td>Color</td>
</tr>
<tr>
<td>ILI9486</td>
<td>320 x 480</td>
<td>Color</td>
</tr>
<tr>
<td>ILI9341</td>
<td>240 x 320</td>
<td>Color</td>
</tr>
<tr>
<td>ST7789</td>
<td>240 x 320</td>
<td>Color</td>
</tr>
<tr>
<td>ST7796</td>
<td>320 x 480</td>
<td>Color</td>
</tr>
<tr>
<td>SSD1351</td>
<td>128 x 128</td>
<td>Color</td>
</tr>
<tr>
<td>SSD1306</td>
<td>128 x 64</td>
<td>Mono</td>
</tr>
<tr>
<td>SSD1307</td>
<td>128 x 39</td>
<td>Mono</td>
</tr>
<tr>
<td>SSD1322</td>
<td>480 x 128</td>
<td>Gray</td>
</tr>
</tbody>
</table>

Note: The 8080 parallel interface is implemented via the LCD mode in the I2S of ESP32, so sometimes it is called I2S interface in this document.

3.1.1 Screen Driver Structure

In order to be more in line with the actual situation where a screen controller has multiple interfaces, the screen driver is divided into two parts: the interface driver and the controller driver.

- The interface driver: conduct basic reads and writes of commands and data
- The controller driver: display information on screen via interfaces

A controller driver can be designed to switch between different interfaces in hardware level by calling corresponding interface drivers.
3.1.2 Screen Types

A discussion about screen types will help us to have a clear understanding of drivers. Here, we use colors that can be displayed on the screen to classify screens, rather than the panel material of them such as OLED, LCD and etc. In general, the colors displayed on screen determines the BPP (Bits Per Pixel), and the differences in BPP lead to differences in how the program handles it. Here, we list some ways in which GRAM is mapped to pixel points in below:

From above figures, we can see that there are mainly two types of mapping:

- When BPP >= 8, it is usually a color screen that supports RGB888, RGB666, RGB565 and other codings.
- When BPP < 8, it is usually a mono screen that may either be black-and-white or gray.

When BPP < 8, a byte is mapped to multiple pixels, so a single pixel cannot be controlled directly. In this case, `draw_pixel()` is not supported in the driver, and the parameters of `set_window()` are also limited. When BPP >= 8, each single pixel can be accessed easily.

**Attention:** For color screens, the driver only supports RGB565 color coding.
Fig. 2: BPP = 16 GRAM Structure

Fig. 3: BPP = 1 GRAM Structure

Fig. 4: BPP = 4 GRAM Structure

3.1. Screen
3.1.3 Interface Driver

A screen controller usually has multiple interfaces. On ESP32, three kinds of interfaces as 8080 parallel interface, SPI and I2C are typically used to connect to the screen. You can choose one of them as the interface when creating interface drivers via `scr_interface_create()`.

**Note:** Please remember to select corresponding parameter types when creating different interfaces using `scr_interface_create()`, e.g., select `i2s_lcd_config_t` for I2S interface; select `scr_interface_spi_config_t` for SPI interface.

To facilitate the use of these interfaces in the driver, all interfaces are defined in `display/screen/screen_utility/interface_drv_def.h`, which can be called easily by less parameters.

**Note:** Most screens use big-endian order to store data, while ESP32 uses small-endian mode. You can switch between them in the interface driver you used, based on `swap_data` configurations. **Please note:** when using the SPI interface, the received data must be stored in RAM because the IDF’s SPI driver itself does not support this swapping function and an additional program in the interface driver will do the work, which require the data to be writable.

3.1.4 Controller Driver

Some common functions of the screen are abstracted using `scr_driver_t` in this section according to display and other functions of different screen controllers, in order to port these common functions to different GUI libraries easily. For some non-generic functions of the screen, you need to call its specific functions.

Not all screens have implemented these common functions, since different screen controller has their own functions. For example, for screens with BPP < 8, the function `draw_pixel()` is not supported. And calling an unsupported function will return `ESP_ERR_NOT_SUPPORTED`.

**Display Direction**

The screen display direction set here is implemented entirely by the screen hardware, and this feature varies from one screen controller to another. There are 8 possible display directions. A display can be rotated by 0°, 90°, 180° or 270° and can also be viewed from the top or bottom, with 0° and top view as its default direction. These 8 (4 × 2) directions can also be represented as a combination of 3 binary switches: X-mirroring, Y-mirroring and X/Y swapping.

The total 8 combinations of display directions are listed in the following table. If the direction of your display is not correct, please check the configuration switches below to make it work properly.
The implementations of display directions are not exactly the same for different screen controllers, and are usually divided into the following cases:

- For color screens, 8 directions are supported.
- For mono screens, e.g., SSD1306, only the first 4 directions defined in `scr_dir_t` are supported, which means they do not support X/Y swapping.

**Note:** The display direction is also related to the screen panel you used, and you may encounter two types of abnormal cases:

- The display direction is set to `SCR_DIR_LRTB`, but the screen does not show as what listed in the above table. This may be because the alignment on the screen panel is mirrored in the X/Y direction, in which case you need to adjust the rotation to get the desired direction.
- After rotated, the screen does not show anything any more. This may be because the resolution of the screen panel is smaller than that of the screen controller, making the display area not falling completely on the screen panel, in which case you need to set a proper offset for the display area.

### Offset of the Display Area

In some small screens, the resolution of the display area is usually smaller than that of the controller window. Please refer to the following figure:
In this figure, **Controller window** is the window for screen controller, with its resolution as 240 × 320; **Panel window** is the window for screen panel, with its resolution as 135 × 240, which is the display area. From this figure, we can see that the display area is shifted by 52 pixels horizontally and by 40 pixels vertically.

When the screen is rotated 90° anticlockwise, the display area is shifted by 40 pixels horizontally and by 53 pixels vertically, as shown in the figure below:

The screen controller driver will help you to change the offset value automatically according to the rotation of the screen.
to maintain a proper display. All you need to do is to properly configure the screen offset and the screen panel size in `scr_controller_config_t` when it is in `SCR_DIR_LRTB` direction.

**Note:**
- This only supports screens with BPP >= 8.
- When the resolution of your screen controller is configurable and you find something wrong with the offset, it may be because the selected resolution does not match the actual one, and you should make modifications accordingly, for example, set the `ILI9806_RESOLUTION_VER` in `ili9806.c` as the actual resolution for ILI9806.

### 3.1.5 Application Example

#### Initialize the Screen

```c
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;
sr_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) {
    return;
    ESP_LOGE(TAG, "screen find failed");
}
/** 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_LRTB,
```
/** Initialize ILI9806 screen */
g_lcd.init(&lcd_cfg);

Note: By default, only the driver of ILI9341 screen is enabled. If you need to use other drivers, please go to menu-config -> Component config -> LCD Drivers -> Select Screen Controller to enable the corresponding screen drivers.

Display Images

/** 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);

Obtain Screen Information

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);

3.1.6 API Reference

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

```c
struct scr_controller_config_t
configuration of screen controller
```

**Public Members**

```c
scr_interface_driver_t *interface_drv
Interface driver for screen

int8_t pin_num_rst
Pin to hard reset 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
```

```c
struct scr_info_t
Information of screen.
```

**Public Members**

```c
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
```

3.1. 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

```c
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

```c
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

```c
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

```c
esp_err_t(*get_info)(scr_info_t *info)
Get screen information.
```

**Return**
- ESP_OK on success
- ESP_FAIL Failed

---

### 3.1. Screen
Parameters

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

**Macros**

```c
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
COLORGRAY
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_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

```c
struct scr_interface_spi_config_t
    SPI interface configuration.
```

Public Members

```c
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
```

```c
struct scr_interface_i2c_config_t
    I2C interface configuration.
```

Public Members

```c
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
```

```c
struct scr_interface_driver_t
    Define common function for screen interface driver.
```

Public Members

```c
scr_interface_type_t type
    Interface bus type, see scr_interface_type_t struct
esp_err_t (*write_cmd)(void *handle, uint16_t cmd)
    Function to write a command
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
```
### 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 Digital Tube

Digital tube and dot matrix LEDs are common display solutions in embedded systems, which occupy fewer pins and memory resources than LCD displays and are simpler to implement, making them more suitable for application scenarios with single display requirements such as timing, counting, status display and etc.

The digital tube and LED display drivers adapted to ESP-IoT-Solution are shown in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Features</th>
<th>Interface</th>
<th>Driver</th>
<th>Datasheet</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH450</td>
<td>Digital tube display driving chip, supports 6-bit digital tube</td>
<td>I2C</td>
<td>ch450</td>
<td>CH450</td>
</tr>
<tr>
<td>HT16C21</td>
<td>20×4/16×8 LCD controller, supports RAM mapping</td>
<td>I2C</td>
<td>ht16c21</td>
<td>HT16C21</td>
</tr>
<tr>
<td>IS31FL3XXX</td>
<td>Dot matrix LED controller</td>
<td>I2C</td>
<td>is31fl3xxx</td>
<td>IS31FL3XXX</td>
</tr>
</tbody>
</table>

#### 3.2.1 CH450 Driver

CH450 is a digital tube display driving chip that can be used to drive a 6-bit digital tube or a 48-dot LED matrix and can communicate with ESP32 via the I2C interface.

