ESP8266
技术参考

版本 1.4

版权 © 2019

关于本手册

本文介绍了 ESP8266 的各个接口，包括功能、参数配置、函数说明、应用示例等内容。

本手册结构如下：

章	标题	内容
第 1 章	概述	对 ESP8266 各接口的简要介绍。
第 2 章	GPIO	描述 GPIO 的功能、寄存器和参数配置。
第 3 章	SDIO 通信 SPI 兼容模式	描述 SDIO 的功能、DEMO 实现方案、ESP8266 端及 STM32 端软件说明。
第 4 章	SPI 模块使用说明	描述 SPI 的功能、SPI 主 / 从机协议格式和 API 函数说明。
第 5 章	SPI Overlap 模式和显示屏控制台 DEMO	描述 SPI 的功能、硬件连接、API 说明和显示屏控制台程序 DEMO。
第 6 章	SPI 透传协议（单线）	描述 SPI 的功能、从机协议格式、从机状态定义与中断线行为和 API 函数说明。
第 7 章	SPI 透传协议（双线）	描述 SPI 的功能、从机协议格式、数据流控制线功能说明和 API 函数说明。
第 8 章	HSPI 主机多设备驱动说明	描述 HSPI 的功能、硬件连接和 API 说明。
第 9 章	I2C 接口说明	描述 I2C 的功能、master 接口和使用示例。
第 10 章	I2S 接口说明	描述 I2S 的功能、模块配置和接口函数说明。
第 11 章	UART 接口说明	描述 UART 的功能、硬件资源、参数配置、配置中断、中断处理函数示例流程和屏蔽上电打印。
第 12 章	PWM 接口说明	PWM 的功能、***pwm.h*** 详解和自定义通道。
第 13 章	IR 红外例程及使用说明	红外发送与接收使用说明、参数配置、例程说明、硬件连接和实验结果。
第 14 章	Sniffer 应用设计说明	Sniffer 模式介绍、应用场景和相关问题、手机 APP 设计和 IOT-device 上固件设计。
附录	附录	GPIO 寄存器、SPI 寄存器、UART 寄存器、定时器寄存器。

发布说明

日期	版本	发布说明
2016.05	V1.0	首次发布。
2016.06	V1.1	增加章节 4.5 SPI 接口说明

日期	版本	发布说明
2016.08	V1.2	更新章节 14.1 Sniffer 模式介绍
2017.05	V1.3	更新章节 4.1.2 SPI 特点
2019.08	V1.4	更新章节 1.1 通用输入 / 输出接口（GPIO）

目录

1. 概述 1

1.1. 通用输入 / 输出接口（GPIO） 1

1.2. 安全数字输入 / 输出接口（SDIO） 1

1.3. 串行外设接口（SPI / HSPI） 1

1.3.1. 通用 SPI（主机 / 从机） 1

1.3.2. HSPI（从机） 2

1.4. I2C 接口 2

1.5. I2S 接口 2

1.6. 通用异步收发器（UART） 3

1.7. 脉冲宽度调制（PWM） 3

1.8. IR 遥控接口 4

1.9. Sniffer 4

2. GPIO 5

2.1. 功能综述 5

2.2. GPIO 寄存器说明 6

2.2.1. GPIO 功能选择寄存器 6

2.2.2. GPIO 输出寄存器 6

2.2.3. GPIO 输入寄存器 7

2.2.4. GPIO 中断寄存器 7

2.2.5. GPIO16 对应接口 8

2.3. 参数配置 8

2.3.1. 应用场景 1 参数配置 9

2.3.2. 应用场景 2 参数配置 10

2.3.3. 应用场景 3 参数配置 10

2.3.4. 中断函数处理流程说明 11

2.3.5. 中断函数处理流程示例 12

3. SDIO 通信 SPI 兼容模式 13

3.1. 功能综述 13

3.2. DEMO 实现方案 13

3.2.1. 平台介绍 13

3.2.2. ESP8266 软件编译与下载准备 13

3.2.3. ESP8266 FLASH 端软件下载 14

3.2.4. ESP8266 FLASH 端软件下载 14

3.3. ESP8266 端软件说明 15

3.3.1. 协议原理：SDIO 中断线行为与 SDIO 状态寄存器 15

3.3.2. 读写缓存与注册链表的使用说明 16

3.3.3. ESP8266 DEMO 中提供的 API 函数 17

3.4. STM32 端软件说明 18

3.4.1. 主要函数说明 18

4. SPI 模块使用说明 21

4.1. 概述 21

4.1.1. 功能综述 21

4.1.2. SPI 特点 21

4.2. ESP8266 SPI 主机协议格式 21

4.2.1. SPI 主机支持的通信格式 21

4.2.2. 现有 API 支持的 SPI 主机通信格式 22

4.3. ESP8266 SPI 从机协议格式 22

4.3.1. SPI 从机时钟极性配置要求 22

4.3.2. SPI 从机支持的通信格式 22

4.3.3. SPI 从机支持命令定义 22

4.3.4. 现有 API 支持的 SPI 从机通信格式 23

4.4. SPI 模块 API 函数说明 23

4.4.1. SPI 主机 API 函数说明 23

4.4.2. SPI 主机 API 函数说明 25
4.5. SPI 接口说明 27
4.5.1. 数据结构 28
4.5.2. 接口说明 30
4.5.3. SPI_Test 示例说明 36
5. SPI Overlap 模式和显示屏控制台 DEMO 47
5.1. 功能综述 47
5.2. SPI OVERLAP 模式下的硬件连接 48
5.3. SPI OVERLAP 模式的 API 说明 48
5.4. 显示屏控制台程序 DEMO 49
5.4.1. 连线说明 49
5.4.2. API 函数说明 49
5.4.3. 预编译宏设定 51
6. SPI 透传协议（单线） 52
6.1. 功能综述 52
6.2. ESP8266 SPI 从机协议格式 52
6.2.1. SPI 从机时钟极性配置要求 52
6.2.2. SPI 从机支持的通信格式 52
6.3. 从机状态定义与中断线行为 53
6.3.1. 状态定义 53
6.3.2. GPIO0 中断线行为 53
6.4. ESP8266 SPI 从机 API 函数说明 53
7. SPI 透传协议（双线） 59
7.1. 功能综述 59
7.2. ESP8266 SPI 从机协议格式 59
7.2.1. SPI 从机时钟极性配置要求 59
7.2.2. SPI 从机支持的通信格式 59

7.3. 数据流控制线功能说明 59
7.3.1. GPIO0 主机发送从机接收缓存状态 60
7.3.2. GPIO2 主机接收从机发送缓存状态 60
7.3.3. 主机通信逻辑实现 60
7.4. ESP8266 SPI 从机 API 函数说明 62
8. HSPI 主机多设备驱动说明 65
8.1. 功能综述 65
8.2. 硬件连接 65
8.3. API 说明 66
9. I2C 使用说明 67
9.1. 功能综述 67
9.2. I2C master 接口 67
9.2.1. 初始化 67
9.2.2. I2C 起始 67
9.2.3. I2C 停止 68
9.2.4. I2C 主机回复 ACK 68
9.2.5. I2C 主机回复 NACK 69
9.2.6. 检查 I2C 从机应答 69
9.2.7. 向 I2C 总线写数据 69
9.2.8. 向 I2C 总线读数据 70
9.3. 使用示例 70
10.I2S 接口说明 72
10.1. 功能综述 72
10.2. 模块配置 72
10.2.1. I2S 模块配置 72
10.2.2. 链表配置 75
10.2.3. SLC 模块配置 76

10.3. 接口函数说明 76
10.3.1. 空隙函数 77
10.3.2. 配置函数 77
10.3.3. 启动函数 78
11.UART 接口说明 79
11.1. 功能综述 79
11.2. 硬件资源 80
11.3. 参数配置 80
11.3.1. 波特率 80
11.3.2. 校验位 81
11.3.3. 数据位 81
11.3.4. 停止位 81
11.3.5. 反相 81
11.3.6. 切换打印函数输出端口 82
11.3.7. 读取 tx/rx 队列内当前剩余的字节数 82
11.3.8. 回环操作（loop-back） 82
11.3.9. 线中止信号 82
11.3.10.流量控制 82
11.3.11.其他接口 83
11.4. 配置中断 83
11.4.1. 中断寄存器 83
11.4.2. 接口 84
11.4.3. 接收 full 中断 84
11.4.4. 接收溢出中断 85
11.4.5. 接收超时中断 tout 85
11.4.6. 发送 fifo 空中断 86
11.4.7. 错误检测类中断 86
11.4.8. 流量控制状态中断 87

11.5. 中断处理函数示例流程 88
11.6. 关于屏蔽上电打印....88
12.PWM 接口说明 90
12.1. 功能综述....90
12.1.1. 特性描述....90
12.1.2. 实现方式....90
12.1.3. 配置说明....91
12.1.4. 参数说明....91
12.2. pwm.h 详解 91
12.2.1. 代码示例....91
12.2.2. 接口说明....92
12.3. 自定义通道 94
13.IR 红外例程及使用说明....96
13.1. 红外发送与接收使用说明....96
13.2. 参数配置....96
13.3. 例程说明....98
13.4. 硬件连接....99
13.5. 实验结果....99
14.Sniffer 应用设计说明....102
14.1. Sniffer 模式介绍....102
14.2. Sniffer 的应用场景和相关问题....106
14.3. 手机 APP 设计....107
14.4. IOT-device 上固件设计 107
附录 108

1. 概述

1.1. 通用输入 / 输出接口（GPIO）

ESP8266EX 共有 17 个 GPIO 管脚，通过配置适当的寄存器可以给它们分配不同的功能。

每个 GPIO 都可以配置为内部上拉/下拉，或者被设置为高阻。当被配置为输入时，可通过读取寄存器获取输入值；输入也可以被设置为边缘触发或电平触发来产生 CPU 中断。简言之，IO 管脚是双向、非反相和三态的，带有三态控制的输入和输出缓冲器。

这些管脚可以与其他功能复用，例如 I2C、I2S、UART、PWM、IR 遥控等。

1.2. 安全数字输入 / 输出接口（SDIO）

ESP8266EX 拥有 1 个从机 SDIO 接口，接口管脚定义如下表 1-1 所示。支持 4 位 25 MHz SDIO v1.1 和 4 位 50 MHz SDIO v2.0。

表 1-1. SDIO 管脚定义

管脚名称	管脚编号	IO	功能名称
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. 串行外设接口（SPI / HSPI）

ESP8266EX 拥有 1 个通用从机 / 主机 SPI，1 个从机 SDIO / SPI，和 1 个通用从机 / 主机 HSPI。所有接口的功能均由硬件实现。接口定义如下所示。

1.3.1. 通用 SPI（主机 / 从机）

表 1-2. SPI 接口定义

管脚名称	管脚编号	IO	功能名称
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

说明：

SPI 模式可由软件编程实现。时钟频率最大为 *80 MHz*。

1.3.2. HSPI（从机）

表 1-3. HSPI（从机）管脚定义

管脚名称	管脚编号	IO	功能名称
MTMS	9	IO14	HSPICLK
MTDI	10	IO12	HSPIQ/MISO
MTCK	12	IO13	HSPID/MOSI
MTDO	13	IO15	HPSICS

1.4. I2C 接口

ESP8266EX 拥有 1 个 I2C 接口，用于连接微控制器以及外围设备，如传感器等。I2C 接口定义如表 1-4 所示。

表 1-4. I2C 管脚定义

管脚名称	管脚编号	IO	功能名称
MTMS	9	IO14	I2C_SCL
GPIO2	14	IO2	I2C_SDA

ESP8266EX 既支持 I2C 主机也支持 I2C 从机功能。I2C 接口功能可由软件编程实现，时钟频率最高约为 100 kHz，需高于从设备最慢速的时钟频率。

1.5. I2S 接口

ESP8266EX 拥有 1 个 I2S 输入接口和 1 个 I2S 输出接口。I2S 主要用于音频数据采集、处理和传输，也可用于串行数据输入输出，如支持 LED 彩灯（WS2812 系列）。I2S 管脚定义如表 1-5 所示。I2C 接口功能可以使用复用 GPIO 通过软件编程实现，支持链表 DMA。

表 1-5. I2S 管脚定义

I2S 数据输入			
管脚名称	管脚编号	IO	功能名称
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. 通用异步收发器（UART）

ESP8266EX 拥有两个 UART 接口，分别为 UART0 和 UART，接口定义如表 1-6 所示。

表 1-6. UART 管脚定义

管脚类型	管脚名称	管脚编号	IO	功能名称
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

2 个 UART 接口的数据传输均由硬件实现。数据传输速度可达 115200*40（4.5 Mbps）。

UART0 可以用做通信接口，支持流控。由于 UART1 目前只有数据传输功能，所以一般用作打印 log。

> 📖 **说明：**
> *UART0* 默认会在上电启动期间输出一些打印，此期间打印内容的波特率与所用的外部晶振频率有关。使用 *40 MHz* 晶振时，该段打印波特率为 *115200*；使用 *26 MHz* 晶振时，该段打印波特率为 *74880*。如果打印信息影响设备功能，建议在上电期间将 *U0TXD*、*U0RXD* 分别与 *U0RTS(MTDO)*，*U0CTS(MTCK)* 交换，以屏蔽打印。

1.7. 脉冲宽度调制（PWM）

ESP8266EX 拥有 4 个 PWM 输出接口，如表 1-7 所示。用户可自行扩展。

表 1-7. PWM 管脚定义

管脚名称	管脚编号	IO	功能名称
MTDI	10	IO12	PWM0
MTDO	13	IO15	PWM1
MTMS	9	IO14	PWM2
GPIO4	16	IO4	PWM3

PWM 接口功能由软件实现。例如，在 LED 智能照明的示例中，PWM 通过定时器的中断实现，最小分辨率可达 44 ns。PWM 频率的可调节范围为 1000 μs 到 10000 μs，即 100 Hz 到 1 kHz 之间。当 PWM 频率为 1 kHz，占空比为 1/22727，1 kHz 的刷新率下可达超过 14 位的分辨率。

1.8. IR 遥控接口

ESP8266EX 芯片目前定义了 1 个 IR 红外遥控接口，该接口定义如表 1-8 所示。

表 1-8. IR 红外遥控管脚定义

管脚名称	管脚编号	IO	功能名称
MTMS	9	IO14	IR Tx
GPIO5	24	IO5	IR Rx

IR 红外遥控接口由软件实现，接口使用 NEC 编码及调制解调，采用 38 kHz 的调制载波，占空比为 1/3 的方波。传输范围在 1m 左右，传输范围由 2 个因素决定，一个是 GPIO 口 的最大额定电流，另一个是红外接收管内部的限流电阻的大小。电阻越大，电流越小，功耗也越小，反之亦然。传输半角度为 15° 到 30°，取决于红外接收管的辐射方向。

