# ESP8266
# Technical Reference

Version 1.4

Copyright © 2019

# About This Guide

This document provides introduction to the interfaces integrated on ESP8266. Functional overview, parameter configuration, function description, application demos and other information is included.

The document is structured as below.

| Chapter | Title | Subject |
|---|---|---|
| Chapter 1 | Overview | Overall introduction to the interfaces. |
| Chapter 2 | GPIO | Description of GPIO functions, registers and parameter configuration. |
| Chapter 3 | SPI Compatibility Mode User Guide | Description of functions, DEMO solution, ESP8266 software instruction and STM32 software solution. |
| Chapter 4 | SPI Communication User Guide | Description of SPI functions, master/slave protocol format and API functions. |
| Chapter 5 | SPI Overlap & Display Application Guide | Description of SPI functions, hardware connection of SPI overlap mode, API description and display screen console program demo. |
| Chapter 6 | SPI Wi-Fi Passthrough 1-Interrupt Mode | Description of SPI functions, SPI slave protocol format, slave status and line breakage and API functions. |
| Chapter 7 | SPI Wi-Fi Passthrough 2-Interrupt Mode | Description of SPI functions, SPI slave protocol format, data flow control line and API functions. |
| Chapter 8 | HSPI Host Multi-device API | Description of HSPI functions, hardware connection and API functions. |
| Chapter 9 | I2C User Guide | Description of I2C functions, master interface and demo. |
| Chapter 10 | I2S Module Description | Description of I2S functions, system configuration and API functions. |
| Chapter 11 | UART Introduction | Description of UART functions, hardware resources, parameter configuration, interrupt configuration, example of interrupt handler process and abandon serial output during booting. |
| Chapter 12 | PWM Interface | Description of PWM functions PWM, detailed on ***pwm.h***, and custom channels. |
| Chapter 13 | IR Remote Control User Guide | Introduction on infrared transmission, parameter configuration and functions of sample codes. |
| Chapter 14 | Sniffer Introduction | Introduction on Sniffer, application scenarios, phone App and IOT-device firmware. |
| Appendix | Appendix | GPIO registers, SPI registers, UART registers, Timer registers. |

## Release Notes

| Date | Version | Release notes |
|---|---|---|
| 2016.05 | V1.0 | First release. |

| Date | Version | Release notes |
| --- | --- | --- |
| 2016.06 | V1.1 | Added ***Section 4.5 Interface Description***. |
| 2016.08 | V1.2 | Updated ***Section 14.1 Sniffer Introduction***. |
| 2017.05 | V1.3 | Updated ***Section 4.1.2 SPI Features***. |
| 2019.08 | V1.4 | Updated ***Section 1.1 General Purpose Input/Output Interface (GPIO).*** |

# Table of Contents


1. Overview ..... 1
   1.1. General Purpose Input/Output Interface (GPIO) ..... 1
   1.2. Secure Digital Input/Output Interface (SDIO) ..... 1
   1.3. Serial Peripheral Interface (SPI/HSPI) ..... 1
      1.3.1. General SPI (Master/Slave) ..... 2
      1.3.2. HSPI (Slave) ..... 2
   1.4. I2C Interface ..... 2
   1.5. I2S Interface ..... 3
   1.6. Universal Asynchronous Receiver Transmitter (UART) ..... 3
   1.7. Pulse-Width Modulation (PWM) ..... 4
   1.8. IR Remote Control ..... 4
   1.9. Sniffer ..... 5

2. GPIO ..... 6
   2.1. Functional Overview ..... 6
   2.2. Instruction on GPIO Registers ..... 7
      2.2.1. GPIO Function Selection Register ..... 7
      2.2.2. GPIO Output Registers ..... 7
      2.2.3. GPIO Input Register ..... 8
      2.2.4. GPIO Interrupt Registers ..... 8
      2.2.5. GPIO16 Related APIs ..... 9
   2.3. Parameter onfiguration ..... 9
      2.3.1. Parameter Configuration for Scene 1 ..... 9
      2.3.2. Parameter Configuration for Scene 2 ..... 10
      2.3.3. Parameter Configuration for Scene 3 ..... 11
      2.3.4. Interrupt Function Processing Procedures ..... 12
      2.3.5. Example of The Interrupt Function Processing Procedures ..... 12

3. SPI Compatibility Mode User Guide ..... 13
   3.1. Functional Overview ..... 13
   3.2. DEMO Solution ..... 13
      3.2.1. Introduction ..... 13
      3.2.2. ESP8266 Software Compiling and Downloading ..... 13
      3.2.3. ESP8266 FLASH Software Downloading ..... 14

3.2.4. ESP8266 FLASH Software Downloading ....14
3.3. ESP8266 Software Instruction ....15
3.3.1. Protocol Principle: SDIO Line Breakage and SDIO Status Register ....15
3.3.2. Instructions on The Read/Write Buffer and The Registration Linked List ....16
3.3.3. API Functions in The ESP8266 DEMO ....17
3.4. STM32 Software Instruction ....18
3.4.1. Important functions ....18

4. SPI Communication User Guide ....21
4.1. Overview ....21
4.1.1. Functional Overview ....21
4.1.2. SPI Features ....21
4.2. ESP8266 SPI Master Protocol Format ....21
4.2.1. Communication Format Supported by Master SPI ....21
4.2.2. Master SPI Communication Format Supported by Current API ....22
4.3. ESP8266 SPI Slave Protocol Format ....22
4.3.1. SPI Slave Clock Polarity Configuration Requirement ....22
4.3.2. Communication Format Supported by Slave SPI ....22
4.3.3. Command Definition Supported by Slave SPI ....22
4.3.4. Slave SPI Communication Format Supported by Current API ....23
4.4. API Function Description of SPI Module ....23
4.4.1. API Function Description of Master SPI ....23
4.4.2. Master SPI API Function Description ....25
4.5. SPI Interface Description ....27
4.5.1. Data Structure ....27
4.5.2. API Description ....30
4.5.3. SPI_Test Demo ....35

5. SPI Overlap & Display Application Guide ....46
5.1. Functional Overview ....46
5.2. Hardware Connection of SPI Overlap Mode ....47
5.3. API Description of SPI Overlap Mode ....47
5.4. Display Screen Console Program DEMO ....48
5.4.1. Connection Description ....48
5.4.2. API Function Description ....48
5.4.3. Pre-compiled Macro Setting ....50

6. SPI Wi-Fi Passthrough 1-Interrupt Mode ....51

6.1. Functional Overview ..... 51
6.2. ESP8266 SPI Slave Protocol Format ..... 51
6.2.1. SPI Slave Clock Polarity Configuration ..... 51
6.2.2. Communication Format Supported by The SPI Slave ..... 51
6.3. Slave Status Definition and Line Breakage ..... 52
6.3.1. Status Definition ..... 52
6.3.2. GPIO0 Line Breakage ..... 52
6.4. ESP8266 SPI Slave API Functions ..... 52

7. SPI Wi-Fi Passthrough 2-Interrupt Mode ..... 58
7.1. Functional Overview ..... 58
7.2. ESP8266 SPI Slave Protocol Format ..... 58
7.2.1. SPI Slave Clock Polarity Configuration ..... 58
7.2.2. Communication Format Supported by The SPI Slave ..... 58
7.3. Instruction on The Data Flow Control Line ..... 59
7.3.1. GPIO0 MOSI Buffer Status ..... 59
7.3.2. GPIO2 Master Receives The Slave Send Buffer Status ..... 59
7.3.3. Master Communication Logic Implementation ..... 59
7.4. ESP8266 SPI Slave API Functions ..... 61

8. HSPI Host Multi-device API ..... 64
8.1. Functional Overview ..... 64
8.2. Hardware Connection ..... 64
8.3. API Description ..... 65

9. I2C User Guide ..... 67
9.1. Functional Overview ..... 67
9.2. I2C master Interface ..... 67
9.2.1. Initialization ..... 67
9.2.2. Start I2C ..... 67
9.2.3. Stop I2C ..... 68
9.2.4. I2C Master Responds ACK ..... 68
9.2.5. I2C Master Responds NACK ..... 68
9.2.6. Check I2C Slave Response ..... 69
9.2.7. Write Data on I2C Bus ..... 69
9.2.8. Read Data from I2C Bus ..... 69
9.3. Demo ..... 69

10.I2S Module Description ..... 71
10.1. Functional Overview ..... 71
10.2. System Configuration ..... 71
10.2.1. I2S Module Configuration ..... 71
10.2.2. Link List Configuration ..... 74
10.2.3. SLC Module Configuration ..... 75
10.3. API Function Description ..... 75
10.3.1. Void Function ..... 76
10.3.2. CONF Function ..... 76
10.3.3. START Function ..... 77

11.UART Introduction ..... 78
11.1. Functional Overview ..... 78
11.2. Hardware Resources ..... 79
11.3. Parameter Configuration ..... 79
11.3.1. The Baud Rate ..... 79
11.3.2. Parity Bit ..... 80
11.3.3. Data Bit ..... 80
11.3.4. Stop Bit ..... 80
11.3.5. Inverting ..... 80
11.3.6. Switch Output Port of Print Function ..... 81
11.3.7. Read The Remaining Number of Bytes in tx / rx Queue ..... 81
11.3.8. Loopback Operation (loop-back) ..... 81
11.3.9. Line Stop Signal ..... 81
11.3.10.Flow Control ..... 81
11.3.11.Other Interfaces ..... 82
11.4. Configure Interrupt ..... 82
11.4.1. Interrupt register ..... 82
11.4.2. Interface ..... 83
11.4.3. Interrupt Type ..... 83
11.5. Example of Interrupt Handler Process ..... 87
11.6. Abandon Serial Output During Booting ..... 87

12.PWM Interface ..... 89
12.1. Functional Overview ..... 89
12.1.1. Features ..... 89
12.1.2. Implementation ..... 89

12.1.3. Configuration .....90
12.1.4. Parameter Specification .....90
12.2. Details on pwm.h .....90
12.2.1. Sample Codes .....90
12.2.2. Interface Specifications .....91
12.3. Custom Channels.....93

13.IR Remote Control User Guide.....95
13.1. Introduction to Infrared Transmission .....95
13.1.1. Transmitting .....95
13.1.2. Receiving .....95
13.2. Parameters Configuration .....96
13.3. Functions of Infrared Sample Codes .....97

14.Sniffer Introduction .....98
14.1. Sniffer Introduction.....98
14.2. Sniffer Application Scenarios.....101
14.3. Phone APP .....103
14.4. IOT-device Firmware .....103

Appendix .....104

# 1. Overview

## 1.1. General Purpose Input/Output Interface (GPIO)

ESP8266EX has 17 GPIO pins which can be assigned to various functions by programming the appropriate registers.

Each GPIO can be configured with internal pull-up or pull-down, or set to high impedance, and when configured as an input, the data are stored in software registers; the input can also be set to edge-trigger or level trigger CPU interrupts. In short, the IO pads are bi-directional, non-inverting and tristate, which includes input and output buffer with tristate control inputs.

These pins can be multiplexed with other functions such as I2C, I2S, UART, PWM, IR Remote Control, etc.

## 1.2. Secure Digital Input/Output Interface (SDIO)

ESP8266EX has one Slave SDIO, the definitions of which are described below. 4-bit 25 MHz SDIO v1.1 and 4-bit 50 MHz SDIO v2.0 are supported.

**Table 1-1: Pin Definitions of SDIOs**

| Pin Name | Pin Num | IO | Function Name |
|---|---|---|---|
| SDIO_CLK | 21 | IO6 | SDIO_CLK |
| SDIO_DATA0 | 22 | IO7 | SDIO_DATA0 |
| SDIO_DATA1 | 23 | IO8 | SDIO_DATA1 |
| SDIO_DATA_2 | 18 | IO9 | SDIO_DATA_2 |
| SDIO_DATA_3 | 19 | IO10 | SDIO_DATA_3 |
| SDIO_CMD | 20 | IO11 | SDIO_CMD |

## 1.3. Serial Peripheral Interface (SPI/HSPI)

ESP8266EX has 3 SPIs.

One general Slave/Master SPI

One Slave SDIO/SPI

One general Slave/Master HSPI

Functions of all these pins can be implemented via hardware. The pin definitions are described as below.

### 1.3.1. General SPI (Master/Slave)

Table 1-2. Pin Definitions of SPIs

| Pin Name | Pin Num | IO | Function Name |
|---|---|---|---|
| SDIO_CLK | 21 | IO6 | SPICLK |
| SDIO_DATA0 | 22 | IO7 | SPIQ/MISO |
| SDIO_DATA1 | 23 | IO8 | SPID/MOSI |
| SDIO_DATA_2 | 18 | IO9 | SPIHD |
| SDIO_DATA_3 | 19 | IO10 | SPIWP |
| U0TXD | 26 | IO1 | SPICS1 |
| GPIO0 | 15 | IO0 | SPICS2 |

📖 ***Note:***

*SPI mode can be implemented via software programming. The clock frequency is 80 MHz at maximum.*

### 1.3.2. HSPI (Slave)

Table 1-3. Pin Definitions of HSPI (Slave)

| Pin Name | Pin Num | IO | Function Name |
|---|---|---|---|
| MTMS | 9 | IO14 | HSPICLK |
| MTDI | 10 | IO12 | HSPIQ/MISO |
| MTCK | 12 | IO13 | HSPID/MOSI |
| MTDO | 13 | IO15 | HPSICS |

## 1.4. I2C Interface

ESP8266EX has one I2C used to connect with micro-controller and other peripheral equipments such as sensors. The pin definition of I2C is as below.

Table 1-4. Pin Definitions of I2C

| Pin Name | Pin Num | IO | Function Name |
|---|---|---|---|
| MTMS | 9 | IO14 | I2C_SCL |
| GPIO2 | 14 | IO2 | I2C_SDA |

Both I2C Master and I2C Slave are supported. I2C interface functionality can be realized via software programming, the clock frequency reaches 100 kHz at a maximum. It should be noted that I2C clock frequency should be higher than the slowest clock frequency of the slave device.

## 1.5. I2S Interface

ESP8266EX has one I2S data input interface and one I2S data output interface. I2S interfaces are mainly used in applications such as data collection, processing, and transmission of audio data, as well as the input and output of serial data. For example, LED lights (WS2812 series) are supported. The pin definition of I2S is as below. I2S functionality can be enabled via software programming by using multiplexed GPIOs, and linked list DMA is supported.

**Table 1-5. Pin Definitions of I2S**

| I2S Data Input | | | |
|---|---|---|---|
| Pin Name | Pin Num | IO | Function Name |
| MTDI | 10 | IO12 | I2SI_DATA |
| MTCK | 12 | IO13 | I2SI_BCK |
| MTMS | 9 | IO14 | I2SI_WS |
| MTDO | 13 | IO15 | I2SO_BCK |
| U0RXD | 25 | IO3 | I2SO_DATA |
| GPIO2 | 14 | IO2 | I2SO_WS |

## 1.6. Universal Asynchronous Receiver Transmitter (UART)

ESP8266EX has two UART interfaces UART0 and UART, the definitions are as below.

**Table 1-6. Pin Definitions of UART**

| Pin Type | Pin Name | Pin Num | IO | Function Name |
|---|---|---|---|---|
| UART0 | U0RXD | 25 | IO3 | U0RXD |
| | U0TXD | 26 | IO1 | U0TXD |
| | MTDO | 13 | IO15 | U0RTS |
| | MTCK | 12 | IO13 | U0CTS |
| UART1 | GPIO2 | 14 | IO2 | U1TXD |
| | SD_D1 | 23 | IO8 | U1RXD |

Data transfers to/from UART interfaces can be implemented via hardware. The data transmission speed via UART interfaces reaches 115200 x 40 (4.5 Mbps).

UART0 can be used for communication. It supports fluid control. Since UART1 features only data transmit signal (Tx), it is usually used for printing log.

> 📖 ***Note:***
>
> *By default, UART0 outputs some printed information when the device is powered on and booting up. The baud rate of the printed information is relevant to the frequency of the external crystal oscillator. If the frequency of the crystal oscillator is 40 MHz, then the baud rate for printing is 115200; if the frequency of the crystal oscillator is 26 MHz, then the baud rate for printing is 74880. If the printed information exerts any influence on the functionality of the device, it is suggested to block the printing during the power-on period by changing (U0TXD,U0RXD) to (MTDO,MTCK).*

## 1.7. Pulse-Width Modulation (PWM)

ESP8266EX has four PWM output interfaces. They can be extended by users themselves. The pin definitions of the PWM interfaces are defined as below.

**Table 1-7. Pin Definitions of PWM**

| Pin Name | Pin Num | IO | Function Name |
|---|---|---|---|
| MTDI | 10 | IO12 | PWM0 |
| MTDO | 13 | IO15 | PWM1 |
| MTMS | 9 | IO14 | PWM2 |
| GPIO4 | 16 | IO4 | PWM3 |

The functionality of PWM interfaces can be implemented via software programming. For example, in the LED smart light demo, the function of PWM is realized by interruption of the timer, the minimum resolution reaches as much as 44 ns. PWM frequency range is adjustable from 1000 μs to 10000 μs, i.e., between 100Hz and 1 kHz. When the PWM frequency is 1 kHz, the duty ratio will be 1/22727, and over 14 bit resolution will be achieved at 1 kHz refresh rate.

## 1.8. IR Remote Control

One Infrared remote control interface is defined as below.

**Table 1-8. Pin Definitions of IR Remote Control**

| Pin Name | Pin Num | IO | Function Name |
|---|---|---|---|
| MTMS | 9 | IO14 | IR Tx |
| GPIO5 | 24 | IO5 | IR Rx |

The functionality of Infrared remote control interface can be implemented via software programming. NEC coding, modulation, and demodulation are used by this interface. The frequency of modulated carrier signal is 38 kHz, while the duty ratio of the square wave is 1/3. The transmission range is around 1m which is determined by two factors: one is the maximum value of rated current, the other is internal current-limiting resistance value in the infrared receiver. The larger the resistance value, the lower the current, so is the power, and vice versa. The transmission angle is between 15° and 30° which is determined by the radiation direction of the infrared receiver.

## 1.9. Sniffer

ESP8266 can enter promiscuous mode (sniffer). ESP8266 can capture complete IEEE 802.11 packets in the air or it can obtain the length of the packets.

# 2. GPIO

## 2.1. Functional Overview

The ESP8266 has 16 general IOs. Their pin numbers and names are shown in the table below:

**Table 2-1. GPIO Pin Definition**

| GPIO NO. | Pin NO. | Pin name |
|---|---|---|
| GPIO0 | pin15 | GPIO0_U |
| GPIO1 | pin26 | U0TXD_U |
| GPIO2 | pin14 | GPIO2_U |
| GPIO3 | pin25 | U0RXD_U |
| GPIO4 | pin16 | GPIO4_U |
| GPIO5 | pin24 | GPIO5_U |
| GPIO6 | pin21 | SD_CLK_U |
| GPIO7 | pin22 | SD_DATA0_U |
| GPIO8 | pin23 | SD_DATA1_U |
| GPIO9 | pin18 | SD_DATA2_U |
| GPIO10 | pin19 | SD_DATA3_U |
| GPIO11 | pin20 | SD_CMD_U |
| GPIO12 | pin10 | MTDI_U |
| GPIO13 | pin12 | MTCK_U |
| GPIO14 | pin9 | MTMS_U |
| GPIO15 | pin13 | MTDO_U |

In the QUAD mode flash, 6 IO interfaces are used for flash communication.

In the DUAL mode flash, 4 IO interfaces are used for flash communication.

📖 ***Note:***

*Users may find the following documents helpful:*

- ***Appendix 1 - GPIO Registers***
- *List of ESP8266 pin functions:* ***ESP8266_Pin_List.xlsx****:* http://bbs.espressif.com/viewtopic.php?f=21&t=412&p=1545#p1545。

## 2.2. Instruction on GPIO Registers

### 2.2.1. GPIO Function Selection Register

The ESP8266 MTDI is used to demonstrate the GPIO function selection.

Function selection register `PERIPHS_IO_MUX_MTDI_U` (this register differs for different GPIOs)

```
PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U,FUNC_GPIO12);
```

`FUNC_GPIO12`=3.

Configurations differ for different pins.

During the configuration, refer to ***ESP8266_Pin_List.xlsx***. On the Digital Die Pin List page, users can see the general GPIO and their multiple functions. On the Reg page, users can find registers related to GPIO function selection.

On the Digital Die Pin List page, users can find the function configuration in the FUNCTION column.

> ⚠️ ***Notice:***
>
> *If you want to configure it to be FUNCTION X, write X -1 into the bit in the register. For example, if you want to configure it to be FUNCTION 3, write 2 into the bit in the register.*

### 2.2.2. GPIO Output Registers

- **Output enable register: GPIO_ENABLE_W1TS**

bit[15:0] the output enable bit (readable and writable):

If the related bit is set to be 1, the IO output is enabled. Bit[15:0] contains 16 GPIO output enable bits.

- **Output disable register: GPIO_ENABLE_W1TC**

bit[15:0] the output disable bit (readable and writable):

If the related bit is set to be 1, the IO output is disabled. Bit[15:0] contains 16 GPIO output disable bits.

- **Output enable status register: GPIO_ENABLE**

bit[15:0] the output enable status bit (readable and writable):

Value of bit[15:0] of this register shows the related pin output enable status.

By writing data into bit[15:0] of GPIO_ENABLE_W1TS and bit[15:0] of GPIO_ENABLE_W1TC, users can control bit[15:0] of GPIO_ENABLE. For example, when bit[0] of GPIO_ENABLE_W1T is set to be 1, then bit[0] of GPIO_ENABLE =1; when bit[1] of GPIO_ENABLE_W1TC is set to be 1, then bit[1] of GPIO_ENABLE = 0.

- **Output low level register GPIO_OUT_W1TC**

bit[15:0] output low level bit (write only register):

If the related bit is set to be 1, the IO output is low level (at the same time, users should enable the output). Bit[15:0] contains 16 GPIO output statuses.

> 📖 ***Note:***
>
> *If users need to set the pin to high level, they need to configure the GPIO_OUT_W1T register.*

- **Output high level register GPIO_OUT_W1TS**

bit[15:0] output high level bit (write only register):

If the related bit is set to be 1, it means the IO output is high level (at the same time, users should enable the output). Bit[15:0] contains 16 GPIO output statuses.

> 📖 ***Note:***
>
> *If users need to set the pin to low level, they need to configure the GPIO_OUT_W1TC register.*

- **Output status register GPIO_OUT**

bit[15:0] output status bit (read/write register):

Value of bit[15:0] of this register shows the related pin output status.

Bit[15:0] of GPIO_OUT is decided by bit[15:0] of GPIO_OUT_W1TS and bit[15:0] of GPIO_OUT_W1TC. For example, when bit[1] of GPIO_OUT_W1TS =1, then GPIO_OUT[1] =1; when bit[2] of GPIO_OUT_W1TC = 1, then GPIO_OUT[2]=0.

### 2.2.3. GPIO Input Register

bit[15:0] the input status bit (readable and writable):

If the related bit is set to be 1, the IO pin status is high level. If the related bit is set to be 0, the IO pin status is low level. Bit[15:0] contains 16 GPIO input status bits.

> 📖 ***Note:***
>
> *The GPIO input detection function is enabled by default.*

### 2.2.4. GPIO Interrupt Registers

- **Interrupt type register GPIO_PIN12 (this register differs for different GPIOs)**

bit[9:7] (readable and writable):

0: the GPIO interrupt is disabled

1: rising edge triggered interrupt

2: falling edge triggered interrupt

3: double-edge triggered interrupt

4: low level

5: high level

- **Interrupt status register GPIO_STATUS**

Bit[15:0] (readable and writable):

If the related bit is set to be 1, the IO interrupts. Bit[15:0] contains 16 GPIOs.

- **Interrupt clearing register GPIO_STATUS_W1TC**

Bit[15:0] (readable and writable):

Write 1 into the related bit, the related GPIO interrupt status will be cleared.

### 2.2.5. GPIO16 Related APIs

Different from other IO interfaces, GPIO16(XPD_DCDC) belongs to the RTC module instead of the general GPIO module. It can be used to wake up the chip during deep-sleep; it can be configured to input or output mode; but it cannot trigger the IO interrupt. the APIs are shown below.