This driver encapsulates the basic operations of CH450, and users can directly call `ch450_write()` or `ch450_write_num()` to display numbers on the digital tube.
Example

```c
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 Driver

HT16C21 is a LCD control/driver chip which supports RAM mapping and can be used to drive 20 x 4 or 16 x 8 segmented LCDs. The chip communicates with ESP32 via the I2C interface.

This driver encapsulates the basic operations of HT16C21. After creating an example using `ht16c21_create`, users can configure its parameters via `ht16c21_param_config` and then call `ht16c21_ram_write` directly to write data.
Example

```c
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_DISABLE,
   .scl_io_num = I2C_MASTER_SCL_IO,
   .scl_pullup_en = GPIO_PULLUP_DISABLE,
   .master.clk_speed = I2C_MASTER_FREQ_HZ,
};
i2c_bus = i2c_bus_create(I2C_MASTER_NUM, &conf);
seg = ht16c21_create(i2c_bus, HT16C21_I2C_ADDRESS_DEFAULT);

ht16c21_config_t ht16c21_conf = {
   .duty_bias = HT16C21_4DUTY_3BIAS,
   .oscillator_display = HT16C21_OSCILLATOR_ON_DISPLAY_ON,
   .frame_frequency = HT16C21_FRAME_160HZ,
   .blinking_frequency = HT16C21_BLINKING_OFF,
   .pin_and_voltage = HT16C21_VLCD_PIN_VOL_ADJ_ON,
   .adjustment_voltage = 0;
};
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 Driver

The IS31FL3XXX series chips can be used to drive dot matrix LED screens with different sizes. The IS31FL3218 supports 18 constant current channels, with each channel controlled by an independent PWM. It has a maximum output current of 38 mA and can directly drive LEDs for display. The IS31FL3736 supports more channels and can compose a maximum size of LED matrix as $12 \times 8$. With each channel driven by an 8-bit PWM driver, the IS31FL3736 can support up to 256 gradients.

![Fig. 7: IS31FL3218 Typical Application Circuit](image)

This driver encapsulates the basic operations of IS31FL3XXX. The example is shown in the next section.

**IS31FL3218 Example**

```c
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);
```

3.2. Digital Tube
3.3 LED Indicator

As one of the simplest output peripherals, LED indicators can indicate the current operating state of the system by blinking in different types. ESP-IoT-Solution provides an LED indicator component with the following features:

- Can define multiple groups of different blink types
- Can define the priority of blink types
- Can set up multiple indicators

3.3.1 Instructions

Pre-define Blink Types

The blink step structure `blink_step_t` defines the type of a step, indicator state and the state duration. Multiple steps are combined into one blink type, and different blink types can be used to identify different system states. The blink type is defined as follows:

Example 1. define a blink loop: turn on 0.05 s, turn off 0.1 s, and loop.

```c
const blink_step_t test_blink_loop[] = {
    {LED_BLINK_HOLD, LED_STATE_ON, 50},    // step1: turn on LED 50 ms
    {LED_BLINK_HOLD, LED_STATE_OFF, 100},   // step2: turn off LED 100 ms
    {LED_BLINK_LOOP, 0, 0},                // step3: loop from step1
};
```

Example 2. define a blink loop: turn on 0.05 s, turn off 0.1 s, turn on 0.15 s, turn off 0.1 s, and stop blink.

```c
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)
};
```

After defining a blink type, you need to add its corresponding enumeration member to `led_indicator_blink_type_t` and then add the type to the blink type list `led_indicator_blink_lists`, as the following example:

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

blink_step_t * led_indicator_blink_lists[] = {
    [BLINK_TEST_BLINK_ONE_TIME] = test_blink_one_time,
    [BLINK_TEST_BLINK_LOOP] = test_blink_loop,
    [BLINK_MAX] = NULL,
};
```
Pre-define Blink Priorities

For the same indicator, a high-priority blink can interrupt an ongoing low-priority blink, which will resume execution after the high-priority blink stop. The blink priority can be adjusted by configuring the enumeration member order of the blink type `led_indicator_blink_type_t`, the smaller order value the higher execution priority.

For instance, in the following example, `test_blink_one_time` has higher priority than `test_blink_loop`, and should blink first:

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

Control Indicator Blinks

Create an indicator by specifying an IO and a set of configuration information.

```c
led_indicator_config_t config = {
    .off_level = 0,              // attach led positive side to esp32...
    .gpio pin                   
        .mode = LED_GPIO_MODE,
};
led_indicator_handle_t led_handle = led_indicator_create(8, &config); // attach to...
```

Start/stop blinking: control your indicator to start/stop a specified type of blink by calling corresponding functions. The functions are returned immediately after calling, and the blink process is controlled by the internal timer. The same indicator can perform multiple blink types in turn based on their priorities.

```c
led_indicator_start(led_handle, BLINK_TEST_BLINK_LOOP);  // call to start, the...
/*
*/
led_indicator_stop(led_handle, BLINK_TEST_BLINK_LOOP);   // call stop
```

Delete an indicator: you can also delete an indicator to release resources if there are no further operations required.

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

Note: This component supports thread-safe operations. You can share the LED indicator handle `led_indicator_handle_t with global variables, or use `led_indicator_get_handle to get the handle in other threads via the LED's IO number for operation.
### 3.3.2 API Reference

#### 3.3.3 Header File

- led/indicator/include/led_indicator.h

#### 3.3.4 Functions

**led_indicator_handle_t led_indicator_create** (int io_num, 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**
  - io_num: gpio number of the led
  - config: configuration of the led, eg. gpio level when led off

**led_indicator_handle_t led_indicator_get_handle** (int io_num)

- get the handle of created led_indicator with gpio number

  **Return** led_indicator_handle_t handle of the created led indicator, NULL if not created

  **Parameters**
  - io_num: gpio number of the led

**esp_err_t led_indicator_delete** (led_indicator_handle_t *p_handle)

- delete the led indicator and release resource

  **Return** esp_err_t

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

  **Parameters**
  - p_handle: pointer to led indicator handle

**esp_err_t led_indicator_start** (led_indicator_handle_t handle, led_indicator_blink_type_t 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, led_indicator_blink_type_t 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

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

blink_step_state_t on_off
hold on or off, set NULL if not LED_BLINK_HOLD

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

bool off_level
gpio level of turn off. 0 if attach led positive side to esp32 gpio pin, 1 if attach led negative side

led_indicator_mode_t mode
led work mode, eg. gpio or pwm mode

3.3.6 Type Definitions

typedef void *led_indicator_handle_t
led indicator operation handle


3.3.7 Enumerations

enum blink_step_state_t
led on-off state

    Values:

    LED_STATE_OFF = 0
    turn off the led

    LED_STATE_ON = 1
    turn on the led

denum 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_LOOP
    loop from first step

denum led_indicator_mode_t
led indicator blink mode, as a member of led_indicator_config_t

    Values:

    LED_GPIO_MODE
    blink with max brightness

denum led_indicator_blink_type_t
The blink type with smaller index has the higher priority eg. BLINK_FACTORY_RESET priority is higher than BLINK_UPDATING.

    Values:

    BLINK_FACTORY_RESET
    restoring factory settings

    BLINK_UPDATING
    updating software

    BLINK_CONNECTED
    connected to AP (or Cloud) succeed

    BLINK_PROVISIONED
    provision done

    BLINK_CONNECTING
    connecting to AP (or Cloud)

    BLINK_RECONNECTING
    reconnecting to AP (or Cloud), if lose connection

    BLINK_PROVISIONING
    provisioning

    BLINK_MAX
    INVALIDED type
4.1 PWM Audio

The PWM audio function uses the internal LEDC peripheral in ESP32 to generate PWM audio, which does not need an external audio Codec chip. This is mainly used for cost-sensitive applications with low audio quality requirements.

4.1.1 Features

- Allows any GIPO with output capability as an audio output pin
- Supports 8-bit ~ 10-bit PWM resolution
- Supports stereo
- Supports 8 \( \frac{1}{2} \) 48 KHz sampling rate

4.1.2 Structure

![Fig. 1: Structure]
1. First, the data is recoded to meet the PWM input requirements, including the shift and offset of the data;
2. Send data to the ISR (Interrupt Service Routines) function of the Timer Group via Ring Buffer;
3. The Timer Group reads data from the Ring Buffer according to the pre-defined sampling rate, and write the data into LEDC.

**Note:** Since the output is a PWM signal, it needs to be low-pass filtered to get the audio waveform.

### 4.1.3 PWM Frequency

The frequency of the output PWM cannot be configured directly, but needs to be calculated by configuring the number of PWM resolution bits, as shown below:

\[
 f_{\text{pwm}} = \frac{f_{\text{APB_CLK}}}{2^{\text{res_bits}}} - \left(\frac{f_{\text{APB_CLK}}}{2^{\text{res_bits}}} \mod 1000\right)
\]

The \( f_{\text{APB_CLK}} \) here is 80 MHz, and \( \text{res_bits} \) is the number of PWM resolution bits. When the resolution is LEDC_TIMER_10_BIT, the PWM frequency is 78 KHz. As we all know, a higher PWM frequency and resolution can better reproduce the audio signal. However, this formula shows that increasing PWM frequency means lower resolution and increasing resolution means lower PWM frequency. Thus, please adjust these parameters according to the actual application scenario to achieve a good balance.

### 4.1.4 Application Example

```c
pwm_audio_config_t pac;
pac.duty_resolution = LEDC_TIMER_10_BIT;
pac.gpio_num_left = LEFT_CHANNEL_GPIO;
pac.ledc_channel_left = LEDC_CHANNEL_0;
pac.gpio_num_right = RIGHT_CHANNEL_GPIO;
pac.ledc_channel_right = LEDC_CHANNEL_1;
pac.ledc_timer_sel = LEDC_TIMER_0;
pac.tg_num = TIMER_GROUP_0;
pac.timer_num = TIMER_0;
pac.ringbuf_len = 1024 * 8;
pwm_audio_init(&pac)); //** Initialize pwm audio */
pwm_audio_set_param(48000, 8, 2); //** Set sample rate, bits and channel */
pwm_audio_start(); //** Start to run */

while(1) {
    //** Prepare audio data, such as decode mp3/wav file */
    //** Write data to play */
    pwm_audio_write(audio_data, length, &written, 1000 / portTICK_PERIOD_MS);
}
```
4.1.5 API Reference

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 port-MAX_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

```c
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

```c
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

```c
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**

```c
timer_group_t tg_num
    timer group number (0 - 1)

timer_idx_t timer_num
    timer number (0 - 1)

int gpio_num_left
    the LEDC output gpio_num, Left channel

int gpio_num_right
    the LEDC output gpio_num, Right channel

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

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

ledc_timer_t ledc_timer_sel
    Select the timer source of channel (0 - 3)

ledc_timer_bit_t duty_resolution
    ledc pwn bits

uint32_t ringbuf_len
    ringbuffer size
```

**Enumerations**

```c
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)
```

```c
PWM_AUDIO_CH_MAX
```
4.2 DAC Audio

ESP32 has two independent DAC channels and can play audio using I2S directly via DMA. The APIs in this document have been simplified on the basis of ESP-IDF, and the related data has been recoded to support more types of sampling bit width.