1.9. Sniffer

ESP8266 可以进入混杂模式（sniffer），ESP8266 可以接收完整的 IEEE802.11 包，或者可以获得包的长度。

2. GPIO

2.1. 功能综述

ESP8266 的 16 个通用 IO 的管脚位置和名称如下表所示。

表 2-1. GPIO 管脚定义

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

其中，在四线（QUAD）模式 Flash 下，有 6 个 IO 口用于 Flash 通讯。

在两线（DUAL）模式 Flash 下，有 4 个 IO 口用于与 Flash 通讯。

> 📖 **说明：**
>
> 本说明如果对照以下资料阅读，会更有助理解：
>
> - *"附录 1 – GPIO 寄存器"*
> - 引脚功能复用表：《*ESP8266_Pin_List.xlsx*》：*http://bbs.espressif.com/viewtopic.php?f=21&t=412&p=1545#p1545*。

2.2. GPIO 寄存器说明

2.2.1. GPIO 功能选择寄存器

下面以 ESP8266 的 MTDI 为例，说明 GPIO 功能的选择。

功能选择寄存器 PERIPHS_IO_MUX_MTDI_U（不同的 GPIO，该寄存器不同）

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

此处的 FUNC_GPIO12=3。

不同的 PIN 脚，配置不同。

配置的时候，请参考《ESP8266_Pin_List.xlsx》表格，在该表格的 Digital Die Pin List 页中可以查到通用的 GPIO 以及复用功能，在 Reg 页可以查阅到 GPIO 功能选择相关的寄存器。

Digital Die Pin List 该页中的 FUNCTION 下拉选择项就是功能的配置选项。需要注意的是，如果需要配置为 FUNCTION3，应该往寄存器对应的位中写 2，如果需要配置为 FUNCTION2，应该往寄存器对应的位中写 1，以此类推。

2.2.2. GPIO 输出寄存器

- **输出使能寄存器 GPIO_ENABLE_W1TS**

bit[15:0] 输出使能位（可读写）：

若对应的位被置 1，表示该 IO 的输出被使能。Bit[15:0] 对应 16 个 GPIO 的输出使能位。

- **输出禁用寄存器 GPIO_ENABLE_W1TC**

bit[15:0] 输出禁用位（可读写）：

若对应的位被置 1，表示该 IO 的输出被禁用。Bit[15:0] 对应 16 个 GPIO 的输出禁用位。

- **输出使能状态寄存器 GPIO_ENABLE**

Bit[15:0] 输出使能状态位（可读写）：

该寄存器的 bit[15:0] 的值，反映的是对应的 PIN 脚输出使能状态。

GPIO_ENABLE 的 bit[15:0] 通过给 GPIO_ENABLE_W1TS 的 bit[15:0] 和 GPIO_ENABLE_W1TC 的 bit[15:0] 写值来控制。例如 GPIO_ENABLE_W1TS 的 bit[0] 置 1，则 GPIO_ENABLE 的 bit[0]=1。GPIO_ENABLE_W1TC 的 bit[1] 置 1，则 GPIO_ENABLE 的 bit[1]=0。

- **输出低电平寄存器 GPIO_OUT_W1TC**

bit[15:0] 输出低电平位（只写寄存器）：

若对应的位被置 1，表示该 IO 的输出为低电平（同时需要使能输出）。Bit[15:0] 对应 16 个 GPIO 的输出状态。

> 📖 **说明：**
> 如果需要将该 *PIN* 配置为高电平，需要配置 *GPIO_OUT_W1TS* 寄存器。

- **输出高电平寄存器 GPIO_OUT_W1TS**

bit[15:0] 输出高电平位（只写寄存器）：

若对应的位被置 1，表示该 IO 的输出为高电平（同时需要使能输出）。Bit[15:0] 对应 16 个 GPIO 的输出状态。

> 📖 **说明：**
> 如果需要将该 *PIN* 配置为低电平，需要配置 *GPIO_OUT_W1TC* 寄存器。

- **输出状态寄存器 GPIO_OUT**

bit[15:0] 输出状态位（读写寄存器）：

该寄存器的 [15:0] 的值，反映的是对应 PIN 脚输出的状态。

GPIO_OUT 的 bit[15:0] 是由 GPIO_OUT_W1TS 的 bit[15:0] 和 GPIO_OUT_W1TC 的 bit[15:0] 共同决定的。例如：GPIO_OUT_W1TS 的 Bit[1]=1，那么 GPIO_OUT[1]=1。GPIO_OUT_W1TC 的 Bit[2]=1，那么 GPIO_OUT[2]=0。

2.2.3. GPIO 输入寄存器

输入状态寄存器 GPIO_IN

bit[15:0] 输入状态位（可读写）：

若对应的位为 1，表示该 IO 的引脚状态为高电平，若对应的位为 0 表示该 IO 的引脚状态低电平。Bit[15:0] 对应 16 个 GPIO 的输入状态位。

> 📖 **说明：**
> *GPIO* 的输入检测功能，是缺省设置，无需使能。

2.2.4. GPIO 中断寄存器

- **中断类型寄存器 GPIO_PIN12（不同的 GPIO 该寄存器不同）**

bit[9:7]（可读写）：

0：该 GPIO 的中断禁用

1：上升沿触发中断

2：下降沿触发中断

3：双沿触发中断

4：低电平

5：高电平

- **中断状态寄存器 GPIO_STATUS**

Bit[15:0]（可读写）：

若对应的位被置 1，表示该 IO 中断发生。Bit[15:0] 对应 16 个 GPIO。

- **清中断寄存器 GPIO_STATUS_W1TC**

Bit[15:0]（可读写）：

向对应的位写 1，对应的 GPIO 的中断状态就会被清除。

2.2.5. GPIO16 对应接口

与其他 IO 口不同，GPIO16(XPD_DCDC) 不属于通用 GPIO 模块，它属于 RTC 模块，可以用来在深度睡眠时候唤醒整个芯片，可以配置为输入或者输出模式，但无法触发 IO 中断。使用接口如下：

- `gpio16_output_conf(void)`

将 GPIO16 配置为输出模式。

- `gpio16_output_set(uint8 value)`

从 GPIO16 输出高电平 / 低电平，需要先配置为输出模式。

- `gpio16_input_conf(void)`

将 GPIO16 配置为输入模式。

- `gpio16_input_get(void)`

读取 GPIO16 的输入电平状态，需要先配置为输入模式。

2.3. 参数配置

在参数配置过程中，给出 3 个应用场景。用户可以此应用场景为例，配置其他的 GPIO。

应用场景：

- 配置 MTDI 输出高电平，并使能其上拉。
- 配置 MTDI 为输入模式，并获取其电平状态。
- 配置 MTDI 为下降沿触发中断。

2.3.1. 应用场景 1 参数配置

1. 配置 MTDI 为 GPIO 模式

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

该语句的作用是向 PERIPHS_IO_MUX_MTDI_U 寄存器的第 4 位、第 5 位写 1。PERIPHS_IO_MUX_MTDI_U 寄存器的第 4 位和第 5 位置 1 表示将 MTDI 配置为 GPIO 功能。关于 PERIPHS_IO_MUX_MTDI_U 寄存器，请参阅第 2.2 节 GPIO 寄存器说明。

2. 配置 MTDI 输出高电平

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

该语句有两个作用：

- 向 GPIO_ENABLE_W1TS 的寄存器第 12 位写 1，该位置 1 表示使能 MTDI 的输出功能。
- 向 GPIO_OUT_W1TS 的寄存器第 12 位写 1，该位置 1 表示将 MTDI 输出为高电平。

备注：需要 MTDI 配置输出低电平，将该函数的第 2 个参数设置为 0 即可。

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

该语句有两个作用：

- 向 GPIO_ENABLE_W1TS 的寄存器第 12 位写 1，该位置 1 表示使能 MTDI 的输出功能。
- 向 GPIO_OUT_W1TC 的寄存器第 12 位写 1，该位置 1 表示将 MTDI 输出为低电平。

3. 使能 MTDI 上拉

```
PIN_PULLUP_EN(PERIPHS_IO_MUX_MTDI_U);
```

该语句作用是向 PERIPHS_IO_MUX_MTDI_U 的第 7 位写 1。该位置 1 表示使能 MTDI 的上拉功能。

> 📖 **说明：**
> 如果需要关闭 *MTDI* 的上拉功能，请使用如下语句。

```
PIN_PULLUP_DIS(PERIPHS_IO_MUX_MTDI_U);
```

2.3.2. 应用场景 2 参数配置

1. 配置 MTDI 为 GPIO 模式

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

该语句的作用是向 PERIPHS_IO_MUX_MTDI_U 寄存器的第 4 位、第 5 位写 1。PERIPHS_IO_MUX_MTDI_U 寄存器的第 4 位和第 5 位置 1 表示将 MTDI 配置为 GPIO 功能。

2. 配置 MTDI 为输入模式

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

3. 获取 MTDI 管脚的电平状态

Uint8 level=0;

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

GPIO_INPUT_GET(GPIO_ID_PIN(12)) 语句实际是获取 GPIO_IN 寄存器第 12 位的状态，该寄存器的值反映的是对应的管脚的输入电平（必须使能对应的管脚的输入功能，该寄存器的状态才有效）。

> 📖 **说明：**
>
> - 如果 *MTDI* 的电平为高电平，那么 *GPIO_INPUT_GET* 的返回值为 *1*，*level=1;*
> - 如果 *MTDI* 的电平为低电平，那么 *GPIO_INPUT_GET* 的返回值为 *0*，*level=0;*

2.3.3. 应用场景 3 参数配置

```
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;
```

该结构体用来配置 GPIO 的中断触发方式，该结构体在 ***gpio.h*** 中声明。

1. 配置 MTDI 为 GPIO 模式

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



该语句的作用是向 PERIPHS_IO_MUX_MTDI_U 寄存器的第 4 位、第 5 位写 1。PERIPHS_IO_MUX_MTDI_U 寄存器的第 4 位和第 5 位置 1 表示将 MTDI 配置为 GPIO 功能。

2. 配置 MTDI 为输入模式

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

3. 禁止所有的 IO 中断

```
ETS_GPIO_INTR_DISABLE();
```

4. 设置中断处理函数

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

5. 配置 MTDI 为下降沿中断触发的方式

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

该语句的作用是向 GPIO_PIN12 的寄存器的 [9:7] 位写入 0x02，向该位域内写入 0x02，表示配置为下降沿中断。

> 📖 **说明：**
> 若需要禁用 *MTDI* 的中断功能，只需要将向 *GPIO_PIN12* 的寄存器的 *[9:7]* 位写入 *0x00* 即可。如需配置为其他的中断触发模式，请参阅“第 *2.2* 节 *GPIO* 寄存器说明”。

6. 使能 GPIO 中断

```
ETS_GPIO_INTR_ENABLE();
```

2.3.4. 中断函数处理流程说明

1. 清除该中断

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

GPIO_STATUS和GPIO_STATUS_W1TC 说明请参阅“第 2.2 节 GPIO 寄存器说明”。

2. 判断是哪个 IO 触发的中断（当有多个 IO 都配置为中断方式时）

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

3. 如果是双沿中断，应该判断此次中断为上升沿中断还是下降沿中断。

```
if(!GPIO_INPUT_GET(GPIO_ID_PIN(12))) //如果MTDI此次触发方式为下降沿
```

2.3.5. 中断函数处理流程示例

```
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. SDIO 通信 SPI 兼容模式

3.1. 功能综述

该协议使用 ESP8266 的 SDIO 模块与其他的处理器的 SPI 主机进行通信。在电气接口方面，协议通过 4 路信号线实现，包括 SPI 协议中的 SCLK、MOSI、MISO（注意没有 CS 信号）与 1 路中断信号。

要使用 EPS8266 SDIO 通信，其程序下载方式与常规情况有所不同。由于 ESP8266 启动时，默认读取程序的 SPI 接口与 SDIO 接口复用相同的芯片管脚，因此要使用 SDIO 模块通信协议，ESP8266 需要以 SDIO 模式下启动，随后主机通过 SDIO 下载部分的程序到 ESP8266 的 RAM 中以启动芯片，而大部分直接由 CPU CACHE 调用 FLASH 的程序则可以事先用烧录工具写入与 HSPI 接口相连接的 FLASH 芯片中。

ESP8266 SDIO 的接收发送的数据直接由内部支持链表检索的 DMA 模块操作。

ESP8266 可以不通过 CPU 参与，高效地通过内存映射链表的地址完成 SDIO 数据包的收发。

3.2. DEMO 实现方案

3.2.1. 平台介绍

通信主机端是以 STM32F103ZET6 为核心的红龙开发板，软件由 IAR 平台开发使用 FreeRTOS 操作系统。从机为 ESP_IOT Reference board，基于 v0.9.3 的 SDK 开发。

3.2.2. ESP8266 软件编译与下载准备

- 将 ESP8266 的 DEMO 工程在 ***SDIO communication demo\esp_iot_sdk_v0.9.3_sdio_demo \app*** 通过编译器编译并生成下载使用的 BIN 文件。
- 注意 ***SDIO communication demo\esp_iot_sdk_v0.9.3_sdio_demo\lib*** 中的 ***libmain.a*** 与 v0.9.3 release 版本中的不同，如果使用 release 版本的 SDK，需要用 DEMO 中的 ***libmain.a*** 替换掉原来的，修改版的 ***libmain.a*** 会使芯片启动后，将读取 Flash 程序的 SPI 模块与原 HSPI 的映射管脚互相交换。使用 DEMO 工程直接编译生成即可。
- 将 ***SDIO communication demo\esp_iot_sdk_v0.9.3_sdio_demo\bin*** 中的 ***eagle.app.v6.irom0text.bin*** 复制到 ***SDIO communication demo\XTCOM_UTIL*** 目

录下，***eagle.app.v6.irom0text.bin*** 为 ESP8266 程序中由 CPU CACHE 直接通过 SPI 读取 Flash 芯片的所有函数。

- 运行 ***SDIO communication demo*** 中的 ***BinToArray.exe*** 选择 ***SDIO communication demo\esp_iot_sdk_v0.9.3_sdio_demo\bin*** 中的 ***eagle.app.v6.flash.bin*** 将其转换为 ANSI C 格式的数组。转换后的文件会保存在 D:\ 中，***BinToArray.exe*** 的目标路径一定是 D 盘不能修改。如果没有 D 盘，只能用有 D 盘的虚拟机，或连个 U 盘名叫 D，或分一个 D 盘，或者直接去网上找个 BIN 转数组的工具。
- 假如电脑有 D 盘，把 D:\ 中的 ***hexarray.c*** 以 ***eagle_fw.h*** 命名，并且把数组名定义为 ***const unsigned char eagle_fw[] =.....***，替换掉 ***SDIO communication demo\STM32\ Eagle_Wifi_Driver\ egl_drv_simulation*** 中的 ***eagle_fw.h***，（可以先从老的 ***eagle_fw.h*** 中复制出数组名和文件名，修改 ***hexarray.c*** 后替换掉老的 ***eagle_fw.h***）。***eagle.app.v6.flash.bin*** 是芯片在启动前先要载入 ESP8266 内存的部分，这里需要转成数组，通过 STM32 写入 ESP8266。
- 用 IAR 平台打开 ***SDIO communication demo\STM32\IAR*** 中的 ***EglWB.ewp.eww***，编译工程。

3.2.3. ESP8266 FLASH 端软件下载

1. 用串口线连接 ESP_IOT Reference board 和电脑，连上 5V 电源。板子上 J67 连接右边的两针（使能 HSPI 接口上的 FLASH 芯片），J66 连接左边的两针（断开 SPI 接口上的 FLASH 芯片）。将 MTD0、GPIO0、GPIO2 这 3 个跳线设置为：0、0、1（上、上、下）的 UART 模式。
2. 双击 ***SDIO communication demo\XTCOM_UTIL*** 目录下 ***XTCOM_UTIL.exe***，点击 ***Tools -> Config Device*** 选择 Com 口，Baud Rate：115200，点击 ***Open***，出现 open Success，点击 ***Connect***，然后按 H Flash 板电源，出现连接成功。
3. 点击 ***API TEST(A)->(5) HSpiFlash Image Download***，选择 ***SDIO communication demo\XTCOM_UTIL*** 目录下的 ***eagle.app.v6.irom0text.bin*** 然后 Offset：0x40000，点击 DownLoad，下载完成。

3.2.4. ESP8266 FLASH 端软件下载

使用排线针连接 ESP_IOT Reference board 和红龙开发板具体连线如下：

红龙开发板 JP1 中：

ESP_IOT 板中 J62 排针从下往上数

GND	->	1	VSS/GND
SPI_CLK	->	4	SDIO_CLK
SPI_MOSI	->	5	SDIO_CMD
SPI_MISO	->	3	SDIO_DAT0
IRQ	->	2	SDIO_DAT1

ESP_IOT Reference board 上：将跳线 MTD0 换到 1（下方两针短接），GPIO0、GPIO2 任意（1，x，x 为 SDIO 启动模式），CHIP_PD:ON（开关拨在下方），保持跳线 J66 连接左边两针，跳线 J67 连接右边两针。

5V 电源适配器连接 ESP_IOT Reference board 和红龙开发板。打开红龙开发板电源，在 IAR 环境中将之前 3.2.2 节中编译完成的工程下载到 STM32 中。启动 STM32 程序，打开 ESP_IOT Reference board 电源。STM32 会先向 ESP8266 写入启动程序，几秒后自动运行 SDIO 返回测试程序。

3.3. ESP8266 端软件说明

3.3.1. 协议原理：SDIO 中断线行为与 SDIO 状态寄存器

ESP8266 的 SD_DATA1 管脚在 SDIO 运行于 SPI 兼容模式时作为通知 SPI 主机的中断线，且为低电平有效。当 ESP8266 中 SDIO 状态寄存器被软件更新时，中断线由高电平变为低电平，需要主机通过 SDIO 写入恢复中断线。（具体为：主机需要通过 CMD53 或 52 命令向地址为 `0x30` 的寄存器写入 1 来使中断线恢复高电平。）

SDIO 状态寄存器为 32 bits，由 ESP8266 软件修改，并可以主机由 CMD53 或 52 命令读取，地址为 `0x20` - `0x23`，其数据结构被定义为：