- `gpio16_output_conf(void)`

Set the GPIO16 to the output mode.

- `gpio16_output_set(uint8 value)`

Output high/low level from GPIO16. Configure GPIO16 to the output mode first.

- `gpio16_input_conf(void)`

Set the GPIO16 to the input mode.

- `gpio16_input_get(void)`

Read the GPIO16 input level status. Configure GPIO16 to the input mode first.

## 2.3. Parameter onfiguration

Three scenes are given as examples for parameter configuration:

- Configure the MTDI output high level, and enable the pull up.
- Configure the MTDI to the input mode, and get its level status.
- Configure the MTDI to falling edge triggers interrupt.

### 2.3.1. Parameter Configuration for Scene 1

1. Configure the MTDI to GPIO mode.

```
PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U,FUNC_GPIO12);
```

This sentence writes 1 into bits 4-5 of PERIPHS_IO_MUX_MTDI_U register. When bits 4-5 of PERIPHS_IO_MUX_MTDI_U are set to be 1, the MTDI is configured to the GPIO mode. For details of PERIPHS_IO_MUX_MTDI_U register, refer to ***Section 2.2, Instruction on GPIO Register***.

2. Configure the MTDI output high level.

```
GPIO_OUTPUT_SET(GPIO_ID_PIN(12), 1);
```

This sentence has two functions:

- Write 1 into bit 12 of GPIO_ENABLE_W1TS register. It enables the MTDI output function.
- Write 1 into bit 12 of GPIO_OUT_W1TS register. It sets MTDI output to high level.

> 📖 ***Note:***
>
> *To set MTDI output to low level, set the second parameter of this function to be 0.*

```
GPIO_OUTPUT_SET(GPIO_ID_PIN(12), 0);
```

This sentence has two functions:

- Write 1 into bit 12 of GPIO_ENABLE_W1TS register. It enables the MTDI output function.
- Write 1 into bit 12 of GPIO_OUT_W1TC register. It sets MTDI output to low level.

3. Enable the MTDI pull up.

```
PIN_PULLUP_EN(PERIPHS_IO_MUX_MTDI_U);
```

It writes 1 into bit 7 of PERIPHS_IO_MUX_MTDI_U. It enables the MTDI pull up.

> 📖 ***Note:***
>
> *To disable the MTDI pull up, use the following sentence:*

```
PIN_PULLUP_DIS(PERIPHS_IO_MUX_MTDI_U);
```

### 2.3.2. Parameter Configuration for Scene 2

1. Configure the MTDI to GPIO mode.

```
PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U,FUNC_GPIO12);
```

This sentence writes 1 into bits 4-5 of PERIPHS_IO_MUX_MTDI_U register. When bits 4-5 of PERIPHS_IO_MUX_MTDI_U are set to be 1, the MTDI is configured to the GPIO mode.

2. Configure the MTDI to the input mode.

```
GPIO_DIS_OUTPUT(GPIO_ID_PIN(12));
```

3. Get the MTDI pin level status.

Uint8 level=0;

```
level=GPIO_INPUT_GET(GPIO_ID_PIN(12));
```

GPIO_INPUT_GET(GPIO_ID_PIN(12)) gets the status of bit 12 of GPIO_IN register. The value of this register shows the input level of related pin. (Enable the input function of the related pin first to get effective register status)

> 📖 ***Note:***
>
> - *If MTDI is at high level, then the return value of GPIO_INPUT_GET is 1, level = 1;*
> - *If MTDI is at low level, then the return value of GPIO_INPUT_GET is 0, level = 0.*

### 2.3.3. Parameter Configuration for Scene 3

```
typedef enum {
GPIO_PIN_INTR_DISABLE = 0,
GPIO_PIN_INTR_POSEDGE = 1,
GPIO_PIN_INTR_NEGEDGE = 2,
GPIO_PIN_INTR_ANYEGDE = 3,
GPIO_PIN_INTR_LOLEVEL = 4,
GPIO_PIN_INTR_HILEVEL = 5
} GPIO_INT_TYPE;
```

This structure is used to configure the GPIO interrupt trigger manner. It is declared in ***gpio.h***.

1. Configure the MTDI to GPIO mode.

```
PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U,FUNC_GPIO12);
```

This sentence writes 1 into bits 4-5 of PERIPHS_IO_MUX_MTDI_U register. When bits 4-5 of PERIPHS_IO_MUX_MTDI_U are set to be 1, the MTDI is configured to the GPIO mode.

2. Configure the MTDI to the input mode.

```
GPIO_DIS_OUTPUT(GPIO_ID_PIN(12));
```

3. Disable all IO interrupts.

```
ETS_GPIO_INTR_DISABLE();
```

4. Set the interrupt handler function.

```
ETS_GPIO_INTR_ATTACH(GPIO_INTERRUPT,NULL);
```

5. Configure MTDI to falling edge triggers interrupt.

```
gpio_pin_intr_state_set(GPIO_ID_PIN(12),GPIO_PIN_INTR_NEGEDGE);
```

This sentence writes 0x02 into bit[9:7] of GPIO_PIN12 register. It sets MTDI to falling edge triggers interrupt.

> 📖 ***Note:***
>
> *If users want to disable the MTDI interrupt function, write 0x02 into bit[9:7] of GPIO_PIN12 register.*
>
> *For other interrupt triggering mode configuration, refer to **2.2 Instruction on GPIO Registers**.*

6. Enable the GPIO interrupt.

```
ETS_GPIO_INTR_ENABLE();
```

### 2.3.4. Interrupt Function Processing Procedures

1. Clear the interrupt.

```
Uint16 gpio_status=0;
gpio_status = GPIO_REG_READ(GPIO_STATUS_ADDRESS);
GPIO_REG_WRITE(GPIO_STATUS_W1TC_ADDRESS, gpio_status);
```

For instructions on GPIO_STATUS and GPIO_STATUS_W1TC, refer to ***Section 2.2 Instruction on GPIO Registers***.

2. check which IO triggered the interrupt (when multiple IOs are configured to be in interrupt mode)

```
If(gpio_status==GPIO_Pin_12)
```

3. If it is double-edge triggered interrupt, check whether this interrupt is triggered by rising or falling edge.

```
if(!GPIO_INPUT_GET(GPIO_ID_PIN(12))) //if this MTDI interrupt is
triggered by falling edge.
```

### 2.3.5. Example of The Interrupt Function Processing Procedures

```
void gpio_intr_handler()
{

    uint32 gpio_status = GPIO_REG_READ(GPIO_STATUS_ADDRESS);//read interrupt status
    uint8  level=0;
     GPIO_REG_WRITE(GPIO_STATUS_W1TC_ADDRESS, gpio_status);//clear interrupt mask
     if(gpio_status & (BIT(12))){  //judge whether interrupt source is gpio12
      if(GPIO_INPUT_GET(12)){   // if gpio 12 is high level

        }else{ // if gpio 12 is low level

      }
    }
     else{
     }

}
```

# 3. SPI Compatibility Mode User Guide

## 3.1. Functional Overview

This protocol uses the SDIO mode of the ESP8266 to communicate with other processor's SPI hosts. The electrical interface is connected through signal line No.4, including the SCLK, MOSI, MISO and interrupt signal No.1 in the SPI protocol (note: no CS signal).

Downloading the ESP8266 SDIO can be different from downloading other programs. When the ESP8266 starts, the system reads the pin shared by the SPI interface and the SDIO interface by default. Therefore, the SDIO module communication protocol should be used. The ESP8266 should start in the SDIO mode, and then, the host will start the chip in the ESP8266 RAM through the SDIO downloaded programs. The majority of the programs that directly use CPU CACHE to call FLASH can be burnt to the FLASH chip connected to the HSPI interface beforehand.

Data received or sent by the ESP8266 SDIO is processed directly by the DMA module that supports linked list index.

The ESP8266 can receive and send the SDIO packets efficiently without using the CPU. It does so through the address of the memory map linked list.

## 3.2. DEMO Solution

### 3.2.1. Introduction

The host is the Red Dragon demo board with STM32F103ZET6 as its core.The software is the FreeRTOS system developed by the IAR platform. The slave is the ESP_IOT reference board, which is based on the v0.9.3 SDK development.

### 3.2.2. ESP8266 Software Compiling and Downloading

- In the SDIO communication demo ***\esp_iot_sdk_v0.9.3_sdio_demo\ap***, use the compiler to compile and generate the bin documents for downloading in order to complete the ESP8266 DEMO work.
- ibmain.a in SDIO communication demo ***\esp_iot_sdk_v0.9.3_sdio_demo\lib*** is different from the version released in v0.9.3. When you use the released version of the SDK, use ***libmain.a*** in the DEMO to replace the original one. The new ***libmain.a*** will start the chip, and exchange the SPI module that reads FLASH and the HSPI mapping pin. Then, you can use DEMO to compile and generate.
- Copy ***eagle.app.v6.irom0text.bin*** in SDIO communication demo ***\esp_iot_sdk_v0.9.3_sdio_demo\bin*** to SDIO communication demoboard

\XTCOM_UTIL. ***eagle.app.v6.irom0text.bin*** is all the functions of FLASH chip read directly through the SPI by CPU CACHE in the ESP8266 program.

- Run ***BinToArray.exe*** in SDIO communication demo\. Transfer ***eagle.app.v6.flash.bin*** in SDIO communication demo ***\esp_iot_sdk_v0.9.3_sdio_demo*** bin to ANSI C format array. The new array will be saved in ***D:***. The target route of BinToArray.exe must be D:\. If there is not a D:\, you can (1) use a virtual machine with a D:\; (2) connect the device to a U disk named D:\; or (3) search online for a tool that can transfer bin to array.
- If there is a D:\, name ***hexarray.c*** in ***D:*** as ***eagle_fw.h***, and define the array name as const unsigned char eagle_fw[] =..... Replace ***eagle_fw.h*** in SDIO communication ***demo\STM32\ Eagle_Wifi_Driver\ egl_drv_simulation*** (you can copy the array name and document name in the old ***eagle_fw.h***, rename the ***hexarray.c*** and use it to replace the old ***eagle_fw.h.***). Before starting the chip, write eagle.app.v6.flash.bin into the ESP8266 memory. ***eagle.app.v6.flash.bin*** should be transferred to array, and be written into the ESP8266 through STM32.
- Use the IAR platform to open ***EglWB.ewp.eww*** in SDIO communication ***demo\STM32\IAR*** to compile the programs.

### 3.2.3. ESP8266 FLASH Software Downloading

1. Use the serial line to connect the ESP_IOT reference board and the computer, and connect them with a 5V power supply. Connect J67 to the 2 pins on the right (enable the FLASH chip in the HSPI interface), and J66 to the 2 pins on the left (disable the FLASH chip in the SPI interface). Set MTD0, GPIO0 and GPIO2 to the UART mode 0, 0,1 (up, up, down).
2. Double-click XTCOM_UTIL.exe in SDIO communication demo\XTCOM_UTIL. Click Tools -> Config Device, and choose Com interface. Baud Rate: 115200. Click Open, and you will see open Success. Click Connect, and push the H Flash board power, you will see the connection is completed.
3. Click API TEST(A)->(5) HSpiFlash Image Download, and choose ***eagle.app.v6.irom0text.bin*** in SDIO communication demo\XTCOM_UTIL. Offset: 0x40000. Click Download, and the downloading will be completed.

### 3.2.4. ESP8266 FLASH Software Downloading

Use the pin header to connect the ESP_IOT reference board and the Red Dragon demo board. The details are shown below:

In the Red Dragon demo board JP1:

J62 pin headers in the ESP_IOT reference board (bottom-up)

| | | | |
|---|---|---|---|
| GND | -> | 1 | VSS/GND |
| SPI_CLK | -> | 4 | SDIO_CLK |

| | | | |
|---|---|---|---|
| SPI_MOSI | -> | 5 | SDIO_CMD |
| SPI_MISO | -> | 3 | SDIO_DAT0 |
| IRQ | -> | 2 | SDIO_DAT1 |

The ESP_IOT reference board: change the jumper MTD0 to 1 (short the 2 pins below), GPIO0, GPIO2 random (1, x, x is the SDIO starting mode), CHIP_PD:ON (flip the switch downward). Keep jumper J66 connected to the 2 pins on the left, and jumper J67 connected to the 2 pins on the right.

Connect the 5V power adapter to the ESP_IOT reference board and the Red Dragon demo board. Turn on the demoboard power, download the compiled programs mentioned in Section 2.2 to STM32 in the IAR environment. Start the STM32 program, and turn on the ESP_IOT reference board power. the STM32 will write the starting program into the ESP8266, and after several seconds, it will automatically run the SDIO to return to the testing program.

## 3.3. ESP8266 Software Instruction

### 3.3.1. Protocol Principle: SDIO Line Breakage and SDIO Status Register

In the SDIO SPI compatibility mode, pin SD_DATA1 of the ESP8266 is used as the interrupt line to send signals to the SPI host, and the signals are active low. When the ESP8266 SDIO status register is upgraded by software, the interrupt line will change from active high to active low. The host should write in data to resume the active high through SDIO. (to be specific, the host should write 1 into register with the address 0x30 through CMD53 or CMD52 command in order to resume the active high of the interrupt line.)

the SDIO status register is 32 bits, it is revised by ESP8266 software, and it can be read by the host through CMD53 or CMD52 command. The address is 0x20-0x23. The data structure is shown as below:

```
struct sdio_slave_status_element
{
u32 wr_busy:1;
u32 rd_empty :1;
u32 comm_cnt :3;
u32 intr_no :3;
u32 rx_length:16;
u32 res:8;
};
```

To be specific:

- wr_busy, bit 0: 1, write buffer of the slave is full, and the ESP8266 is processing data from the host; 0, write buffer is empty, users can write data into the buffer.

- rd_empty, bit 1: 1, read buffer of the slave is empty, no data has been updated; 0, there is new data in the buffer for the host to read.
- comm_cnt, bit 2-4: count the read/write communication. Each time the ESP8266 SDIO module finishes an effective packet-reading/packet-writing, the count will increase by 1. Therefore, the host can judge whether a read/write communication has been effectively responded by the ESP8266.
- intr_no, bit 5-7: the protocol does not use this variable; reserved.
- rx_length, bit 8-23: actual length of the packets prepared in the read buffer.
- res, bit 24-31: reserved.

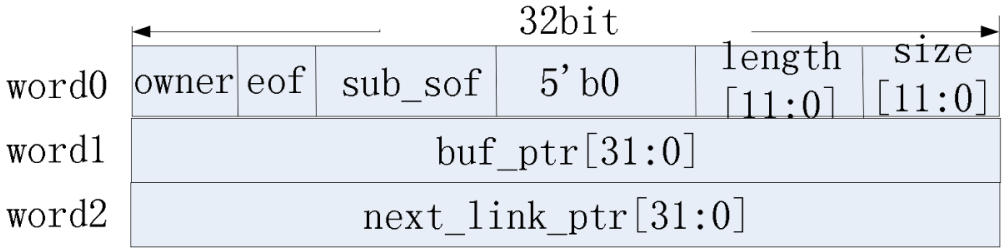
The communication procedures of the host are shown as below:

- upon receiving the interrupt request, the host reads the SDIO status register, and then clears the interuption, and reads/writes the packets according to the status register;
- it checks the SDIO status register regularly, and reads/writes the packets according to the status register.

### 3.3.2. Instructions on The Read/Write Buffer and The Registration Linked List

DMA will directly send packets received and sent by the ESP8266 SDIO to corresponding memories. The ESP8266 software will define the linked list registration structure (or array), and buffer(s). In this example, only one buffer is used, and there is only one element in the linked list. Write the first address of the buffer into the linked list registration structure, and write in other information. When you write the first address of the linked list structure into the corresponding hardware register in the ESP8266, the DMA can automatically process the SDIO and the buffer.

The linked list registration structure is shown as below:




- owner: 1'b0: operator of the current link buffer is SW; operator of the current link buffer. MAC does not use this bit. 1'b1: operator of the current link buffer is HW.
- eof: flag of the end of the frame (for the end of AMPDU sub-frames, this flag is not needed). When the MAC sends the frames, it is used to mark the end of the frames. For links in eof, buffer_length[11:0] must be equal to the length of the remaining part of the frame. Otherwise, the MAC will report an error. When the MAC receives frames, it is used to indicate that the reception has been completed, and the value is set by hardware.

- sub_sof: the flag of the start of the sub-frame. It is used to distinguish different AMPDU sub-frames. It is only used when the MAC is sending packets.
- length[11:0]: actual size of the buffer.
- size[11:0]: total size of the buffer.
- buf_ptr[31:0]: starting address of the buffer.
- next_link_ptr[31:0]: starting address of the next discripter. When the MAC is receiving frames, the value is 0, indicating that there is no empty buffer to receive the frames.

### 3.3.3. API Functions in The ESP8266 DEMO

1. `void sdio_slave_init(void)`

Function: Initialise the SDIO module, including initialising the status register, initialising the Rx and Tx registration linked list, configuring the communication interrupt line mode, configuring packet-sending/receiving interruption, and registering the interrupt service routine, etc.

2. `void sdio_slave_isr(void *para)`

Function and trigger condition: The SDIO interrupt processing function; this function will be triggered when the SDIO successfully receives or sends a packet. in DEMO, all the ESP8266 testing procedures are completed in the interrupt processing function. All the processing procedures of the registration linked lists, status registers and data during the communication process can be found in this function.

3. `void rx_buff_load_done(uint16 rx_len)`

Function: When rx_buffer receives new packets, this function should be called to change the status of the new packets to "to be read". This function contains related operations of the software/hardware of the registration linked list, and the status register. In DEMO, this function will be called in the interrupt service routine.

Parameter: rx_len: actual length of the new packet (unit: byte).

4. `void tx_buff_handle_done(void)`

Function: When data in tx_buffer has been processed, this function should be called to change the SDIO status to "sent" in order to receive the next packet. This function contains related operations of the software/hardware of the registration linked list, and the status register. In DEMO, this function will be called in the interrupt service routine.

5. `void rx_buff_read_done(void)`

Function: When data in rx_buffe has been read, this function should be called to change the SDIO status to "non-readable". This function contains related operations of the status register, and should be called at the beginning of the RX_EOF interrupt service.

6. `void tx_buff_write_done(void)`

Function: When tx_buffer receives new packets, this function should be called to change the SDIO status to "non-writable". This function contains related operations of the status register, and should be called at the beginning of the TX_EOF interrupt service.

7. `TRIG_TOHOST_INT()`

Function: Macro, pull low the communication interrupt line, inform the host.

8. **Other functions**

Other functions are used for tests.

## 3.4. STM32 Software Instruction

### 3.4.1. Important functions

1. `void SdioRW(void *pvParameters)`

Function:

SDIO testing thread, it contains all the read/write procedures.

Location:

***egl_thread.c***. Registered by `SPITest()` of the same file in ***egl_thread.c***.

2. `int esp_sdio_probe(void)`

Function:

Enable related programs in the ESP8266.

Location:

*esp_main_sim.c*. Called by `SPITest()` in *egl_thread.c*.

3. `int sif_spi_write_bytes(u32 addr, u8*src,u16 count,u8 func)`

Function:

Write the SDIO byte mode into the API; encapsulate the write-in function of the CMD53 byte mode. It can process the register and the packets. According to the SDIO protocol, the maximum data length is 512 Bytes.

Location:

*port_spi.c*. Called by `SdioRW` in *egl_thread.c*.

Parameters:

`src`: starting address of the packet to be sent.

`count`: length of the packet to be sent, (unit: Byte).

`func`: function number. It is 0 for communication of block_size in the block mode used to revise the SDIO CMD53, and 1 for all other communications.

`addr`: starting address of the data to be written in. If you want to process the register, input the corresponding address, for example, 0x30, interrupt line clearance register, 0x110, revise block_size (func=0). If you want to process the packets, input a value that equals to 0x1f800 - tx_length, and 0x1f800 - tx_length should equal to count. If count > tx_lengt, the SPI host will send packets of count length. But data between tx_length + 1 and count will

be discarded by the ESP8266 SDIO module. Therefore, when sending packets, addr is related to the actual length of the effective data.

4. `int sif_spi_read_bytes(u32 addr,u8* dst,u16 count,u8 func)`

Function:

The SDIO byte mode reads the API; encapsulate the read function of the CMD53 byte mode. It can process the register or the packets. According to the SDIO protocol, the maximum data length is 512 Bytes.

Location:

***port_spi.c***. Called by `SdioRW` in ***egl_thread.c***.

Parameters:

`dst`: starting address of the receiving buffer

`count`: length of the packet to be received (unit: Byte)

`func`: function number. It is 0 for communication of block_size in the block mode used to read the SDIO CMD53, and 1 for all other communications.

`addr`: starting address of the data to be read. If you want to operate the register, input the corresponding address. For example, 0x20, the SDIO status register. If you want to operate the packets, input a value that equals 0x1f800 - `tx_length`, and 0x1f800 - `tx_length` equals count. If count > `tx_length`, the SPI host will send packets of count length. But data between `tx_length` + 1 and count will be discarded by the ESP8266 SDIO module. Therefore, when sending packets, `addr` is related to the actual length of the effective data.

5. `int sif_spi_write_blocks(u32 addr, u8 * src, u16 count,u16 block_size)`

Function:

Write the SDIO block mode into the API; encapsulate the write-in function of the CMD53 byte mode. It can only transport the packets, According to the SDIO protocol, the maximum data length is 512 blocks.

Location:

***port_spi.c***. Called by `dioRW` in `egl_thread.c` and `sif_io_sync` used by the program downloader in ***esp_main_sim.c***.

Parameters:

`src`: starting address of the packet to be sent.

`count`: length of the packet to be sent (unit: block)

`block_size`: the number of bytes in 1 block. It should be equal to the 16 bit value whose `func`=0, and whose `addr`=0x110-111. In general, when initialising the SDIO, block_size of the ESP8266 SDIO should be configured. The starting value of DEMO is 512. During the operation, it is configured to be 1024. block_size should be an integer multiple of 4.

`addr`: starting address of the data to be written in. Input a value that equals 0x1f800 - `tx_length` (the same as the byte mode), and the `tx_length` should equal to count.

6. `int sif_spi_read_blocks(u32 addr, u8 *dst, u16 count,u16 block_size)`

Function:

Write the SDIO block mode into the API; encapsulate the write-in function of the CMD53 byte mode. It can only transport the packets, According to the SDIO protocol, the maximum data length is 512 blocks.

Location:

***port_spi.c.*** Called by `dioRW` in `egl_thread.c` and `sif_io_sync` used by the program downloader in ***esp_main_sim.c.***

Parameters:

`src`: starting address of the receiving buffer

`count`: length of the packet to be received (unit: block)

`block_size`: the number of bytes in 1 block. It should be equal to the 16 bit value whose `func` = 0, and whose `addr`=0x110-111. In general, when initialising the SDIO, `block_size` of the ESP8266 SDIO should be configured. The starting value of DEMO is 512. During the operation, it is configured to be 1024. `block_size` should be an integer multiple of 4.

`addr`: starting address of the data to be read. Input a value that equals 0x1f800 - `tx_length` (the same as the byte mode), and the `tx_length` should equal to count.

7. `void EXTI9_5_IRQHandler(void)`

Function:

The communication interrupt processing function offers enable signal for `egl_arch_sem_wait` (& `BusIrqReadSem`,1000) in thread function `SdioRW`, so that `SdioRW` thread can exit the wait state, and read the SDIO status register.

Location:

***spi_cfg.c***

# 4. SPI Communication User Guide

## 4.1. Overview

### 4.1.1. Functional Overview

ESP8266 SPI module is used for communication with devices supporting SPI protocols. It supports the SPI protocol standard of 4 line communication（CS, SCLK, MOSI, MISO） in the electrical interface. ESP8266 SPI module has special support for FLASH memory in the SPI interface. Therefore, master and slave SPI module have its corresponding hardware protocol to match with the SPI communication device.

### 4.1.2. SPI Features

- Supports standard master and slave modes;
- Supports length-programmable hardware commands and addresses, up to 16 bits and 64 bits;
- Word-aligned data buffer, up to 64 bytes;
- Programmable read/write status register in slave mode;
- Selection of 3 CS pins;
- Clock frequency up to 80 MHz in master mode and 20 MHz in slave mode;
- Programmable clock polarity;
- MSB or LSB first;
- Selection of byte order in SPI buffer transmission;
- Selection of multiple interrupt sources, including transmit end, read/write data and read/write status.