4.2.1 API Reference

Header File

- audio/dac_audio/include/dac_audio.h

Functions

```c
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
```

```c
esp_err_t dac_audio_deinit (void)
deinitialize dac

Return
- ESP_OK Success
- ESP_FAIL Encounter error
```

```c
esp_err_t dac_audio_start (void)
Start dac to play.

Return
- ESP_OK Success
- ESP_FAIL Encounter error
```

```c
esp_err_t dac_audio_stop (void)
Stop play.

Return
```
ESP_OK Success

ESP_FAIL Encounter error

dac_audio_set_param(int rate, int bits, int ch)
Configuration dac parameter.

Return

• ESP_OK Success
• ESP_FAIL Encounter error

Parameters

• rate: sample rate (ex: 8000, 44100…)
• bits: bit width
• ch: channel number

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_FAIL Encounter 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

Write data to play.

Return

• ESP_OK Success
• ESP_FAIL Write encounter error

Parameters

• inbuf: Pointer source data to write
• len: length of data in bytes
• 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 port-MAX_DELAY for no timeout.
Structures

```c
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
}
```
5.1 LVGL Graphics Library

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

5.1.1 Features

LVGL has the following features:

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

5.1.2 Requirements

The minimum requirements for running LVGL are listed as follows:

- 16, 32 or 64 bit micro-controller or processor
- Clock frequency: > 16 MHz
- Flash/ROM: > 64 kB (180 kB is recommended)
- RAM: 8 kB (24 kB is recommended)
- 1 frame buffer
ESP-IoT-Solution

• Graphics buffer: > "horizontal resolution" pixels
• C99 or newer compiler

5.1.3 Online Tools
LVGL provides online Font Converter and Image Converter.

5.1.4 Demo Examples

Official Demo
LVGL provides a demo project of using LVGL on ESP32 in LVGL project for ESP32.
On top of that, ESP-IoT-Solution also provides some application examples of using LVGL:

Thermostat
A thermostat control interface designed using LVGL:

Please find details of this example in hmi/lvgl_thermostat.
Coffee

An interactive interface of a coffee machine designed using LVGL:

Please find details of this example in hmi/lvgl_coffee.

Wificonfig

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

Please find details of this example in hmi/lvgl_wificonfig.
6.1

Note:

- GPIO
- ADC
6.1.1 

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUTTON_PRESS_DOWN</td>
<td>1</td>
</tr>
<tr>
<td>BUTTON_PRESS_UP</td>
<td>2</td>
</tr>
<tr>
<td>BUTTON_PRESS_REPEAT</td>
<td>&gt;=2</td>
</tr>
<tr>
<td>BUTTON_SINGLE_CLICK</td>
<td>1</td>
</tr>
<tr>
<td>BUTTON_DOUBLE_CLICK</td>
<td>2</td>
</tr>
<tr>
<td>BUTTON_LONG_PRESS_START</td>
<td>1</td>
</tr>
<tr>
<td>BUTTON_LONG_PRESS_HOLD</td>
<td>2</td>
</tr>
</tbody>
</table>

**Attention:** TaskDelay

6.1.2 

```c
// create gpio button
button_config_t gpio_btn_cfg = {
    .type = BUTTON_TYPE_GPIO,
    .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,
    .adc_button_config = {
        .adc_channel = 0,
        .button_index = 0,
        .min = 100,
        .max = 400,
    },
};
button_handle_t adc_btn = iot_button_create(&adc_btn_cfg);
if(NULL == adc_btn) {
    ESP_LOGE(TAG, "Button create failed");
}
```
```c
static void button_single_click_cb(void *arg)
{
    ESP_LOGI(TAG, "BUTTON_SINGLE_CLICK");
}

iot_button_register_cb(gpio_btn, BUTTON_SINGLE_CLICK, button_single_click_cb);
```

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

### 6.1.3 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)`
  
  Register the button event callback function.

  **Return**

  - ESP_OK on success
  - ESP_ERR_INVALID_ARG Arguments is invalid.

  **Parameters**
• \texttt{btn\_handle}: A button handle to register
• \texttt{event}: Button event
• \texttt{cb}: Callback function.

\begin{verbatim}
esp\_err\_t \textit{iot\_button\_unregister\_cb} (\textit{button\_handle\_t btn\_handle}, \textit{button\_event\_t event})
Unregister the button event callback function.
\end{verbatim}

Return
• ESP\_OK on success
• ESP\_ERR\_INVALID\_ARG Arguments is invalid.

Parameters
• \texttt{btn\_handle}: A button handle to unregister
• \texttt{event}: Button event

\begin{verbatim}
button\_event\_t \textit{iot\_button\_get\_event} (\textit{button\_handle\_t btn\_handle})
Get button event.
\end{verbatim}

Return Current button event. See \texttt{button\_event\_t}

Parameters
• \texttt{btn\_handle}: Button handle

\begin{verbatim}
uint8\_t \textit{iot\_button\_get\_repeat} (\textit{button\_handle\_t btn\_handle})
Get button repeat times.
\end{verbatim}

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

Parameters
• \texttt{btn\_handle}: Button handle

Structures

\begin{verbatim}
\textbf{struct button\_config\_t}
Button configuration.
\end{verbatim}

Public Members

\begin{verbatim}
\textbf{button\_type\_t type}
button type. The corresponding button configuration must be filled
\textbf{button\_gpio\_config\_t gpio\_button\_config}
gpio button configuration
\textbf{button\_adc\_config\_t adc\_button\_config}
adc button configuration
\end{verbatim}

union \textbf{button\_config\_t::[anonymous]} [anonymous]
button configuration
Type Definitions

typedef void (*button_cb_t)(void *)
typedef void *button_handle_t

Enumerations

enum button_event_t
Button events.
Values:
    BUTTON_PRESS_DOWN = 0
    BUTTON_PRESS_UP
    BUTTON_PRESS_REPEAT
    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

6.2 Touch Panel

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

<table>
<thead>
<tr>
<th>Resistive Touch Panel</th>
<th>Capacitive Touch Panel</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPT2046</td>
<td>FT5216</td>
</tr>
<tr>
<td>NS2016</td>
<td>FT5436</td>
</tr>
<tr>
<td></td>
<td>FT6336</td>
</tr>
<tr>
<td></td>
<td>FT5316</td>
</tr>
</tbody>
</table>

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

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

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

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

6.2.2 Press Touch Panel

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

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

6.2.3 Touch Panel Rotation

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

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

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

6.2.4 Application Example

Initialize a Touch Panel

touch_panel_driver_t touch; // a touch panel driver

i2c_config_t i2c_conf = {
    .mode = I2C_MODE_MASTER,
    .sda_io_num = 35,
    .sda_pullup_en = GPIO_PULLUP_ENABLE,
    .scl_io_num = 36,
    .scl_pullup_en = GPIO_PULLUP_ENABLE,
    .master.clk_speed = 100000,
};  

i2c_bus_handle_t i2c_bus = i2c_bus_create(I2C_NUM_0, &i2c_conf);

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

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

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

Note:

• When using a capacitive touch panel, the call to the calibration function will return ESP_OK directly.

• By default, only FT5x06 touch panel driver is enabled, please go to menuconfig -> Component config -> Touch Screen Driver -> Choose Touch Screen Driver to do configurations if you need to enable other drivers.

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

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

6.2.5 API Reference

Header File

• display/touch_panel/touch_panel.h
Functions

```c
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

```c
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

```c
struct touch_panel_config_t
```

Configuration of touch panel.

**Public Members**

- `i2c_bus_handle_t i2c_bus`: Handle of i2c bus
- `int clk_freq`: i2c clock frequency
  - spi clock frequency
- `uint8_t i2c_addr`: Screen i2c slave address

```c
struct touch_panel_config_t::[anonymous]::[anonymous] interface_i2c
```

I2c interface
```c
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: Now this line is not used, you can set to -1 and no connection with hardware

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)
   Deinitialize 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.

6.2. Touch Panel
```
Return

• ESP_OK Success
• ESP_FAIL Fail

Parameters

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

```c
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

```c
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

```c
define enum touch_panel_event_t

Touch events.

Values:

TOUCH_EVT_RELEASE = 0x0
Release event

TOUCH_EVT_PRESS = 0x1
Press event
```

```c
define 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**

6.2. Touch Panel
7.1 Sensor Hub

Sensor Hub is a sensor management component that can realize hardware abstraction, device management and data distribution for sensor devices. When developing applications based on Sensor Hub, users do not have to deal with complex sensor implementations, but only need to make simple selections for sensor operation, acquisition interval, range, etc. and then register callback functions to the event messages of your interests. By doing so, users can receive notifications when sensor states are switched or when data is collected.

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

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

2. Register callback functions for sensor events: when a sensor event occurs, the callback functions will be called in sequence. There are two ways to register a callback function, and the instance handler of the event callback function will be returned after the registration succeed:
   - Use `iot_sensor_handler_register()` to register a callback function with the sensor handler
   - Use `iot_sensor_handler_register_with_type()` to register a callback function with the sensor type

3. Start a sensor: use `iot_sensor_start()` to start a specific sensor. After started, it will trigger a `SENSOR_STARTED` event, then it will collect the sensor data continuously with a set of period and trigger `SENSOR_XXXX_DATA_READY` event. The event callback function can obtain the specific data of each event via the `event_data` parameter;

4. Stop a sensor: use `iot_sensor_stop()` to stop a specified sensor temporarily. After stopped, the sensor will send out a `SENSOR_STOPED` event and then stop the data collecting work. If the driver of this sensor supports power management, the sensor will be set to sleep mode in this stage;

5. Unregister callback functions for sensor events: the user program can unregister an event at any time using the instance handler of this event callback function, and this callback function will not be called again when this event occurs afterwards. There are also two ways to do so:
   - Use `iot_sensor_handler_unregister()` to unregister the callback function with the sensor handler
   - Use `iot_sensor_handler_unregister_with_type()` to unregister the callback function with the sensor type
6. Delete sensors: use `iot_sensor_delete()` to delete the corresponding sensor to release the allocated memory and other resources.

### 7.1.2 Examples

1. Sensor control LED example: `sensors/sensor_control_led`.
2. Sensor hub monitor example: `sensors/sensor_hub_monitor`.

### 7.1.3 API Reference

#### Header File

- `sensors/sensor_hub/include/sensor_type.h`

#### Structures