```
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;
};
```

其中：

- wr_busy，bit0：1 表示从机写缓存满，ESP8266 正在处理主机发送的数据，0 表示写缓存空可以进行一次写入操作。
- rd_empty，bit1：1 表示从机读缓存为空，没有新数据更新，0 表示读缓存有新数据需要主机读取。
- comm_cnt，bit2-4：读写通信计数。每次 ESP8266 SDIO 模块完成一次有效读/写数据包操作时，计数值会加 1，主机可以由此判断一次读/写数据通信是否已经被 ESP8266 有效响应。
- intr_no，bit5-7：协议最终未使用该变量，保留。
- rx_length，bit8-23：读缓存中，所准备数据包的实际长度。
- res，bit24-31：保留。

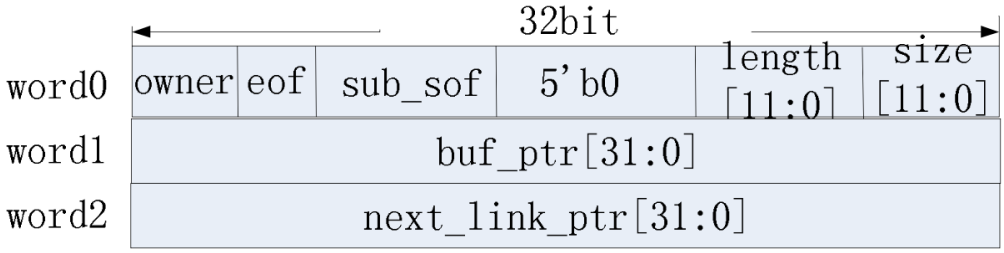
因此，主机通信流程为：

- 在收到中断请求后，先读取 SDIO 状态寄存器，再清除中断，并根据状寄存器来进行读写数据包的操作。
- 定时轮询 SDIO 状态寄存器，根据状态寄存器来进行读写数据包的操作。

3.3.2. 读写缓存与注册链表的使用说明

ESP8266 的 SDIO 收发数据包直接会被 DMA 传输到对应的内存。ESP8266 软件中会定义链表注册结构体（或数组）以及一个（或多个）缓存空间，本例中只使用一个缓存空间链表也只有一个元素。将缓存的首地址写入链表注册结构体，并完成其他信息，在将链表结构体首地址写入 ESP8266 对应硬件寄存器，DMA 就能自动操作 SDIO 与缓存空间。

表注册结构体具体为：



- owner：1'b0：当前 link 对应 buffer 的操作者为 SW；当前 link 对应 buffer 的操作者。MAC 不使用该 bit。1'b1：当前 link 对应 buffer 的操作者为 HW；
- eof：帧结束标志（对于 AMPDU 的子帧结束，该标识是不会置上的）。在 MAC 发送帧时用于指示帧结束 link。对于 eof 位置上的 link，它的 buffer_length[11:0] 必须等于该帧剩下的长度，否者 MAC 会上报 error。在 MAC 接收帧时用于指示帧已接收完成。此时该值由硬件置上。

- sub_sof：子帧起始 link 标识，区分 AMPDU 帧内的不同子帧，仅用于 MAC 发送时。
- length[11:0]：buffer 实际占用的大小。
- size[11:0]：buffer 的总大小。
- buf_ptr[31:0]：buffer 的起始地址。
- next_link_ptr[31:0]：下 1 个 discripter 的起始地址。在 MAC 接收帧时该值为 0，表示已无空 buffer 用于接收。

3.3.3. ESP8266 DEMO 中提供的 API 函数

1. `void sdio_slave_init(void)`

功能：SDIO 模块初始化，其中包括状态寄存器初始化，RX 和 TX 注册链表初始化，通信中断线的模式配置，收发中断的配置与注册中断服务程序等。

2. `void sdio_slave_isr(void *para)`

功能与触发条件：SDIO 中断处理函数，SDIO 正确接收或发送了一个数据包后就会触发该中断程序。在 DEMO 中，ESP8266 所有的测试操作都在中断处理函数中完成。通信过程中所有对注册链表，状态寄存器，数据处理都可以在该函数中找到。

3. `void rx_buff_load_done(uint16 rx_len)`

功能：`rx_buffer` 装入新数据包后，必须调用该函数使新数据变为待读取状态。该函数包含了注册链表软硬件相关操作，与状态寄存器相关操作。DEMO 中该函数在中断服务程序中被调用。

参数：rx_len 新数据包实际长度，以字节为单位。

4. `void tx_buff_handle_done(void)`

功能：`tx_buffer` 中数据处理完成后，必须调用该函数使 SDIO 变为可发送状态已准备接受下个数据包。该函数包含了注册链表软硬件相关操作，与 状态寄存器相关操作。DEMO 中该函数在中断服务程序中被调用。

5. `void rx_buff_read_done(void)`

功能：rx_buffe 中数据被读取后，必须调用该函数使 SDIO 变为不可读状态。该函数包含了状态寄存器相关操作。函数应在 `RX_EOF` 中断服务的一开始调用。

6. `void tx_buff_write_done(void)`

功能：tx_buffer 收到新数据包时，必须调用该函数 SDIO 变为不可写状态。该函数包含了状态寄存器相关操作。函数应在 TX_EOF 中断服务的一开始调用。

7. `TRIG_TOHOST_INT()`

功能：宏，将通信中断线拉低，通知主机。

8. 其他函数

其他函数为测试使用。

3.4. STM32 端软件说明

3.4.1. 主要函数说明

1. `void SdioRW(void *pvParameters)`

功能：

SDIO 测试线程，其中包含了所有读写的操作流程。

位置：

egl_thread.c 中由同文件中的函数 `SPITest()` 中注册。

参数：

未使用。

2. `int esp_sdio_probe(void)`

功能：

ESP8266 启动程序下载相关程序。

位置：

esp_main_sim.c 中，由 ***egl_thread.c*** 中的函数 `SPITest()` 调用。

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

功能：

SDIO byte 模式写入 API，封装了 CMD53 Byte 模式写入功能，可以对寄存器或数据包进行操作。SDIO 协议规定最大有效数据长度是 512 字节。

位置：

port_spi.c 中，由 ***egl_thread.c*** 中的函数 `SdioRW` 调用。

参数：

`src`：发送数据包首地址。

`count`：发送长度。字节为单位

`func`：功能号，目前除了使用修改 SDIO CMD53 的 block 模式中的 `block_size` 大小通信用的是 0，其余所有通信都使用 1。

addr：写入的目标首地址。如果操作寄存器直接输入对应地址，如 0x30 中断线清除寄存器，0x110 修改 block_size（func 为 0）；注意如果操作数据包，要输入 0x1f800-tx_length 的数值，且 tx_length 与 count 相等，如果 count>tx_length，SPI 主机会发送count 长度的数据包，但是从第 tx_length+1 到 count 的数据会被 ESP8266 的 SDIO 模块丢弃。因此，在发送数据包是 addr 关系到实际传输有效数据的长度。

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

功能：

SDIO Byte 模式读取 API，封装了 CMD53 Byte 模式读取功能，可以对寄存器或数据包进行操作。SDIO 协议规定最大有效数据长度是 512 字节。

位置：

port_spi.c 中，由 ***egl_thread.c*** 中的函数 SdioRW 调用。

参数：

dst：接收缓存首地址。

count：接收长度。字节为单位。

func：功能号，目前除了使用读取 SDIO CMD53 的 block 模式中的 block_size 大小通信用的是 0，其余所有通信都使用 1。

addr：读取的目标首地址。如果操作寄存器直接输入对应地址，如 0x20 为 SDIO 状态寄存器；注意如果操作数据包，要输入 0x1f800-tx_length 的数值，且 tx_length 与 count 相等，如果 count>tx_length，SPI 主机会读取 count 长度的数据包，但是从第 tx_length+1 到 count 的数据会被 ESP8266 的 SDIO 模块丢弃读出无效数据。因此，在发送数据包是 addr 关系到实际传输有效数据的长度。

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

功能：

sdio block 模式写入 API，封装了 CMD53 block 模式写入功能，只能传输数据包。SDIO 协议规定最大有效数据长度是 512 个 block。

位置：

port_spi.c 中，由 egl_thread.c 中的函数 SdioRW 和 esp_main_sim.c 中程序下载器所使用的函数 sif_io_sync 调用。

参数：

src：发送数据包首地址。

count：发送长度。以 block 为单位

block_size：1 个 block 有多少字节，注意必须与 fund 为 0 addr 为 0x110-111 中的 16 bit 值相同。一般在 SDIO 初始化时需要配置 ESP8266 SDIO 的 block_size。DEMO 的初始值为 512。在运行过程中，有配置为 1024。block_size 必须为 4 的整数倍。

addr：写入的目标首地址。与 Byte 模式相同输入 0x1f800-tx_length 的数值，且 tx_length 与 count 相等。

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

功能：

sdio block 模式写入 API，封装了 CMD53 block 模式写入功能，只能传输数据包。SDIO 协议规定最大有效数据长度是 512 个 block。

位置：

port_spi.c 中，由 ***egl_thread.c*** 中的函数 SdioRW 和 esp_main_sim.c 中程序下载器所使用的函数 sif_io_sync 调用。

参数：

src：接收缓存首地址。

count：接收长度。以 block 为单位

block_size：一个 block 有多少字节，注意必须与 func 为 0 addr 为 0x110-111 中的 16 bit 值相同。一般在 SDIO 初始化时需要配置 ESP8266 SDIO 的 block_size。DEMO 的初始值为 512。在运行过程中，有配置为 1024。block_size 必须为 4 的整数倍。

addr：读取的目标首地址。与 Byte 模式相同输入 0x1f800-tx_length 的数值，且 tx_length 与 count 相等。

7. `void EXTI9_5_IRQHandler(void)`

功能：

通信中断处理函数为线程函数 SdioRW 中的函数 egl_arch_sem_wait (&BusIrqReadSem,1000) 提供使能信号，使 SdioRW 线程跳出等待读取 SDIO 状态寄存器。

位置：

spi_cfg.c

4. SPI 模块使用说明

4.1. 概述

4.1.1. 功能综述

ESP8266 SPI 模块用于与各种支持 SPI 协议的设备进行通信，在电气接口方面，支持 SPI 协议标准的 4 线通信（CS、SCLK、MOSI、MISO）。与普通 SPI 模块不同的是，ESP8266 SPI 模块对 SPI 接口的 FLASH 存储器做了特殊的支持。因此 ESP8266 SPI 模块的主机与从机模式都有着相应的固定硬件控制协议，与之通信的 SPI 设备需要做对应的匹配处理。

4.1.2. SPI 特点

- 支持标准的主机（Master）和从机（Slave）模式。
- 支持长度可编程的硬件指令（CMD）和地址（ADDR），指令最大 16 位，地址最长 64 位。
- 数据缓冲区最大 64 字节，以 word 对齐。
- Slave 模式下可编程的读写状态寄存器。
- 3 个片选（CS）管脚。
- 主机模式时钟频率高达 80 MHz，从机模式钟频率最高为 20 MHz。
- 可选的时钟极性。
- 可选的 MSB（最高有效位）和 LSB（最低有效位）传输。
- 可选的数据缓冲区高字节或低字节传输。
- 传输结束、读写数据、读写状态多个中断源可选择。

4.2. ESP8266 SPI 主机协议格式

4.2.1. SPI 主机支持的通信格式

ESP8266 SPI 主机通信格式为命令 + 地址 + 读 / 写数据，具体为：

- 命令：必须存在；长度，1 ～ 16 位；主机输出从机输入（MOSI）。
- 地址：可选；长度，0 ～ 32 位；主机输出从机输入（MOSI）。

- 数据写或读：可选；长度，0 - 512 位（64 字节）；主机输出从机输入（MOSI）或主机输入从机输出（MISO）。

4.2.2. 现有 API 支持的 SPI 主机通信格式

ESP8266 SPI 的 API 函数中给出两个固定的主机初始化模式，一个模式支持大多数以字节单位的常规 SPI 通信，另一个模式专为驱动一种彩色 LCD 屏设计，该设备需要一次 9 位的非标准 SPI 通信格式，详细叙述参见“第 4.4.1 节 SPI 主机 API 函数说明”。

4.3. ESP8266 SPI 从机协议格式

4.3.1. SPI 从机时钟极性配置要求

与 ESP8266 SPI 从机通信的主机设备时钟极性需配置为：空闲低电平，上升沿采样，下降沿变换数据。并且在一次 16 位读 / 写过程中，务必保持片选信号 CS 的低电平，如果在发送过程中 CS 被拉高，从机内部状态将会重置。

4.3.2. SPI 从机支持的通信格式

ESP8266 SPI 从机通信格式与主机模式基本相同为命令 + 地址 + 读 / 写数据，而从机读写操作有固定硬件命令，且地址部分不能去除，具体为：

- 命令：必须存在；长度，3 ～ 16 位；主机输出从机输入（MOSI）。
- 地址：必须存在；长度，1 ～ 32 位；主机输出从机输入（MOSI）。
- 数据写或读：可选；长度，0 ～ 512 位（64 字节）；主机输出从机输入（MOSI）或主机输入从机输出（MISO）。

4.3.3. SPI 从机支持命令定义

从机接收命令长度至少是 3 位且低 3 位对应有固定的硬件读写操作，具体为：

- 010（从机接收）：将主机发送数据通过 MOSI 写入从机数据缓存对应寄存器 SPI_FLASH_C0 至 SPI_FLASH_C15。
- 011（从机发送）：将从机缓存对应寄存器 SPI_FLASH_C0 至 SPI_FLASH_C15 中的数据通过 MISO 发送到主机。
- 110（从机同时收发）：将从机数据缓存发送至 MISO 同时将 MOSI 上的主机数据写入数据缓存 SPI_FLASH_C0 至 SPI_FLASH_C15。

> ⚠ **注意：**
> 其余数值用于读写 *SPI* 从机的状态寄存器 *SPI_FLASH_STATUS*，由于其通信格式与读写数据缓存不同，会造成从机读写错误，请勿使用。

4.3.4. 现有 API 支持的 SPI 从机通信格式

ESP8266 SPI 的 API 函数中给出一个固定的从机初始化模式，该模式为兼容大多数以字节为单位的设备，将从机通信格式设定为：7 位命令 + 1 位地址 + 8 位读 / 写数据。这样其他 SPI 主机设备进行一次 16 位（或两次 8 位且 CS 需要保持低电平）的通信就可以对 ESP8266 的 SPI 从机进行一字节的读或写操作。详细参见“第 4.4.2 节 SPI 主机 API 函数说明”。

4.4. SPI 模块 API 函数说明

4.4.1. SPI 主机 API 函数说明

1. `void spi_lcd_mode_init(uint8 spi_no)`

功能：

为驱动 TM035PDZV36 彩色液晶屏提供的 SPI 主机初始化程序。

参数	说明
`uint8 spi_no`	所使用 SPI 模块号，只能输入 SPI（0）与 HSPI（1）其他输入无效。

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

功能：

为驱动 TM035PDZV36 彩色液晶屏提供的 SPI 主机发送函数，该液晶模块需要一次需要 9 位传输。

参数	说明
`uint8 spi_no`	所使用 SPI 模块号，只能输入 SPI 与 HSPI，其他输入无效。
`uint8 high_bit`	第 9 位数据，0 代表第 9 位为 0，其余数据都表示第 9 位为 1。
`uint8 low_8bit`	低 8 位数据。

3. `void spi_master_init(uint8 spi_no)`

功能：

常规 SPI 主机初始化函数，波特率为 CPU 时钟的 4 分频，初始化后可以使用除 `spi_lcd_9bit_write` 以外的所有主机函数。

参数	说明
uint8 spi_no	所使用 SPI 模块号，只能输入 SPI 与 HSPI，其他输入无效。

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

功能：

完成基本的一字节的主机发送。

参数	说明
uint8 spi_no	所使用 SPI 模块号，只能输入 SPI 与 HSPI，其他输入无效。
uint8 data	8 位发送数据。

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

功能：

对 ESP8266 的 SPI 从机写入 1 字节数据。

由于从机设置为：7 位命令 + 1 位地址 + 8 位数据，因此完成发送需要一次 16 位的传输并且第一字节的固定为 0b0000010+0（原理参见“第 4.3.3 节”）即 0x04，第 2 字节为发送数据 data。实际发送波形如图 4-1 所示。

参数	说明
uint8 spi_no	所使用 SPI 模块号，只能输入 SPI 与 HSPI，其他输入无效。
uint8 data	8 位发送数据。



图 4-1. spi_byte_write_espslave 写入 ESP8266 从机波形

> 📖 **说明：**
>
> 黄线 *CS*，蓝线 *CLK*，红线 *MOSI*，绿线 *MISO*。

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

功能：

对 ESP8266 的 SPI 从机读取 1 字节数据，也可以读取其他 SPI 从机设备。

由于 ESP8266 从机设置为：7 位命令 + 1 位地址 + 8 位数据，因此完成发送需要一次 16 bits 的传输并且第一字节的固定为 0b0000011+0（原理参见“第 4.3.3 节”）即 0x06，第 2 字节为接收数据。

实际运行波形如图 4-2 所示。

对于其他全双工 SPI 从机设备，需要设置从机完成一次 16 位的通信，并将有效数据放置在从机发送缓存的第 2 个字节，该字节会被 ESP8266 主机接收。

参数	说明
`uint8 spi_no`	所使用 SPI 模块号，只能输入 SPI 与 HSPI，其他输入无效。
`uint8* data`	8 位接收数据所在内存地址。



图 4-2. spi_byte_read_espslave 读取 ESP8266 从机波形

> 📖 **说明：**
> 黄线 *CS*，蓝线 *CLK*，红线 *MOSI*，绿线 *MISO*。

4.4.2. SPI 主机 API 函数说明

1. `void spi_slave_init(uint8 spi_no)`

功能：

SPI 从机模式初始化，将 IO 口配置为 SPI 模式，启用 SPI 传输中断，并注册 `spi_slave_isr_handler` 函数。

通信格式设定为 7 位命令 + 1 位地址 + 8 位读 / 写数据。命令与地址组成高 8 位，且地址位必须为 0，根据 4.3.3 节叙述，支持：0x04 主机写从机读，0x06 主机读从机写，0x0c 主从同时读写，3 个主机命令。

通信波形参见图 4-1、4-2。

参数	说明
spi_no	SPI 模块的序号，ESP8266 处理器有两组功能相同的 SPI 模块，分别为 SPI 和 HSPI。 可选配的值：SPI 或 HSPI。

2. `spi_slave_isr_handler(void *para)`

功能与触发条件：

SPI 中断处理函数，主机如正确进行了传输操作（读或写从机），中断就会触发。

代码：