## 4.2. ESP8266 SPI Master Protocol Format

### 4.2.1. Communication Format Supported by Master SPI

Master ESP8266SPI communication format is command+address+read/write data, which is,

- Command: a must; length: 1 ~ 16 bits; master output and slave input (MOSI).
- Address: optional; length: 0 ~ 32 bits； master output and slave input (MOSI).
- Read/write data: optional; length: 0 ~ 512 bits (64 Bytes); master output and slave input (MOSI) or master input and slave output (MISO).

### 4.2.2. Master SPI Communication Format Supported by Current API

The API function of ESP8266 SPI has two master initialization modes: one supports most of the general signals and the other is designed for driving a colored LCD screen. The device needs non-standard 9 bits SPI communication format. Please refer to ***Section 4.4.1*** for detailed information.

## 4.3. ESP8266 SPI Slave Protocol Format

### 4.3.1. SPI Slave Clock Polarity Configuration Requirement

The master device clock polarity configuration of ESP8266 SPI slave communication should be set with idle low power , rising edge sampling and falling edge data transformation. Please make sure to keep low power for CS in a 16's reading/writing process. If the CS power is raised to high level while sending, the internal state of slave will be reset.

### 4.3.2. Communication Format Supported by Slave SPI

Slave ESP8266SPI communication format is almost the same as that of the master mode, i.e. command+address+read/write data, but the slave read/write operation has its hardware command and undeletable address, which is,

- Command: a must; length: 3 ~ 16 bits; master output and slave input (MOSI).
- Address: a must; length: 1 ~ 32 bits； master output and slave input (MOSI).
- Read/write data: optional; length: 0 ~ 512 bits (64 Bytes); master output and slave input (MOSI) or master input and slave output (MISO).

### 4.3.3. Command Definition Supported by Slave SPI

The length of slave receiving command should at least be 3 bits. For low 3 bits, there are hardware reading and writing operation, which is,

- 010 (slave receiving) : Write the data sent by master into the register of slave data caching via MOSI, i.e. SPI_FLASH_C0 to SPI_FLASH_C15.
- 011 (slave sending): Send the data in the register of slave data caching (from SPI_FLASH_C0 to SPI_FLASH_C15) to master via MOSI.
- 110 (slave receiving and sending): Send slave data caching to MISO and write the master data in MOSI into data caching SPI_FLASH_C0 to SPI_FLASH_C15.

> ⚠️ ***Notice:***
> *Other vales are used to read and write the status register of slave SPI, SPI_FLASH_STATUS. Please do not use it because the difference between communication format and data caching reading/writing might lead to slave read/write error.*

### 4.3.4. Slave SPI Communication Format Supported by Current API

The API function of ESP8266 SPI has a slave initialization mode which is compatible with most of the devices in bytes. Set the slave communication format of 7 bits command+8 bits read/write data so that other master SPI devices could read and write bytes of slave SPI via the 16 bits communication (or two times 8 bits with low lever CS). Please refer to ***Section 4.4.2*** for detailed information.

## 4.4. API Function Description of SPI Module

### 4.4.1. API Function Description of Master SPI

1. `void spi_lcd_mode_init(uint8 spi_no)`

Function:

Provide master SPI initialization program for driving the chromatic LCD TM035PDZV36.

| Parameter | Description |
|---|---|
| `uint8 spi_no` | The number of SPI module. Only input SPI(0) and HSPI(1). Any other inputs are invalid. |

2. `void spi_lcd_9bit_write(uint8 spi_no,uint8 high_bit,uint8 low_8bit)`

Function:

Provide master SPI transmitting program for driving the chromatic LCD TM035PDZV36. The LCD module needs a 9 bits transmitting.

| Parameter | Description |
|---|---|
| `uint8 spi_no` | The number of SPI module. Only input SPI(0) and HSPI(1). Any other inputs are invalid. |
| `uint8 high_bit` | he 9's data. 0 represents the 9's 0 and other data represents the 9's 1. |
| `uint8 low_8bit` | Low 8 bit data. |

3. `void spi_master_init(uint8 spi_no)`

Function:

Normal master SPI initialization function. Baud rate is the 1/4 frequency of CPU clock. All the master functions except `spi_lcd_9bit_write` can be used after initialization.

| Parameter | Description |
|---|---|
| `uint8 spi_no` | The number of SPI module. Only input SPI(0) and HSPI(1). Any other inputs are invalid. |

4. `void spi_mast_byte_write(uint8 spi_no,uint8 data)`

Function:

Master data sending of one byte.

| Parameter | Description |
|---|---|
| `uint8 spi_no` | The number of SPI module. Only input SPI(0) and HSPI(1). Any other inputs are invalid. |
| `uint8 data` | 8 bit data sending. |

5. `void spi_byte_write_espslave(uint8 spi_no,uint8 data)`

Function:

Write a Byte data for slave SPI.

As the slave is set at 7bits command+1bit address+8bits data, data sending requires 16 bits transmission and the first byte is 0b0000010+0（refer to 3.3 ）, i.e. 0x04. The second byte is data sending. The actual transmitting waveform is illustrated in Figure 4-1.

| Parameter | Description |
|---|---|
| `uint8 spi_no` | The number of SPI module. Only input SPI(0) and HSPI(1). Any other inputs are invalid. |
| `uint8 data` | 8 bit data sending. |

**Figure 4-1. The waveform of spi_byte_write_espslave written into slave ESP8266**

> 📖 ***Note:***
>
> *Yellow line: CS, blue line: CLK, red line: MOSI, green line: MISO.*

6. `void spi_byte_read_espslave(uint8 spi_no,uint8 *data)`

Function:

Read one byte data from slave SPI and read other SPI slave devices. As the slave device is set at 7bits command+1bit address+8bits data, data sending requires 16 bits transmission and the first byte is 0b0000011+0 (refer to ***Section 4.3.3***), i.e. 0x06. The second Byte is data sending. The actual operating waveform is illustrated in Figure 4-2.

For other full duplex slave devices, 16 bits slave communication should be set. The effective data should be put to the second byte of slave sending caching which will be received by master ESP8266.

| Parameter | Description |
|---|---|
| `uint8 spi_no` | The number of SPI module. Only input SPI(0) and HSPI(1). Any other inputs are invalid. |
| `uint8 data` | 8 bit memory address data receiving. |

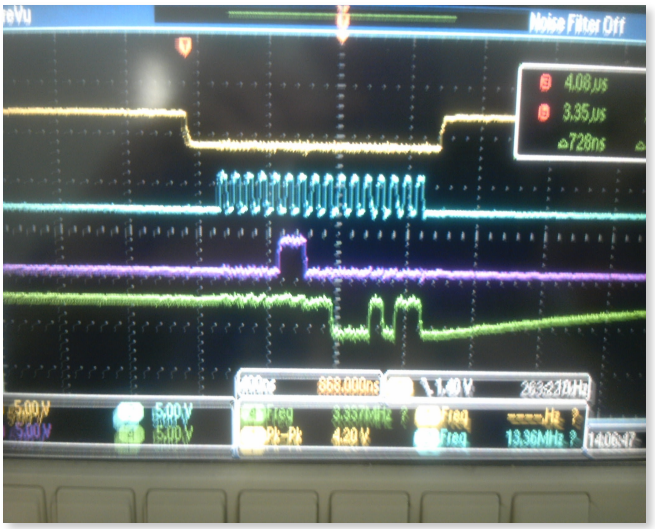



**Figure 4-2. The slave waveform of spi_byte_read_espslave read from ESP8266**

> 📖 ***Note:***
>
> *Yellow line: CS, blue line: CLK, red line: MOSI, green line: MISO.*

### 4.4.2. Master SPI API Function Description

1. `void spi_slave_init(uint8 spi_no)`

Function:

Initialization of slave SPI mode. Configure IO interface to SPI mode, enable SPI transmission interruption and register the function spi_slave_isr_handler.

Communication format is set at 7bits command +1bit address+8bits read/write data. Command and address combines to be high 8 bits and the address must be 0. According to descriptions in 3.3, it supports the three master commands, i.e. 0x04 master write and slave read, 0x06master write and slave read, 0x0c master and slave read/write. The communication waveform is illustrated in Figure 4-1, 4-2.

| Parameter | Description |
|---|---|
| `spi_no` | The number of SPI module. ESP8266 processor has two SPI modules with the same function, i.e. SPI and HSPI.<br>Optional values: SPI or HSPI. |

2. `spi_slave_isr_handler(void *para)`

Function and trigger condition:

SPI interrupt processing function. Interruption will be triggered if the master operates the correct transmission operation(read/write slave).

Code:

```
      //0x3ff00020 is isr flag register, bit4 is for spi isr,
      if(READ_PERI_REG(0x3ff00020)&BIT4){
            //following 3 lines is to close spi isr enable
            regvalue=READ_PERI_REG(SPI_FLASH_SLAVE(SPI));
            regvalue&=~(0x3ff);
            WRITE_PERI_REG(SPI_FLASH_SLAVE(SPI),regvalue);
            //os_printf("SPI ISR is trigged\n"); //debug code
      }else if(READ_PERI_REG(0x3ff00020)&BIT7){ //bit7 is for hspi
isr,
            //following 3 lines is to clear hspi isr signal
            regvalue=READ_PERI_REG(SPI_FLASH_SLAVE(HSPI));
            regvalue&=~(0x1f);
            WRITE_PERI_REG(SPI_FLASH_SLAVE(HSPI),regvalue);
       //when master command is write slave 0x04,
       //recieved data will be occur in register SPI_FLASH_C0's low 8
bit,
       //also if master command is read slave 0x06,
       //the low 8bit data in register SPI_FLASH_C0 will transmit to
master,
            //so prepare the transmit data in SPI_FLASH_C0' low 8bit,
            //if a slave transmission needs
            recv_data=(uint8)READ_PERI_REG(SPI_FLASH_C0(HSPI));
            /*put user code here*/
      //    os_printf("recv data is %08x\n", recv_data);//debug code
      }else if(READ_PERI_REG(0x3ff00020)&BIT9){ //bit9 is for i2s
isr,
      }
```

Code description: As SPI store the FLASH chip by the read/write program, HSPI is used for communication. For ESP8266 processor, there are multiple devices that share the interruption function, including SPI module, HSPI module, I2S module, the 4's, 7's and 9's 0x3ff00020 in the register.

As SPI module triggers transmission interruption frequently, 5 interruption source enabled should be closed. The corresponding codes are as follows:

```
regvalue=READ_PERI_REG(SPI_FLASH_SLAVE(SPI));
regvalue&=~(0x3ff);
WRITE_PERI_REG(SPI_FLASH_SLAVE(SPI),regvalue);
```

If HSPI is triggered, software that resets the 5 interruption source is needed, in order to avoid the repeated interruption function. The corresponding codes are as follows:

```
regvalue=READ_PERI_REG(SPI_FLASH_SLAVE(HSPI));
regvalue&=~(0x1f);
WRITE_PERI_REG(SPI_FLASH_SLAVE(HSPI),regvalue);
```

Data receiving and transmitting data share one register, SPI_FLASH_C0. The corresponding codes of readout register are as follows:

```
recv_data=(uint8)READ_PERI_REG(SPI_FLASH_C0(HSPI));
```

`recv_data` is a global variable. Processing program defined by users can be inserted to the tail of the sentence.

> ⚠ ***Notice:***
> *Interruption program is unfit for time-consuming processing code because long-time interruption program will cause watchdog timer unable to realize normal reset and will also lead to unexpected restart of processor.*

## 4.5. SPI Interface Description

> ⚠ ***Notice:***
> *The contents of this chapter are applicable only for Non-OS SDK V1.5.3 and above.*

### 4.5.1. Data Structure

#### 4.5.1.1. Enumerated Values

**SpiMode**

| Value | Description |
|---|---|
| SpiMode_Master | Master mode |
| SpiMode_Slave | Slave mode |

**SpiSubMode**

| Value | Description |
|---|---|
| SpiSubMode_0 | SPI_CPOL（0） SPI_CPHA（0） |
| SpiSubMode_1 | SPI_CPOL（0） SPI_CPHA（1） |

| Value | Description |
|---|---|
| SpiSubMode_2 | SPI_CPOL（1） SPI_CPHA（0） |
| SpiSubMode_3 | SPI_CPOL（1） SPI_CPHA（1） |

### SpiSpeed

| Value | Description |
|---|---|
| SpiSpeed_0_5MHz | SPI speed at 0.5 MHz |
| SpiSpeed_1MHz | SPI speed at 1 MHz |
| SpiSpeed_2MHz | SPI speed at 2 MHz |
| SpiSpeed_5MHz | SPI speed at 5 MHz |
| SpiSpeed_8MHz | SPI speed at 8 MHz |
| SpiSpeed_10MHz | SPI speed at 10 MHz |

### SpiBitOrder

| Value | Description |
|---|---|
| SpiBitOrder_MSBFirst | MSB first |
| SpiBitOrder_LSBFirst | LSB first |

### SpiIntSrc

| Value | Description |
|---|---|
| SpiIntSrc_TransDone | Transmit complete interrupt |
| SpiIntSrc_WrStaDone | Write status register interrupt |
| SpiIntSrc_RdStaDone | Read status register interrupt |
| SpiIntSrc_WrBufDone | Write data register interrupt |
| SpiIntSrc_RdBufDone | Read data register interrupt |

### SpiPinCS

| Value | Description |
|---|---|
| SpiPinCS_0 | CS0 pin |
| SpiPinCS_1 | CS1 pin |
| SpiPinCS_2 | CS2 pin |

#### 4.5.1.2. Structure

> 📖 ***Note:***
>
> *For details that require attention, please refer to [ESP8266 SDK API Guide](#).*

**SpiAttr**

SPI parameters configuration

```
typedef struct
{
    SpiMode        mode;            ///< Master or slave mode
    SpiSubMode     subMode;         ///< SPI SPI_CPOL SPI_CPHA mode
    SpiSpeed       speed;           ///< SPI Clock
    SpiBitOrder    bitOrder;        ///< SPI bit order
} SpiAttr;
```

**SpiData**

Data structure of SPI transmission

```
typedef struct
{
    uint16_t    cmd;            ///< Command value
    uint8_t     cmdLen;         ///< Command byte length
    uint32_t    *addr;          ///< Point to address value
    uint8_t     addrLen;        ///< Address byte length
    uint32_t    *data;          ///< Point to data buffer
    uint8_t     dataLen;        ///< Data byte length.
} SpiData;
```

**SpiIntInfo**

Information structure of SPI interrupt configuration

```
typedef struct
{
    SpiIntSrc     src;          ///< Interrupt source
    void          *isrFunc;     ///< SPI interrupt callback function.
} SpiIntInfo;
```

#### 4.5.1.3. Constants

**ESP8266 Commands**

| Name | Value | Description |
|---|---|---|
| MASTER_WRITE_DATA_TO_SLAVE_CMD | 2 | Write data command in ESP8266 slave mode. |
| MASTER_READ_DATA_FROM_SLAVE_CMD | 3 | Read data command in ESP8266 slave mode. |
| MASTER_WRITE_STATUS_TO_SLAVE_CMD | 1 | Write status register command in ESP8266 slave mode. |

| | | |
|---|---|---|
| MASTER_READ_STATUS_FROM_SLAVE_CMD | 4 | Read status register command in ESP8266 slave mode. |

### 4.5.2. API Description

> 📖 ***Note:***
>
> *For details that require attention, please refer to [ESP8266 SDK API Guide](#).*

#### 4.5.2.1. SPIInit

**Description**

SPI module initialization.

**Function**

```
void SPIInit(SpiNum spiNum, SpiAttr* pAttr);
```

| Parameter | Description |
|---|---|
| `spiNum` | [in] choose SPI and HSPI. |
| `pAttr` | [in] a pointer to SpiAttr structure. |

**Return value**

Null

> 📖 ***Notes:***
>
> - *In slave mode, the default CMD length is 8 bits, ADDR length 8 bits, DATA length 32 bytes.*
> - *No support currently for transmission with only DATA.*
> - *The maximum DATA length is 64 bytes in a single transmission.*

#### 4.5.2.2. SPIMasterCfgAddr

**Description**

Configure address register.

**Function**

```
void SPIMasterCfgAddr(SpiNum spiNum, uint32_t addr);
```

| Parameter | Description |
|---|---|
| `spiNum` | [in] choose SPI and HSPI. |
| `addr` | [in] address to set. |

**Return value**

Null

> 📖 ***Notes:***
>
> - *If the address length is over 32 bits, the user needs to configure the SPI_WR_STATUS register.*
> - *Address transmission is in high-byte order.*

#### 4.5.2.3. SPIMasterCfgCmd

**Description**

Configure SPI command register.

**Function**

```
Void SPIMasterCfgCmd(SpiNum spiNum, uint32_t cmd);
```

| Parameter | Description |
|---|---|
| `spiNum` | [in] choose SPI and HSPI. |
| `cmd` | [in] command value to set. |

**Return value**

Null

> 📖 ***Note:***
>
> *CMD length is up to 16 bits and the transmission is in low-byte order.*

#### 4.5.2.4. SPIMasterSendData

**Description**

Master sends data according to the pInData buffer.

**Function**

```
int SPIMasterSendData(SpiNum spiNum, SpiData* pInData);
```

| Parameter | Description |
|---|---|
| `spiNum` | [in] choose SPI and HSPI. |
| `pInData` | [in] a pointer to SpiData structure. The command, address, data buffer and length should be specified. |

**Return value**

- 0: Success
- Others: Failure

> 📖 ***Note:***
>
> *DATA transmission is in low-byte order.*

#### 4.5.2.5. SPIMasterRecvData

**Description**

Master receives data.

**Function**

```
int SPIMasterRecvData(SpiNum spiNum, SpiData* pOutData);
```

| Parameter | Description |
|---|---|
| `spiNum` | [in] choose SPI and HSPI. |
| `pOutData` | [in] a pointer to SpiData structure. The command, address, data buffer and length should be specified. |

**Return value**

- 0: Success
- Others: Failure

#### 4.5.2.6. SPISlaveSendData

**Description**

Upload data to SPI W8 ~ W15.

**Function**

```
int SPISlaveSendData(SpiNum spiNum, uint32_t *pInData, uint8_t
inLen);
```

| Parameter | Description |
|---|---|
| `spiNum` | [in] choose SPI and HSPI. |
| `pInData` | [in] a pointer to buffer. |
| `inLen` | [in] buffer length. |

**Return value**

- 0: Success
- Others: Failure

📖 ***Notes:***

- *This function is only used to upload the data to SPI W8 ~ W15. Upon receiving MASTER_READ_DATA_FROM_SLAVE_CMD, ESP8266 will automatically transmit data.*
- *The default value is 32 bytes, with 64 bytes the maximum.*

#### 4.5.2.7. SPISlaveRecvData

**Description**

Slave receives data.

#### Function

```
int SPISlaveRecvData(SpiNum spiNum);
```

| Parameter | Description |
|---|---|
| `spiNum` | [in] choose SPI and HSPI. |

#### Return value

- 0: Success
- Others: Failure

### 4.5.2.8. SPIMasterSendStatus

#### Description

Master writes data to slave's status register.

#### Function

```
void SPIMasterSendStatus(SpiNum spiNum, uint8_t data);
```

| Parameter | Description |
|---|---|
| `spiNum` | [in] choose SPI and HSPI. |
| `data` | [in] data to write into status register. |

#### Return value

Null

### 4.5.2.9. SPIMasterRecvStatus

#### Description

Master reads data from slave's status register.

#### Function

```
int SPIMasterRecvStatus(SpiNum spiNum);
```

| Parameter | Description |
|---|---|
| `spiNum` | [in] choose SPI and HSPI. |

#### Return value

- 0: Success
- Others: Failure

> 📖 ***Note:***
>
> *The status register value of the slave is stored in SPI buffer W0.*

#### 4.5.2.10.SPICsPinSelect

**Description**

Select CS pin.

**Function**

```
void SPICsPinSelect(SpiNum spiNum, SpiPinCS pinCs);
```

| Parameter | Description |
|---|---|
| `spiNum` | [in] choose SPI and HSPI. |
| `pinCs` | [in] pin to select. |

**Return value**

Null

> 📖 ***Note:***
>
> *CS Pin can only be changed after transmission ends.*

#### 4.5.2.11.SPIIntCfg

**Description**

Set interrupt source and terminal callback function.

**Function**

```
void SPIIntCfg(SpiNum spiNum, SpiIntInfo *pIntInfo)
```

| Parameter | Description |
|---|---|
| `spiNum` | [in] choose SPI and HSPI. |
| `pIntInfo` | [in] a pointer to SpiIntInfo with interrupt source and interrupt callback function. |

**Return value**

Null

#### 4.5.2.12.SPIIntEnable

**Description**

Set the available interrupt source.

**Function**

```
void SPIIntEnable(SpiNum spiNum, SpiIntSrc intSrc);
```

| Parameter | Description |
|---|---|
| `spiNum` | [in] choose SPI and HSPI. |
| `intSrc` | [in] interrupt to set, please refer to ***Section 4.5.1.1 SpiIntSrc***. |

#### Return value

Null

#### 4.5.2.13.SPIIntDisable

#### Description

Set disable interrupt source.

#### Function

```
void SPIIntDisable(SpiNum spiNum, SpiIntSrc intSrc);
```

| Parameter | Description |
|---|---|
| `spiNum` | [in] choose SPI and HSPI. |
| `intSrc` | [in] interrupt to set, please refer to ***Section 4.5.1.1 SpiIntSrc***. |

#### Return value

Null

#### 4.5.2.14.SPIIntClear

#### Description

Clear all interrupt sources.

#### Function

```
void SPIIntClear(SpiNum spiNum);
```

| Parameter | Description |
|---|---|
| `spiNum` | [in] choose SPI and HSPI. |

#### Return value

Null

### 4.5.3. SPI_Test Demo

The communication format is CMD + ADDR + Data when ESP8266 is in slave mode. The transmission only with DATA is not supported currently. As the slave, ESP8266 can respond to different commands. The CMD default values are as follows:

- CMD = 2, write data to the ESP8266 data register W0 ~ W15;
- CMD = 3, read data from the ESP8266 data register；
- CMD = 1, write data to the ESP8266 status register;
- CMD = 4, read data from the ESP8266 status register.

Spi_test demo is based on the SPI communication between two ESP8266. The communication test followed the steps below.

1. Master sends 32-byte data to slave.
2. Master receive data from slave.

3. Master read data from the status register of the slave.
4. Master writes data to the status register of the slave.

The slave will receive interrupts in order from SPI_SLV_WR_BUF_DONE, SPI_SLV_RD_BUF_DONE, SPI_SLV_RD_STA_DONE, SPI_SLV_WR_STA_DONE.

#### 4.5.3.1. Hardware Connection

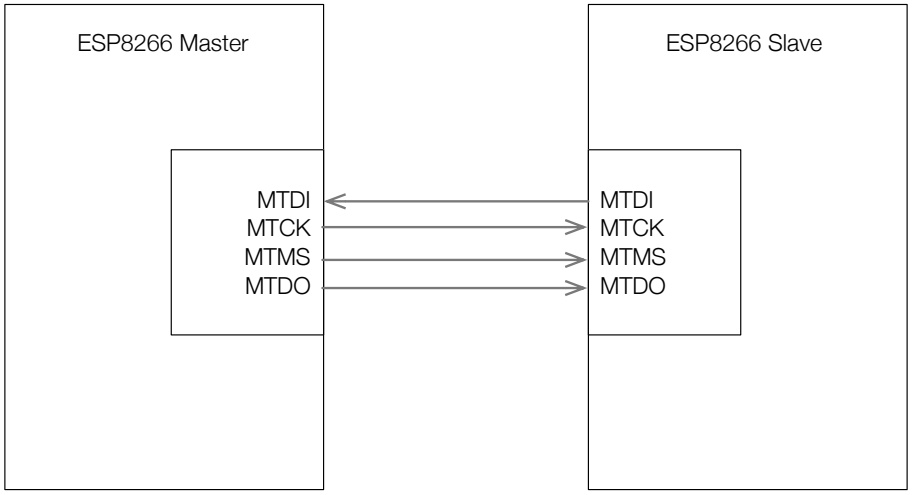



**Figure 4-3. Test Demo Hardware Connection**

Figure 4-3 shows the test demo hardware connection. The master and the slave are connected via HSPI. MTCK pin is SPI. MOSI, MTDI pin is SPI MISO, MTMS pin is SPI Clock and MTMO pin is SPI CS pin.