```c
struct sensor_data_t
    sensor data type
```

#### Public Members

```c
int64_t timestamp  // timestamp
```

```c
uint8_t sensor_id  // sensor id
```

```c
int32_t event_id  // reserved for future use
```

```c
uint32_t min_delay  // minimum delay between two events, unit: ms
```

```c
axis3_t acce  // Accelerometer. unit: G
```

```c
axis3_t gyro  // Gyroscope. unit: dps
```

```c
axis3_t mag  // Magnetometer. unit: Gauss
```

```c
float temperature  // Temperature. unit: °Celsius
```

```c
float humidity  // Relative humidity. unit: percentage
```

```c
float baro  // Pressure. unit: pascal (Pa)
```

```c
float light  // Light. unit: lux
```

---

**7.1. Sensor Hub**
Esp-IOT-Solution

rgbw_t rgbw
    Color. unit: lux

uv_t uv
    ultraviolte 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 mode
    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
        wake up 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


```c
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**

```c
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 succeed.

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 declared in sensor_type_t.  
- `event_id`: sensor event declared in sensor_event_id_t and sensor_data_event_id_t  
- `handler`: the handler function which gets called when the event is dispatched  
- `context`: An event handler instance object related to the registered event handler and data, can be NULL. This needs to be kept if the specific callback instance should be unregistered before deleting the whole event loop. Registering the same event handler multiple times is possible and yields distinct instance objects. The data can be the same for all registrations. If no unregistration is needed but the handler should be deleted when the event loop is deleted, instance can be NULL.
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 declared in sensor_type_t.
- event_id: sensor event declared in sensor_event_id_t and sensor_data_event_id_t
- context: the instance object of the registration to be unregistered

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

custom uint8_t *addr
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 detailed 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:

7.2 Humidity and Temperature Sensor

The humidity and temperature sensor can be used as a temperature sensor, a humidity sensor or a sensor with both functions. It is mainly used for environmental temperature and humidity detections in smart home, smart farm and smart factory applications.

7.2.1 Adapted Products

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Bus</th>
<th>Vendor</th>
<th>Datasheet</th>
<th>HAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDC2010</td>
<td>Temperature, Humidity</td>
<td>I2C</td>
<td>TI</td>
<td><a href="#">Datasheet</a></td>
<td></td>
</tr>
<tr>
<td>HTS221</td>
<td>Temperature, Humidity</td>
<td>I2C</td>
<td>ST</td>
<td><a href="#">Datasheet</a></td>
<td>√</td>
</tr>
<tr>
<td>SHT3X</td>
<td>Temperature, Humidity</td>
<td>I2C</td>
<td>Sensirion</td>
<td><a href="#">Datasheet</a></td>
<td>√</td>
</tr>
<tr>
<td>MVH3004D</td>
<td>Temperature, Humidity</td>
<td>I2C</td>
<td>–</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7.2.2 API Reference

The following APIs have implemented hardware abstraction on the humidity and temperature sensor. Users can call the code from this layer directly to write a sensor application, or use the sensor interface in `sensor_hub` for easier development.

Header File

- `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)

```c
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)

```c
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

```c
esp_err_t humiture_wakeup(sensor_humiture_handle_t sensor)
```

Wake up 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

```c
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

7.2. Humidity and Temperature Sensor
ESP-IoT-Solution

- data_group: acquired data

```c
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

```c
typedef void *sensor_humiture_handle_t
humiture sensor handle
```

Enumerations

```c
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
```

7.3 Inertial Measurement Unit (IMU)

The Inertial Measurement Unit (IMU) can be used as a gyroscope sensor, an acceleration sensor, a sensor with multiple functions or etc. It is mainly used to measure the acceleration and angular velocity of an object, and then calculate the motion attitude of the object.
7.3.1 Adapted Products

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Bus</th>
<th>Vendor</th>
<th>Datasheet</th>
<th>HAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIS2DH12</td>
<td>3-axis acceler</td>
<td>I2C</td>
<td>ST</td>
<td>Datasheet</td>
<td>√</td>
</tr>
<tr>
<td>MPU6050</td>
<td>3-axis acceler + 3-axis gyro</td>
<td>I2C</td>
<td>InvenSense</td>
<td>Datasheet</td>
<td>√</td>
</tr>
</tbody>
</table>

7.3.2 API Reference

The following APIs have implemented hardware abstraction on the IMU. Users can call the code from this layer directly to write a sensor application, or use the sensor interface in `sensor_hub` for easier development.

Header File

- 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)
Wake up 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 \textbf{imu\_acquire} (sensor_imu_handle_t sensor, sensor_data_group_t *data_group)
acquire a group of sensor data

\textbf{Return} esp_err_t
• ESP_OK Success
• ESP_FAIL Fail

\textbf{Parameters}
• sensor: imu sensor handle to operate
• data_group: acquired data

esp_err_t \textbf{imu\_control} (sensor_imu_handle_t sensor, sensor_command_t cmd, void *args)
control sensor mode with control commands and args

\textbf{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

\textbf{Type Definitions}

\texttt{typedef void *sensor\_imu\_handle\_t}
imu sensor handle

\textbf{Enumerations}

\texttt{enum imu\_id\_t}
imu sensor id, used for imu_create

\texttt{Values:}
\begin{itemize}
  \item \texttt{MPU6050\_ID} = 0x01
      MPU6050 imu sensor id
  \item \texttt{LIS2DH12\_ID}
      LIS2DH12 imu sensor id
  \item \texttt{IMU\_MAX\_ID}
      max imu sensor id
\end{itemize}
7.4 Ambient Light Sensor

The ambient light sensor can be used as a light intensity sensor, a color sensor, a UV sensor or a sensor with multiple functions.

7.4.1 Adapted Products

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Bus</th>
<th>Vendor</th>
<th>Datasheet</th>
<th>HAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BH1750</td>
<td>Light</td>
<td>I2C</td>
<td>rohm</td>
<td>Datasheet</td>
<td>✓</td>
</tr>
<tr>
<td>VEML6040</td>
<td>Light RGBW</td>
<td>I2C</td>
<td>Vishay</td>
<td>Datasheet</td>
<td>✓</td>
</tr>
<tr>
<td>VEML6075</td>
<td>Light UVA UVB</td>
<td>I2C</td>
<td>Vishay</td>
<td>Datasheet</td>
<td>✓</td>
</tr>
</tbody>
</table>

7.4.2 API Reference

The following APIs have implemented hardware abstraction on the ambient light sensor. Users can call the code from this layer directly to write a sensor application, or use the sensor interface in sensor_hub for easier development.

Header File

- sensors/sensor_hub/include/hal/light_sensor_hal.h

Functions

```c
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
```

```c
esp_err_t light_sensor_delete(sensor_light_handle_t *sensor)
Delete and release the sensor resource.