```
      //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,
        }
```

代码说明：由于 SPI 用于读写程序存储 Flash 芯片，因此使用 HSPI 进行通信。对于 ESP8266 处理器而言应该有多个设备公用此中断函数，包括 SPI 模块，HSPI 模块，I2S 模块，寄存器 0x3ff00020 的第 4 位、第 7 位、第 9 位分别对应。

SPI 模块会频繁触发传输中断，因此需要关闭其 5 个中断源使能，对应代码如下：

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

而在 HSPI 触发的情况下，需要软件清零其 5 个中断源以避免反复触发中断函数。代码对应如下：

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

而接收和发送数据都共用一个寄存器 SPI_FLASH_C0。其中读出寄存器对应代码：

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

`recv_data` 为全局变量。在该语句之后可以加入用户自定义处理程序。

> ⚠ **注意：**
> 中断程序不适宜执行时间过长的处理代码，因为长时间的中断程序会使看门狗定时器无法正常清零，造成处理器意外重启。

4.5. SPI 接口说明

> ⚠ **注意：**
> 以下内容仅适用于 *Non-OS SDK V1.5.3* 及以上版本。

4.5.1. 数据结构

4.5.1.1. 枚举值

SpiMode

值	说明
SpiMode_Master	主机模式
SpiMode_Slave	从机模式

SpiSubMode

值	说明
SpiSubMode_0	SPI_CPOL（0） SPI_CPHA（0）
SpiSubMode_1	SPI_CPOL（0） SPI_CPHA（1）
SpiSubMode_2	SPI_CPOL（1） SPI_CPHA（0）
SpiSubMode_3	SPI_CPOL（1） SPI_CPHA（1）

SpiSpeed

值	说明
SpiSpeed_0_5MHz	SPI 速率 0.5 MHz
SpiSpeed_1MHz	SPI 速率 1 MHz
SpiSpeed_2MHz	SPI 速率 2 MHz
SpiSpeed_5MHz	SPI 速率 5 MHz
SpiSpeed_8MHz	SPI 速率 8 MHz
SpiSpeed_10MHz	SPI 速率 10 MHz

SpiBitOrder

值	说明
SpiBitOrder_MSBFirst	首先传输 MSB
SpiBitOrder_LSBFirst	首先传输 LSB

SpiIntSrc

值	说明
SpiIntSrc_TransDone	传输完成中断标志
SpiIntSrc_WrStaDone	写状态寄存器中断标志
SpiIntSrc_RdStaDone	读状态寄存器中断标志

值	说明
SpiIntSrc_WrBufDone	写数据寄存器中断标志
SpiIntSrc_RdBufDone	读数据寄存器中断标志

SpiPinCS

值	说明
SpiPinCS_0	CS0 管脚
SpiPinCS_1	CS1 管脚
SpiPinCS_2	CS2 管脚

4.5.1.2. 结构体

> 📖 **说明：**
> 相关注意事项请参考《*ESP8266 SDK 编程手册*》。

SpiAttr

SPI 配置参数

```
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

SPI 传输的数据结构体

```
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

SPI 中断配置信息结构体

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

4.5.1.3. 常量

ESP8266 的命令

名称	数值	描述
MASTER_WRITE_DATA_TO_SLAVE_CMD	2	ESP8266 从机模式的写数据命令。
MASTER_READ_DATA_FROM_SLAVE_CMD	3	ESP8266 从机模式的读数据命令。
MASTER_WRITE_STATUS_TO_SLAVE_CMD	1	ESP8266 从机模式的写状态寄存器命令。
MASTER_READ_STATUS_FROM_SLAVE_CMD	4	ESP8266 工作从机模式的读状态寄存器命令。

4.5.2. 接口说明

> 📖 **说明：**
> 相关注意事项请参考《*ESP8266 SDK 编程手册*》。

4.5.2.1. SPIInit

接口描述

SPI 模块的初始化。

接口函数

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

参数	说明
spiNum	[in] 选择 SPI 和 HSPI。
pAttr	[in] 一个指向 SpiAttr 结构体的指针。

返回值

无

> 📖 **说明：**
>
> - 从机模式默认的配置为 *CMD* 长度 *8* 位，*ADDR* 长度 *8* 位，*DATA* 长度 *32* 字节。
> - 暂不支持只有 *DATA* 的传输。
> - 单次传输 *DATA* 的长度最大 *64* 字节。

4.5.2.2. SPIMasterCfgAddr

接口描述

配置地址寄存器。

接口函数

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

参数	说明
spiNum	[in] 选择 SPI 和 HSPI。
addr	[in] 需要设置的地址。

返回值

无

> 📖 **说明：**
>
> - 如果地址长度超过 *32* 位，需要设置 *SPI_WR_STATUS* 寄存器。
> - *ADDR* 先发送 *word* 的高字节。

4.5.2.3. SPIMasterCfgCmd

接口描述

配置 SPI 的命令寄存器。

接口函数

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

参数	说明
spiNum	[in] 选择 SPI 和 HSPI。
cmd	[in] 需要设置的命令值。

返回值

无

> 📖 **说明：**
> *CMD* 长度最大为 *16* 位，先发送 *word* 的低字节。

4.5.2.4. SPIMasterSendData

接口描述

主机模式下根据 pInData 指定的命令地址和数据完成一次传输。

接口函数

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

参数	说明
spiNum	[in] 选择 SPI 和 HSPI。
pInData	[in] 一个指向 SpiData 结构体的指针，需要指明传输的命令、地址和数据的缓冲区和长度。

返回值

- 0：成功
- 其他：失败

> 📖 **说明：**
> *DATA* 先发送 *word* 的低字节。

4.5.2.5. SPIMasterRecvData

接口描述

主机模式接收数据。

接口函数

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

参数	说明
spiNum	[in] 选择 SPI 和 HSPI。
pOutData	[in] 一个指向 SpiData 结构体的指针，需要指明传输的命令、地址和数据的缓冲区和长度。

返回值

- 0：成功
- 其他：失败

4.5.2.6. SPISlaveSendData

接口描述

向 W8 ~ W15 装载主机需要发送的数据。

接口函数

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

参数	说明
spiNum	[in] 选择 SPI 和 HSPI。
pInData	[in] 一个指向缓冲区的指针。
inLen	[in] 缓冲区的长度。

返回值

- 0：成功
- 其他：失败

> 📖 **说明：**
>
> - 这个函数只是把需要发送的数据装载到 *SPI W8 ~ W15* 中，当 *ESP8266* 在从机模式下接收到 *MASTER_READ_DATA_FROM_SLAVE_CMD* 命令后会自动传输数据。
> - 默认长度 *32* 字节，最大到 *64* 字节。

4.5.2.7. SPISlaveRecvData

接口描述

从机模式接收数据。

接口函数

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

参数	说明
spiNum	[in] 选择 SPI 和 HSPI。

返回值

- 0：成功
- 其他：失败

4.5.2.8. SPIMasterSendStatus

接口描述

主机向从机的状态寄存器写数据。

接口函数

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

参数	说明
spiNum	[in] 选择 SPI 和 HSPI。
data	[in] 待写入状态寄存器的值。

返回值

无

4.5.2.9. SPIMasterRecvStatus

接口描述

主机读取从机的状态寄存器数值。

接口函数

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

参数	说明
spiNum	[in] 选择 SPI 和 HSPI。

返回值

- 0：成功
- 其他：失败

> 📖 **说明：**
> 从机的状态寄存器值在 *SPI* 缓冲区 *W0* 中。

4.5.2.10. SPICsPinSelect

接口描述

选择 CS 管脚。

接口函数

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

参数	说明
spiNum	[in] 选择 SPI 和 HSPI。
pinCs	[in] 需要选择的管脚。

返回值

无

> 📖 **说明：**
> 必须在一次传输结束后，才能修改 *CS* 管脚。

4.5.2.11.SPIIntCfg

接口描述

设置中断的源和终端回调函数。

接口函数

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

参数	说明
spiNum	[in] 选择 SPI 和 HSPI。
pIntInfo	[in] 带有中断源和中断回调函数的 SpiIntInfo 结构体指针。

返回值

无

4.5.2.12.SPIIntEnable

接口描述

设置允许的中断源。

接口函数

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

参数	说明
spiNum	[in] 选择 SPI 和 HSPI。
intSrc	[in] 需要设置的中断，参见“4.5.1.1 SpiIntSrc”。

返回值

无

4.5.2.13.SPIIntDisable

接口描述

设置禁止的中断源。

接口函数

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

参数	说明
spiNum	[in] 选择 SPI 和 HSPI。
intSrc	[in] 需要设置的中断，参见“4.5.1.1 SpiIntSrc”。

返回值

无

4.5.2.14.SPIIntClear

接口描述

清除所有中断源。

接口函数

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

参数	说明
spiNum	[in] 选择 SPI 和 HSPI。

返回值

无

4.5.3. SPI_Test 示例说明

ESP8266 做 Slave 的通信格式为 CMD + ADDR + DATA，暂不支持只有 DATA 的传输。从机的 ESP8266 会根据不同的 CMD 响应不同的操作，CMD 默认值如下：

- CMD 为 2，向 ESP8266 的数据寄存器 W0 ~ W15 写入数据；
- CMD 为 3，读取 ESP8266 数据寄存器的数据；
- CMD 为 1，向 ESP8266 的状态寄存器写入数据；
- CMD 为 4，读取 ESP8266 状态寄存器的数据。

Spi_test 示例为两个 ESP8266 的 SPI 通信。其通信测试步骤为：

1. 主机发送 32 字节数据给从机；

2. 主机接收从机的数据；
3. 主机读取从机的状态寄存器数据；
4. 主机向从机的状态寄存器写入数据。

从机端会依次收到 SPI_SLV_WR_BUF_DONE、SPI_SLV_RD_BUF_DONE、SPI_SLV_RD_STA_DONE、SPI_SLV_WR_STA_DONE 的中断。

4.5.3.1. 硬件连接

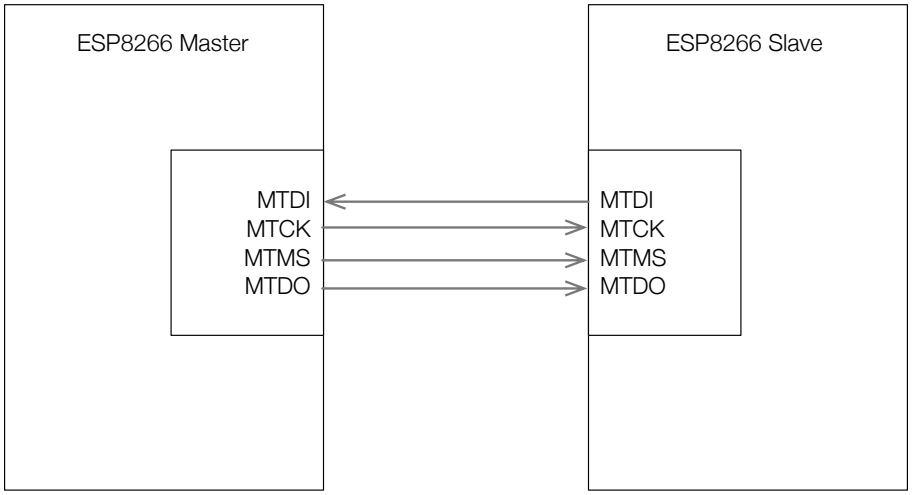


图 4-3. 测试 Demo 硬件连接图

图 4-3 为测试 Demo 的硬件连接图。主机和从机通过 HSPI 连接，MTCK 管脚为SPI。MOSI、MTDI 管脚为 SPI MISO，MTMS 管脚为 SPI Clock，MTMO 管脚为 SPI CS 管脚。

4.5.3.2. 程序介绍

spi_master_test

主机使用从 W0 开始的 SPI 缓冲区。