#### 4.5.3.2. Program Introduction

**spi_master_test**

Master uses SPI buffer starting from W0.

```
void ICACHE_FLASH_ATTR spi_master_test()
{
    SpiAttr hSpiAttr;
    hSpiAttr.bitOrder = SpiBitOrder_MSBFirst;
    hSpiAttr.speed = SpiSpeed_10MHz;
    hSpiAttr.mode = SpiMode_Master;
    hSpiAttr.subMode = SpiSubMode_0;

    // Init HSPI GPIO
    WRITE_PERI_REG(PERIPHS_IO_MUX, 0x105);
    PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U, 2);//configure io to spi
mode
```

```
    PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTCK_U, 2);//configure io to spi
mode
    PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTMS_U, 2);//configure io to spi
mode
    PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDO_U, 2);//configure io to spi
mode


    SPIInit(SpiNum_HSPI, &hSpiAttr);
    uint32_t value = 0xD3D4D5D6;
    uint32_t sendData[8] ={ 0 };
    SpiData spiData;

    os_printf("\r\n =============   spi init master   =============
\r\n");

//  Test 8266 slave.Communication format: 1byte command + 1bytes
address + x bytes Data.
    os_printf("\r\n Master send 32 bytes data to slave(8266)\r\n");
    os_memset(sendData, 0, sizeof(sendData));
    sendData[0] = 0x55565758;
    sendData[1] = 0x595a5b5c;
    sendData[2] = 0x5d5e5f60;
    sendData[3] = 0x61626364;
    sendData[4] = 0x65666768;
    sendData[5] = 0x696a6b6c;
    sendData[6] = 0x6d6e6f70;
    sendData[7] = 0x71727374;
    spiData.cmd = MASTER_WRITE_DATA_TO_SLAVE_CMD;
    spiData.cmdLen = 1;
    spiData.addr = &value;
    spiData.addrLen = 4;
    spiData.data = sendData;
    spiData.dataLen = 32;
    SPIMasterSendData(SpiNum_HSPI, &spiData);
```

```
    os_printf("\r\n Master receive 24 bytes data from
slave(8266)\r\n");
    spiData.cmd = MASTER_READ_DATA_FROM_SLAVE_CMD;
    spiData.cmdLen = 1;
    spiData.addr = &value;
    spiData.addrLen = 4;
    spiData.data = sendData;
    spiData.dataLen = 24;
    os_memset(sendData, 0, sizeof(sendData));
    SPIMasterRecvData(SpiNum_HSPI, &spiData);
    os_printf(" Recv Slave data0[0x%08x]\r\n", sendData[0]);
    os_printf(" Recv Slave data1[0x%08x]\r\n", sendData[1]);
    os_printf(" Recv Slave data2[0x%08x]\r\n", sendData[2]);
    os_printf(" Recv Slave data3[0x%08x]\r\n", sendData[3]);
    os_printf(" Recv Slave data4[0x%08x]\r\n", sendData[4]);
    os_printf(" Recv Slave data5[0x%08x]\r\n", sendData[5]);

    // read the value of slave status register
    value = SPIMasterRecvStatus(SpiNum_HSPI);
    os_printf("\r\n Master read slave(8266) status[0x%02x]\r\n",
value);
    // write 0x99 into the slave status register
    SPIMasterSendStatus(SpiNum_HSPI, 0x99);
    os_printf("\r\n Master write status[0x99] to slavue(8266).\r\n");
    SHOWSPIREG(SpiNum_HSPI);



//  Test others slave.Communication format:0bytes command + 0 bytes
address + x bytes Data
#if 0
    os_printf("\r\n Master send 4 bytes data to slave\r\n");
    os_memset(sendData, 0, sizeof(sendData));
    sendData[0] = 0x2D3E4F50;
    spiData.cmd = MASTER_WRITE_DATA_TO_SLAVE_CMD;
    spiData.cmdLen = 0;
```

```
        spiData.addr = &addr;
        spiData.addrLen = 0;
        spiData.data = sendData;
        spiData.dataLen = 4;
        SPIMasterSendData(SpiNum_HSPI, &spiData);

        os_printf("\r\n Master receive 4 bytes data from slaver\n");
        spiData.cmd = MASTER_READ_DATA_FROM_SLAVE_CMD;
        spiData.cmdLen = 0;
        spiData.addr = &addr;
        spiData.addrLen = 0;
        spiData.data = sendData;
        spiData.dataLen = 4;
        os_memset(sendData, 0, sizeof(sendData));
        SPIMasterRecvData(SpiNum_HSPI, &spiData);
        os_printf(" Recv Slave data[0x%08x]\r\n", sendData[0]);
    #endif

    }
```

### spi_slave_test

The SPI buffer used by the slave starts from W8. The program configures SPI mode first and initializes GPIO. Then it receives the data from the master and uploads the data to SPI buffer, waiting for the master to read. Finally, the program will modify the value of the status register.

```
void ICACHE_FLASH_ATTR spi_slave_test()
{
    // SPI initialization configuration, speed = 0 in slave mode
    SpiAttr hSpiAttr;
    hSpiAttr.bitOrder = SpiBitOrder_MSBFirst;
    hSpiAttr.speed = 0;
    hSpiAttr.mode = SpiMode_Slave;
    hSpiAttr.subMode = SpiSubMode_0;

    // Init HSPI GPIO
    WRITE_PERI_REG(PERIPHS_IO_MUX, 0x105);
```

```
    PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U, 2);//configure io to spi
mode
    PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTCK_U, 2);//configure io to spi
mode
    PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTMS_U, 2);//configure io to spi
mode
    PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDO_U, 2);//configure io to spi
mode

    os_printf("\r\n ============= spi init slave =============\r\n");
    SPIInit(SpiNum_HSPI, &hSpiAttr);

    // Set spi interrupt information.
    SpiIntInfo spiInt;
    spiInt.src = (SpiIntSrc_TransDone
        | SpiIntSrc_WrStaDone
        |SpiIntSrc_RdStaDone
        |SpiIntSrc_WrBufDone
        |SpiIntSrc_RdBufDone);
    spiInt.isrFunc = spi_slave_isr_sta;
    SPIIntCfg(SpiNum_HSPI, &spiInt);
    // SHOWSPIREG(SpiNum_HSPI);

    SPISlaveRecvData(SpiNum_HSPI);
    uint32_t sndData[8] = { 0 };
    sndData[0] = 0x35343332;
    sndData[1] = 0x39383736;
    sndData[2] = 0x3d3c3b3a;
    sndData[3] = 0x11103f3e;
    sndData[4] = 0x15141312;
    sndData[5] = 0x19181716;
    sndData[6] = 0x1d1c1b1a;
    sndData[7] = 0x21201f1e;
// write 8 word (32 byte) data to SPI buffer W8~W15
SPISlaveSendData(SpiNum_HSPI, sndData, 8);
// set the value of status register
```

```
    WRITE_PERI_REG(SPI_RD_STATUS(SpiNum_HSPI), 0x8A);
    WRITE_PERI_REG(SPI_WR_STATUS(SpiNum_HSPI), 0x83);
}
```

**spi_slave_isr_sta**

```
// SPI interrupt callback function.
void spi_slave_isr_sta(void *para)
{
    uint32 regvalue;
    uint32 statusW, statusR, counter;
    if (READ_PERI_REG(0x3ff00020)&BIT4) {
        //following 3 lines is to clear isr signal
        CLEAR_PERI_REG_MASK(SPI_SLAVE(SpiNum_SPI), 0x3ff);
    } else if (READ_PERI_REG(0x3ff00020)&BIT7) { //bit7 is for hspi
isr,
        regvalue = READ_PERI_REG(SPI_SLAVE(SpiNum_HSPI));
        os_printf("spi_slave_isr_sta SPI_SLAVE[0x%08x]\n\r",
regvalue);
        SPIIntClear(SpiNum_HSPI);
        SET_PERI_REG_MASK(SPI_SLAVE(SpiNum_HSPI), SPI_SYNC_RESET);
        SPIIntClear(SpiNum_HSPI);

        SPIIntEnable(SpiNum_HSPI, SpiIntSrc_WrStaDone
                 | SpiIntSrc_RdStaDone
                 | SpiIntSrc_WrBufDone
                 | SpiIntSrc_RdBufDone);

        if (regvalue & SPI_SLV_WR_BUF_DONE) {
            // User can get data from the W0~W7
            os_printf("spi_slave_isr_sta : SPI_SLV_WR_BUF_DONE\n\r");
        } else if (regvalue & SPI_SLV_RD_BUF_DONE) {
            // TO DO
            os_printf("spi_slave_isr_sta : SPI_SLV_RD_BUF_DONE\n\r");
        }
        if (regvalue & SPI_SLV_RD_STA_DONE) {
            statusR = READ_PERI_REG(SPI_RD_STATUS(SpiNum_HSPI));
```

```
                    statusW = READ_PERI_REG(SPI_WR_STATUS(SpiNum_HSPI));
                    os_printf("spi_slave_isr_sta :
SPI_SLV_RD_STA_DONE[R=0x%08x,W=0x%08x]\n\r", statusR,
                              statusW);
            }

            if (regvalue & SPI_SLV_WR_STA_DONE) {
                    statusR = READ_PERI_REG(SPI_RD_STATUS(SpiNum_HSPI));
                    statusW = READ_PERI_REG(SPI_WR_STATUS(SpiNum_HSPI));
                    os_printf("spi_slave_isr_sta :
SPI_SLV_WR_STA_DONE[R=0x%08x,W=0x%08x]\n\r", statusR, tatusW);
            }
            if ((regvalue & SPI_TRANS_DONE) && ((regvalue & 0xf) == 0)) {
                    os_printf("spi_slave_isr_sta : SPI_TRANS_DONE\n\r");

            }
            SHOWSPIREG(SpiNum_HSPI);
        }
}
```

#### 4.5.3.3. Running Log and Waveform Graphs

**ESP8266 Master**

Master log is as shown in Figure 4-4.

```
=====================================================
        ESP8266 spi_interface_test application
                SDK version:1.5.3(827143cc)
                Complie time:17:13:39

=====================================================

=============   spi init master   =============

Master send 32 bytes data to slave(8266)

Master receive 24 bytes data from slave(8266)
Recv Slave data0[0x38373635]
Recv Slave data1[0x3c3b3a39]
Recv Slave data2[0x103f3e3d]
Recv Slave data3[0x14131211]
Recv Slave data4[0x18171615]
Recv Slave data5[0x1c1b1a19]

Master read slave(8266) status[0x83]

Master write status[0x99] to slavue(8266).

FUNC[spi_master_test],line[176]
SPI_ADDR       [0xd3d4d5d6]
SPI_CMD        [0x00001001]
SPI_CTRL       [0x0028a737]
SPI_CTRL2      [0x00040011]
SPI_CLOCK      [0x000070c7]
SPI_RD_STATUS  [0x00000000]
SPI_WR_STATUS  [0x00000000]
SPI_USER       [0x88000070]
SPI_USER1      [0x7c0e0700]
SPI_USER2      [0x70000001]
SPI_PIN        [0x0000001e]
SPI_SLAVE      [0x02000210]
SPI_SLAVE1     [0x02000000]
SPI_SLAVE2     [0x00000000]
ADDR[0x60000140],Value[0x00000099]
ADDR[0x60000144],Value[0x3c3b3a39]
ADDR[0x60000148],Value[0x103f3e3d]
ADDR[0x6000014c],Value[0x14131211]
ADDR[0x60000150],Value[0x18171615]
ADDR[0x60000154],Value[0x1c1b1a19]
```

Figure 4-4. Master Log

In Figure 4-5, the yellow area is the command 0x02 which means the master writes data to the slave, the red area is the address register 0x00, and the green area is the data written, with the low byte being transmitted first.

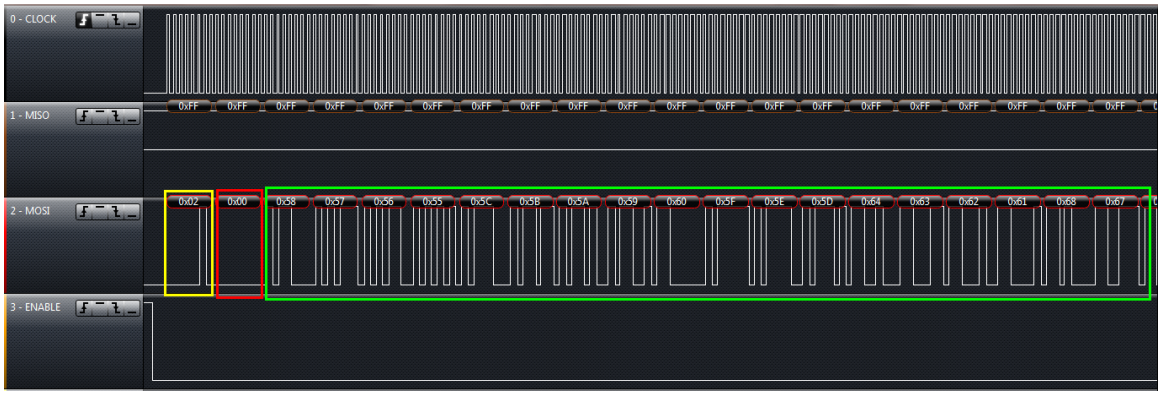



Figure 4-5. Waveform Graph 1

In Figure 4-6, the yellow area is the command 0x03 which means the master reads data from the slave, the red area is the address register 0x00, and the green MISO area is the data in SPI buffer.

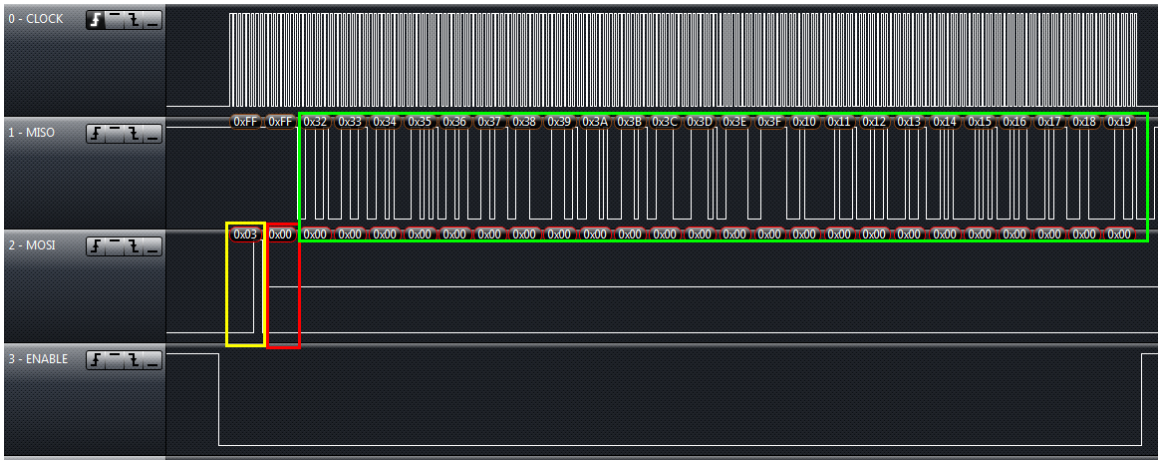



Figure 4-6. Waveform Graph 2

In Figure 4-7, the yellow area is the command 0x04 which means the master reads data from the slave, the red area is the address register 0x00, and the green MISO area is the value of the slave's status register.

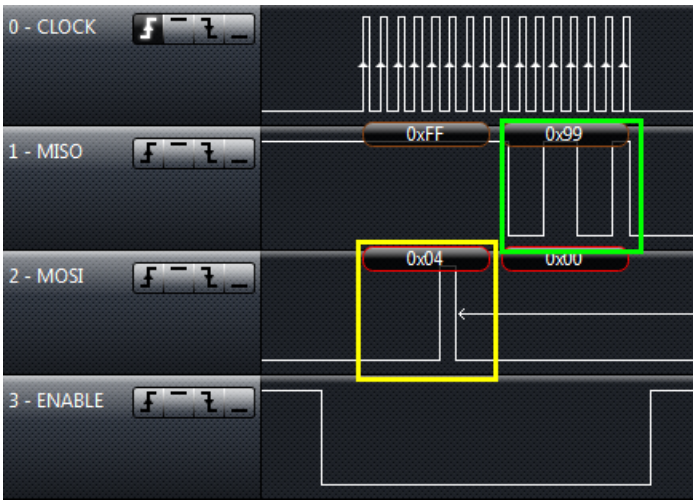



Figure 4-7. Waveform Graph 3

In Figure 4-8, the yellow area is the command 0x01 which means the master writes to the slave's status register, the purple area is the value written to the slave's status register.

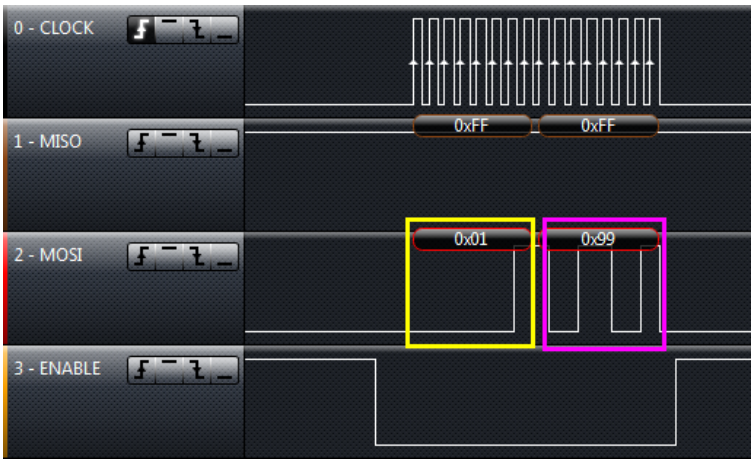



Figure 4-8. Waveform Graph 4

### ESP8266 Slave

Slave log is as shown in Figure 4-9.

```
 ============= spi init slave =============
mode : softAP(1a:fe:34:a1:32:d7)
add if1
dhcp server start:(ip:192.168.4.1,mask:255.255.255.0,gw:192.168.4.1)
bcn 100
spi_slave_isr_sta SPI_SLAVE[0x47f401f2]
spi_slave_isr_sta : SPI_SLV_WR_BUF_DONE

 FUNC[spi_slave_isr_sta],line[108]
 SPI_ADDR       [0xd3000000]
 SPI_CMD        [0x00049002]
 SPI_CTRL       [0x0028a000]
 SPI_CTRL2      [0x00800011]
 SPI_CLOCK      [0x00000000]
 SPI_RD_STATUS  [0x0000008a]
 SPI_WR_STATUS  [0x00000083]
 SPI_USER       [0xd1000040]
 SPI_USER1      [0x1dfeff00]
 SPI_USER2      [0x70000004]
 SPI_PIN        [0x0008001e]
 SPI_SLAVE      [0x45f201fd]
 SPI_SLAVE1     [0x3aff1c70]
 SPI_SLAVE2     [0x00000000]
 ADDR[0x60000140],Value[0x58d6d5d4]
 ADDR[0x60000144],Value[0x5c555657]
 ADDR[0x60000148],Value[0x60595a5b]
 ADDR[0x6000014c],Value[0x645d5e5f]
 ADDR[0x60000150],Value[0x68616263]
 ADDR[0x60000154],Value[0x6c656667]
 ADDR[0x60000158],Value[0x70696a6b]
 ADDR[0x6000015c],Value[0x746d6e6f]
 ADDR[0x60000160],Value[0x35343332]
 ADDR[0x60000164],Value[0x39383736]
 ADDR[0x60000168],Value[0x3d3c3b3a]
 ADDR[0x6000016c],Value[0x11103f3e]
 ADDR[0x60000170],Value[0x15141312]
 ADDR[0x60000174],Value[0x19181716]
 ADDR[0x60000178],Value[0x1d1c1b1a]
 ADDR[0x6000017c],Value[0x21201f1e]
spi_slave_isr_sta SPI_SLAVE[0x45f201fd]
spi_slave_isr_sta : SPI_SLV_RD_BUF_DONE
spi_slave_isr_sta : SPI_SLV_RD_STA_DONE[R=0x0000008a,W=0x00000099]
spi_slave_isr_sta : SPI_SLV_WR_STA_DONE[R=0x0000008a,W=0x00000099]
```

Figure 4-9. Slave Log

# 5. SPI Overlap & Display Application Guide

## 5.1. Functional Overview

The Overlap mode of ESP8266 Host SPI allows for two SPI modes (SPI and HSPI) to reuse the same IO interface (such as SCLK, MOSI and MISO) for the operation of multiple slave SPI devices. The hardware supports 3 line chip selection. If there are additional 3 slave devices, GPIO can be adopted as CS signal for the communication of multiple slave device.

Generally speaking, in order to ensure that the CPU can be running at high efficiency, SPI module is used to read the running program from external Flash to CPU CACHE, while HSPI module is used to operate slave devices of other users. Under Overlap mode, the hardware will automatically arbitrate the control of two SPI modules to the current pin signal for time-sharing application. If the software starts HSPI communication, the arbitration signal will delay the start of HSPI block communication via the working of SPI. The arbitration signal is then allowed to start the communication of HSPI IO interface after SPI finishes reading the program codes for communication. This is illustrated in Figure 1. For user software, only a switch of the corresponding CS signal before the start of communicator is needed. Other operations are of no difference to the use of single HSPI communication.

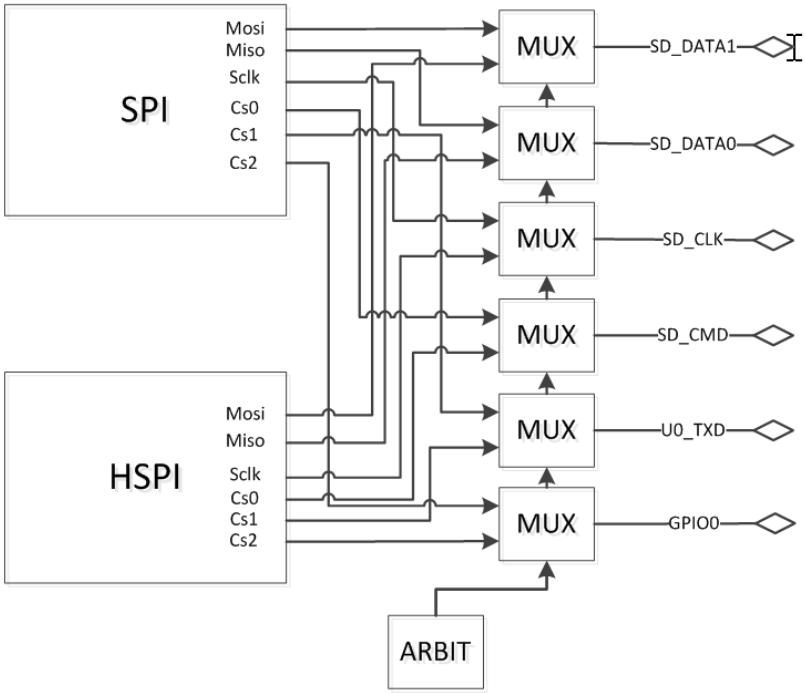



Figure 5-1. SPI Overlap Block Diagram

Please refer to ***Chapter 4 EPS8266 SPI Communication User Guide*** for more information about the application method of Host SPI Module. The configuration method of Overlap mode is discussed in detail below.

## 5.2. Hardware Connection of SPI Overlap Mode

Pins including SD_CLK, SD_DATA0, and SD_DATA1 correspond to pins SCLK, MISO and MOSI in two SPI modes, while pins SD_CMD, U0TXD, and GPIO0 correspond to chip selection (CS) signals CS0, CS1, and CS2 respectively. Generally, SD_CMD connects to the CS signal of an external Flash, while U0TXD and GPIO0 can be connect with the CS signals of two slave devices. It can connect to the CS signal of two salve devices. Besides, HSPI can read and write Flash data through enabled CS0, independent of SPI (e.g. Read some pre-stored user data).