Return  esp_err_t

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

```c
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)
Wake up 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

7.5 Pressure Sensor

The pressure sensor can be used to detect the absolute pressure of gases, calculate altitude and etc. It is mainly used in environmental monitoring, altitude measurement and space positioning equipments.

7.5.1 Adapted Products

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Bus</th>
<th>Vendor</th>
<th>Datasheet</th>
<th>HAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BME280</td>
<td>Pressure</td>
<td>I2C/SPI</td>
<td>BOSCH</td>
<td>Datesheet</td>
<td></td>
</tr>
</tbody>
</table>

7.6 Gesture Sensor

Gesture sensors are generally sensors that convert measurements of reflected infrared light into relevant physical motions and can be used to achieve non-contact interaction between human and machines, etc.
### 7.6.1 Adapted Products

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Bus</th>
<th>Vendor</th>
<th>Datasheet</th>
<th>HAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>APDS9960</td>
<td>Light, RGB and Gesture Sensor</td>
<td>I2C</td>
<td>Avago</td>
<td>Datasheet</td>
<td></td>
</tr>
</tbody>
</table>
8.1 Storage Media

The supported storage media is listed in the following table:

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

Note:

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

8.1.1 SPI Flash

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

Flash can be partitioned using the partition table. Based on functions of the partition table, flash can not only be used to store the binary code generated by users, but can also act as a non-volatile storage (NVS) to store application programming parameters. On top of that, specific flash areas can be mounted to a file system (e.g., FatFS) to store text, images and other files.
The flash chip supports 2-line (DOUT/DIO) and 4-line (QOUT/QIO) operation modes and can be configured to work in 40 MHz or 80 MHz modes. Since the main flash chip can be used directly for data storage without needs for additional memory chips, it is particularly suitable for cost-sensitive applications with small capacity requirements (MB) and high integration needs.

**Related documents:**
- SPI Flash API

### 8.1.2 SD Card

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

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

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

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

### 8.1.3 eMMC

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

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

**Related documents:**
- SD/SDIO/MMC Driver;
- Supported eMMC Speed Modes.
8.1.4 EEPROM

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

Adapted EEPROM chips:

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Bus</th>
<th>Vendor</th>
<th>Datasheet</th>
<th>Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT24C01/02</td>
<td>1024/2048 bits EEPROM</td>
<td>I2C</td>
<td>Atmel</td>
<td>Datasheet</td>
<td>eeprom</td>
</tr>
</tbody>
</table>

8.2 File System

Supported file systems:

<table>
<thead>
<tr>
<th>Key Features</th>
<th>NVS Library</th>
<th>FAT File System</th>
<th>SPIFFS File System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Features</td>
<td>Operates on key-value pairs, with safe interfaces</td>
<td>Operation system supported, strong compatibility</td>
<td>Developed for embedded systems, low resource occupancy</td>
</tr>
<tr>
<td>Application Scenarios</td>
<td>Stores parameters</td>
<td>Stores audio, video and other files</td>
<td>Stores audio, video and other files</td>
</tr>
<tr>
<td>Size</td>
<td>KB-MB</td>
<td>GB</td>
<td>&lt; 128 MB</td>
</tr>
<tr>
<td>Directory Support</td>
<td>X</td>
<td>√</td>
<td>X</td>
</tr>
<tr>
<td>Wear Leveling</td>
<td>√</td>
<td>Optional</td>
<td>√</td>
</tr>
<tr>
<td>R/W Efficiency</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Resources Occupancy</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Power Failure Protection</td>
<td>√</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Encryption</td>
<td>√</td>
<td>√</td>
<td>X</td>
</tr>
</tbody>
</table>

Note:
- 0: data not available or not for comparison.
- 1: low RAM occupancy.
8.2.1 NVS Library

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

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

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

8.2.2 FAT File System

ESP-IDF uses the FatFs library to work with FAT file system. FatFs is a file system layer independent to platform and storage media that can realize access to physical devices (e.g., flash, SD card) via a unified interface. Although the library can be used directly, many of its features can be accessed via VFS, using the C standard library and POSIX API functions. The operating system of FAT is compatible with a wide range of mobile storage devices such as USB memory disc or SD cards. And ESP32 series chips can access these common storage devices by supporting the FAT file system.

Related documents:
- Using FatFs with VFS.
- Using FatFs with VFS and SD cards.

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

8.2.3 SPIFFS File System

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

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

Related documents:
- SPIFFS Filesystem.
- Two Tools to Generate SPIFFS Images.

Examples:
- storage/spiffs: SPIFFS examples.
8.2.4 Virtual File System (VFS)

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

**Related documents:**

- Virtual Filesystem Component.
9.1 Servo

This component uses the LEDC peripheral to generate PWM signals for independent control of servos with up to 16 channels (ESP32 chips support 16 channels and ESP32-S2 chips support 8 channels) at a selectable frequency of 50 ~ 400 Hz. When using this layer of APIs, users only need to specify the servo group, channel and target angle to realize the angle control of a servo.

Generally, there is a reference signal inside the servo generating a fixed period and pulse width, which is used to compare with the input PWM signal to output a voltage difference so as to control the rotation direction and angle of a motor. A common 180 angular rotation servo usually takes 20 ms (50 Hz) as a clock period and 0.5 ~ 2.5 ms as its high level pulse, making it rotates between 0 ~ 180 degrees.

This component can be used in scenarios with lower control accuracy requirements, such as toy cars, remote control robots, home automation, etc.

9.1.1 Instructions

1. Initialization: Use `servo_init()` to initialize a channel. Please note that ESP32 contains two sets of channels as `LEDC_LOW_SPEED_MODE` and `LEDC_HIGH_SPEED_MODE`, while some chip may only support one channel. The configuration items in this step mainly include maximum angle, signal frequency, and minimum and maximum input pulse width to calculate the correspondence between angle and duty cycle; as well as pins and channels to specify the correspondence with chip pins and LEDC channels, respectively;

2. Set a target angle: use `servo_write_angle()` to specify the servo group, channel and target angle so as to realize angle control of the servo;

3. Read the current angle: you can use `servo_read_angle()` to read the current angle of the servo. Please note that this is a theoretical number calculated based on the input signal;

4. De-initialization: you can use `servo_deinit()` to de-initialize a group of channels when a group of servos is used any more.
9.1.2 Application Example

```c
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;

// 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);
```
9.1.3 API Reference

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_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 lede_channel_t
- angle: The angle to go

esp_err_t iot_servo_read_angle (ledc_mode_t speed_mode, uint8_t_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 TEN

SECURITY & ENCRYPTION

10.1 Flash

10.1.1

- Flash encryption
- 256-bit AES key
- Boot loader
- Boot flash
- OTA

10.1.2

1. `make menuconfig` “Security features”->”Enable flash encryption on boot”
2. Bootloader, partition table
3. Boot flash

10.1.3 boot

1. Bootloader
2. Bootloader
3. Efuse
4. Efuse
5. Efuse
6. Bootloader
10.1.4 **flash**

- **flash**
  1. make menuconfig “Security features”->”Enable flash encryption on boot”
  2. `esp-idf` components/esptool_py/esptool espefuse.py burn\_efuse FLASH\-_CRYPT\_CNT efuse \_FLASH\_CRYPT\_CNT
  3. `bootloader` FLASH\_CRYPT\_CNT `flash`

10.1.5 **FLASH\_CRYPT\_CNT**

- FLASH\_CRYPT\_CNT `flash`
- `bootloader` FLASH\_CRYPT\_CNT
- FLASH\_CRYPT\_CNT `flash`
- FLASH\_CRYPT\_CNT `flash`
- FLASH\_CRYPT\_CNT `flash`
- FLASH\_CRYPT\_CNT `flash`
- FLASH\_CRYPT\_CNT `flash`
- FLASH\_CRYPT\_CNT `flash`
- FLASH\_CRYPT\_CNT `flash`
10.1.6 • Bootloader
• Secure boot bootloader digest
• Secure Boot flash
• Partition table
• Partition table Type “app”
• Partition table Flags “encrypted” NVS

10.1.7 • flash
• API esp_spi_flash_mmap()
• ROM bootloader bootloader image

10.1.8 • API esp_spi_flash_read() ROM
• ROM SPIRead()

10.1.9 • API esp\partition\write() partition
• esp\spi\flash\write\encrypted true
• ROM esp\rom\spiflash\write\encrypted() flash ROM SPIWrite()

10.2 Secure Boot Secure Boot
• Secure Boot ECDSA partition table app images
• Secure Boot bootloader bootloader Secure Boot ROM bootloader bootloader image secure boot key efuse
10.2.2

- ECDSA
  - flash PC
  - ECDSA bootloader image boot loader image app images
  - partition table app images image boot

- secure bootloader key
  - 256-bit AES key Secure Boot efuse
  - key boot loader image

10.2.3

1. bootloader image menuconfig secure boot menuconfig bootloader image secure boot
2. partition table app images
3. bootloader menuconfig JTAG ROM BASIC bootloader efuse
4. boot rom bootloader efuse ABS_DONE_0 secure boot
5. bootloader bootloader image flash partition table app images

10.2.4

1. make menuconfig “enable secure boot in bootloader”
2. make menuconfig
3. make “make” openssl “openssl ecparam -name prime256v1 -genkey -nout -out my_secure_bootSigning_key.pem”
4. “make bootloader” secure boot bootloader image
5. bootloader bootloader image
6. “make flash” partition table app images
7. bootloader bootloader secure boot secure boot
10.2.5 bootloader

- bootloader image
- partition table
- app images
- secure boot
- OTA
- image
- software
- Bootloader

10.2.6 bootloader

- bootloader image
- bootloader image
- secure boot
- image
- secure boot
- key
- PC
- secure boot key
- efuse
- secure digest
- bootloader image
- flash

1. make menuconfig “secure bootloader mode” -> “Reflashable”
2. “make menuconfig bootloader”
3. “make bootloader” 256-bit secure boot key
   - PC secure boot key
   - efuse
   - secure digest bootloader image

10.2.7 Secure Boot Flash Encryption

- boot secure boot
- flash encrypt
- secure boot
- flash encrypt

10.2.8 flash Secure Boot Flash encryption

1. make menuconfig “secure boot flash encrypt”
   “Secure bootloader mode” “Reflashable”
2. bootloader secure boot key
3. key bootloader digest 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. partition_table

make partition_table
make app
ESP-IoT-Solution

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
5.  

```bash
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.bin
python $IDF_PATH/components/esptool_py/esptool/espsecure.py encrypt_flash_data --
  keyfile flash_encrypt_key.bin --address 0x8000 -o build/partitions_singleapp_
  digest_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.  

```bash
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.  

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

8.  

```bash
python $IDF_PATH/components/esptool_py/esptool/espefuse.py burn_key secure_boot ./
  build/bootloader/secure_boot_key.bin
```

9.  