```
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

从机使用的 SPI 缓冲区从 W8 开始，程序首先配置 SPI 模式，初始化 GPIO，然后接收主机写入的数据，向 SPI 缓冲区装载数据，等待主机来读取。最后是修改状态寄存器的值。

```
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. 运行结果和波形图

ESP8266 Master

调试串口输出日志如图 4-4 所示：

```
=====================================================
        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]
```

图 4-4. 调试串口输出日志 1

图 4-5 中黄色区域为 Master 向 Slave 写数据的命令 0x02，红色为地址寄存器 0x00。绿色区域为写入的数据，数据格式为低字节先发送。

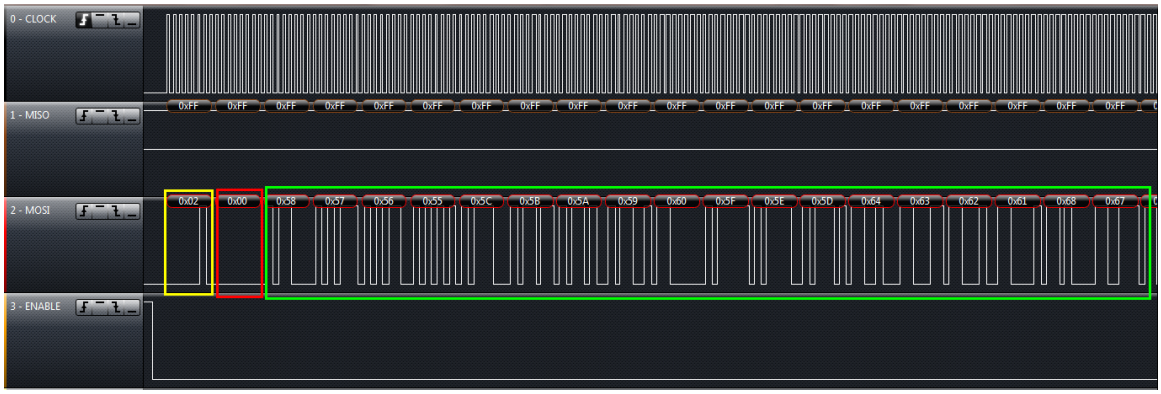


图 4-5. 波形图 1

图 4-6 中黄色为命令 0x03 读取 Slave 的数据，红色为地址寄存器 0x00。绿色区域 MISO 为 SPI 缓冲区的数据。

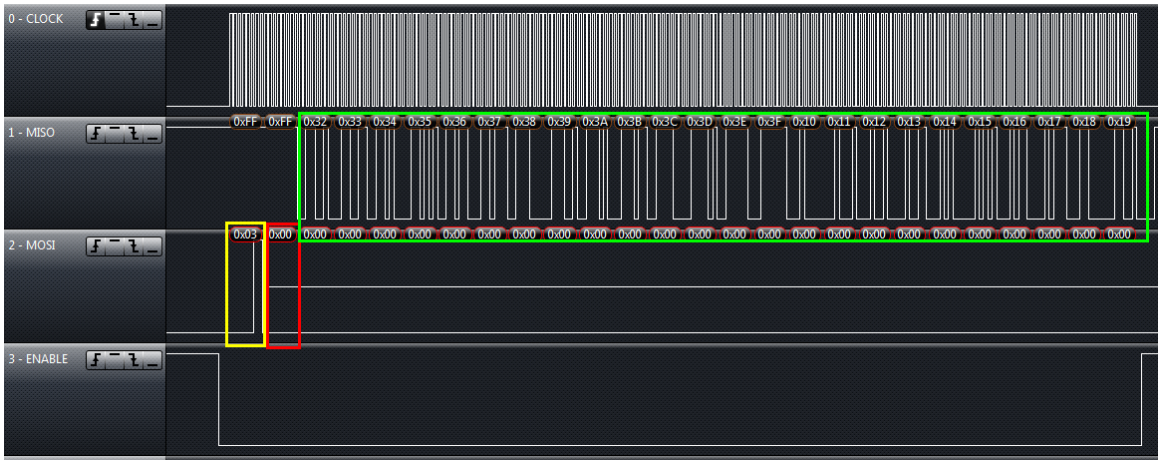


图 4-6. 波形图 2

图 4-7 中黄色为命令 0x04 读取 Slave 的状态寄存器，绿色区域 MISO 为 Slave 的状态寄存器数值。

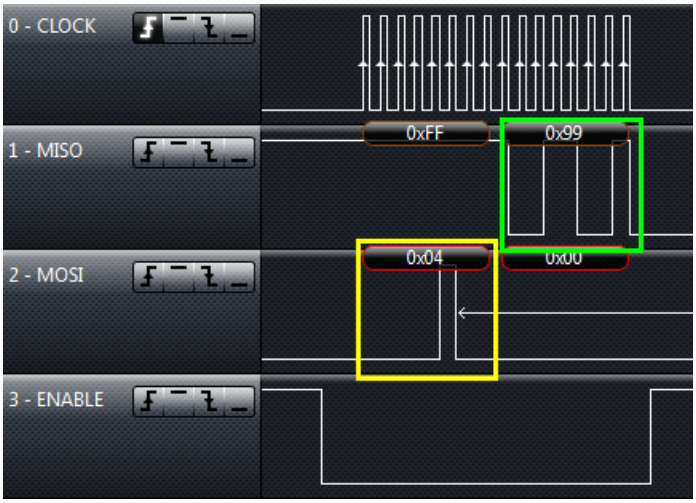


图 4-7 波形图 3

图 4-8 中黄色为命令 0x01 向 Slave 的状态寄存器写入数据，紫色区域为写入 Slave 状态寄存器的数值。

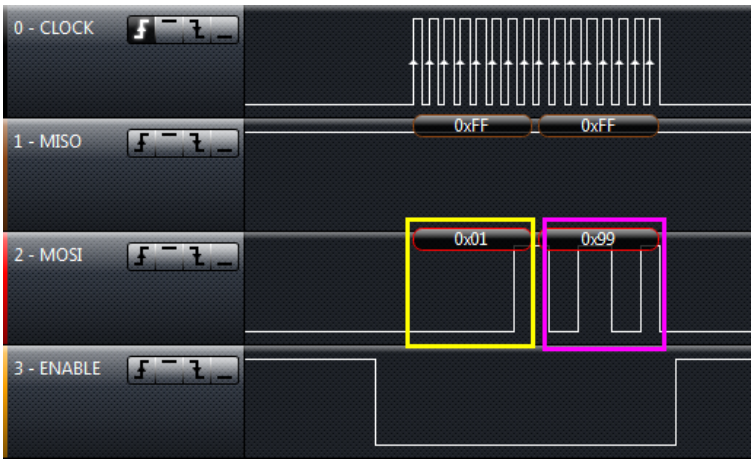


图 4-8 波形图 4

ESP8266 Slave

调试串口输出日志如图 4-9 所示：

```
 ============= 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]
```

图 4-9 调试串口输出日志 2

5. SPI Overlap 模式和显示屏控制台 DEMO

5.1. 功能综述

ESP8266 的 SPI 主机 overlap 模式允许两组 SPI 模块（SPI 与 HSPI）复用相同的 IO 口（如 SCLK、MOSI、MISO）来操作多个 SPI 从机设备，其中硬件支持 3 路 CS 片选，如果从机设备多余 3 个可以使用 GPIO 作为片选信号实现多从机设备通信。

一般情况下为了保持处理器的高效运行，SPI 模块通常专门用于从外接 FLASH 中读取运行程序到 CPU 的 CACHE 中，而 HSPI 模块则用于操作其他用户从机设备。在 overlap 模式下，硬件会自动仲裁两组 SPI 模块对当前管脚信号的控制权，以实现高效的分时应用。如果，软件要启动一次 HSPI 的通信，仲裁信号通过 SPI 是否正在工作来延时阻塞 HSPI 通信的启动。在 SPI 完成一次读取程序代码的通信操作后，仲裁信号才允许 HSPI 接管 IO 口启动通信，如图 5-1 所示。对于用户软件来说，在每次启动 HSPI 通信前，只需切换对应的片选信号，其余操作与单独使用 HSPI 通信并无差别。

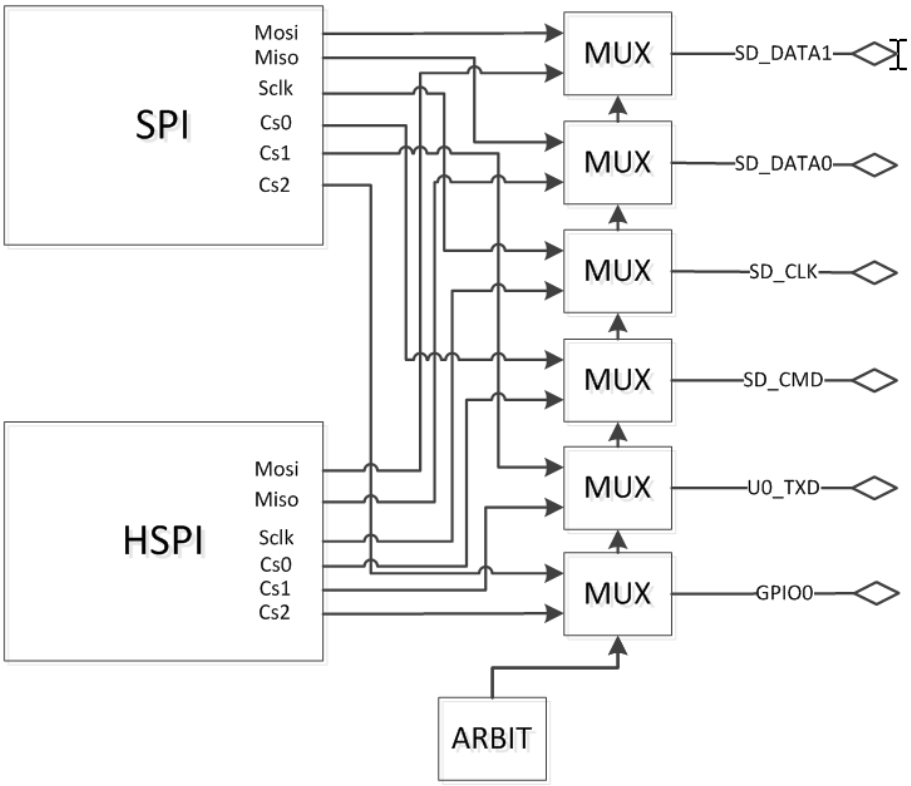


图 5-1. SPI OVERLAP 原理框图

SPI 模块的主机模式的使用方法参见“第 4 章 EPS8266 SPI 模块使用说明”。以下主要详细介绍 OVERLAP 模式的配置使用方法。

5.2. SPI OVERLAP 模式下的硬件连接

EPS8266 中的管脚 SD_CLK、SD_DATA0、SD_DATA1 分别对应两组 SPI 中的 SCLK、MISO、MOSI。而 SD_CMD、U0TXD、GPIO0 分别对应两组 SPI 的片选信号 CS0、CS1、CS2。通常情况下，SD_CMD 与外接 FLASH 的 CS 信号相连，而 U0TXD、GPIO0 可以连接两个从机设备的片选信号，并且 HSPI 可以通过使能 CS0 来独立于 SPI 来读取写入FLASH 数据（如读取一些预存的用户数据）。

如果要使用更多的 SPI 设备，则可以通过配置寄存器禁用所有的 CS0-CS2 通过其他的 GPIO 来对设备进行选定。

5.3. SPI OVERLAP 模式的 API 说明

1. `void hapi_overlap_init(void)`

功能：

初始化 SPI 的 overlap 模式，调用之后 SPI 与 HSPI 就可以共用 SCLK、MOSI、MISO 与不同的设备进行通信。另外默认状态下 HSPI 使用 CS2 作为片选信号。在通信需注意 CS 信号的切换。

位置：

DEMO 中 ***\app\user\user_main.c***

2. `SELECT_OLED(),SELECT_TFT()`

功能：

切换 HSPI 使用的 CS 管脚，在 DEMO 中 OLED 显示屏连接在片选 CS2 上，TFTLCD 屏连接在片选 CS1 上。在每次启动 HSPI 通信前需要调用宏，宏定义具体为：