If more SPI devices are needed, device can be selected via other GPIOs, while CS0, CS1, and CS2 are blocked by the configuration register.

## 5.3. API Description of SPI Overlap Mode

1. `void hapi_overlap_init(void)`

Function:

After SPI Overlap mode has been initialized, and SPI and HSPI interfaces are invoked, interfaces including CLK, MOSI, and MISO can be shared with SPI and HSPI interfaces to communicate with different devices. By default, CS2 is the CS signal of HSPI interface. Please be careful when switching CS signals during communication.

Location:

***\app\user\user_main.c*** in the DEMO.

2. `SELECT_OLED(),SELECT_TFT()`

Function:

Switch the CS pin of HSPI and OLED in DEMO connects to CS2. TFTLCD connects to CS1. Before the start of HSPI communication, macro needs to be called. The macro definition is as follows:

```
#define SELECT_OLED()  CLEAR_PERI_REG_MASK(SPI_PIN(HSPI),
SPI_CS2_DIS);\
SET_PERI_REG_MASK(SPI_PIN(HSPI), SPI_CS0_DIS |SPI_CS1_DIS)
#define SELECT_TFT()   CLEAR_PERI_REG_MASK(SPI_PIN(HSPI),
SPI_CS1_DIS);\
SET_PERI_REG_MASK(SPI_PIN(HSPI), SPI_CS0_DIS |SPI_CS2_DIS)
```

Therefore, users can change the macro definition. For example, the following macro can be defined if HSPI is used to the operate Flash:

```
#define SELECT_FLASH()   CLEAR_PERI_REG_MASK(SPI_PIN(HSPI),
SPI_CS0_DIS);\
```

```
SET_PERI_REG_MASK(SPI_PIN(HSPI), SPI_CS1_DIS |SPI_CS2_DIS)
```

If normal GPIO is used for CS, the following is needed:

```
#define DISABLE_CS()\
SET_PERI_REG_MASK(SPI_PIN(HSPI), SPI_CS0_DIS |SPI_CS1_DIS |
SPI_CS2_DIS)
```

Location:

***\app\include\user_lcd.h*** in the DEMO.

Please refer to ***Chapter 4 EPS8266 SPI Communication User Guide*** for more information about other host SPI communication.

## 5.4. Display Screen Console Program DEMO

The DEMO is used to print simple strings on display screens, including LCD for parameter display and debug printing. DEMO driver supports two screens currently, i.e. 3.5-inch TM035PDZV36 480*320 TFT colored LCD and Zhong JY. Tech 1.3-inch 128*64OLED. The driver programs can communicate with the display screen via ESP8266 HSPI interface under Overlap mode.

Under SPI Overlap mode, the two screens and 8266 external program flash chip share SCLK, MOSI and MISO signals on the SPI bus. Different CS signals are used in different device.

### 5.4.1. Connection Description

**Zhong JY. Tech 1.3-inch OLED Connection**

The signals in OLED, i.e. SCLK, MOSI, CS, DC, RESET connects to the pins in 8266, i.e. SD_CLK, SD_DATA1, GPIO0, MTCK, GPIO5 respectively. The VCC in OLED and GND connects to 3.3V network and GND on DEMO board.

**Tian Ma 3.5-inch TFT LCD**

The signals in TFT, i.e. SCLK, MOSI, CS, RESET connects to the pins in 8266, i.e. SD_CLK, SD_DATA1, U0TXD, GPIO5 respectively. The VCC in OLED and GND connects to 3.3V network and GND on DEMO board.

### 5.4.2. API Function Description

1. `void screen_init(void)`

Function:

Display screen initialization program. Call the function after it is enabled.

Location:

***\app\user\user_lcd.c*** and ***\app\include\user_lcd.h***

2. `void scr_param_config(uint8 bkg_color,uint8 ft_color,uint8 ft_size, uint8 scr_size_clr_row, uint8 scr_size_x,uint8 scr_size_y)`

Function:

Display parameter for the global variable configuration string of the `scr_font_param` structure.

Parameters:

| Parameter | Description |
|---|---|
| uint8 bkg_color | Background color of TFT can change between BLACK_8COLOR and WHITE_8COLOR. Do not use OLED display screen. |
| uint8 ft_color | Font color of TFT can change between BLACK_8COLOR and WHITE_8COLOR. Do not use OLED display screen. |
| uint8 ft_size | Font size with 12*6 ASCII character. The parameter is the multiple of pixels under the character.<br>For example, if ft_size is 2, the actual font size is 24*12. Input non-zero value. |
| uint8 scr_size_clr_row | Rows should be removed after the screen is refreshed.Input non-zero value. |
| uint8 scr_size_x | Each line shows the character number.Please note that it should not exceed the pixel range of the screen. |
| uint8 scr_size_y | This parameter shows the character lines. Please note that it should not exceed the pixel range of the screen. |

Location:

***\app\user\user_lcd.c*** and ***\app\include\user_lcd.h***, call in the function `screen_init`.

3. `void scr_printf(const char* fmt, ...)`

Function:

used for standard printing of functions displayed on the screen, similar to the using method of printf in C programming language.

Parameters:

- `const char* fmt` —— shows the character string.
- ***...*** —— variable parameters that needs to be displayed in the corresponding string.

Location:

***\app\user\user_lcd.c*** and ***\app\include\user_lcd.h***

4. `void at_lcd_print(uint8* str)`

Function:

shows the assigned character string displayed on the screen order.

Parameters:

`uint8* str` —— the starting address of string array.

### 5.4.3. Pre-compiled Macro Setting

```
#define OLED_SCR        1
#define TFT_SCR         1
#define OVERLAP_TEST    0
```

Location:

***\app\include\user_lcd.h***

`OLED_SCR` and `TFT_SCR` can control the debugging characters displayed on the corresponding screen. The program supports the  same character shown in two screens. Overlap_TEST is used for SPI Overlap test when TFT is used to display image. TFT should be set at 0 as it conflicts with the displayed characters.

# 6. SPI Wi-Fi Passthrough 1-Interrupt Mode

## 6.1. Functional Overview

This protocol uses the ESP8266 slave mode to communicate with other processor's SPI master. Signal line No.5 is used to implement this protocol. Apart from signal line No.4 needed for standard SPI, signal line No.1 is also needed to inform the master of the update of the slave status register.

## 6.2. ESP8266 SPI Slave Protocol Format

### 6.2.1. SPI Slave Clock Polarity Configuration

Clock polarity of the master clock which communicates with the ESP8266 SPI slave should be set to be low in the idle state, sampling for rising edge, and changing data for falling edge. When it reads/writes 34 bytes at a time, or when it reads 2 bytes at a time to get information of the slave status register, selection signal CS must be kept at low level. If CS is pulled high when data is being sent, the slave interior status will be reset.

### 6.2.2. Communication Format Supported by The SPI Slave

The ESP8266 SPI slave communication format should be command+address+read/write data or command+slave status value. To be specific:

- Command: length, 8 bits; master output slave input (MOSI).

0x02 is the data sent by the master and received by the slave. The master writes 32 bytes of data through MOSI into SPI_W7 in corresponding register SPI_W0 of the slave data buffer.

0x03 is the data received by the master and sent by the slave. 32 bytes of data from corresponding register of the slave buffer between SPI_FLASH_C8 and SPI_FLASH_C15 are sent to the master through MISO.

0x04 and 0x05 can read the lower 8 bits of SPI_FLASH_STATUS in the slave status register.

> ⚠ ***Notice:***
>
> *Other values are used to read/write the SPI slave status register SPI_FLASH_STATUS. Their communication formats are different from those of the read/write buffer, using them will cause read/write errors for the slave. So users should not use these values.*

- address: length, 8 bits; master output slave input (MOSI). The address content must be 0.

- read/write data: length, 256 bits (32 Bytes). Master output slave input (MOSI) the 0x02 command, or master input slave output (MISO) the 0x03 command.
- slave status: length, 8 bits; master input slave output (MISO), use 0x04 or 0x05 to read the slave communication status.

## 6.3. Slave Status Definition and Line Breakage

### 6.3.1. Status Definition

The slave status contains 8 bits:

- `wr_busy`, bit0: 1, slave write buffer is full, and is processing the data received; 0, slave write buffer is empty, new data can be written in.
- `rd_empty`, bit1: 1, slave read buffer is empty, no data has been updated; 0, there is new data in the buffer for the master to read.
- `comm_cnt`, bit2-4: count value of the read/write communication. Each time when the slave SPI read/write buffer is interrupted, this 3-bit count value will increase by 1. Therefore, the master can judge whether the readwrite communication has been recognised by the slave, and whether the communication is completed.

> ⚠ ***Notice:***
> *When the master completed a read/write communication, if it wants to conduct the next read operation, rd_empty must be 0, and comm_cnt value must be the previous value +1; if it wants to conduct the next write operation, wr_busy must be 0, and comm_cnt value must be the previous value +1.*

### 6.3.2. GPIO0 Line Breakage

When there are changes in the slave status register, interrupt line GPIO0 will be set to be 1; when the master uses 0x04, 0x05 to read the slave status register, interrupt line GPIO0 will be set 0.

## 6.4. ESP8266 SPI Slave API Functions

> ⚠ ***Notice:***
> *Configure in **spi.h** if SPI status register single-threaded passthrough protocol is used.*

```
//SPI protocol selection
#define TWO_INTR_LINE_PROTOCOL      0
#define ONE_INTR_LINE_31BYTES       0
#define ONE_INTR_LINE_WITH_STATUS   1
```

The interrupt response function will use `spi_slave_isr_sta(void *para)`.

1. `void spi_slave_init(uint8 spi_no)`

Function:

Initialise the SPI slave mode, set the IO interface to SPI mode, start the SPI transmission interrupt, and register spi_slave_isr_handler. The communication format is set to be 8 bits command + 8 bits address + 256 bits (32 Bytes) read/write data.

Parameters:

`spi_no`: number of the SPI module. The ESP8266 processor has two SPI modules (SPI and HSPI) with the same functions.

Value to be selected: SPI or HSPI.

2. `spi_slave_isr_sta(void *para)`

Function and trigger condition:

It's the SPI interrupt handler function. When the master successfully reads data from or writes data into the slave, the interrupt will be triggered. Users can revise the interrupt service routine in order to attain the communication functions they need. The code is shown as below:

```
struct spi_slave_status_element
{
      uint8 wr_busy:1;
      uint8 rd_empty :1;
      uint8 comm_cnt :3;
      uint8 res :3;
};

union spi_slave_status
{
      struct spi_slave_status_element elm_value;
      uint8 byte_value;
};
void spi_slave_isr_sta(void *para)
{
      uint32 regvalue,calvalue;
      uint32 recv_data,send_data;
      union spi_slave_status spi_sta;

      if(READ_PERI_REG(0x3ff00020)&BIT4){
        //following 3 lines is to clear isr signal
```

```
                CLEAR_PERI_REG_MASK(SPI_SLAVE(SPI), 0x3ff);
        }else if(READ_PERI_REG(0x3ff00020)&BIT7){ //bit7 is for hspi
isr,
                // record the interrupt status
                regvalue=READ_PERI_REG(SPI_SLAVE(HSPI));
                //***********interrupt handler flag, end this
passthrough************//
                CLEAR_PERI_REG_MASK(SPI_SLAVE(HSPI),
                                                SPI_TRANS_DONE_EN|

SPI_SLV_WR_STA_DONE_EN|

SPI_SLV_RD_STA_DONE_EN|

SPI_SLV_WR_BUF_DONE_EN|

SPI_SLV_RD_BUF_DONE_EN);
                SET_PERI_REG_MASK(SPI_SLAVE(HSPI), SPI_SYNC_RESET);
                CLEAR_PERI_REG_MASK(SPI_SLAVE(HSPI),
                                                SPI_TRANS_DONE|
                                                SPI_SLV_WR_STA_DONE|
                                                SPI_SLV_RD_STA_DONE|
                                                SPI_SLV_WR_BUF_DONE|
                                                SPI_SLV_RD_BUF_DONE);
                SET_PERI_REG_MASK(SPI_SLAVE(HSPI),
                                                SPI_TRANS_DONE_EN|

SPI_SLV_WR_STA_DONE_EN|

SPI_SLV_RD_STA_DONE_EN|

SPI_SLV_WR_BUF_DONE_EN|

SPI_SLV_RD_BUF_DONE_EN);
                //
*******************************************************//
                /****************master writes interrupt
handler***************/
```

```
                if(regvalue&SPI_SLV_WR_BUF_DONE){
                //*****complete the write operation, wr_busy set to be
1, communication count increases by 1****//

spi_sta.byte_value=READ_PERI_REG(SPI_STATUS(HSPI))&0xff;
                    spi_sta.elm_value.wr_busy=1;
                    spi_sta.elm_value.comm_cnt++;
                    WRITE_PERI_REG(SPI_STATUS(HSPI),
(uint32)spi_sta.byte_value);
                //*********************************************//
                    //*******move the data received by the register
into the memory******//
                          idx=0;
                          while(idx<8){
                                recv_data=READ_PERI_REG(SPI_W0(HSPI)+
(idx<<2));
                                //os_printf("rcv data : 0x%x
\n\r",recv_data);
                                spi_data[idx<<2] = recv_data&0xff;
                                spi_data[(idx<<2)+1] =
(recv_data>>8)&0xff;
                                spi_data[(idx<<2)+2] =
(recv_data>>16)&0xff;
                                spi_data[(idx<<2)+3] =
(recv_data>>24)&0xff;
                                idx++;
                    }
                    //*********************************//
                    //************data transmission completed, wr_busy
set to be 0*************//

spi_sta.byte_value=READ_PERI_REG(SPI_STATUS(HSPI))&0xff;
                    spi_sta.elm_value.wr_busy=0;
                    WRITE_PERI_REG(SPI_STATUS(HSPI),
(uint32)spi_sta.byte_value);
                    //
*********************************************//
                    /***testing part, it can be revised. This part of
the program is used to copy the data read to the read buffer**/
```

```
                    for(idx=0;idx<8;idx++)
                    {
                          WRITE_PERI_REG(SPI_W8(HSPI)+(idx<<2),

READ_PERI_REG(SPI_W0(HSPI)+(idx<<2)));
                    }
                    /
*************************************************************/
                    /***testing part, it can be revised. rd_empty is
set to be 0, the slave can read**/

spi_sta.byte_value=READ_PERI_REG(SPI_STATUS(HSPI))&0xff;
                    spi_sta.elm_value.rd_empty=0;
                    WRITE_PERI_REG(SPI_STATUS(HSPI),
(uint32)spi_sta.byte_value);
                    /*****************************************/
                    GPIO_OUTPUT_SET(0, 1);  // interrupt line set to be
1, inform the master to read the slave status
            /****************master reads the interrupt
handler***************/
            }else if(regvalue&SPI_SLV_RD_BUF_DONE){
                    //*****complete the read operation, rd_empty set to
be 1, communication count increases by 1****//

spi_sta.byte_value=READ_PERI_REG(SPI_STATUS(HSPI))&0xff;
                    spi_sta.elm_value.comm_cnt++;
                    spi_sta.elm_value.rd_empty=1;
                    WRITE_PERI_REG(SPI_STATUS(HSPI),
(uint32)spi_sta.byte_value);


                    GPIO_OUTPUT_SET(0, 1);  // interrupt line set to be
1, inform the master to read the slave status
            }
            /****************master reads status interrupt
handler***************/
            if(regvalue&SPI_SLV_RD_STA_DONE){
                    GPIO_OUTPUT_SET(0,0);   // interrupt line set to be
0, the master has read the status
            }
```

```
                }else if(READ_PERI_REG(0x3ff00020)&BIT9){ //bit7 is for i2s isr,


                }
            }
```

# 7. SPI Wi-Fi Passthrough 2-Interrupt Mode

## 7.1. Functional Overview

This protocol uses the ESP8266 slave mode to communicate with other processor's SPI masters. Signal line No.6 is used to implement this protocol. Apart from signal line No.4 needed for standard SPI, signal line No.2 is also needed to inform the master of information of the slave receive and send buffer status, so as to control the data flow.

## 7.2. ESP8266 SPI Slave Protocol Format

### 7.2.1. SPI Slave Clock Polarity Configuration

Clock polarity of the master clock which communicates with the ESP8266 SPI slave should be set to be low in the idle state, sampling for rising edge, and changing data for falling edge. When it reads/writes 34 Bytes at a time, selection signal CS must be kept at low level. If CS is pulled high when data is being sent, the slave interior status will be reset.

### 7.2.2. Communication Format Supported by The SPI Slave

The ESP8266 SPI slave communication format is similar to that of the master, it should be command + address+ read/write data. To be specific:

- command: length, 8 bits; master output slave input (MOSI).

0x02 is the data sent by the master and received by the slave. The host writes 32 Bytes of data through MOSI into SPI_W0 to SPI_W7 in the corresponding register of the slave data buffer.

0x03 is the data received by the master and sent by the slave. 32 Bytes of data from corresponding register of the slave buffer between SPI_W8 and SPI_W15 are sent to the master through MISO.

> ⚠ ***Note:***
> *other values are used to read/write the SPI slave status register SPI_STATUS. Their communication formats are different from those of the read/write buffer, using them will cause read/write errors for the slave. So users should not use these values.*

- address: length, 8 bits; master output slave input (MOSI). The address content must be 0.
- read/write data: length, 256 bits (32 Bytes). Master output slave input (MOSI) the 0x02 command, or master input slave output (MISO) the 0x03 command.

## 7.3. Instruction on The Data Flow Control Line

The ESP8266 uses 2 GPIOs to output the slave receive buffer status and send buffer status.

### 7.3.1. GPIO0 MOSI Buffer Status

When GPIO0 enters the slave receive interrupt, the interrupt program will resume the SPI slave to communicable status in order to prepare for the next communication. Then, GPIO0 will be written to be low level, data received will be processed, and GPIO0 will be written to be high level to exit the interrupt program. Therefore:

- Between the master enables an SPI write communication to GPIO0 generates a falling edge, if users enable any other SPIs, communication errors will occur.
- When GPIO0 is at low level, if the master enables any SPI to write (0x02 command), SPI_W0 to SPI_W7 in the slave receive register will be covered. But if there is effective data in the slave send register (refer to GPIO2 instructions), when GPIO0 is at low level, master can be started to read (0x03 command) data between SPI_W8 to SPI_W15 in the slave send register.
- If GPIO0 shifts from low level to high level, it means the slave has processed data from SPI_W0 to SPI_W7 in the receive register, and the master can start another write operation (0x02 command).

### 7.3.2. GPIO2 Master Receives The Slave Send Buffer Status

GPIO2 activities are slightly different from those of GPIO0. In the slave send interrupt, the interrupt program will resume the SPI slave to communicable status in order to prepare for the next communication. Then, GPIO0 will be written to be low level, and quit the interrupt program. After that, if data is sent to the ESP8266 through WiFi and is required to be forwarded by SPI, ESP8266 software will be written into SPI_W8 to SPI_W15, and GPIO2 will be set to be high level. Therefore:

- Between the master enables an SPI read communication to GPIO2 generates a falling edge, if users enable any other SPIs, communication errors will occur.
- When GPIO2 is at low level, if the master enables any SPI to read (0x03 command), it can only read data the same as the previous data, or incomplete data. But if data in the slave receive register has been processed (refer to GPIO2 instructions), when GPIO2 is at low level, master can be started to write (0x02 command).
- If GPIO2 shifts from low level to high level, it means the slave has updated data from SPI_W8 to SPI_W15 in the send register, and the master can start the another read operation (0x03 command).

### 7.3.3. Master Communication Logic Implementation

Incomplete C code is used to briefly introduce the communication logic:

```
//wr_rdy: ready to conduct the next SPI write operation
//rd_rdy: ready to conduct the next SPI read operation
unsigned char wr_rdy=1,rd_rdy=0;
void spi_read_func(....)
{
      // before starting the read operation, check if there is new
      data for the slave to read (rd_rdy is non-0);
      // also, check if the previous write operation is completed;
      write operationcompleted and processing data (signal GPIO0 is
      0), or new data can be written into the slave (wr_rdy is non-0)
      if(rd_rdy&&((GPIO0= =0)||wr_rdy)){
      rd_rdy=0;    //rd_rdy set to be 0
      spi_transmit(0x03,0,*read_buff);// start the SPI transmission,
      command 3 + address 0 + 32 bytes of data
      ….
      }
}
void spi_write_func(...)
{
      // before starting the write operation, check if there is new
      data for the slave to receive (rd_rdy is non-0);
      // also, check if the previous read operation is completed;
      completed, no new data to be read (signal GPIO2 is 0), or new
      data to be read (rd_rdy is non-0)
      if(wr_rdy&&((GPIO2= =0)||rd_rdy)){
      wr_rdy=0;    //wr_rdy set to be 0
      spi_transmit(0x02,0,*write_buff);// start the SPI transmission,
      command 2 + address 0 + 32 bytes of data
      ...
      }
}


GPIO0_Raising_Edge_ISR() // rising edge interrupt program connected
to the ESP8266 GPIO0
{
      wr_rdy=1;     // data sent by the master has been processed,
      ready for the next write operation
```

```
}


GPIO2_Raising_Edge_ISR() // rising edge interrupt program connected
to the ESP8266 GPIO2
{
      rd_rdy=1; // the slave updates the send buffer, the master is
      ready to read
```

## 7.4. ESP8266 SPI Slave API Functions

1. `void spi_slave_init(uint8 spi_no)`

Function:

Initialise the SPI slave mode, set the IO interface to SPI mode, start the SPI transmission interrupt, and register spi_slave_isr_handler. The communication format is set to be 8 bits command + 8 bits address + 256 bits (32 Bytes) read/write data.

Parameters:

`spi_no`: number of the SPI module. The ESP8266 processor has two SPI modules (SPI and HSPI) with the same functions.

value to be selected: SPI or HSPI.

2. `spi_slave_isr_handler(void *para)`

Function and trigger condition:

It is the SPI interrupt handler function. When the master successfully reads data from or writes data into the slave, the interrupt will be triggered. Users can revise the interrupt service routine in order to complete the communication. The code is shown below.

Code:

```
      uint32 regvalue;
      static uint32 t1 =0;
      static uint32 t2 =0;
      t1=system_get_time();


      if(READ_PERI_REG(0x3ff00020)&BIT4){      //bit4: SPI interrupt
      CLEAR_PERI_REG_MASK(SPI_SLAVE(SPI), 0x3ff);
      }else if(READ_PERI_REG(0x3ff00020)&BIT7){ //bit7: HSPI
      interrupt,
      regvalue=READ_PERI_REG(SPI_SLAVE(HSPI));   // record the
      interrupt type
            // turn off the SPI interrupt enable
```

```
CLEAR_PERI_REG_MASK(SPI_SLAVE(HSPI),
                                        SPI_TRANS_DONE_EN|
                                        SPI_SLV_WR_STA_DONE_EN|
                                        SPI_SLV_RD_STA_DONE_EN|
                                        SPI_SLV_WR_BUF_DONE_EN|
                                        SPI_SLV_RD_BUF_DONE_EN);
     // resume the SPI slave to communicable status, in order
     to prepare for the next communication
SET_PERI_REG_MASK(SPI_SLAVE(HSPI), SPI_SYNC_RESET);
     // clear the interrupt flag
CLEAR_PERI_REG_MASK(SPI_SLAVE(HSPI),
                                        SPI_TRANS_DONE|
                                        SPI_SLV_WR_STA_DONE|
                                        SPI_SLV_RD_STA_DONE|
                                        SPI_SLV_WR_BUF_DONE|
                                        SPI_SLV_RD_BUF_DONE);
     // turn on the SPI interrupt enable
     SET_PERI_REG_MASK(SPI_SLAVE(HSPI),
                                        SPI_TRANS_DONE_EN|
                                        SPI_SLV_WR_STA_DONE_EN|
                                        SPI_SLV_RD_STA_DONE_EN|
                                        SPI_SLV_WR_BUF_DONE_EN|
                                        SPI_SLV_RD_BUF_DONE_EN);
     //MISO processing program
     if(regvalue&SPI_SLV_WR_BUF_DONE){
     GPIO_OUTPUT_SET(0, 0);   //GPIO0 set to be 0
     idx=0;
     //read the data received
          while(idx<8){
          recv_data=READ_PERI_REG(SPI_W0(HSPI)+4*idx);
          //os_printf("rcv data : 0x%x \n\r",recv_data);
          spi_data[4*idx+0] = recv_data&0xff;
          spi_data[4*idx+1] = (recv_data>>8)&0xff;
          spi_data[4*idx+2] = (recv_data>>16)&0xff;
          spi_data[4*idx+3] = (recv_data>>24)&0xff;
```

```
                    idx++;
                    }
                system_os_post(USER_TASK_PRIO_1,MOSI,0);// send the
                reception completed message
                GPIO_OUTPUT_SET(0, 1);                          //GPIO0
                set to be 1
                SET_PERI_REG_MASK(SPI_SLAVE(HSPI),
                SPI_SLV_WR_BUF_DONE_EN);
                //master reads, slave sends the processing program
            if(regvalue&SPI_SLV_RD_BUF_DONE){
                GPIO_OUTPUT_SET(2, 0);   //GPIO2 set to be 0
            }
        }else if(READ_PERI_REG(0x3ff00020)&BIT9){ //bit7: I2S interrupt


        }
```

# 8. HSPI Host Multi-device API

## 8.1. Functional Overview

ESP8266 encapsulates two SPI (Serial Peripheral Interfaces) bus segments, shortly named SPI and HSPI. SPI bus is especially used to read CPU programming code from the external Flash, while HSPI bus is used for SPI device communication.