```bash
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
```

10.3  

10.3.1 Windows  

- configure security.conf
- configure configure_security.conf
### 10.3.2 ESPTool

- **ESPTool**
  - `esptool`
    - `esptool` command
    - `esptool` command
  - `pip install esptool`
  - `pip3 install esptool`

### 10.3.3 Bootloader Security

- **Bootloader Security**
  - `espsecure.py` generate signing key `secure_boot_signing_key.pem` or
  - `openssl ecparam -name prime256v1 -genkey -noout -out secure_boot_signing_key.pem`
122 Chapter 10. Security & Encryption

![Flowchart Image]
2. **menuconfig** Sign binaries during build.

3. **make bootloader**
   
   ```
   make bootloader
   make
   ```

4. **esptool**
   
   ```
   esptool.bin --chip esp32 /dev/cu.SLAB_USBtoUART --port $PORT --baud 1152000 --before default_reset --after no_reset --reset write_flash --flash_mode dio --flash_freq 40m --flash_size detect
   --0x1000 $IDF_PATH/esp-idf/examples/get-started/hello_world/build/bootloader/\n   bootloader.bin 0xf000 $IDF_PATH/esp-idf/examples/get-started/hello_world/\n   build/phy_init_data.bin 0x10000 $IDF_PATH/examples/get-started/hello_world/\n   build/hello-world.bin 0x8000 $IDF_PATH/examples/get-started/hello_world/\n   build/partitions_singleapp.bin
   ```

**Note:**

5. **Window**

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

**Note:**
6. combine bin flash -
   - bin flash
   - 'DoNotChgBin' bin (SPI)
   - 'CombineBin' bin
   - ‘combine’ target.bin Flash 0x0
   - 0xff

7. bootloader security AES EFUSE

Note:

- app.bin
- OTA app
- bootloader flash partition
- bootloader flash
  - secure boot images flash boot
  - secure boot flash boot
2: ESP-IoT-Solution security

- image flash efuse, secure boot flash
- secure boot flash encryption make menuconfig enable secure boot in bootloader
- secure boot flash encryption EFUSE

1. RSA

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

2. menuconfig Sign binaries during build

3. bootloader

   ```
   make bootloader
   make
   ```

4. [DEBUG MODE]

   ```
   [DEBUG MODE]
   debug_enable * False
   debug_pem_path *
   [SECURE BOOT]
   secure_boot_en * True
   burn_secure_boot_key * True
   secure_boot_force_write * False
   ```

(continues on next page)
secure_boot_rw_protect * True  # secure boot key

[FLASH ENCRYPTION]
flash_encryption_en * True  # flash
burn_flash_encryption_key * True  # flash encrypt key
flash_encrypt_force_write * False  # flash encrypt key block
flash_encrypt_rw_protect * True  # flash encrypt key

[AES KEY]
aes_key_en * False  # aes
burn_aes_key * False  # aes

[DISABLE FUNC]
jtag_disable * True  # JTAG
dl_encrypt_disable * True  # flash
dl_decrypt_disable * True  # flash
dl_cache_disable * True  # flash cache

5. flash 'DoNotChgBin'

- app.bin
- OTA app
- app image menuconfig secure boot efuse secure boot app image

10.3. 127
11.1 GPIO Expander

With further expansions of the ESP32 chip family, more application scenarios with diverse demands are being introduced, including some that have more requirements on GPIO numbers. The subsequent release of ESP32-S2 and other products have included up to 43 GPIOs, which can greatly alleviate the problem of GPIO resource constraint. If this still could not meet your demand, you can also add GPIO expansion chips to ESP32 to have more GPIO resources, such as using the I2C-based GPIO expansion module MCP23017, which can expand 16 GPIO ports per module and mount up to 8 expansion modules simultaneously thus expanding additional 128 GPIO ports totally.

11.1.1 Adapted Products

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Bus</th>
<th>Vendor</th>
<th>Datasheet</th>
<th>Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCP23017</td>
<td>16-bit I/O expander</td>
<td>I2C</td>
<td>Microchip</td>
<td>Datasheet</td>
<td>mcp23017</td>
</tr>
</tbody>
</table>
CONTRIBUTIONS GUIDE

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

12.1 How to Contribute

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

12.2 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.

12.3 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.
12.4 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.

12.5 Related Documents

12.5.1 esp-iot-solution

- components
- docs
- examples
- tools
- .c
- .h
- .hpp

#include _IOT_I2C_BUS_H_
#define _IOT_I2C_BUS_H_
@endef

- extern “C” C/C++
### ESP-IoT-Solution

```c
#ifndef __cplusplus
extern "C"
{
#endif
//c code
#ifndef __cplusplus
}
#endif
```

---

- VSCODE
- Doxygen Documentation Generator

```c
/**
 * @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.
 */

typedef i2c_bus_handle_t iot_i2c_bus_create(i2c_port_t port, i2c_config_t* conf);
```

---

```
/**
 * @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.
 */

typedef i2c_bus_handle_t iot_i2c_bus_create(i2c_port_t port, i2c_config_t* conf);
```

---

```
// 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
```

(continues on next page)
// 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.

- static
- static
- static
- static
- static
- static
- static
- static
- static
- static

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>xxx</td>
<td>xxx</td>
</tr>
<tr>
<td>iot_type_xxx</td>
<td>iot_sensor_xxx;</td>
<td>iot_board_xxx;</td>
</tr>
<tr>
<td></td>
<td>iot_storage_...</td>
<td></td>
</tr>
<tr>
<td>type_xxx</td>
<td>imu_xxx; light_xxx; eeprom_xxx</td>
<td></td>
</tr>
<tr>
<td>name_xxx</td>
<td>mpu6050_xxx;</td>
<td></td>
</tr>
<tr>
<td>xxx_creat / xxx_delete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xxx_read / xxx_write</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- static
- static
- static
- static
- static
- static
- NULL;
- static
- static
- data
- data
- data
```c
typedef int signed_32_bit_t;

typedef enum {
    MODULE_FOO_ONE,
    MODULE_FOO_TWO,
    MODULE_FOO_THREE
} module_foo_t;

void function1()
{
    do_one_thing();
    do_another_thing();
    // INCORRECT, don't place empty line here
    // place empty line here
}
```

12.5. Related Documents
void function2()
{
    int var = 0;
    while (var < SOME_CONSTANT) {
        do_stuff(&var);
    }
}

3. /**/}

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;
++y_cur;
const int y = y0+(x-x0)*(y1-y0)/(x1-x0); // INCORRECT

• GPIO
• PIN CAM_VSYNC
4. // This is correct:

```c
void function(int arg) {
}
```

// NOT like this:
```c
void function(int arg) {
}
```

5. // This is correct: // not like this:

```c
if (condition) {
    do_one();
} else if (other_condition) {
    do_two();
}
```

• /* * * /

```c
void init_something() {

    setup_dma();
    // load_resources(); // WHY is this thing commented, asks the reader?
    start_timer();
}
```

• /* * * /

```c
void init_something() {

    setup_dma();

    // load_resources(); // TODO: we should load resources here, but loader is not fully integrated yet.
    start_timer();
}
```

• #if 0 ... #endif

```c
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

```c
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

```c
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

```c
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

```c
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

```c
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

```c
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

```c
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

```c
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

```c
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

```c
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

```c
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

```c
void init_something() {

    setup_dma();