```
#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)
```

因此，用户可以以此来修改制作宏定义，例如想要使用 HSPI 来操作 FLASH 可以制作宏。

```
#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)
```

如果想要使用普通 GPIO 来做片选则需要：

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

位置：

DEMO 中 ***\app\include\user_lcd.h***

其他 SPI 主机通信 API 参见“第 4 章 EPS8266 SPI 模块使用说明”。

5.4. 显示屏控制台程序 DEMO

该 DEMO 用于在液晶等显示屏制作简易的字符串打印以便于各种参数显示及调试打印。DEMO 驱动目前支持两种屏幕，天马 3.5 寸 TM035PDZV36 480*320 TFT 彩色 LCD 与中景园电子 1.3 寸 128*64OLED。驱动程序均使用 ESP8266 HSPI 在 OVERLAP 模式下与显示屏通信。SPI 主机详细说明参见“第 4 章 EPS8266 SPI 模块使用说明”。

在 SPI OVERLAP 模式下，两种屏幕与 ESP8266 外部程序 Flash 芯片共用 SPI 总线的 SCLK、MOSI、MISO 信号，不同设备之间使用不同的 CS 信号加以区分。

5.4.1. 连线说明

中景园电子 1.3 寸 OLED 连接

OLED 屏信号 SCLK、MOSI、CS、DC、RESET 分别与 ESP8266 的 SD_CLK、SD_DATA1、GPIO0、MTCK、GPIO5 管脚相连，OLED 屏的 VCC 与 GND 分别连接 DEMO 板的 3.3V 网络与 GND。

天马 3.5 寸 TFT LCD

TFT 屏信号 SCLK、MOSI、CS、RESET 分别与 ESP8266的 SD_CLK、SD_DATA1、U0TXD、GPIO5 管脚相连，TFT 屏的 VCC 与 GND 分别连接 DEMO 板的 3.3V 网络与 GND。

5.4.2. API 函数说明

1. `void screen_init(void)`

功能：

显示屏初始化程序。启动时调用。

位置：

\app\user\user_lcd.c 与 ***\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)`

功能：

对 `scr_font_param` 结构的全局变量配置字符串显示参数。

参数说明：

- `uint8 bkg_color`——TFT 背景颜色可选 BLACK_8COLOR 与 WHITE_8COLOR。OLED 屏不使用。
- `uint8 ft_color`——TFT 字体颜色可选 BLACK_8COLOR 与 WHITE_8COLOR。OLED 屏不使用。
- `uint8 ft_size`——字体大小，字符为 12*6 的 ASCII 字符，参数为字符下像素倍数，如 ft_size 为 2，实际字体为 24*12 大小。输入非 0 值。
- `uint8 scr_size_clr_row`——屏幕充满后，刷新的屏幕显示需要清除行数。输入非 0 值。
- `uint8 scr_size_x`——每行显示字符个数。注意不要超过屏幕像素范围。
- `uint8 scr_size_y`——显示字符行数。注意不要超过屏幕像素范围。

位置：

\app\user\user_lcd.c 与 ***\app\include\user_lcd.h***，在 ***screen_init*** 函数中调用。

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

功能：

用于屏幕显示的标准打印函数，与 C 语言标准 ***printf*** 函数使用方法相同。

参数说明：

- `const char* fmt`——显示字符串。
- ***...***——对应字符串中需要显示的可变个数参数。

位置：

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

4. `void at_lcd_print(uint8* str)`

功能：

在屏幕上显示顺序显示指定字符串。

参数说明：

`uint8* str`——字符串数组首地址。

5.4.3. 预编译宏设定

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

位置：

\app\include\user_lcd.h

`OLED_SCR` 与 `TFT_SCR` 分别可以控制在各自屏幕显示调试字符，程序支持在两个屏幕同时显示相同字符。`OVERLAP_TEST` 用于 SPI OVERLAP 测试，该模式使用 TFT 显示图片，会与 TFT 显示字符冲突，因此需要设置为 0。

6. SPI 透传协议（单线）

6.1. 功能综述

该协议使用 ESP8266 的从机模式与其他的处理器的 SPI 主机进行通信，连线上需要 5 路信号线实现该协议，除了标准 SPI 所需要的 4 路信号线外，还需要额外的 1 路信号用于告知主机当前从机的状态寄存器状态更新情况。

6.2. ESP8266 SPI 从机协议格式

6.2.1. SPI 从机时钟极性配置要求

与 ESP8266 SPI 从机通信的主机设备时钟极性需配置为：空闲低电平，上升沿采样，下降沿变换数据。并且在一次 34 字节读 / 写或通过一次 2 字节的通信读取丛机状态寄存器的过程中，务必保持片选信号 CS 的低电平，如果在发送过程中 CS 被拉高，从机内部状态将会重置。

6.2.2. SPI 从机支持的通信格式

ESP8266 SPI 从机通信格式为命令 + 地址 + 读 / 写数据或命令 + 从机状态值，具体：

- 命令：长度，8 bits；主机输出从机输入（MOSI）。

其中 0x02 为主机发送从机接收数据，主机通过 MOSI 将 32 Bytes 写入从机数据缓存对应寄存器 SPI_W0 至 SPI_W7；

而 0x03 为主机接收从机发送数据，将从机缓存对应寄存器 SPI_FLASH_C8 至 SPI_FLASH_C15 中的 32 Bytes 数据通过 MISO 发送到主机。

另外，0x04 或 0x05 均可读取从机状态寄存器 SPI_FLASH_STATUS 中的低 8 位。

> ⚠ **注意：**
> 其余数值用于读写 *SPI* 从机的状态寄存器 *SPI_FLASH_STATUS*，由于其通信格式与读写数据缓存不同，会造成从机读写错误，请勿使用。

- 地址：长度，8 bits；主机输出从机输入（MOSI）。地址内容必须为 0。
- 读/写数据：长度，256 bits（32 Bytes）；主机输出从机输入（MOSI）对应 0x02 命令或主机输入从机输出（MISO）对应 0x03 命令。
- 从机状态：长度，8 bits；主机输入从机输出（MISO），使用 0x04 或 0x05 读取表示从机通信状态。

6.3. 从机状态定义与中断线行为

6.3.1. 状态定义

从机状态一共有 8 bits 其中：

- `wr_busy`，bit0：1 表示从机写缓存满，并正在处理接收数据，0 表示写缓存空可以进行下一次写入操作。
- `rd_empty`，bit1：1 表示从机读缓存为空，没有新数据更新，0 表示读缓存已更新需要主机读取。
- `comm_cnt`，bit2-4：读写通信计数。每次从机进去 SPI 读 / 写缓存中断时，该 3 位计数值会加 1，主机可以由此判断一次读 / 写数据通信是否已经被从机识别并通信完毕。

> ⚠ **注意：**
> 因此主机在一次读 / 写数据通信后，如果想要进行下一次读操作必须满足：*rd_empty* 为 *0*，并且 *comm_cnt* 的值为上一次通信时加 *1*；如果想要进行下一次写操作必须满足：*wr_busy* 为 *0*，并且 *comm_cnt* 的值为上一次通信时加 *1*。

6.3.2. GPIO0 中断线行为

在从机状态寄存器产生变化时，中断线 GPIO0 会置 1，当主机使用 0x04、0x05 命令读取从机状态寄存器后，中断线 GPIO0 会清 0。

6.4. ESP8266 SPI 从机 API 函数说明

> ⚠ **注意：**
> 如果需要使用 *SPI* 带状态寄存器的单线透传协议需要在 ***spi.h*** 文件中配置。

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

中断响应函数会采用 `spi_slave_isr_sta(void *para)`

1. `void spi_slave_init(uint8 spi_no)`

功能：

SPI 从机模式初始化，将 IO 口配置为 SPI 模式，启用 SPI 传输中断，并注册 `spi_slave_isr_handler` 函数。通信格式设定为 8 bits 命令 + 8 bits 地址 + 256 bits（32 Bytes）读/写数据。

参数：

`spi_no`：SPI 模块的序号，ESP8266 处理器有两组功能相同的 SPI 模块，分别为 SPI 和 HSPI。

可选配的值：SPI 或 HSPI。

2. `spi_slave_isr_sta(void *para)`

功能与触发条件：

SPI 中断处理函数，主机如正确进行了传输操作（读或写从机），中断就会触发。用户可以修改中断服务程序实现所需通信功能，代码如下：

```
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,
            //记录中断状态
        regvalue=READ_PERI_REG(SPI_SLAVE(HSPI));
        //***********处理中断标志结束本次通行过程***********//
            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);
        //
**********************************************************//
        /***************主机写中断处理***************/
         if(regvalue&SPI_SLV_WR_BUF_DONE){
         //*****写入完成，写入忙状态位置1，通信计数器加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);
        //*********************************************//
            //*******将寄存器接收数据搬入内存******//
                  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++;
            }
            //*********************************//
            //************数据搬完，清0写入忙状态*************//

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);
            //
**********************************************//
            /***测试部分，可以修改，该段程序作用是将读到数据复制到读缓存
**/
            for(idx=0;idx<8;idx++)
            {
```

```
                    WRITE_PERI_REG(SPI_W8(HSPI)+(idx<<2),
READ_PERI_REG(SPI_W0(HSPI)+(idx<<2)));
            }
            /
*************************************************************/
            /***测试部分，可以修改，读缓存空状态清0，从机可以进行读取操
作**/

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);  //中断线置1，提醒主机读取从机状
态
      /****************主机读中断处理***************/
            }else if(regvalue&SPI_SLV_RD_BUF_DONE){
            //*****读取完成，读取空状态位置1，通信计数器加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);  //中断线置1，提醒主机读取从机状
态
            }
      /****************主机读状态中断处理***************/
      if(regvalue&SPI_SLV_RD_STA_DONE){
            GPIO_OUTPUT_SET(0,0);   //中断线清0，主机读取状态完毕
      }
   }else if(READ_PERI_REG(0x3ff00020)&BIT9){ //bit7 is for i2s
isr,
```

```
                    }
}
```

7. SPI 透传协议（双线）

7.1. 功能综述

该协议使用 ESP8266 的从机模式与其他的处理器的 SPI 主机进行通信，连线上需要 6 路信号线实现该协议，除了标准 SPI 所需要的 4 路信号线外，还需要额外的 2 路信号用于告知主机当前从机的接收与发送缓存状态以实现数据流控制。

7.2. ESP8266 SPI 从机协议格式

7.2.1. SPI 从机时钟极性配置要求

与 ESP8266 SPI 从机通信的主机设备时钟极性需配置为：空闲低电平，上升沿采样，下降沿变换数据。并且在一次 34 字节读 / 写过程中，务必保持片选信号 CS 的低电平，如果在发送过程中 CS 被拉高，从机内部状态将会重置。

7.2.2. SPI 从机支持的通信格式

ESP8266 SPI 从机通信格式与主机模式基本相同为命令 + 地址 + 读 / 写数据，具体为：

- 命令：长度，8 bits；主机输出从机输入（MOSI）。

其中 0x02 为主机发送从机接收数据，主机通过 MOSI 将 32 Bytes 写入从机数据缓存对应寄存器 SPI_W0 至 SPI_W7；

而 0x03 为主机接收从机发送数据，将从机缓存对应寄存器 SPI_W8 至 SPI_W15 中的32 Bytes 数据通过 MISO 发送到主机。

> ⚠ **注意：**
> 其余数值用于读写 *SPI* 从机的状态寄存器 *SPI_STATUS*，由于其通信格式与读写数据缓存不同，会造成从机读写错误，请勿使用。

- 地址：长度，8 bits；主机输出从机输入（MOSI）。地址内容必须为 0。
- 读/写数据：长度，256 bits（32 Bytes）；主机输出从机输入（MOSI）对应 0x02 命令或主机输入从机输出（MISO）对应 0x03 命令。

7.3. 数据流控制线功能说明

EPS8266 使用两个 GPIO 输出从机接收和发送缓存的状态。

7.3.1. GPIO0 主机发送从机接收缓存状态

GPIO0 在进入从机接收中断后中断程序会：将 SPI 从机恢复到可通信状态准备下一次通信；然后把 GPIO0 写为低电平；再处理所接收到的数据；最后将 GPIO0 写为高电平退出中断程序。因此：

- 在主机启动一次 SPI 写入通信到 GPIO0 产生下降沿期间，再次启动任何其他的 SPI 通信都会出错。
- 在 GPIO0 低电平期间，主机启动任何 SPI 写入（0x02 命令）操作会覆盖从机接收寄存器 SPI_W0 至 SPI_W7。然而，如果此时从机发送寄存器准备好了有效数据（参见 GPIO2 说明），GPIO0 低电平期间可以启动主机读取操作（0x03 命令），将从机发送寄存器 SPI_W8 至 SPI_W15 中的数据读出。
- 在 GPIO0 低电平跳转到高电平后，表示从机已经将接收寄存器 SPI_W0 至 SPI_W7 中的数据处理完毕，主机就可以再一次启动写入操作（0x02 命令）。

7.3.2. GPIO2 主机接收从机发送缓存状态

GPIO2 与 GPIO0 动作略有区别。在进入从机发送中断后，中断程序会：将 SPI 从机恢复到可通信状态准备下一次通信；然后把 GPIO0 写为低电平；最后将退出中断程序。假如之后 WIFI 端有数据传入 ESP8266 并要求从 SPI 转发，则 8266 软件会写入 SPI_W8 至 SPI_W15 并将 GPIO2 置为高电平，因此：

- 在主机启动一次 SPI 读取通信到 GPIO2 产生下降沿期间，再次启动任何其他的 SPI 通信都会出错。
- 在 GPIO2 低电平期间，主机启动任何 SPI 读取（0x03 命令）操作一般只能读出与上一次相同数据或写入不完整的数据。然而，如果此时从机接收寄存器数据处理完毕（参见 GPIO0 说明），GPIO2 低电平期间可以启动主机写入操作（0x02 命令）。
- 在 GPIO2 低电平跳转到高电平后，表示从机已经将发送寄存器 SPI_W8 至 SPI_W15 中的数据更新，主机就可以再一次启动读取操作（0x03 命令）。

7.3.3. 主机通信逻辑实现

下面以不完整 C 代码简要说明通信逻辑：