When ESP8266 is working as a host, HSPI bus can operate with three user devices, besides, it also supports one external Flash writing operation. User devices are supported through selection with CS lines. To be more specific,

| Mode | Device Name |
|---|---|
| HSPI Default IO | User device 1 |
| SPI OVERLAP and CS1 | User device 2 |
| SPI OVERLAP and CS2 | User device 3 |
| SPI OVERLAP and CS0 | Flash |

In the above-mentioned ways of connection, SPI bus shares the same external Flash with HSPI bus. Apart from the memory occupied by programs and related configurations, the rest Flash memory can all be used for reading and writing of user programs.

⚠ ***Notice:***

- *Operation with devices via HSPI host implemented by software programming is not supported in the API functions.*
- *When downloading user programs, the clock frequency of SPI bus used for reading Flash data should be set at 80 MHz. SPI clock frequency should be specified as 80 MHz at SPI OVERLAP and CS1 mode or SPI OVERLAP and CS2 mode.*

## 8.2. Hardware Connection

Generally speaking, SPI slave devices specify four logic signals: SCLK, MOSI, MISO, and CS.

HSPI bus can operate with three different user devices, the ways of connection are explained below:

| Mode | Pin Name of Host ESP8266 | SPI bus Signal Line |
|---|---|---|
| HSPI Default IO | MTDO | CS |
| | MTCK | MOSI |
| | MTDI | MISO |

| Mode | Pin Name of Host ESP8266 | SPI bus Signal Line |
|---|---|---|
| HSPI Default IO | MTDI | MISO |
| | MTMS | CLK |
| SPI OVERLAP and CS1 | U0TXD | CS1 |
| | SD_CLK | SCLK |
| | SD_DATA0 | MISO |
| | SD_DATA1 | MOSI |
| SPI OVERLAP and CS2 | GPIO0 | CS2 |
| | SD_CLK | SCLK |
| | SD_DATA0 | MISO |
| | SD_DATA1 | MOSI |

📖 ***Note:***

*The pins used when HSPI operates with the Flash in OVERLAP mode is completely the same with that of SPI communication.*

## 8.3. API Description

Names of the connection modes supported by the system are defined by macro definitions in ***\app\include \driver\spi_overlap.h***.

- HSPI_CS_DEV (HSPI default IO)
- SPI_CS1_DEV (SPI OVERLAP and CS1)
- SPI_CS2_DEV (SPI OVERLAP and CS2)

Operation with the Flash is defined as SPI_CS0_FLASH. If HSPI operates with two user devices, the API function is shown as below:

```
void hspi_master_dev_init(uint8 dev_no,uint8 clk_polar,uint8 clk_div)
```

| | |
|---|---|
| Function | This function is used to initialize connection of HSPI host. Altogether four user devices can be operated. If multi devices communicate with the host using SPI communication mode, the function should be called each time when that certain device is operated. |
| Location | Defined in directory ***\app\include\driver\spi_overlap.h***, implemented in directory ***\app\driver \spi_overlap.c***. |

| | |
|---|---|
| Parameters | • `uint8 dev_no`: only HSPI_CS_DEV, SPI_CS1_DEV, SPI_CS2_DEV, and SPI_CS0_FLASH are supported, the corresponding values of these four parameters are 0, 1, 2, and 3 respectively. If the parameter should be other values, ERROR will be printed and the function will be returned.<br>• `uint8 clk_polar`: clock polarity.<br>- If the clock polarity is 0, data are captured on the clock's rising edge, and are propagated on a falling edge.<br>- If the clock polarity is 1, data are captured on the clock's falling edge, and are propagated on a rising edge.<br>- If the clock polarity should be other values, ERROR will be printed and the function will be returned.<br>• `uint8 clk_div`: clock frequency division. 40 MHz is reference frequency, the number of division is clk_div+1. To be more specific, 0 stands for reference frequency, 1 stands for 20 MHz, while 2 stands for 40/3 MHz, and so forth. |

⚠ ***Notice:***

*ONLY when the clock frequency of SPI bus used for reading Flash data is set at 80 MHz. If the device is defined by SPI_CS1_DEV and SPI_CS2_DEV via SPI OVERLAP, the clock frequency of host SPI is unadjustable, and should be 80 MHz.*

```
void hspi_dev_sel(uint8 dev_no)
```

| | |
|---|---|
| Function | Convert and select host communication devices. |
| Location | Defined in directory ***\app\include\driver\spi_overlap.h***, implemented in directory ***\app\driver\spi_overlap.c***. |
| Parameters | `uint8 dev_no`: only HSPI_CS_DEV, SPI_CS1_DEV, SPI_CS2_DEV, and SPI_CS0_FLASH are supported, the corresponding values of these four parameters are 0, 1, 2, and 3 respectively. If the device has not been initialized, ERROR will be printed and the function will be returned. If it the parameter should be other values, ERROR will be printed and the function will be returned. |

# 9. I2C User Guide

## 9.1. Functional Overview

ESP8266EX now has interfaces for I2C master devices, and allows control and reading and writing over other I2C slave devices (e.g. most digital sensors).

All GPIO pins can be configured with open-drain mode, thus easily enabling GPIO interface for I2C data or clock functionalities.

Besides that, the chip has pull-up resistance inside which can help save the pull-up resistance outside.

As an I2C master, ESP8266EX has its waveforms of the SDA and SCL lines simulated from SPIO, where SDA access is behind the positive edge of SCL. SCL high and low levels will maintain 5us and thus I2C clock pulse will be around 100KHz.

## 9.2. I2C master Interface

### 9.2.1. Initialization

`i2c_master_gpio_init`: GPIO hardware initialization.

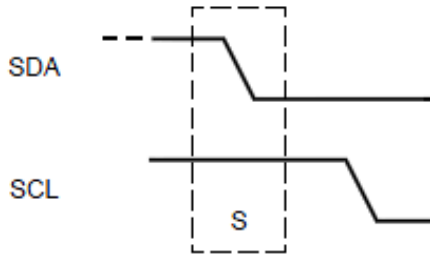
Steps are as follows:

1. Select pin functionality and set as GPIO
2. Set the GPIO into open-drain mode
3. Initialize SDA and SCL as high levels
4. Disconnect GPIO and reset slave state

`i2c_master_init(void)`: Reset slave state

### 9.2.2. Start I2C

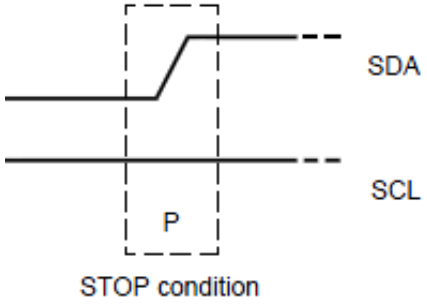
`i2c_master_start(void)`: master generates I2C start conditions.




START condition

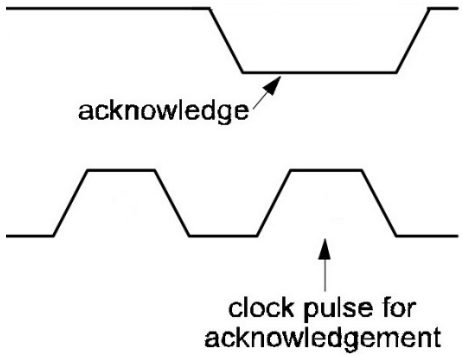
### 9.2.3. Stop I2C

`i2c_master_stop(void)`: master generates I2C stop conditions.



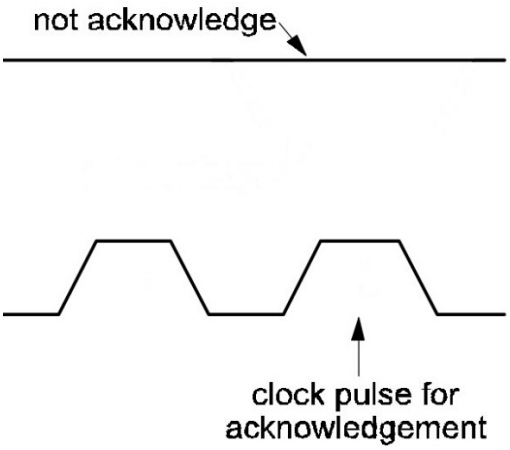

### 9.2.4. I2C Master Responds ACK

`i2c_master_send_ack(void)`: sets I2C master to respond ACK.




### 9.2.5. I2C Master Responds NACK

`I2C_master_send_nack(void)`: sets I2C master to respond NACK.



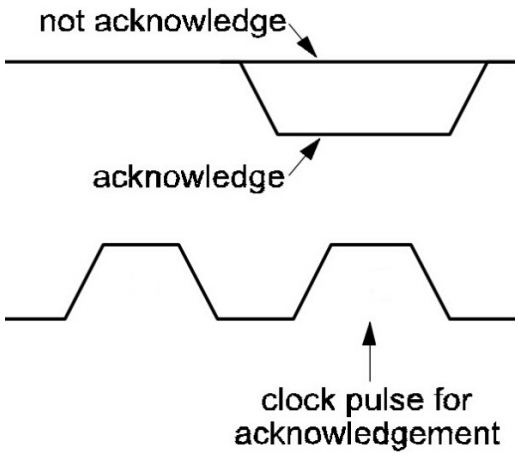
### 9.2.6. Check I2C Slave Response

`bool i2c_master_checkAck(void)`: check slave response state.

Return value:

- TRUE: "acknowledge" from slave
- FALSE: "not acknowledge" from slave

Details shown below:




### 9.2.7. Write Data on I2C Bus

`i2c_master_writeByte(uint8 wrdata)`: write data on I2C bus

Parameters:

1 Byte of data

> 📖 ***Note:***
>
> *Data at the highest place will be sent first and that at the lowest place sent last.*
>
> *Either slave address or data can be sent.*

### 9.2.8. Read Data from I2C Bus

`i2c_master_readByte (void)`: read a byte from SPI slave.

Return value:

Read 1 Byte of data.

## 9.3. Demo

Please refer to IOT_Demo provided by ***esp_iot_sdk***, for example:

```
void ICACHE_FLASH_ATTR
user_mvh3004_init(void)
{
    i2c_master_gpio_init();
}
```

```
LOCAL bool ICACHE_FLASH_ATTR
user_mvh3004_burst_read(uint8 addr, uint8 *pData, uint16 len)
{
    uint8 ack;
    uint16 i;

    i2c_master_start();
    i2c_master_writeByte(addr);
    ack = i2c_master_checkAck();

    if (!ack) {
        os_printf("addr not ack when tx write cmd \n");
        i2c_master_stop();
        return false;
    }

    i2c_master_stop();
    i2c_master_wait(40000);

    i2c_master_start();
    i2c_master_writeByte(addr + 1);
    ack = i2c_master_checkAck();

    if (!ack) {
        os_printf("addr not ack when tx write cmd \n");
        i2c_master_stop();
        return false;
    }

    for (i = 0; i < len; i++) {
        pData[i] = i2c_master_readByte();

        if (i == (len - 1))
            i2c_master_send_nack();
        else
            i2c_master_send_ack();


    }

    i2c_master_stop();

    return true;
} ? end user_mvh3004_burst_read ?
```

# 10. I2S Module Description

## 10.1. Functional Overview

The I2S module of the ESP8266 contains a Tx (transport) unit and a Rx (receive) unit. Both the Tx and the Rx unit have a three-wire interface that includes:

- Clock line;
- Data line;
- Channel selection line (the line for selecting the left or the right channel).

> 📖 ***Note:***
> *The clock and data output will stop when 0 is written into the data line.*

The transmission direction of the I2S module is shown in Table 10-1.

**Table 10-1. Transmission Direction of The I2S Module**

| | Tx unit | Rx unit |
|---|---|---|
| Clock line | output / input | output / input |
| Data line | output | input |
| Channel selection line | output | input |

> 📖 ***Note:***
> *Both the Tx and Rx unit have a separate FIFO, which has a depth of 128 and a width of 32 bits, and can be visited by software directly. You can also make an automatic DMA operation to FIFO by the SLC module.*

## 10.2. System Configuration

### 10.2.1. I2S Module Configuration

#### 10.2.1.1.I2S Module Reset

Bits 0 ~ 3 in the I2SCONF register provide the software reset feature to the I2S. Write 1 and then 0 to complete the reset operation. Different bits are used for:

- Bit 0: I2S_TX_RESET
- Bit 1: I2S_RX_RESET
- Bit 2: I2S_TX_FIFO_RESET
- Bit 3: I2S_RX_FIFO_RESET

#### 10.2.1.2.I2S Module Start

**Provide a running clock**

To start the I2S module to transport or receive data, firstly you need to provide a running clock for the I2S by invoking the system function below:

```
i2c_writeReg_ Mask_def (i2c_bbpll, i2c_bbpll_en_audio_clock_out, 1)
```

**Start the Tx module**

Bit 8 in the I2SCONF register is used to start the Tx module.

- In the master Tx mode, when bit 8 is 1, the Tx mode will output the clock signal, the left and right channel signals and data. The first frame data is 0, and then the FIFO data will be shifted out.
  - If no data is written into the the FIFO, the data line will remain 0.
  - If the FIFO has transported all the written data and no new data is written in the FIFO, the data line will loop the last data in the FIFO.
- In the slave passive Tx mode, the Tx module will be started when it receives a clock signal from the Rx module.

**Start the Rx module**

Bit 9 in the I2SCONF register is used to start the Rx module. In the master receive mode:

- When bit 9 is 1, the Rx mode will output the clock signal, and sample the data line and the channel selection line.
- When bit 9 is 0, it will stop the clock signal transport.
- In the slave receive mode, it is prepared to receive any data from the master.

#### 10.2.1.3.Tx/Rx FIFO Mode

**FIFO access mode**

Bit 12 of I2S_FIFO_CONF defines the access mode of the FIFO.

- When bit 12 is 1, the SLC will make a DMA operation to the FIFO. Direct access to the FIFO will be invalid.
- When bit 12 is 0, the FIFO can be accessed directly by software.
- The default value of bit12 is 1.

**Tx FIFO mode**

Bits 13 ~ 15 of ***I2S_FIFO_CONF*** are used to control the transport data format for `i2s_tx_fifo_mod`.

| Value | Description |
|---|---|
| 0 | 16bits_per_channel full data (dual channel, FIFO data organisation, 16 bits data in the left channel,16 bits data in the right channel, and 16 bits data in the left channel) |

| Value | Description |
|---|---|
| 1 | 16bits_per_channel half data (single channel, FIFO data organisation, 16 bits data, 16 bits invalid , 16 bits data) |
| 2 | 24bits_per_channel full data discontinue (dual channel, FIFO data organisation, 24 bits data in the left channel, 8 bits invalid, 24 bits data in the right channel, 8 bits empty) |
| 3 | 24bits_per_channel half data discontinue (single channel, FIFO data organisation, 24 bits data, 8 bits invalid, 24 bits data, 8 bits empty) |
| 4 | 24bits_per_channel full data continue (left and right channels, FIFO data organisation, 24 bits data in the left channel, 24 bits data in the right channel) |
| 5 | 24bits_per_channel half data continue (single channel, FIFO data organisation, 24 bits data, 24 bits data) |
| 6 ~ 7 | Invalid |

#### RX FIFO mode

Bits 16~18 of ***I2S_FIFO_CONF*** is used to control the receive data format for `i2s_rx_fifo_mod`.

| Value | Description |
|---|---|
| 0 | 16bits_per_channel full data |
| 1 | 16bits_per_channel half data |
| 2 | 24bits_per_channel full data discontinue |
| 3 | 24bits_per_channel half data discontinue |
| 4 ~ 7 | Invalid |

### 10.2.1.4.Channel Mode

#### Tx channel mode

Bits 0 ~ 2 in the ***I2SCONF_CHAN*** are used for the Tx channel mode (`tx_chan_mod`).

| Value | Description |
|---|---|
| 0 | Dual-channel |
| 1 | Right channel (left and right audio channels are used to put the data of the right channel) |
| 2 | Left channel (left and right audio channels are used to put the data of the left channel) |
| 3 | Right channel (put a constant from regfile in the left channel) |
| 4 | Left channel (put a constant from regfile in the right channel) |

#### Rx channel mode

Bits 3~4 in the ***I2SCONF_CHAN*** are used for the Rx channel mode (`rx_chan_mod`).

| Value | Description |
|---|---|
| 0 | Dual-channel |
| 1 | Right channel |
| 2 | Left channel |

#### 10.2.1.5.Clock Mode

in the I2SCONF:

- Bits16 ~ 21 are the prescaler of the input clock (I2S_CLKM_DIV_NUM).
- Bits 22 ~ 27 are the frequency divider of the communication clock signal (I2S_BCK_DIV_NUM).

#### 10.2.1.6.Other Configurations

Register I2SRXEOF_NUM sets the number of data to be received when the Rx FIFO triggers the SLC transport (unit: 4 bytes).

See the definitions of ***i2s_reg.h*** in DEMO. Other instructions will be updated.

### 10.2.2. Link List Configuration

In the ESP8266, the DMA transfers the receive and transport packets in the SDIO to the corresponding memory. The software will define the structure (or group) of the registration list and cache space(s).

As shown in Figure 10-1, there is only one cache space and one registration list. Write the first address of the cache and other information to the registration list, and then write the first address of the registration list to the hardware register of the ESP8266. Therefore, the DMA will automatically operate the SDIO and the cache space.

| | | | | | | |
|---|---|---|---|---|---|---|
| word 0 | owner | eof | sub_sof | 5'b0 | length [11:0] | size [11:0] |
| word 1 | buf_ptr [31:0] | | | | | |
| word 2 | next_link_ptr [31:0] | | | | | |

**Figure 10-1. Registration List**

| Field name | | Description |
|---|---|---|
| owner | 1'b0 | Software operates the buffer of the current link. The MAC shouldn't use this bit. |
| | 1'b1 | Hardware operates the buffer of the current link. |

| Field name | Description |
|---|---|
| eof | Flag of frame end（for the end of AMPDU sub-frame，the mark isn't needed).<br>▸ When the MAC transports the frames, it's used in the end of the frame. For the link in the position of eof，the buffer_length[11:0] should be equal to the length of the remaining frame; otherwise, the mac will report an error.<br>▸ When the MAC receives the frames, it's used to indicate that the frame has been received completely and the value is set by hardware. |
| sub_sof | Flag of sub-frame start. It's used to differentiate different sub-frames in the AMPDU. It's only for MAC transport. |
| length[11:0] | The actual size of the buffer. |
| size[11:0] | The total size of the buffer. |
| buf_ptr[31:0] | The start address of the buffer. |
| next_link_ptr[31:0] | The start address of the next descripter. When the MAC is receiving the flame, the value is "0", indicating that there is no empty buffer to receive any flames. |

### 10.2.3. SLC Module Configuration

#### 10.2.3.1. Basic Configuration

The SLC module provides the ESP8266 with DMA service of several modules.

Follow the instructions below so that the SLC module is used for the FIFO transmission of I2S.

- Set Bits 12~13 (SLC_MODE) of the SLC_CONF0 to 01.
- Set Bit 17 (SLC_INFOR_NO_REPLACE) and Bit 16 (SLC_TOKEN_NO_REPLACE) of the SLC_RX_DSCR_CONF to 01.

#### 10.2.3.2. Write The First Address

Bits 0~19 of SLC_RX_LINK (SLC_TX_LINK) register are the first 20 bits of the Rx (Tx) registration list address. The first address of the registration list should be written to be the register before the SLC hardware is started.

#### 10.2.3.3. Start The SLC Transmission

Bit 29 of SLC_RX_LINK (SLC_TX_LINK) register is the control bit for starting the SLC transmission. In the cache space, register a link list and write the first 20 bits of the link table address to the hardware, and then set bit 29 to 1 to start the SLC transmission.

## 10.3. API Function Description

The following functions can be found in:

```
/app/driver/i2s.c and /app/include/driver/i2s.h
```

### 10.3.1. Void Function

| | |
|---|---|
| | `void i2s_test` |
| Function | `void i2s_test(void)` |
| Feature | I2S Programs for read and write testing of the module. It is the core function of the DEMO, which can be used to test the transporting and receiving communications of the I2S. |
| Parameter | null |
| | `void i2s_init` |
| Function | `void i2s_init(uint8 slc_en)` |
| Feature | Configure the related registers of the I2S. |
| Parameter | `slc_en`: Enable the SLC module access. When it's 0, the software will operate the FIFO， For other values for the SLC module directly access FIFO， refer to 2.1.3. Tx/Rx FIFO mode. |
| | `void creat_one_link` |
| Function | `void creat_one_link` (uint8 own, uint8 eof,uint8 sub_sof, uint16 size, uint16 length, uint32* buf_ptr, uint32* nxt_ptr, struct sdio_queue* i2s_queue) |
| Feature | Set up a link register structure. |
| Parameter | struct sdio_queue* i2s_queue: The first address to be configured structure space.<br>For details of other parameters, refer to ***Section 10.2.2. Link list Configuration***. |
| | `void slc_init` |
| Function | `void slc_init (uint8 trans_dev)` |
| Feature | Basic configuration of the SLC module. For configuration instructions, refer to ***Section 10.2.3. SLC module configuration***. |
| Parameter | `uint8 trans_dev`: SLCModule access device, 1 is I2S， 0 is SDIO， other input values are not valid. |

### 10.3.2. CONF Function

| | |
|---|---|
| | `CONF_RXLINK_ADDR` |
| Function | `CONF_RXLINK_ADDR(addr)` |
| Feature | Configure the Rx link list address to the register. For configuration instructions, refer to ***Section 10.2.3. SLC module configuration***. |
| Parameter | `addr`: link list address |
| | `CONF_TXLINK_ADDR` |
| Function | `CONF_TXLINK_ADDR(addr)` |
| Feature | Configure the TX link list address to the register. For configuration instructions, refer to ***Section 10.2.3. SLC module configuration***. |

| | |
|---|---|
| Parameter | addr: link list address |

### 10.3.3. START Function

`START_RXLINK`

| | |
|---|---|
| Function | `START_RXLINK()` |
| Feature | Start the Rx transmission of the SLC module. For configuration instructions, refer to ***Section 10.2.3. SLC module configuration***. |
| Parameter | null |

`START_TXLINK`

| | |
|---|---|
| Function | `START_TXLINK()` |
| Feature | Start the Tx transmission of the SLC module. For configuration instructions, refer to ***Section 10.2.3. SLC module configuration***. |
| Parameter | null |

# 11. UART Introduction

## 11.1. Functional Overview

There are two group ESP8266 UART interfaces, respectively:

- UART0:
  - U0TXD: pin26 (U0TXD)
  - U0RXD: pin25 (U0RXD)
  - U0CTS: pin12 (MTCK)
  - U0RTS: pin13 (MTDO)
- UART1:
  - U1TXD: pin14 (GPIO2)

The basic working process of transmission FIFO:

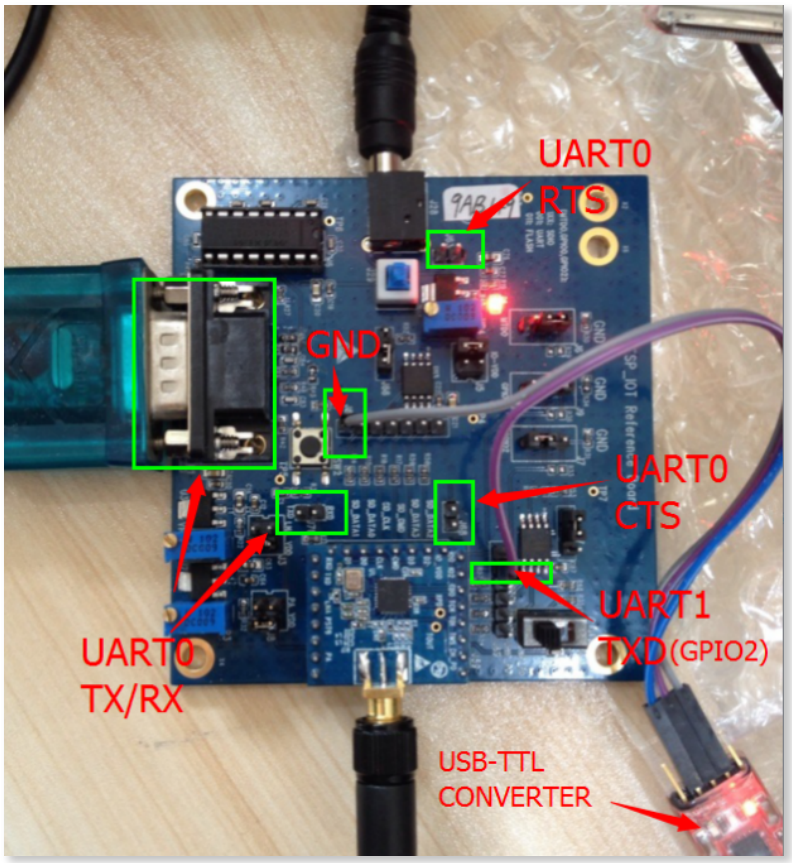
As long as there has data filling into transmission FIFO, it will immediately start sending process.Since transmission itself is a relatively slow process,other data can be sent to the transmission FIFO simultaneously.  Data sending should be paused when the transmission FIFO is full ,or it will cause data loss.Transmission FIFO will sent out one by one in accordance with the order of the data filling in,until the transmission FIFO is completely empty.Data has been sent will be automatically cleared, at the same time transmission FIFO will be more of a vacancy.