    // TODO: we should load resources here, but loader is not fully integrated yet.
    // load_resources();
    start_timer();
}
```
setup_dma();
// XXX add 2016-09-01
init_dma_list();
fill_dma_item(0);
// end XXX add
start_timer();
}

6. Commit

Windows

commit LF

Github

commit LF

MSYS2

Unix

ESP-IDF

LF

Unix

commit LF

MSYS2

Unix

Unix IDF

checkout

git rebase --exec 'git diff-tree --no-commit-id --name-only -r HEAD | xargs dos2unix &
& git commit --amend --no-edit --allow-empty' master

dos2unix FILENAME

git commit --amend

7. Astyle

astyle

tools/format.sh components/my_component/file.c
CMake

- 4
- 120
- endforeach() endif()
- (with_underscores)
- (with_underscores)
- (WITH_UNDERSCORES)
- cmake-lint
INDEX

B
BH1750_ID (C++ enumerator), 101
BLINK_CONNECTED (C++ enumerator), 54
BLINK_CONNECTED (C++ enumerator), 54
BLINK_FACTORY_RESET (C++ enumerator), 54
BLINK_MAX (C++ enumerator), 54
BLINK_PROVISIONED (C++ enumerator), 54
BLINK_PROVISIONING (C++ enumerator), 54
BLINK_RECONNECTING (C++ enumerator), 54
blink_step_state_t (C++ enum), 54
blink_step_t (C++ class), 53
blink_step_t::hold_time_ms (C++ member), 53
blink_step_t::on_off (C++ member), 53
blink_step_t::type (C++ member), 53
blink_step_type_t (C++ enum), 54
bus_handle_t (C++ type), 85
button_cb_t (C++ type), 73
button_config_t (C++ class), 72
button_config_t::adc_button_config (C++ member), 72
button_config_t::gpio_button_config (C++ member), 72
button_config_t::type (C++ member), 72
button_config_t::[anonymous] (C++ member), 72
BUTTON_DOUBLE_CLICK (C++ enumerator), 73
BUTTON_EVENT_MAX (C++ enumerator), 73
button_event_t (C++ enum), 73
button_handle_t (C++ type), 73
BUTTON_LONG_PRESS_HOLD (C++ enumerator), 73
BUTTON_LONG_PRESS_START (C++ enumerator), 73
BUTTON_NONE_PRESS (C++ enumerator), 73
BUTTON_PRESS_DOWN (C++ enumerator), 73
BUTTON_PRESS_REPEAT (C++ enumerator), 73
BUTTON_PRESS_UP (C++ enumerator), 73
BUTTON_SINGLE_CLICK (C++ enumerator), 73
BUTTON_TYPE_ADC (C++ enumerator), 73
BUTTON_TYPE_GPIO (C++ enumerator), 73
button_type_t (C++ enum), 73

C
COLOR_BLACK (C macro), 42
COLOR_BLUE (C macro), 42
COLOR_CYAN (C macro), 42
COLOR DARKCYAN (C macro), 42
COLOR DARKGREEN (C macro), 42
COLOR DARKGREY (C macro), 42
COLOR ESP_BKGD (C macro), 42
COLOR_FUCHSIA (C macro), 42
COLOR_GRAY (C macro), 42
COLOR_GREEN (C macro), 42
COLOR_GRENNEYELLOW (C macro), 42
COLOR_LIGHTGREY (C macro), 42
COLOR_LIME (C macro), 42
COLOR_MAGENTA (C macro), 42
COLOR_MAROON (C macro), 42
COLOR NAVY (C macro), 42
COLOR OLIVE (C macro), 42
COLOR ORANGE (C macro), 42
COLOR PINK (C macro), 42
COLOR PURPLE (C macro), 42
COLOR_RED (C macro), 42
COLOR SILVER (C macro), 42
COLOR_TEAL (C macro), 42
COLOR WHITE (C macro), 42
COLOR YELLOW (C macro), 42
COMMAND_MAX (C++ enumerator), 85
COMMAND SELF_TEST (C++ enumerator), 85
COMMAND_SET_MODE (C++ enumerator), 85
COMMAND_SET_ODR (C++ enumerator), 85
COMMAND_SET_POWER (C++ enumerator), 85
COMMAND_SET_RANGE (C++ enumerator), 85

D
dac_audio_config_t (C++ class), 63
dac_audio_config_t::bits_per_sample (C++ member), 63
dac_audio_config_t::dac_mode (C++ member), 63
dac_audio_config_t::dma_buf_count (C++ member), 63

141
i2c_bus_write_bits (C++ function), 17
i2c_bus_write_bit (C++ function), 17
i2c_bus_write_bytes (C++ function), 16
i2c_bus_write_reg16 (C++ function), 18
i2s_lcd_acquire (C++ function), 25
i2s_lcd_config_t (C++ class), 25
i2s_lcd_config_t::buffer_size (C++ member), 25
i2s_lcd_config_t::clk_freq (C++ member), 25
i2s_lcd_config_t::data_width (C++ member), 25
i2s_lcd_config_t::i2s_port (C++ member), 25
i2s_lcd_config_t::pin_data_num (C++ member), 25
i2s_lcd_config_t::pin_num_cs (C++ member), 25
i2s_lcd_config_t::pin_num_rs (C++ member), 25
i2s_lcd_config_t::pin_num_wr (C++ member), 25
i2s_lcd_config_t::swap_data (C++ member), 25
i2s_lcd_driver_deinit (C++ function), 23
i2s_lcd_driver_init (C++ function), 23
i2s_lcd_handle_t (C++ type), 26
i2s_lcd_release (C++ function), 25
i2s_lcd_write (C++ function), 24
i2s_lcd_write_cmd (C++ function), 24
i2s_lcd_write_command (C++ function), 24
i2s_lcd_write_data (C++ function), 24
imu_acquire (C++ function), 96
imu_acquire_acc (C++ function), 95
imu_acquire_gyro (C++ function), 96
imu_control (C++ function), 97
imu_create (C++ function), 95
imu_delete (C++ function), 95
IMU_ID (C++ enumerator), 85
imu_id_t (C++ enum), 97
IMU_MAX_ID (C++ enumerator), 97
imu_sleep (C++ function), 96
imu_test (C++ function), 95
imu_wakeup (C++ function), 96
iot_button_create (C++ function), 71
iot_button_delete (C++ function), 71
iot_button_get_event (C++ function), 72
iot_button_get_repeat (C++ function), 72
iot_button_register_cb (C++ function), 71
iot_button_unregister_cb (C++ function), 72
iot_sensor_create (C++ function), 87
iot_sensor_delete (C++ function), 88
iot_sensor_handler_register (C++ function), 88
iot_sensor_handler_register_with_type (C++ function), 89
iot_sensor_handler_unregister (C++ function), 89
iot_sensor_handler_unregister_with_type (C++ function), 89
iot_sensor_scan (C++ function), 88
iot_sensor_start (C++ function), 88
iot_sensor_stop (C++ function), 88
iot_servo_deinit (C++ function), 111
iot_servo_init (C++ function), 111
iot_servo_read_angle (C++ function), 111
iot_servo_write_angle (C++ function), 111

L

LCD_CMD_LEV (C macro), 26
LCD_DATA_LEV (C macro), 26
LED_BLINK_HOLD (C++ enumerator), 54
LED_BLINK_LOOP (C++ enumerator), 54
LED_BLINK_STOP (C++ enumerator), 54
LED_GPIO_MODE (C++ enumerator), 54
led_indicator_blank_type_t (C++ enum), 54
led_indicator_config_t (C++ class), 53
led_indicator_config_t::mode (C++ member), 53
led_indicator_config_t::off_level (C++ member), 53
led_indicator_create (C++ function), 52
led_indicator_delete (C++ function), 52
led_indicator_get_handle (C++ function), 52
led_indicator_handle_t (C++ type), 53
led_indicator_mode_t (C++ enum), 54
led_indicator_start (C++ function), 52
led_indicator_stop (C++ function), 52
LED_STATE_OFF (C++ enumerator), 54
LED_STATE_ON (C++ enumerator), 54
LIGHT_MAX_ID (C++ enumerator), 101
light_sensor_acquire (C++ function), 100
light_sensor_acquire_light (C++ function), 99
light_sensor_acquire_rgbw (C++ function), 99
light_sensor_acquire_uv (C++ function), 99
light_sensor_control (C++ function), 100
light_sensor_create (C++ function), 98
light_sensor_delete (C++ function), 98
LIGHTSENSOR_ID (C++ enumerator), 85
light_sensor_id_t (C++ enum), 101
light_sensor_sleep (C++ function), 99
light_sensor_test (C++ function), 98
light_sensor_wakeup (C++ function), 100
LIS2DH12_ID (C++ enumerator), 97

M

MODE_DEFAULT (C++ enumerator), 86
MODE_INTERRUPT (C++ enumerator), 86
MODE_MAX (C++ enumerator), 86
MODE_POLLING (C++ enumerator), 86
MPU6050_ID (C++ enumerator), 97

N

NULL_I2C_DEV_ADDR (C macro), 19
NULL_I2C_MEM_ADDR (C macro), 19
NULL_ID (C++ enumerator), 85
NULL_SPI_CS_PIN (C macro), 23

P

POWER_MAX (C++ enumerator), 86
POWER_MODE_SLEEP (C++ enumerator), 85
POWER_MODE_WAKEUP (C++ enumerator), 85
PWM_AUDIO_CH_MAX (C++ enumerator), 60
PWM_AUDIO_CH_MONO (C++ enumerator), 60
PWM_AUDIO_CH_STEREO (C++ enumerator), 60
pwm_audio_channel_t (C++ enum), 60
pwm_audio_config_t (C++ class), 59
pwm_audio_config_t::duty_resolution
(C++ member), 60
pwm_audio_config_t::gpio_num_left
(C++ member), 60
pwm_audio_config_t::gpio_num_right
(C++ member), 60
pwm_audio_config_t::ledc_channel_left
(C++ member), 60
pwm_audio_config_t::ledc_channel_right
(C++ member), 60
pwm_audio_config_t::ledc_timer_sel
(C++ member), 60
pwm_audio_config_t::ringbuff_len
(C++ member), 60
pwm_audio_config_t::tg_num
(C++ member), 60
pwm_audio_config_t::timer_num
(C++ member), 60
pwm_audio_deinit (C++ function), 57
pwm_audio_get_param (C++ function), 59
pwm_audio_get_status (C++ function), 59
pwm_audio_get_volume (C++ function), 59
pwm_audio_init (C++ function), 57
pwm_audio_set_param (C++ function), 58
pwm_audio_set_sample_rate (C++ function), 58
pwm_audio_set_volume (C++ function), 58
pwm_audio_start (C++ function), 57
PWM_AUDIO_STATUS_BUSY (C++ enumerator), 60
PWM_AUDIO_STATUS_IDLE (C++ enumerator), 60
pwm_audio_status_t (C++ enum), 88
PULSAR_AUDIO_STATUS_UN_INIT (C++ enumerator), 88
pwm_audio_stop (C++ function), 57
pwm_audio_write (C++ function), 57
R
RANGE_DEFAULT (C++ enumerator), 86
RANGE_MAX (C++ enumerator), 86
RANGE_MEDIUM (C++ enumerator), 86
RANGE_MIN (C++ enumerator), 86

S
SCR_COLOR_TYPE_GRAY (C++ enumerator), 43
SCR_COLOR_TYPE_MONO (C++ enumerator), 43
SCR_COLOR_TYPE_RGB565 (C++ enumerator), 43
scr_color_type_t (C++ enum), 43
scr_controller_config_t (C++ class), 39
scr_controller_config_t::bck1_active_level
(C++ member), 39
scr_controller_config_t::height
(C++ member), 39
scr_controller_config_t::interface_drv
(C++ member), 39
scr_controller_config_t::offset_hor
(C++ member), 39
scr_controller_config_t::offset_ver
(C++ member), 39
scr_controller_config_t::pin_num_bck1
(C++ member), 39
scr_controller_config_t::pin_num_rst
(C++ member), 39
scr_controller_config_t::rotate
(C++ member), 39
scr_controller_config_t::rst_active_level
(C++ member), 39
scr_controller_config_t::width
(C++ member), 39
scr_controller_t (C++ enum), 43
SCR_DIR_BTLR (C++ enumerator), 43
SCR_DIR_BTRL (C++ enumerator), 43
SCR_DIR_LRBT (C++ enumerator), 43
SCR_DIR_LRBL (C++ enumerator), 43
SCR_DIR_MAX (C++ enumerator), 43