```
//wr_rdy变量表示可以进行下一次SPI写通信
//rd_rdy变量表示可以进行下一次SPI读通信
unsigned char wr_rdy=1,rd_rdy=0;
```

```
void spi_read_func(....)
{
      //在启动读传输之前，要判断从机是否有新数据可以读（即rd_rdy不为0）；
      //此外还要判断上次写入传输是否完毕：完毕正在处理数据(信号GPIO0为0)或者从机可以接受下一次写入 （wr_rdy不为0）
      if(rd_rdy&&((GPIO0= =0)||wr_rdy)){
       rd_rdy=0;  //清零可读标识 rd_rdy
      spi_transmit(0x03,0,*read_buff);//启动SPI传输,命令3+地址0+加32字节数据
      ….
      }
}
void spi_write_func(...)
{
      //在启动写传输之前，要判断从机是否可以接收新数据（即rd_rdy不为0）；
      //此外还要判断上次读取传输是否完毕：完毕无新数据需要读取(信号GPIO2为0)或者从机有新数据需要读取 （rd_rdy不为0）
      if(wr_rdy&&((GPIO2= =0)||rd_rdy)){
      wr_rdy=0;   //清零可写标识 rd_rdy
      spi_transmit(0x02,0,*write_buff);//启动SPI传输,命令2+地址0+加32字节数据
      ...
      }
}

GPIO0_Raising_Edge_ISR() //与8266的GPIO0相连的上升沿中断程序
{
       wr_rdy=1;  //主机发送数据处理完毕可以进行下一次写入
}

GPIO2_Raising_Edge_ISR()//与8266的GPIO2相连的上升沿中断程序
{
       rd_rdy=1; //从机更新发送缓冲，主机可以读取
```

```
}
```

7.4. ESP8266 SPI 从机 API 函数说明

1. `void spi_slave_init(uint8 spi_no)`

功能：

SPI 从机模式初始化，将 IO 口配置为 SPI 模式，启用 SPI 传输中断，并注册 ***spi_slave_isr_handler*** 函数。通信格式设定为 8 bits 命令 + 8 bits 地址 + 256 bits（32 Bytes）读 / 写数据。

参数：

`spi_no`：SPI 模块的序号，ESP8266 处理器有两组功能相同的 SPI 模块，分别为 SPI 和 HSPI

可选配的值：SPI 或 HSPI

2. `spi_slave_isr_handler(void *para)`

功能与触发条件：

SPI 中断处理函数，主机如正确进行了传输操作（读或写从机），中断就会触发。用户可以修改中断服务程序实现所需通信功能，代码如下：

代码：

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

      if(READ_PERI_REG(0x3ff00020)&BIT4){        //bit4表示SPI中断
        //following 3 lines is to clear isr signal
            CLEAR_PERI_REG_MASK(SPI_SLAVE(SPI), 0x3ff);
      }else if(READ_PERI_REG(0x3ff00020)&BIT7){ //bit7 表示HSPI中断,
            regvalue=READ_PERI_REG(SPI_SLAVE(HSPI));  // 记录中断类型
            // 关闭SPI中断使能
            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);
                        // 将SPI从机恢复到可通信状态，准备下一次通信
                        SET_PERI_REG_MASK(SPI_SLAVE(HSPI), SPI_SYNC_RESET);
                        // 清除中断标志
                        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);
                        // 打开SPI中断使能
                        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);
                        //主机写入，从机接收处理程序
                         if(regvalue&SPI_SLV_WR_BUF_DONE){
                                    GPIO_OUTPUT_SET(0, 0);  //GPIO0清0
                                    idx=0;
                                    //读取接收数据
                               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);//投
送接收完成消息
                                GPIO_OUTPUT_SET(0, 1);
//GPIO0置1
                                SET_PERI_REG_MASK(SPI_SLAVE(HSPI),
SPI_SLV_WR_BUF_DONE_EN);
                    //主机读取，从机发送处理程序
                     if(regvalue&SPI_SLV_RD_BUF_DONE){
                                GPIO_OUTPUT_SET(2, 0);  //GPIO2清0
                    }
        }else if(READ_PERI_REG(0x3ff00020)&BIT9){ //bit7 表示I2S中断


        }
```

8. HSPI 主机多设备驱动说明

8.1. 功能综述

ESP8266 有两组 SPI 通信模块命名分别为 SPI 与 HSPI。其中 SPI 通常专门用于从片外 Flash 读取 CPU 程序代码。而 HSPI 则用于用户 SPI 设备的通信操作。

HSPI 在主机通信模式下，硬件支持 3 个用户设备以及一个片外 Flash 读写操作。连接方式具体为：

模式	设备
HSPI Default IO	用户设备 1
SPI OVERLAP and CS1	用户设备 2
SPI OVERLAP and CS2	用户设备 3
SPI OVERLAP and CS0	Flash

此连接与 SPI 共用一个片外 Flash，除去程序与相关配置所使用的空间外，剩余的 Flash 空间均可用于用户数据的读写。

⚠ **注意：**

- *API* 函数暂不支持使用 *HSPI* 主机加软件 *CS* 对设备的操作。
- 当下载用户程序时，读取 *FLASH* 所使用的 *SPI* 时钟频率被设定为 *80 MHz*，*SPI OVERLAP* 加 *CS1* 与 *SPI OVERLAP* 加 *CS2* 的两种接法的 *SPI* 时钟固定为 *80 MHz*。

8.2. 硬件连接

SPI 从机设备通常使用四线通信，分别为 SCLK、MOSI、MISO 和 CS。

HSPI 主机三种不同的用户设备连接方法如下表所示。

HSPI 默认管脚	MTDO 对应 CS，MTCK 对应 MOSI，MTDI 对应 MISO，MTMS 对应 CLK。
SPI OVERLAP 加 CS1	U0TXD 对应 CS1，SD_CLK 对应 SCLK，SD_DATA0 对应 MISO，SD_DATA1 对应 MOSI。
SPI OVERLAP 加 CS2	GPIO0 对应 CS2，SD_CLK 对应 SCLK，SD_DATA0 对应 MISO，SD_DATA1 对应 MOSI。

> 📖 **说明：**
> 通过 *OVERLAP* 模式 *HSPI* 操作 *FLASH* 的管脚与 *SPI* 所使用的完全相同。

8.3. API 说明

系统所支持的连接模式分别通过 ***\app\include\driver\spi_overlap.h*** 中的宏定义命名为：

- HSPI_CS_DEV
- SPI_CS1_DEV
- SPI_CS2_DEV

以上分别对应第 8.2 节中的三种硬件连接方式。

此外，对于 FLASH 的操作则定义为 `SPI_CS0_FLASH`。两个用户 API 函数为：

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

功能	初始化一个 HSPI 主机连接，该函数支持四种设备连接，如果连接多个 SPI 设备需要多次调用函数分别初始化。
位置	定义于工程目录 ***\app\include\driver\spi_overlap.h***，实现于工程目录 ***\app\driver\spi_overlap.c***。
参数	• `uint8 dev_no`：只支持 HSPI_CS_DEV，SPI_CS1_DEV，SPI_CS2_DEV，SPI_CS0_FLASH 四种情况对应数值 0 ～ 3，其余数值函数提示打印出错直接返回。 • `uint8 clk_polar`：设备时钟极性，0 代表时钟上升沿采样，下降沿变换数据，1 代表时钟下降沿采样，上升沿变换数据。其余数值函数提示打印出错直接返回。 • `uint8 clk_div`：时钟分频，40 MHz 为基准频率，分频数为 clk_div+1 即，0 代表基准频，1 代表20 MHz，2 代表 40/3 MHz 等。

> ⚠ **注意：**
> 当且仅当下载时 *SPI* 读取 *FLASH* 时钟频率被设定为 *80 MHz*，那么通过 *OVERLAP* 连接的两种情况 *SPI_CS1_DEV*，*SPI_CS2_DEV* 主机 *SPI* 时钟不可调，只能为 *80 MHz*。

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

功能	切换并选择主机通信设备。
位置	定义于工程目录 ***\app\include\driver\spi_overlap.h***，实现于工程目录 ***\app\driver\spi_overlap.c***。
参数	`uint8 dev_no`：只支持 HSPI_CS_DEV，SPI_CS1_DEV，SPI_CS2_DEV，SPI_CS0_FLASH 四种情况对应数值 0 ～ 3。设备未初始化打印出错直接返回。其余数值函数提示打印出错直接返回。

9. I2C 使用说明

9.1. 功能综述

ESP8266 目前提供作为 I2C 主设备的接口，可以对其他 I2C 从设备（例如大多数数字传感器）进行控制与读写。

每个 GPIO 管脚内部都可以配置为开漏模式（open-drain），从而可以灵活的将 GPIO 口用作 I2C data 或 clock 功能。

同时，芯片内部提供上拉电阻，以节省外部的上拉电阻。

ESP8266 作为 I2C 主机的 SDA 与 SCL 线波形由 GPIO 模拟产生，在 SCL 的上升沿之后 SDA 取数。SCL 高低电平各保持 5 μs，因此 I2C 时钟频率约为 100 kHz。

9.2. I2C master 接口

9.2.1. 初始化

`i2c_master_gpio_init`：GPIO 硬件初始化。

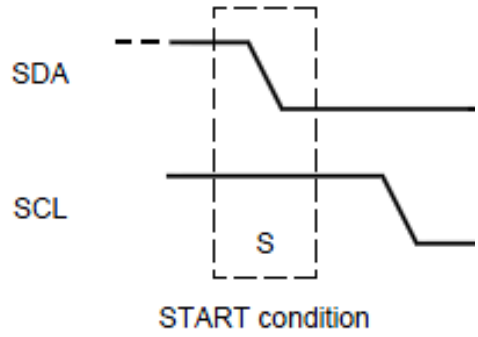
具体如下：

1. 选择 pin 脚功能，配置为 GPIO
2. 配置 GPIO 为开漏模式
3. 初始化 SDA 与 SCL 为高电平
4. 使能 GPIO 中断并复位从机状态。