The basic working process of receiver FIFO:

When the hardware logic receives the data, it will fill them into receiver FIFO. Program should withdraw the data timely ,the data-dequeue is also a process of deleting data from FIFO automatically,thus, there will be one more vacancy in receiver FIFO. If the data in the receiver FIFO can not be removed in time, the receiver FIFO will be full which makes data loss.

Scenario:

UART0 works as data communication interface and UART1 woks as debug port.




UART0 will default output some print while booting ,the baud rate of this period print contents is relate with external crystal frequency.When using the 40M crystal,this section print baud rate is 115200.When use the 26M crystal,this section print baud rate is 74880.

If this print affect application function, you can abandon print output indirectly while power-on period in the fourth quarter method.

## 11.2. Hardware Resources

Both UART0 and UART1 have a length of 128 Byte hardware, read and write FIFO operations are at the same address.

The hardware registers of two UART module are the same, and distinguished by macro definitions of UART0 / UART1.

## 11.3. Parameter Configuration

UART attribute parameters are all in UART_CONF0 register, can be found in the ***Uart_register.h***. You can configure UART properties through modifying the different bit of the register.

### 11.3.1. The Baud Rate

The serial of ESP8266 can support the baud rate range from 300 to 115200 * 40.

Interface: void UART_SetBaudrate (uint8 uart_no,uint32 baud_rate);

### 11.3.2. Parity Bit

#define UART_PARITY_EN (BIT(1)) Enable check: 1: enable; 0: disable

#define UART_PARITY (BIT(0)) Check type setting 1: Odd parity; 0: Even parity

Interface: `void UART_SetParity(uint8 uart_no, UartParityMode Parity_mode);`

### 11.3.3. Data Bit

#define UART_BIT_NUM 0x00000003 //Length of data bit occupies two bit:

Setting these two bit can configure data length 0: 5bit ; 1: 6bit ; 2: 7bit ; 3: 8bit

#define UART_BIT_NUM_S 2 //Offset register is 2 (2 bit start)

Interface: `void UART_SetWordLength(uint8 uart_no, UartBitsNum4Char len)`

### 11.3.4. Stop Bit

#define UART_STOP_BIT_NUM 0x00000003 //The length of data bit occupies two bit:

Configure the length of stop bits through setting these two bit can 1 : 1 bit ; 2 : 1.5 bit ; 3 : 2 bit

#define UART_STOP_BIT_NUM_S 4 //Register offset is 4 (start from 4th bit)

Interface: `void UART_SetStopBits(uint8 uart_no, UartStopBitsNum bit_num);`

### 11.3.5. Inverting

Each input and output UART signals can reverse configuration internal.

#define UART_DTR_INV (BIT(24))

#define UART_RTS_INV (BIT(23))

#define UART_TXD_INV (BIT(22))

#define UART_DSR_INV (BIT(21))

#define UART_CTS_INV (BIT(20))

#define UART_RXD_INV (BIT(19))

Set the corresponding register,you can reverse the corresponding signal line input / output.

Interface: `void UART_SetLineInverse`

`(uint8 uart_no, UART_LineLevelInverse inverse_mask);`

### 11.3.6. Switch Output Port of Print Function

By default, the system os_printf function print output from UART0,you can set to print from UART0 or UART1 port through the following interfaces.

```
void UART_SetPrintPort(uint8 uart_no);
```

### 11.3.7. Read The Remaining Number of Bytes in tx / rx Queue

Tx fifo length：

(READ_PERI_REG(UART_STATUS(uart_no))>>UART_TXFIFO_CNT_S)

&UART_TXFIFO_CNT;

Interface: `TX_FIFO_LEN(uart_no)`

Rx fifo length:

(READ_PERI_REG(UART_STATUS(UART0))>>UART_RXFIFO_CNT_S)

&UART_RXFIFO_CNT;

Interface: `RF_FIFO_LEN(uart_no)`

### 11.3.8. Loopback Operation (loop-back)

Once configured in UART_CONF0 register, uart tx / rx shorted internally.

#define UART_LOOPBACK  (BIT(14))  // loopback enable bit,1: enable;0: disable

ENABLE: SET_PERI_REG_MASK(UART_CONF0(UART0), UART_LOOPBACK);

Interface: `ENABLE_LOOP_BACK(uart_no)`

DISABLE:CLEAR_PERI_REG_MASK(UART_CONF0(UART0), UART_LOOPBACK);

Interface: `DISABLE_LOOP_BACK(uart_no)`

### 11.3.9. Line Stop Signal

To produce the line stop signal,you can set UART_TXD_BRK 1,then after UART transmission queue complete sending it, it will output  a break signal (tx output low), set it 0 if you need to stop the output.

#define UART_TXD_BRK (BIT(8))  //Line stop signal, 1:enable ; 0: disable

### 11.3.10.Flow Control

Configuration process:

- Configure pin12, pin13 of UART0 pin as U0CTS and U0RTS functions.

#define FUNC_U0RTS  4

#define FUNC_U0CTS  4

PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDO_U, FUNC_U0RTS);

PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTCK_U, FUNC_U0CTS);

- Hardware flow control in the receive direction can configure thresholds,when the length of rx fifo is greater than the set threshold,U0RTS feet will be pulled to prevent the other party sending.

Configured the thresholds of receiving flow control:

The threshold related configurations are generally defined in UART_CONF1 register.

#define UART_RX_FLOW_EN (BIT(23)) The 23rd bit enabled to receive flow control: 0: disable; 1: enable

#define UART_RX_FLOW_THRHD 0x0000007F //Threshold, occupied 7bit, range 0 ~ 127

#define UART_RX_FLOW_THRHD_S 16 //Register offset is 16 (start from 16th bit)

- Once configure enable of the flow control of sending direction configuration,the register in UART_CONF0:

#define UART_TX_FLOW_EN (BIT(15)) Enable transmission flow control: 0: disable ; 1: enable

- Interface:

Void UART_SetFlowCtrl(uint8 uart_no,UART_HwFlowCtrl flow_ctrl,uint8 rx_thresh);

e)demo hardware board connections:

Need to connect the J68 (U0CTS) and J63 (U0RTS) jumper .

### 11.3.11.Other Interfaces

TX_FIFO_LEN(uart_no) //Macro definition, the current length of the transmit queue

RF_FIFO_LEN(uart_no) //Macro definition, the current length of the receiving queue

## 11.4. Configure Interrupt

Since all interrupt events will be conducted together in the "OR" operation before being sent to the interrupt controller,UART can only generate an interrupt request each time.By polling the interrupt state function UART_INT_ST (uart_no),software can deal with multiple interrupt events in one interrupt service function(multiple if parallel statement).

### 11.4.1. Interrupt register

Interruption registers in UART:

UART_INT_RAW Interrupt the original status register

UART_INT_ENA Interrupt enable register: Indicates interrupt the current enable UART

UART_INT_ST Interrupt Status Register: Indicates the currently active interrupt status

UART_INT_CLR Clear Interrupt register:set the corresponding bit to clear the interrupt status register

### 11.4.2. Interface

Open interrupt enable: UART_ENABLE_INTR_MASK(uart_no,ena_mask);

Close interrupt enable:

UART_DISABLE_INTR_MASK (uart_no,disable_mask);

Clear interrupt enable:

UART_CLR_INTR_STATUS_MASK(uart_no,clr_mask);

Get interrupt status： UART_GET_INTR_STATUS(uart_no);

### 11.4.3. Interrupt Type

**Receive full interrupt**

Interrupt status bits:UART_RXFIFO_FULL_INT_ST

Definition: When configure threshold and enable interrupts, triggered will interrupt when rx fifo data length is greater than the threshold.

Application:more applied in receiving UART data ,cooperating with flow control,dealing with directly or posting messages or turn into buffer.For example,when the configuration of the threshold is 100 and the enable full is interruption, the full will interrupt once the serial port receive 100 Bytes.

Configure threshold value:

Full interrupt threshold

In the UART_CONF1 register

#define UART_RXFIFO_FULL_THRHD 0x0000007F //The threshold mask, 7bit long and range 0 ~ 127

#define UART_RXFIFO_FULL_THRHD_S 0 //Shift register is 0 (start from 0bit)

Set enable to interrupt:

In UART_INT_ENA register

#define UART_RXFIFO_FULL_INT_ENA (BIT(0)) //full interrupt enable bit, 1: enable;0: disable

clear interrupt status:

As for special full interrupts， you need first to read all fifo received data empty, then write the clear interruption status register.Otherwise, the interrupt status bit will be set again after exit.

Please see details in examples of interrupt handling.

**Receive overflow interrupt**

Interrupt status bits:UART_RXFIFO_OVF_INT_ST

Definition: When enable receive overflow to interrupt and the length of the receive queue is greater than the total length of the queue (128 Bytes), it will trigger the interrupt signal.

Trigger scene: Generally, it's only under the case of unset flow control,because there will not occur overflow when has flow control.Different from the full interrupt is artificially set the threshold and the data will not lose,overflow interrupt triggering will usually has data loss. Can be used for debugging and error checking.

Set enable to interrupt:

In UART_INT_ENA register

#define UART_RXFIFO_OVF_INT_ENA (BIT(4)) //Overflow interrupt enable bit: 1: enable; 0: disable

Clear interrupt status:

Read queue value to make the queue length less than 128, then set the clear interrupt status register.

**Receive timeout interrupt**

Interrupt status bit: UART_RXFIFO_TOUT_INT_ST

Definition:When configure threshold value of tout,enable interrupts and UART begin to receive data, it will triggered tout interrupt once stop transmission time exceeds the set threshold.

Applications: more applied in handling serial commands or data, process the data directly, or post a message, or turn into deposited buffer.

Configure threshold and function enable:

Tout interrupt threshold (or threshold) in UART_CONF1 register.

Tout unit threshold is about 8 data bits uart time (approximately one byte).

#define UART_RX_TOUT_EN  (BIT(31)) //Timeout function enable bit: 1: enable;0: disable

#define UART_RX_TOUT_THRHD 0x0000007F  //Timeout threshold configuration bits, a total of seven and range 0 ~ 127

#define UART_RX_TOUT_THRHD_S 24  //Register offset is 24 (start from 24th bit)

Set enable to interrupt:

In UART_INT_ENA register

#define UART_RXFIFO_TOUT_INT_ENA (BIT(8))  tout  // Interrupt enable bit:1: enable;0: disable

Clear interrupt status:

Like full interrupts,tout interrupt also need to firstly read out all received fifo data,then clear interrupt status register.Otherwise, interrupt status  bit will still be set after exiting.

Please see details in examples of interrupt handling.

**Send empty fifo interrupt**

Interrupt status bit: UART_TXFIFO_EMPTY_INT_ST

Definition: After configure empty threshold value and enable interrupts ,it will trigger this empty interrupt when the data length of the data-send queue is less than the set threshold.

Application: Can be used in forwarding the buffer data into UART automatically with the cooperation of interrupt handler function.For example,set the empty threshold to 5, then when the tx fifo length be less than 5 bytes, trigger the empty interrupt,in the empty interrupt handler ,take the data from the buffer to fill the tx fifo full(operating speed is much higher than tx fifo fifo transmission speed ). Continue the cycle until the buffer data has totally been sent out, then close the empty interrupt.

Configure threshold:

Empty interrupt threshold (or threshold) in UART_CONF1 register

#define UART_TXFIFO_EMPTY_THRHD 0x0000007F //Send queue empty interrupt threshold configuration bits, seven bits and range 0 ~ 127

#define UART_TXFIFO_EMPTY_THRHD_S 8 //Register Offset is 8 (start from 8th)

To enable interrupt:

In UART_INT_ENA register

#define UART_TXFIFO_EMPTY_INT_ENA (BIT(1)) //empty interrupt enable bit, 1: enable;0: disable

Clear interrupt status:

Fill the sending queue above the threshold,and clear the corresponding interrupt status bit.If there is no data need to send, close the interrupt enable bits.

Please see details in examples of interrupt handling.

**Error detection interrupt**

Interrupt status bit:

Parity Error Interrupt：UART_PARITY_ERR_INT_ST

Termination line error interrupt(line-break)：UART_BRK_DET_INT_ST

Received frame error interrupt： UART_FRM_ERR_INT_ST

Definition:

Parity error interrupt (parity_err): received byte exists parity error.

Termination line error interrupt(BRK_DET):receive break signal,or receive error initial conditions (rx line always stays low)

Receive frame error interrupt (frm_err):stop bit is not 1.

Application:

Generally used for error detection.

To enable interrupt:

In UART_INT_ENA register,

#define UART_PARITY_ERR_INT_ENA (BIT(2)) //Parity error enable interrupt bit, 1:enable; 0:disable

#define UART_BRK_DET_INT_ENA (BIT(7)) //Terminal line error enable interrupt bit

1: enable;0: disable

#define UART_FRM_ERR_INT_ENA (BIT(3)) //Received frame error to enable interrupt bit

1: enable;0: disable

Clear interrupt status:

Clear the interrupt status bit after dealing with corresponding error.

**Flow control status interrupt**

Interruption status bit:

UART_CTS_CHG_INT_ST

UART_DSR_CHG_INT_ST

Definition:

When the CTS, DSR pin-line level changes, trigger this interrupt.

Application:

Generally use with flow control, when the trigger the interrupt, check the corresponding flow control line status,if it's high, stop writing to tx queue.

#define UART_CTS_CHG_INT_ST (BIT(6))

#define UART_DSR_CHG_INT_ST (BIT(5))

Set enable interrupt:

In UART_INT_ENA register,

#define UART_CTS_CHG_INT_ENA (BIT(6)) CTS //Line status enable interrupt bit, 1:enable;0:disable

#define UART_DSR_CHG_INT_ENA (BIT(5)) DSR //Line status enable interrupt bit, 1:enable;0:disable

Clear interrupt status:

After dealing with the corresponding error,clear the interrupt status bit.

## 11.5. Example of Interrupt Handler Process

```
LOCAL void
uart0_rx_intr_handler(void *para)
{
  /* uart0 and uart1 intr combine togther, when interrupt occur, see reg 0x3ff20020, bit2, bit0 represents
   * uart1 and uart0 respectively
   */
  uint8 RcvChar;
  uint8 uart_no = UART0;//UartDev.buff_uart_no;
  uint8 fifo_len = 0;
  uint8 buf_idx = 0;
  uint32 uart_intr_status = READ_PERI_REG(UART_INT_ST(uart_no)) ;//get uart intr status
  while (uart_intr_status != 0x0) { //while intr status is not cleared
    if (UART_FRM_ERR_INT_ST == (uart_intr_status & UART_FRM_ERR_INT_ST)) { //if it is caused by a frm_err interrupt
      WRITE_PERI_REG(UART_INT_CLR(uart_no), UART_FRM_ERR_INT_CLR);
    } else if (UART_RXFIFO_FULL_INT_ST == (uart_intr_status & UART_RXFIFO_FULL_INT_ST)) {//if it is caused by a fifo_full interrupt
      fifo_len = (READ_PERI_REG(UART_STATUS(UART0)) >> UART_RXFIFO_CNT_S)&UART_RXFIFO_CNT;  //read rf fifo length
      buf_idx = 0;
      //os_printf("full len:%d\n\r",fifo_len);//for dbg
      while (buf_idx < fifo_len) {                                                    //read all the data in rx fifo
        uart_tx_one_char(UART0, READ_PERI_REG(UART_FIFO(UART0)) & 0xFF);
        buf_idx++;
      }
      WRITE_PERI_REG(UART_INT_CLR(UART0), UART_RXFIFO_FULL_INT_CLR);              //clear full interrupt state
    } else if (UART_RXFIFO_TOUT_INT_ST == (uart_intr_status & UART_RXFIFO_TOUT_INT_ST)) {//if it is caused by a time_out interrupt
      fifo_len = (READ_PERI_REG(UART_STATUS(UART0)) >> UART_RXFIFO_CNT_S)&UART_RXFIFO_CNT;  //read fifo length
      buf_idx = 0;
      //os_printf("tout len:%d\n\r",fifo_len);//for dbg
      while (buf_idx < fifo_len) {                                                  //read all the data in rx fifo
        uart_tx_one_char(UART0, READ_PERI_REG(UART_FIFO(UART0)) & 0xFF);
        buf_idx++;
      }
      WRITE_PERI_REG(UART_INT_CLR(UART0), UART_RXFIFO_TOUT_INT_CLR);              //clear rx tout interrupt state
    } else if (UART_TXFIFO_EMPTY_INT_ST == (uart_intr_status & UART_TXFIFO_EMPTY_INT_ST)) {//if it is caused by a tx_empty interrupt
      //uart1_sendStr_no_wait("empty\n\r");//for dbg
      WRITE_PERI_REG(UART_INT_CLR(uart_no), UART_TXFIFO_EMPTY_INT_CLR);
      CLEAR_PERI_REG_MASK(UART_INT_ENA(UART0), UART_TXFIFO_EMPTY_INT_ENA);
    } else {
      //skip
    }
    uart_intr_status = READ_PERI_REG(UART_INT_ST(uart_no)) ;                        //update interrupt status
  } ?  end while uart_intr_status!=0x0 ?
} ?  end uart0_rx_intr_handler ?
```

## 11.6. Abandon Serial Output During Booting

When ESP8266 is booting, UART0 will default print out some information,if this should be un-acceptable,we can abandon these print output via setting UART internal switching pin functions,exchange U0TXD, U0RXD with U0RTS, U0CTS during initialization.

Calling interface: `void system_uart_swap(void);`

Before initialization:

UART0:

U0TXD: pin26(u0txd)

U0RXD: pin25(u0rxd)

U0CTS: pin12(mtck)

U0RTS: pin13(mtdo)

After the initialization pin-swap,

U0TXD: pin13(mtdo)

U0RXD: pin12(mtck)

U0CTS: pin25(u0rxd)

U0RTS: pin26(u0txd)

As the transceiver feet of UART0,hardware pin13 and pin12 won't print out duing booting,but be attention to ensure pin13 (mtdo) can not be pulled up by external in ESP8266 is booting.

# 12. PWM Interface

## 12.1. Functional Overview

### 12.1.1. Features

PWM (Pulse Width Modulation) can be implemented on Frame Rate Control 1 (FRC1) via software programming, achieving multi-channelled PWM with the same frequency but different duty ratio. It can be used to control devices such as color lights, buzzer, and electric machines, etc.

> 📖 ***Note:***
> *FRC1 is a 23-bit hardware timer.*

Features of PWM are listed below:

- Apply NMI (Non Maskable Interrupt) to interrupt, more precise.
- Can be extended to 8 channels of PWM signal.
- Resolution ratio higher than 14 bit, the minimum resolution can reach 45 ns.
- Configuration can be completed by call interface functions, without set the register.

> ⚠️ ***Notice:***
> - *PWM can not be used when APIs in hw_timer.c are in use, because they all use the same hardware timer.*
> - *Do not set the system to be Light Sleep mode (Do not call* `wifi_set_sleep_type(LIGT_SLEEP);`*, because that Light Sleep will stop the CPU, it can not be interrupted by NMI during light sleep.*
> - *To enter Deep Sleep mode, PWM needs to be stopped first.*

### 12.1.2. Implementation

An optimized software algorithm provided by ESP8266 system enable the transmission of multi-channel PWM signals via GPIO (General Purpose Input Output) interface by way of mounting NMI on FRC1 timer.

The clock of PWM is provided by high-speed system clock, the frequency speed of which can reach as high as 80MHz. Through pre-frequency divider, the clock source can be divided into 16 separated frequencies, the input clock frequency of which is 5MHz. PWM can issue coarse tuning timing via FRC1, which combined with fine tuning issued by the high-speed system clock, can improve the resolution to as much as 45 ns.

> 📖 ***Note:***
> *The highest priority level of interrupt owned by NMI ensures the precision of PWM output waveform.*

### 12.1.3. Configuration

- In timing interrupt, to exist the program as soon as possible, timing parameters of the next period of PWM waveform can be loaded when PWM period started.
- After the duty ratios of every channel have been configured, the system will call function pwm_start() to calculate timing cycle. Before that, parameters of all current channels will be stored and protected by the system, calculation completion bits will be cleared, too. When PWM period comes, parameters stored by the system will be invoked.
- When PWM period is discontinued new parameters will be applied, and flags should be set when the calculation of timing cycle is completed, so that cycles between different colour shade with each new frame and simulate an intermediate shade, achieving higher quality colour. The control of RGB colour lights is an good example of PWM control.
- The specific GPIO used can be configured in user_light.h. In our demo SDK, 5 channels of PWM is applied, however, it can be extended to 16 channels. Details on how to extend the channels of PWM is explained in Chapter 3. The minimum resolution can reach 45 ns at 1KHz refresh rate, while the minimum duty ratio can reach 1/22222.