SCR_DIR_RLB (C++ enumerator), 43
SCR_DIR_RLBT (C++ enumerator), 43
scr_dir_t (C++ enum), 43
SCR_DIR_TBLR (C++ enumerator), 43
SCR_DIR_TBRL (C++ enumerator), 43
scr_driver_t (C++ class), 39
scr_driver_t::deinit
(C++ member), 40
scr_driver_t::draw_bitmap
(C++ member), 41
scr_driver_t::draw_pixel
(C++ member), 41
scr_driver_t::get_info
(C++ member), 41
scr_driver_t::init
(C++ member), 40
scr_driver_t::set_direction
(C++ member), 40
scr_driver_t::set_window
(C++ member), 40
scr_driver_t::write_ram_data
(C++ member), 41
scr_find_driver (C++ function), 38
scr_info_t (C++ class), 39
scr_info_t::bpp
(C++ member), 39
scr_info_t::color_type
(C++ member), 39
scr_info_t::dir
(C++ member), 39
scr_info_t::height
(C++ member), 39
scr_info_t::name
(C++ member), 39
scr_info_t::width
(C++ member), 39
scr_interface_create
(C++ function), 44
scr_interface_delete
(C++ function), 44
scr_interface_driver_t (C++ class), 45
scr_interface_driver_t::bus_acquire
(C++ member), 45
scr_interface_driver_t::bus_release
(C++ member), 46
scr_interface_driver_t::read
(C++ member), 45
scr_interface_driver_t::type
(C++ member), 45
scr_interface_driver_t::write
(C++ member), 45
scr_interface_driver_t::write_cmd
(C++ member), 45
scr_interface_driver_t::write_command
(C++ member), 45
scr_interface_driver_t::write_data
(C++ member), 45
scr_interface_i2c_config_t (C++ class), 45
scr_interface_i2c_config_t::clk_speed
(C++ member), 45
scr_interface_i2c_config_t::i2c_bus
(C++ member), 45
scr_interface_i2c_config_t::slave_addr
(C++ member), 45
scr_interface_spi_config_t (C++ class), 45
scr_interface_spi_config_t::clk_freq
(C++ member), 45
scr_interface_spi_config_t::pin_num_cs
(C++ member), 45
scr_interface_spi_config_t::pin_num_dc
(C++ member), 45
scr_interface_spi_config_t::spi_bus
(C++ member), 45
scr_interface_spi_config_t::swap_data
(C++ member), 45
scr_interface_type_t (C++ enum), 46
SCR_MIRROR_X (C++ enumerator), 43
SCR_MIRROR_Y (C++ enumerator), 43
SCR_SWAP_XY (C++ enumerator), 43
SCREEN_CONTROLLER_ILI9341 (C++ enumerator), 43
SCREEN_CONTROLLER_ILI9486 (C++ enumerator), 43
SCREEN_CONTROLLER_ILI9488 (C++ enumerator), 43
Index 145

SCREEN_CONTROLLER_ILI9806 (C++ enumerator), 43
SCREEN_CONTROLLER_NT35510 (C++ enumerator), 44
SCREEN_CONTROLLER_RM68120 (C++ enumerator), 44
SCREEN_CONTROLLER_SSD1306 (C++ enumerator), 44
SCREEN_CONTROLLER_SSD1307 (C++ enumerator), 44
SCREEN_CONTROLLER_SSD1322 (C++ enumerator), 44
SCREEN_CONTROLLER_SSD1351 (C++ enumerator), 44
SCREEN_CONTROLLER_SSD1963 (C++ enumerator), 44
SCREEN_CONTROLLER_ST7789 (C++ enumerator), 44
SCREEN_CONTROLLER_ST7796 (C++ enumerator), 44
SCREEN_IFACE_8080 (C++ enumerator), 46
SCREEN_IFACE_I2C (C++ enumerator), 46
SCREEN_IFACE_SPI (C++ enumerator), 46
SENSOR_ACCE_DATA_READY (C++ enumerator), 86
SENSOR_BARO_DATA_READY (C++ enumerator), 87
sensor_command_t (C++ enum), 85
sensor_config_t (C++ class), 90
sensor_config_t::bus (C++ member), 90
sensor_config_t::intr_pin (C++ member), 90
sensor_config_t::intr_type (C++ member), 90
sensor_config_t::min_delay (C++ member), 90
sensor_config_t::mode (C++ member), 90
sensor_config_t::range (C++ member), 90
SENSOR_CURRENT_DATA_READY (C++ enumerator), 87
sensor_data_event_id_t (C++ enum), 86
sensor_data_group_t (C++ class), 84
sensor_data_group_t::number (C++ member), 84
sensor_data_group_t::sensor_data (C++ member), 84
sensor_data_t (C++ class), 83
sensor_data_t::acct (C++ member), 83
sensor_data_t::baro (C++ member), 83
sensor_data_t::current (C++ member), 84
sensor_data_t::data (C++ member), 84
sensor_data_t::event_id (C++ member), 84
sensor_data_t::force (C++ member), 84
sensor_data_t::gyro (C++ member), 83
sensor_data_t::hr (C++ member), 84
sensor_data_t::humidity (C++ member), 83
sensor_data_t::light (C++ member), 83
sensor_data_t::mag (C++ member), 83
sensor_data_t::min_delay (C++ member), 83
sensor_data_t::noise (C++ member), 84
sensor_data_t::proximity (C++ member), 84
sensor_data_t::rgbw (C++ member), 83
sensor_data_t::sensor_id (C++ member), 83
sensor_data_t::step (C++ member), 84
sensor_data_t::temperature (C++ member), 83
sensor_data_t::timestamp (C++ member), 83
sensor_data_t::tvc (C++ member), 84
sensor_data_t::uv (C++ member), 84
sensor_data_t::voltage (C++ member), 84
sensor_driver_handle_t (C++ type), 85
SENSOR_EVENT_ANY_ID (C macro), 84
sensor_event_base_t (C++ type), 91
SENSOR_EVENT_COMMON_END (C++ enumerator), 86
sensor_event_handler_instance_t (C++ type), 91
sensor_event_handler_t (C++ type), 91
SENSOR_EVENT_ID_END (C++ enumerator), 87
sensor_event_id_t (C++ enum), 86
SENSOR_FORCE_DATA_READY (C++ enumerator), 87
SENSOR_GYRO_DATA_READY (C++ enumerator), 86
sensor_handle_t (C++ type), 91
SENSOR_HR_DATA_READY (C++ enumerator), 87
SENSOR_HUMI_DATA_READY (C++ enumerator), 87
sensor_humiture_handle_t (C++ type), 94
sensor_id_t (C++ enum), 91
sensor_imu_handle_t (C++ type), 97
sensor_info_t (C++ class), 90
sensor_info_t::addr (C++ member), 90
sensor_info_t::desc (C++ member), 90
sensor_info_t::name (C++ member), 90
sensor_info_t::sensor_id (C++ member), 90
SENSOR_LIGHT_DATA_READY (C++ enumerator), 87
sensor_light_handle_t (C++ type), 101
SENSOR_MAG_DATA_READY (C++ enumerator), 86
sensor_mode_t (C++ enum), 86
SENSOR_NOISE_DATA_READY (C++ enumerator), 87
sensor_power_mode_t (C++ enum), 85
SENSOR_PROXI_DATA_READY (C++ enumerator), 87
sensor_range_t (C++ enum), 86
SENSOR_RGBW_DATA_READY (C++ enumerator), 87
SENSOR_STARTED (C++ enumerator), 86
SENSOR_STEP_DATA_READY (C++ enumerator), 87
SENSOR_STOPED (C++ enumerator), 86
SENSOR_TEMP_DATA_READY (C++ enumerator), 86
SENSOR_TVOC_DATA_READY (C++ enumerator), 87
SENSOR_TYPE_MAX (C++ enumerator), 85
sensor_type_t (C++ enum), 85
SENSOR_UV_DATA_READY (C++ enumerator), 87
SENSOR_VOLTAGE_DATA_READY (C++ enumerator), 87
servo_channel_t (C++ class), 112
servo_channel_t::ch (C++ member), 112
servo_channel_t::servo_pin (C++ member), 112
servo_config_t (C++ class), 112
servo_config_t::channel_number (C++ member), 112
servo_config_t::channels (C++ member), 112
servo_config_t::freq (C++ member), 112
servo_config_t::max_angle (C++ member), 112
servo_config_t::max_width_us (C++ member), 112
servo_config_t::min_width_us (C++ member), 112
servo_config_t::timer_number (C++ member), 112
SHT3X_ID (C++ enumerator), 94
spi_bus_create (C++ function), 20
spi_bus_delete (C++ function), 20
spi_bus_device_create (C++ function), 20
spi_bus_device_delete (C++ function), 20
spi_bus_device_handle_t (C++ type), 23
spi_bus_handle_t (C++ type), 23
spi_bus_transfer_byte (C++ function), 20
spi_bus_transfer_bytes (C++ function), 21
spi_bus_transfer_reg16 (C++ function), 21
spi_bus_transfer_reg32 (C++ function), 22
spi_bus_transmit_begin (C++ function), 21
spi_config_t (C++ class), 22
spi_config_t::max_transfer_sz (C++ member), 22
spi_config_t::miso_io_num (C++ member), 22
spi_config_t::mosi_io_num (C++ member), 22
spi_device_config_t (C++ member), 22
spi_device_config_t::clock_speed_hz (C++ member), 23
spi_device_config_t::cs_io_num (C++ member), 23
spi_device_config_t::mode (C++ member), 23

T
TOUCH_DIR_BTLR (C++ enumerator), 79
TOUCH_DIR_BTRL (C++ enumerator), 79
TOUCH_DIR_LBRT (C++ enumerator), 79
TOUCH_DIR_LRTL (C++ enumerator), 79
TOUCH_DIR_MAX (C++ enumerator), 79
TOUCH_DIR_RLBT (C++ enumerator), 79
TOUCH_DIR_RLTR (C++ enumerator), 79
TOUCH_DIR_TBLR (C++ enumerator), 79
TOUCH_DIR_TBLR (C++ enumerator), 79
TOUCH_EVT_PRESS (C++ enumerator), 78
TOUCH_EVT_RELEASE (C++ enumerator), 78
TOUCH_MAX_POINT_NUMBER (C macro), 78
TOUCH_MIRROR_X (C++ enumerator), 79
TOUCH_MIRROR_Y (C++ enumerator), 79
touch_panel_config_t (C++ class), 76
touch_panel_config_t::clk_freq (C++ member), 76
touch_panel_config_t::direction (C++ member), 77
touch_panel_config_t::height (C++ member), 77
touch_panel_config_t::i2c_addr (C++ member), 76
touch_panel_config_t::i2c_bus (C++ member), 76
touch_panel_config_t::interface_i2c (C++ member), 76
touch_panel_config_t::interface_spi (C++ member), 77
touch_panel_config_t::interface_type (C++ member), 77
touch_panel_config_t::pin_num_cs (C++ member), 77
touch_panel_config_t::pin_num_int (C++ member), 77
touch_panel_config_t::spi_bus (C++ member), 76
touch_panel_config_t::width (C++ member), 77
touch_panel_config_t::[anonymous] (C++ member), 77
TOUCH_PANEL_CONTROLLER_FT5X06 (C++ enumerator), 79
TOUCH_PANEL_CONTROLLER_NS2016 (C++ enumerator), 79
touch_panel_controller_t (C++ enum), 79
TOUCH_PANEL_CONTROLLER_XPT2046 (C++ enumerator), 79
touch_panel_dir_t (C++ enum), 78
touch_panel_driver_t (C++ class), 77
touch_panel_driver_t::calibration_run (C++ member), 77
touch_panel_driver_t::deinit (C++ member), 77
touch_panel_driver_t::init (C++ member), 77
touch_panel_driver_t::read_point_data (C++ member), 78
touch_panel_driver_t::set_direction (C++ member), 78
touch_panel_event_t (C++ enum), 78
touch_panel_find_driver (C++ function), 76
TOUCH_PANEL_IFACE_I2C (C++ enumerator), 79
TOUCH_PANEL_IFACE_SPI (C++ enumerator), 79

touch_panel_interface_type_t (C++ enum), 79

touch_panel_points_t (C++ class), 76

touch_panel_points_t::curx (C++ member), 76

touch_panel_points_t::cury (C++ member), 76

touch_panel_points_t::event (C++ member), 76

touch_panel_points_t::point_num (C++ member), 76

TOUCH_SWAP_XY (C++ enumerator), 79

V

VEML6040_ID (C++ enumerator), 101

VEML6075_ID (C++ enumerator), 101