### 12.1.4. Parameter Specification

- Minimum resolution: 45 ns (approximately speaking, the PWM input clock frequency is 22.72 MHz): >14 bit PWM @ 1 kHz
- PWM period: 1000 μs (1 KHz) ~ 10000 μs (100 Hz)

## 12.2. Details on pwm.h

### 12.2.1. Sample Codes

```
#ifndef __PWM_H__
#define __PWM_H__
#define PWM_CHANNEL_NUM_MAX 8             //8 channels PWM at most
struct pwm_single_param {                 //define the structure of a
single PWM parameter
    uint16 gpio_set;                      //GPIO needs to be set
    uint16 gpio_clear;                    //GPIO needs to be cleared
    uint32 h_time;                        //time needs to be written
into FRC1_LOAD
};
struct pwm_param {                       //define the structure of
PWM parameter
```

```
    Uint32 period;                          //PWM period
    Uint32 freq;                            //PWM frequency
    uint32 duty[PWM_CHANNEL_NUM_MAX];       //PWM duty ratio
};
void pwm_init(uint32 period, uint32 *duty,uint32
pwm_channel_num,uint32 (*pin_info_list)[3]);
void pwm_start(void);
void pwm_set_duty(uint32 duty, uint8 channel);
uint32 pwm_get_duty(uint8 channel);
void pwm_set_freq(uint32 period);
uint32 pwm_get_freq(void);
```

### 12.2.2. Interface Specifications

#### 1. `pwm_init`

| | |
|---|---|
| Function Name | pwm_init |
| Definition | PWM initialization. |
| Sample code | `pwm_init (uint32 freq, uint32 *duty, uint32 pwm_channel_num,uint32 (*pin_info_list)[3]);` |
| Description | PWM GPIO, initializing parameters and timer. |
| Parameters | • uint32 freq: PWM period.<br>• uint32 *duty: duty ratio of each PWM channel.<br>• uint32 pwm_channel_num: the number of PWM channels.<br>• uint32 (*pin_info_list)[3]: This parameter, which is made up of a n x 3 array pointer, defines the GPIO hardware parameter of each PWM channel. Registers of GPIO, pin multiplexing of IO, and the serial number of each GPIO are defined in the array. Take the initialization of a 3-channel PWM for example:<br>`uint32 io_info[][3] = {{PWM_0_OUT_IO_MUX,PWM_0_OUT_IO_FUNC,PWM_0_OUT_IO_NUM},`<br>`{PWM_1_OUT_IO_MUX,PWM_1_OUT_IO_FUNC,PWM_1_OUT_IO_NUM},`<br>`{PWM_2_OUT_IO_MUX,PWM_2_OUT_IO_FUNC,PWM_2_OUT_IO_NUM}};`<br>`pwm_init(light_param.pwm_period,light_param.pwm_duty, 3,io_info);` |
| Call | Call the function when the system is been initialized. Currently the function can be called only once. |
| Returned Value | Null |

#### 2. `pwm_set_period`

| | |
|---|---|
| Function Name | pwm_set_period |
| Definition | Set PWM period. |

| | |
|---|---|
| Sample code | `pwm_set_period (uint32 period)` |
| Description | Set PWM period, unit: μs.<br>For example, PWM period at 1KHz is1000 μs. |
| Parameters | uint32 period: PWM period. |
| Call | Call `pwm_start()` after the parameters has been set. |
| Returned Value | Null |

### 3. `pwm_set_duty`

| | |
|---|---|
| Function Name | pwm_set_duty |
| Definition | Set the duty ratio of PWM signal at a certain channel |
| Sample code | `pwm_set_duty (uint32 duty, uint8 channel)` |
| Description | Set PWM duty ratio. Set the time period of PWM signal when the voltage is high. The value of duty ratio change with PWM period.<br>PWM duty ratio can reach period*1000/45 at most. For example, the range of duty ratio is between 0 and 22222 at 1kHz refresh rate. |
| Parameters | • uint32 duty: set the time parameter when the voltage is high. Duty ratio is (duty*45)/(period*1000).<br>• uint8 channel: PWM channel that needs to be set at present. This parameter is defined in `PWM_CHANNEL`. |
| Call | Call `pwm_start()` after the parameters has been set. |
| Returned Value | Null |

### 4. `pwm_get_period`

| | |
|---|---|
| Function Name | pwm_get_period |
| Description | Get the current PWM period. |
| Sample code | `pwm_get_period (void)` |
| Description | None. |
| Returned Value | PWM period, unit: μs. |

### 5. `pwm_get_duty`

| | |
|---|---|
| Function Name | pwm_get_duty |
| Description | Get the duty ratio of current PWM signal at a certain channel. |
| Sample code | `pwm_get_duty (uint8 channel)` |
| Parameter | uint8 channel: get the current PWM channel. This parameter is defined in `PWM_CHANNEL`. |

| | |
|---|---|
| Call | Call `pwm_start()` after the parameters has been set. |
| Returned Value | Duty ratio of a certain PWM channel, the value returned is (duty*45)/ (period*1000). |

6. `pwm_start`

| | |
|---|---|
| Function Name | pwm_start |
| Description | Update PWM parameters. |
| Sample code | `pwm_start (void)` |
| Parameter | None. |
| Call | Call `pwm_start()` when PWM related parameters have been set. |
| Returned Value | Null. |

## 12.3. Custom Channels

Users can customize PWM channels. Below is a detailed instruction on how to set GPIO4 as the forth channel for PWM signal output.

1. **Modify initialization parameters.**

```
uint32 io_info[][3]={
    {PWM_0_OUT_IO_MUX,PWM_0_OUT_IO_FUNC,PWM_0_OUT_IO_NUM},
    {PWM_1_OUT_IO_MUX,PWM_1_OUT_IO_FUNC,PWM_1_OUT_IO_NUM},
    {PWM_2_OUT_IO_MUX,PWM_2_OUT_IO_FUNC,PWM_2_OUT_IO_NUM},
    {PWM_3_OUT_IO_MUX,PWM_3_OUT_IO_FUNC,PWM_3_OUT_IO_NUM},
    {PWM_4_OUT_IO_MUX,PWM_4_OUT_IO_FUNC,PWM_4_OUT_IO_NUM},
    };
pwm_init(light_param.pwm_period, light_param.pwm_duty,
PWM_CHANNEL,io_info);
```

2. **Modify *user_light.h*.**

```
#define PWM_0_OUT_IO_MUX PERIPHS_IO_MUX_MTDI_U
#define PWM_0_OUT_IO_NUM 12
#define PWM_0_OUT_IO_FUNC FUNC_GPIO12
#define PWM_1_OUT_IO_MUX PERIPHS_IO_MUX_MTDO_U
#define PWM_1_OUT_IO_NUM 15
#define PWM_1_OUT_IO_FUNC FUNC_GPIO15
#define PWM_2_OUT_IO_MUX PERIPHS_IO_MUX_MTCK_U
#define PWM_2_OUT_IO_NUM 13
```

```
#define PWM_2_OUT_IO_FUN CFUNC_GPIO13
#define PWM_3_OUT_IO_MUX PERIPHS_IO_MUX_GPIO4_U
#define PWM_3_OUT_IO_NUM 4
#define PWM_3_OUT_IO_FUNC FUNC_GPIO4
#define PWM_4_OUT_IO_MUX PERIPHS_IO_MUX_GPIO5_U
#define PWM_4_OUT_IO_NUM 5
#define PWM_4_OUT_IO_FUNC FUNC_GPIO5
#define PWM_CHANNEL 5
```

# 13. IR Remote Control User Guide

## 13.1. Introduction to Infrared Transmission

Users can request the sample codes of infrared transmission by sending an e-mail to feedback@espressif.com.

This document introduces how to implement transmitting or receiving remote control codes using the 32-bit NEC IR transmission protocol as an example.

### 13.1.1. Transmitting

Users can use the following methods to transmit carrier wave:

- BCK of I2S
- 38KHz carrier frequency generated by WS pin
- Carrier wave generated by any GPIO via sigma-delta function. However, the duty ratio of carrier wave generated by sigma-delta is around 20%, thus MTMS pin (GPIO14) is suggested, for this pin can generate standard square wave at a carrier frequency of 38KHz and a duty ratio of 50% exactly.

In the sample codes, data transmission queue is generated via the DSR TIMER interface of system FRC2, while a state machine driving the transmission of infrared data is also generated.

Considering that the timing precision of transmitting NEC infrared code should reach a level of μs, when initiating IR TX, system_timer_reinit should be invoked to improve the timing precision of FRC2. In user_config.h, enable the definition of USE_US_TIMER, then interface function os_timer_arm_us can be invoked to implement precise timing at the level of μs.

### 13.1.2. Receiving

The receiving of remote control codes is implemented via edge-triggered interrupt. When one system is substracted from one another, the result is the duration time of the wave. This can be processed by software state machine `ir_intr_handler`.

> ⚠ ***Notice:***
>
> - *Receiving of infrared remote control codes is implemented via GPIO interrupt. However, the system can only register only one IO interrupt handler program at the same time. If other IOs also need interrupts, please handle these interrupts in the same processing program by determine the source of interrupt and deal with them accordingly.*
> - *In non-OS version of SDK, functions with ICACHE_FLASH_ATTR properties, including print function os_printf defined in IROM section of the Flash, should NOT be invoked in the whole process of interrupt handling process such as GPIO, UART, FRC, etc.*

## 13.2. Parameters Configuration

All kinds of parameters related to transmitting and receiving of infrared remote control codes can be configured in ***ir_tx_rx.h***.

**Config Parameters for Transmitting:**

```
#define GEN_IR_CLK_FROM_IIS 0

// Config the mode of carrier

// 1: IIS clock signal generates carrier wave for transmission

// 0: generate carrier wave for transmission under GPIO sigma-delta
modeI

// Suggest using MTMS pin to implement infrared transmitting
function.


                      // Config the register function and
multiplexing function of infrared pins

#define IR_GPIO_OUT_MUX PERIPHS_IO_MUX_GPIO5_U

#define IR_GPIO_OUT_NUM 5

#define IR_GPIO_OUT_FUNC      FUNC_GPIO5
```

**Config Parameters for Receiving:**

```
                     // Config the buffer size via infrared receiving

#define RX_RCV_LEN      128


                  // Config the GPIO register function and
multiplexing function of infrared pins

#define IR_GPIO_IN_NUM  14

#define IR_GPIO_IN_MUX  PERIPHS_IO_MUX_MTMS_U

#define IR_GPIO_IN_FUNC FUNC_GPIO14
```

**Other parameters:**

#define USE_US_TIMER can be defined in ***user_config.h***.

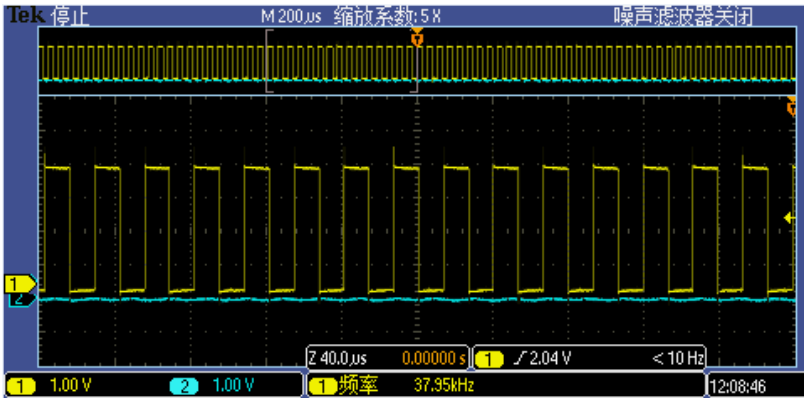
**Modes of Transmitting Carrier Waveform:**

Mode 1: IIS Clock Mode

MTMS pin, or GPIO14 is used to transmit carrier waveform under IIS clock mode. Please refer to Figure 1 below.

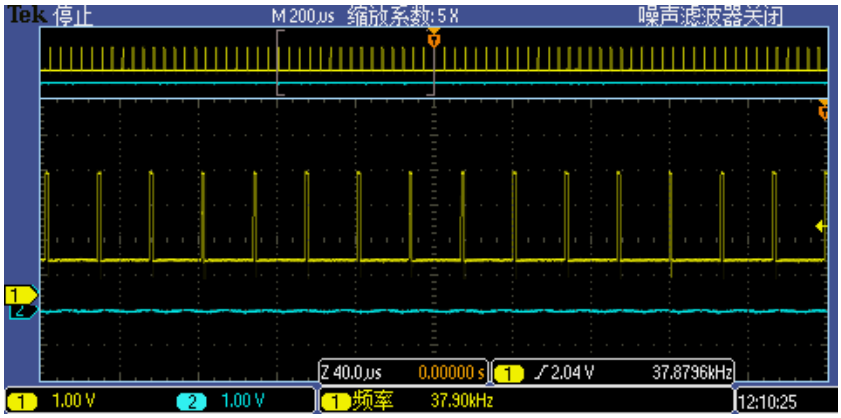
```
#define GEN_IR_CLK_FROM_IIS   1

#define IR_GPIO_OUT_MUX       PERIPHS_IO_MUX_MTMS_U

#define IR_GPIO_OUT_NUM       14
```

```
#define IR_GPIO_OUT_FUNC            FUNC_GPIO14
```




Mode 2: Sigma-delta Mode

```
#define GEN_IR_CLK_FROM_IIS   0
#define IR_GPIO_OUT_MUX       PERIPHS_IO_MUX_GPIO5_U
#define IR_GPIO_OUT_NUM       5
#define IR_GPIO_OUT_FUNC            FUNC_GPIO5
```




## 13.3. Functions of Infrared Sample Codes

The below functions can be implemented using infrared sample codes provided by Espressif Systems:

- Functions of infrared transmitting and receiving can be invoked in the initialization process, and a 4s loop timer can be configured to transmit infrared remote control codes.
- Check the ring buffer of infrared remote control codes simultaneously. If there is any data in the queue, it will be printed out.
- If any carrier waveform in comply with NEC infrared remote control protocol is received by the state machine of infrared receiver, the instruction fields will be stored in the ring buffer of infrared receiving codes.

# 14. Sniffer Introduction

## 14.1. Sniffer Introduction

ESP8266 can enter promiscuous mode (sniffer) and capture IEEE 802.11 packets in the air.

The following HT20 packets are support:

- 802.11b
- 802.11g
- 802.11n (from MCS0 to MCS7)
- AMPDU types of packets

The following are not supported:

- HT40
- LDPC

Although ESP8266 can not completely decipher these kinds of IEEE80211 packets completely, it can still obtain the length of these special packets.

In summary, while in sniffer mode, ESP8266 can either capture completely the packets or obtain the length of the packet:

- Packets that ESP8266 can decipher completely; ESP8266 returns with the
  - MAC address of the both side of communication and encryption type and
  - the length of entire packet.
- Packets that ESP8266 can only partial decipher; ESP8266 returns with
  - the length of packet.

Structure RxControl and sniffer_buf are used to represent these two kinds of packets. Structure sniffer_buf contains structure RxControl.

```
struct RxControl {
    signed rssi:8;              // signal intensity of packet
    unsigned rate:4;
    unsigned is_group:1;
    unsigned:1;
    unsigned sig_mode:2;        // 0:is not 11n packet; non-0:is 11n
packet;
    unsigned legacy_length:12; // if not 11n packet, shows length of
packet.
    unsigned damatch0:1;
    unsigned damatch1:1;
```

```
    unsigned bssidmatch0:1;
    unsigned bssidmatch1:1;
    unsigned MCS:7;            // if is 11n packet, shows the
modulation
                               // and code used (range from 0 to 76)
    unsigned CWB:1; // if is 11n packet, shows if is HT40 packet or
not
    unsigned HT_length:16;// if is 11n packet, shows length of
packet.
    unsigned Smoothing:1;
    unsigned Not_Sounding:1;
    unsigned:1;
    unsigned Aggregation:1;
    unsigned STBC:2;
    unsigned FEC_CODING:1; // if is 11n packet, shows if is LDPC
packet or not.
    unsigned SGI:1;
    unsigned rxend_state:8;
    unsigned ampdu_cnt:8;
    unsigned channel:4; //which channel this packet in.
    unsigned:12;
};

struct LenSeq{
    u16 len; // length of packet
    u16 seq; // serial number of packet, the high 12bits are serial
number,
             //    low 14 bits are Fragment number (usually be 0)
    u8 addr3[6]; // the third address in packet
};

struct sniffer_buf{
    struct RxControl rx_ctrl;
    u8 buf[36]; // head of ieee80211 packet
    u16 cnt;     // number count of packet
    struct LenSeq lenseq[1];  //length of packet
```

```
};

struct sniffer_buf2{
    struct RxControl rx_ctrl;
    u8 buf[112]; //may be 240, please refer to the real source code
    u16 cnt;
    u16 len;  //length of packet
};
```

Callback `wifi_promiscuous_rx` has two parameters (`buf` and `len`). `len` means the length of `buf`, it can be: `len` = sizeof(struct sniffer_buf2), `len` = X * 10, `len` = sizeof(struct RxControl):

**Case of LEN == sizeof (struct sniffer_buf2)**

- `buf` contains structure `sniffer_buf2`: it is the management packet, it has 112 Bytes data.
- `sniffer_buf2.cnt` is 1.
- `sniffer_buf2.len` is the length of packet.

**Case of LEN == X * 10**

- `buf` contains structure `sniffer_buf:` this structure is reliable, data packets represented by it has been verified by CRC.
- `sniffer_buf.cnt` means the count of packets in `buf`. The value of `len` depends on `sniffer_buf.cnt`.
  - `sniffer_buf.cnt`==0, invalid `buf`; otherwise, `len` = 50 + cnt * 10
- `sniffer_buf.buf` contains the first 36 Bytes of IEEE80211 packet. Starting from `sniffer_buf.lenseq[0]`, each structure lenseq represent a length information of packet. `lenseq[0]` represents the length of first packet. If there are two packets where (`sniffer_buf.cnt` == 2), `lenseq[1]` represents the length of second packet.
- If `sniffer_buf.cnt` > 1, it is a AMPDU packet, head of each MPDU packets are similar, so we only provide the length of each packet (from head of MAC packet to FCS)
- This structure contains: length of packet, MAC address of both sides of communication, length of the head of packet.

**Case of LEN == sizeof(struct RxControl)**

- `buf` contains structure RxControl; but this structure is not reliable, we can not get neither MAC address of both sides of communication nor length of the head of packet.
- For AMPDU packet, we can not get the count of packets or the length of packet.

- This structure contains: length of packet, rssi and FEC_CODING.
- RSSI and FEC_CODING are used to guess if the packets are sent from same device.

> 📖 ***Note:***
>
> *For the case of LEN == sizeof(struct RxControl), the methods to calculate the length of packet are as below:*
>
> - *If sig_mode == 0, the length of packet is the legacy_length.*
> - *Otherwise, the length of packet is in struct sniffer_buf and sniffer_buf2, and it is more reliable.*

**Summary**

We should not take too long to process the packets. Otherwise, other packets may be lost.

The diagram below shows the format of a IEEE80211 packet:

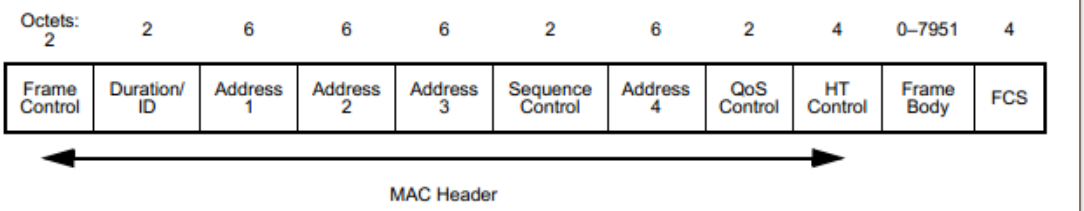



**Figure 8-30—Data frame**

- The first 24 Bytes of MAC Header of data packet are needed:
  - Address 4 field depends on FromDS and ToDS which is in Frame Control;
  - QoS Control field depends on Subtype which is in Frame Control;
  - HT Control field depends on Order Field which is in Frame Control;
  - More details are found in IEEE Std 80211-2012.
- For WEP packets, MAC Header is followed by 4 Bytes IV and before FCS there are 4 bytes ICV.
- For TKIP packet, MAC Header is followed by 4 Bytes IV and 4 bytes EIV, and before FCS there are 8 bytes MIC and 4 bytes ICV.
- For CCMP packet, MAC Header is followed by 8 Bytes CCMP header, and before FCS there are 8 bytes MIC.

## 14.2. Sniffer Application Scenarios

Because some APs won't transmit UDP broadcast packets to WLAN, so only the UDP packets from mobile phone can be listened. These UDP packets are from mobile phone to AP, and are encrypted.

**Scenario 1: IOT_device can get all packets from mobile phone**

This scenario requires:

- The connection between mobile phone and AP is working in 802.11b, or 802.11g, or 802.11n HT20 mode.
- The distance between mobile phone and AP is longer than the distance between mobile phone and IOT_device.

IOT-device firmware can set filter of MAC address or MAC-header (include MAC-cryption-header), it can also set a filter for retransmission.

Meanwhile, for 802.11n AMPDU packets, IOT_device can also get the length of packet and MAC-header (include MAC-cryption-header)

**Scenario 2: IOT_device can not get all packets from mobile phone, signal is strong, but packet format is not supported.**

**Case 1:**

The distance between mobile phone and AP is much longer than the distance between mobile phone and IOT_device. Then the high-frequency packets from mobile phone can be got by AP, but can not be got by IOT_device.

For example, mobile phone sent MCS7 packets which can be got correctly by AP, but IOT_device can only parse its packet header of physical layer (HT-SIG), because packet header of physical layer is encoded on low-speed (6 Mbps).

**Case 2:**

Format of packets that mobile phone sent to AP is not supported by IOT_device, such as:

- HT40;
- LDPC;
- 11n MCS8 and later version, such as MIMO 2x2.

IOT_device can not get the whole packet, but can parse its packet header of physical layer (HT-SIG).

In both case 1 and case 2, IOT_device can get HT-SIG which include the length of packet in physical layer. Please pay attention on following items when using it:

- When it isn't AMPDU packet or only one sub-frame in AMPDU packet, the length of UDP packet can be speculated. If the time interval of UDP packets which sent from phone APP is long ( 20ms ~ 50ms ) , each UDP packet will in different packets in physical layer, may be a AMPDU packet which only has one sub-frame.
- Firmware of IOT_device can filter packets from other devices according to RSSI.
- Packet of retransmission need to be filter according to the packets sequence, it means that length of packets which sent consecutively need to be different. For example:
  - Two useful packets can be separated by a specific packet. The specific packet works like separative sign.
  - Length of packet in odd number to be 0 ~ 511, length of packet in even number to be 512 ~1023。

## 14.3. Phone APP

For Scenario 2, phone APP should notice:

- Time interval of each UDP packet to be longer than 20ms
- Two data packets can be separated by a specific packet. The specific packet works like separative sign.
- Packet with redundant data so that packet can verify each other.
- Set flag-packet at the beginning of sequence. Then phone APP can be cyclic sending the whole sequence.
- Only need to send the lowest 2 Bytes of AP's BSSID ( MAC address ), IOT-device can still get it. If AP will broadcast its SSID, then phone APP need not to send AP's SSID either. So AP beacon need to be analyzed to check if the AP will broadcast its SSID.
- Length of UDP packet need to be multiply by 4. Because when phone APP sent a AMPDU packet which only has one sub-frame, packet length will be filled to be a multiple of 4.

For Scenario 1, phone APP  can send packets as fast as possible.

Phone APP won't know it is Scenario 1 or Scenario 2 for IOT_device.

## 14.4. IOT-device Firmware

For Scenario 2, IOT-device should notice:

- Search the channel which has strongest signal first, according to RSSI.
- Filter useless packets according to RSSI. Considering 10 ~ 15db fluctuations in the air, some packets may be decline 10db or more. We could search the strongest signal at first, then extend the range since find the target sequence.
- Check the Aggregation bit of HT-SIG to distinguish AMPDU packet.
- AMPDU packet can only be encrypt by CCMP(AES).
- To design the length of packet that works as separative sign, different QoS, different encryption algorithm and AMPDU packet will be a multiple of 4, all of these should be taken into consideration.
- Use relative value to transmit information, for example, the value that the length of data packet minus the length of packet that works as separative sign.

# Appendix

📖 ***Note:***

*For GPIO registers, SPI registers, UART registers and Timer registers, please refer to the following appendixes.*

| Chapter | Title | Subject |
|---|---|---|
| Appendix 1 | GPIO Registers | Information on GPIO register names, addresses and description. |
| Appendix 2 | SPI Registers | Information on SPI register names, addresses and description. |
| Appendix 3 | UART Registers | Information on UART register names, addresses and description. |
| Appendix 4 | Timer Registers | Information on Timer register names, addresses and description. |