`i2c_master_init(void)`：复位从机状态

9.2.2. I2C 起始

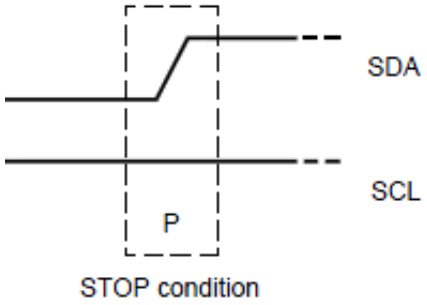
`i2c_master_start(void)`：主机产生 I2C 起始条件。



9.2.3. I2C 停止

`i2c_master_stop(void)`：主机产生 I2C 停止条件。

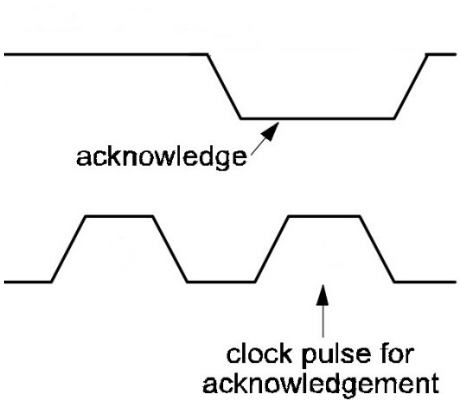
如下图：



9.2.4. I2C 主机回复 ACK

`i2c_master_send_ack(void)`：设置 I2C 主机应答 ACK。

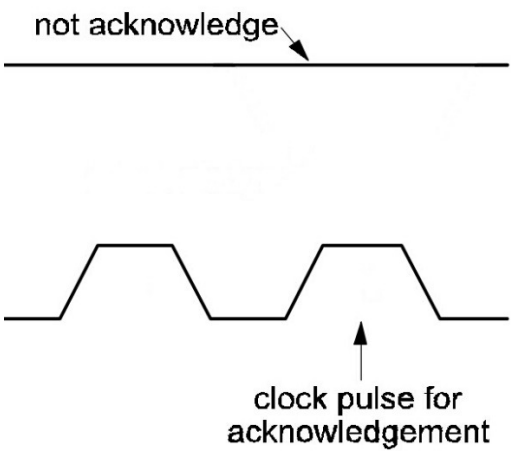
如下图：



9.2.5. I2C 主机回复 NACK

`i2c_master_send_nack(void)`：设置 I2C 主机回复 NACK。

如下图：



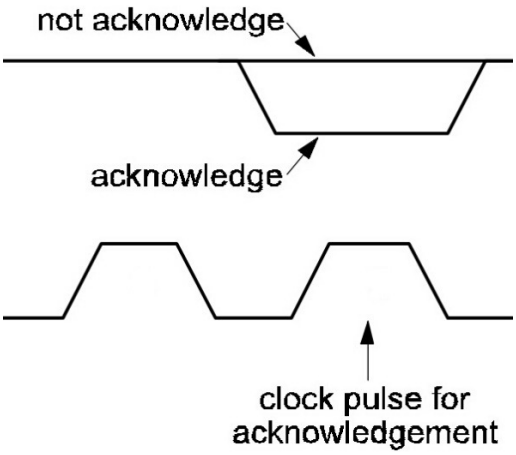
9.2.6. 检查 I2C 从机应答

`bool i2c_master_checkAck(void)`：检查从机应答状态。

返回值：

- TRUE：查询到从机 acknowledge
- FALSE：查询到从机 not acknowledge

如下图：



9.2.7. 向 I2C 总线写数据

`i2c_master_writeByte(uint8 wrdata)`：向 I2C 总线写数

参数：

1 Byte 的数据

> 📖 **说明：**
> 数据最高位先发送，最低位最后发送。

9.2.8. 向 I2C 总线读数据

`i2c_master_readByte (void)`：从 SPI slave 读取一个字节

返回值：

读取到的 1 Byte 数据。

9.3. 使用示例

请参考 ***esp_iot_sdk*** 提供的 IOT_Demo 使用，例如：

```
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 接口说明

10.1. 功能综述

ESP8266 I2S 模块包含一个独立的发送单元和一个独立的接收单元。发送与接收单元各自有一组三线接口：

- 时钟线
- 数据线
- 左右声道选择线

> 📖 **说明：**
> 当数据线写入*"0"*时，时钟和数据输出将停止。

I2S 模块的传输方向见表 10-1。

表 10-1. I2S 模块传输方向

	发送	接收
时钟线	输出 / 输入	输出 / 输入
数据线	输出	输入
频道选择线	输出	输入

> 📖 **说明：**
> *I2S* 模块发送和接收都配有独立的 *FIFO*，其深度为 *128*，宽度为 *32 bits*。两组 *FIFO* 可以通过软件直接访问操作，也可以通过 *SLC* 模块对 *FIFO* 进行自动的 *DMA* 操作。

10.2. 模块配置

10.2.1. I2S 模块配置

10.2.1.1.I2S 模块复位配置

I2SCONF 寄存器中的第 0 位 ～ 第 3 位为 I2S 提供软件复位功能，软件需要写入"1"后，写入"0"来完成对应的复位操作，具体为：

- 第 0 位：`I2S_TX_RESET`
- 第 1 位：`I2S_RX_RESET`
- 第 2 位：`I2S_TX_FIFO_RESET`

- 第 3 位：`I2S_RX_FIFO_RESET`

10.2.1.2.I2S 模块启动配置

提供运行时钟

要启动 I2S 模块接受数据或发送数据，首先要调用系统函数为 I2S 模块提供运行时钟：

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

启动发送模块

I2SCONF 寄存器中的第 8 位用于启动发送模块。

- 如果配置为主机发送模式时，当该位置“1”时，发送模块会一直输出时钟信号、左右声道信号及数据。第一帧数据为 0，随后会移出 FIFO 中的数据。
  - 如果 FIFO 从未写入数据，则数据线会一直保持为 0。
  - 如果 FIFO 将所有写入数据发送完毕，且没有新数据写入时，数据线会循环输出 FIFO 中的最后一个数据。
- 如果配置为从机被动发送模式时，发送模块会等待接收端时钟信号来启动发送模块。

启动接收模块

I2SCONF 寄存器中的第 9 位用于启动接收模块。

- 如果配置为主机接收模式时：
- 当该位置“1”时，接收模块会一直输出时钟信号，并对数据和声道选择线进行采样。
- 该位写“0”，才能停止时钟发送。
- 如果配置为从机被动接收模式时，则准备接收来自于主机的任何发送。

10.2.1.3.发送 / 接收 FIFO 模式配置

FIFO 访问模式

I2S_FIFO_CONF 中的第 12 位定义 FIFO 的访问模式。

- 第 12 位置“1”，则 FIFO 由 SLC 模块 DMA 操作，并且对软件直接访问 FIFO 都会是无效操作
- 第 12 位置“0”，则使用软件直接访问 FIFO。
- 第 12 位默认为“1”。

发送 FIFO 模式

I2S_FIFO_CONF 中的第 13 ~ 15 位为 `i2s_tx_fifo_mod` 控制发送数据格式。

值	描述
0	每声道 16 位全数据（双声道，FIFO 存法 16 位左数据，16 位右数据，16 位左数据）
1	每声道 16 位半数据（单声道，FIFO 存法 16 位数据，16 位无效，16 位数据）
2	每声道 24 位全数据中断（双声道，FIFO 存法 24 位左数据，8 位无效，24 位右数据，8 位空）
3	每声道 24 位半数据中断（单声道，FIFO 存法 24 位数据，8 位无效，24 位数据，8 位空）
4	每声道 24 位全数据继续（左右声道，FIFO 存法 24 位左数据，24 位右数据）
5	每声道 24 位半数据继续（单声道，FIFO 存法 24 位数据，24 位数据）
6 ~ 7	无效

接收 FIFO 模式

`I2S_FIFO_CONF` 中的第 16 ～ 18 位为 `i2s_rx_fifo_mod` 控制接收数据格式。

值	描述
0	每声道 16 位全数据
1	每声道 16 位半数据
2	每声道 24 位全数据中止
3	每声道 24 位半数据中止
4 ~ 7	无效

10.2.1.4.声道模式配置

发送声道模式

I2SCONF_CHAN 中的第 0 ～ 2 位为发送声道模式（`tx_chan_mod`）。

值	描述
0	双声道
1	右声道（左声道和右声道都放右声道的数据）
2	左声道（左声道和右声道都放左声道的数据）
3	右声道（左声道放个常数，从 regfile 来）
4	左声道（右声道放个常数，从 regfile 来）

接收声道模式

I2SCONF_CHAN 中的第 3 ～ 4 位为接收声道模式（`rx_chan_mod`）。

值	描述
0	双声道
1	右声道
2	左声道

10.2.1.5.时钟模式配置

在 I2SCONF 中：

- 第 16 ~ 21 位设置 I2S 模块输入时钟预分频（I2S_CLKM_DIV_NUM）。
- 第 22 ~ 27 位为通信时钟信号分频（I2S_BCK_DIV_NUM）。

10.2.1.6.其他配置

寄存器 I2SRXEOF_NUM 设定了 RX FIFO 在触发 SLC 传输时，需要接收的数据个数，以 4 字节作为单位。

参见 DEMO 中的 ***i2s_reg.h*** 定义，其余说明将会陆续更新。

10.2.2. 链表配置

ESP8266 的 SDIO 收发数据包直接会被 DMA 传输到对应的内存。8266 软件中会定义链表注册结构体（或数组）以及一个（或多个）缓存空间。

如图 10-1，本例中只使用一个缓存空间链表也只有一个元素。将缓存的首地址写入链表注册结构体，并完成其他信息，在将链表结构体首地址写入 8266 对应硬件寄存器，DMA 就能自动操作 SDIO 与缓存空间。链表注册结构体具体为：

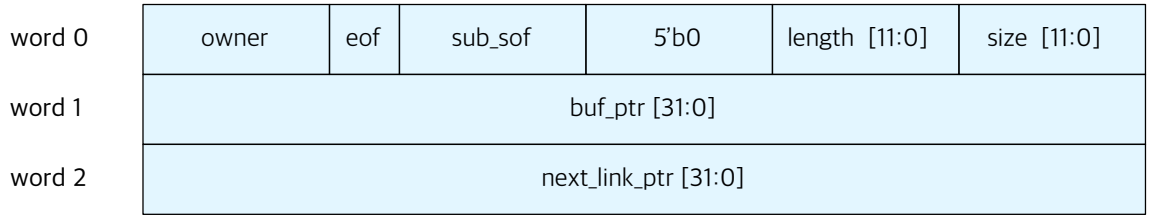


图 10-1. 链表注册结构体

名称		描述
owner	1'b0	当前链接对应缓冲的操作者为软件。MAC 不使用该位。
	1'b1	当前链接对应缓冲的操作者为硬件。

名称	描述
eof	帧结束标志（对于 AMPDU 的子帧结束，该标识是不会置上的）。 • 在 MAC 发送帧时用于指示帧结束 link。对于 eof 位置上的链接，它的 buffer_length[11:0] 必须等于该帧剩下的长度，否者 MAC 会上报错误。 • 在 MAC 接收帧时用于指示帧已接收完成。此时该值由硬件置上。
sub_sof	子帧起始链接标识，区分 AMPDU 帧内的不同子帧，仅用于 MAC 发送时。
length[11:0]	缓冲实际占用的大小。
size[11:0]	缓冲的总大小。
buf_ptr[31:0]	缓冲的起始地址。
next_link_ptr[31:0]	下一个描述符的起始地址。在 MAC 接收帧时该值为0，表示已无空缓冲用于接收。

10.2.3. SLC 模块配置

10.2.3.1.基本配置

SLC 模块为 ESP8266 提供多个模块的 DMA 服务。

进行以下设置，表示 SLC 模块用于 I2S 的 FIFO 传输。

- 配置 SLC_CONF0 中的第 12 ~ 13 位（SLC_MODE）为“01”。
- 将 SLC_RX_DSCR_CONF 的第 17 位（SLC_INFOR_NO_REPLACE）和第 17 位 SLC_TOKEN_NO_REPLACE）都设置为“1”。

10.2.3.2.写入首地址

SLC_RX_LINK 与 SLC_TX_LINK 寄存器的第 0 ～ 19 位为收发链表注册结构体的首地址的前 20 位。在启动 SLC 硬件传输前需要将设定好的注册链表首地址写入寄存器。

10.2.3.3.启动 SLC 传输

SLC_RX_LINK 与 SLC_TX_LINK 寄存器的第 29 位为启动 SLC 传输控制位，在缓存空间，注册链表并把链表地址前 20 位写入硬件后，将该位置 1 启动 SLC 传输。

10.3. 接口函数说明

以下函数可在如下地址中找到：

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

10.3.1. 空隙函数

	`void i2s_test`
函数	`void i2s_test(void)`
功能	I2S 模块读写测试程序，DEMO 中的核心函数，用于测试 I2S 的发送接收通信。
参数	无
	`void i2s_init`
函数	`void i2s_init(uint8 slc_en)`
功能	配置 I2S 相关寄存器。
参数	slc_en：SLC 模块访问使能，0 为软件操作 FIFO，其他数值为 SLC 模块直接访问 FIFO。原理详见“第 10.2.1.3 节”。
	`void creat_one_link`
函数	`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)
功能	设置一个链表注册结构体。
参数	`struct sdio_queue* i2s_queue`：待配置结构体空间的首地址。 其余参数详见“第 10.2.2 节”。
	`void slc_init`
函数	`void slc_init (uint8 trans_dev)`
功能	SLC 模块基础配置。原理详见“第 10.2.3 节”。
参数	`uint8 trans_dev`：SLC 模块访问设备，1 为 I2S，0 为 SDIO，其余输入无效。

10.3.2. 配置函数

	`CONF_RXLINK_ADDR`
函数	`CONF_RXLINK_ADDR(addr)`
功能	配置接收单元链表结构体地址到寄存器，原理详见“第 10.2.3 节”。
参数	`addr`：链表结构体地址。
	`CONF_TXLINK_ADDR`
函数	`CONF_TXLINK_ADDR(addr)`
功能	配置发送单元链表结构体地址到寄存器，原理详见“第 10.2.3 节”。

参数	addr：链表结构体地址。

10.3.3. 启动函数

START_RXLINK

函数	START_RXLINK()
功能	启动 SLC 模块接收单元传输,原理详见“第 10.2.3 节”。
参数	无

START_TXLINK

函数	START_TXLINK()
功能	启动 SLC 模块发送单元传输,原理详见“第 10.2.3 节”。
参数	无

11. UART 接口说明

11.1. 功能综述

ESP8266共有两组 UART 接口，分别为：

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

发送 FIFO 的基本工作过程：

只要有数据填充到发送 FIFO 里，就会立即启动发送过程。由于发送本身是个相对缓慢的过程，因此在发送的同时其它需要发送的数据还可以继续填充到发送 FIFO 里。当发送 FIFO 被填满时就不能再继续填充了，否则会造成数据丢失，此时只能等待。发送 FIFO 会按照填入数据的先后顺序把数据一个个发送出去，直到发送 FIFO 全空时为止。已发送完毕的数据会被自动清除，在发送 FIFO 里同时会多出一个空位。

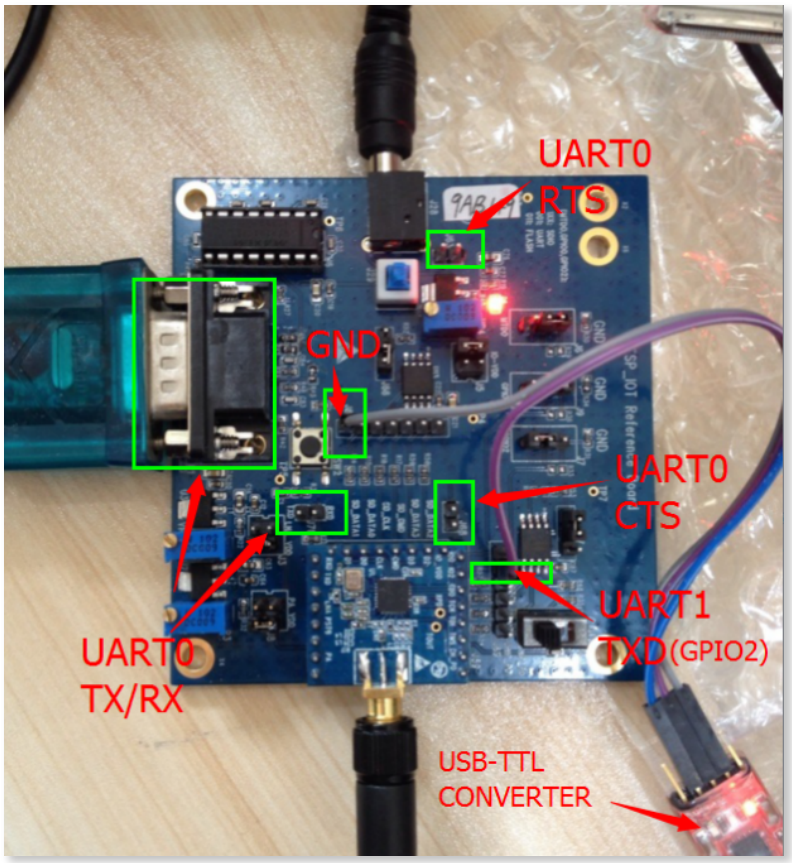
接收 FIFO 的基本工作过程：

当硬件逻辑接收到数据时，就会往接收FIFO 里填充接收到的数据。程序应当及时取走这些数据，数据被取走也是在接收FIFO 里被自动删除的过程，因此在接收 FIFO 里同时会多出一个空位。如果在接收 FIFO 里的数据未被及时取走而造成接收FIFO 已满，则以后再接收到数据时因无空位可以填充而造成数据丢失。

应用场景：

UART0 作为数据通信接口，UART1 作为 debug 信息的打印。

UART0 默认情况会在上电 booting 期间输出一些打印，此期间打印内容的波特率与所用的外部晶振频率有关。使用 40 M 晶振时，该段打印波特率为 115200。使用 26 M 晶振时，该段打印波特率为 74880。



如果这段打印对应用的功能产生影响，可以用第 10.4 节的方法间接屏蔽上电时期的打印输出。

11.2. 硬件资源

UART0 和 UART1 各有一个长度为 128 Bytes 的硬件 FIFO，读写 FIFO 都在同一个地址操作。

两个 UART 模块的硬件寄存器相同，通过 UART0 / UART1 的宏定义来区分。

11.3. 参数配置

UART 属性参数都在 UART_CONF0 定义的寄存器中，可以在 ***uart_register.h*** 中找到。修改该寄存器下的不同对应位，可以配置 UART 属性。

11.3.1. 波特率

ESP8266 的串口波特率范围从 300 到 115200*40 都可以支持。

接口：`void UART_SetBaudrate(uint8 uart_no,uint32 baud_rate);`

11.3.2. 校验位

#define UART_PARITY_EN (BIT(1)) 校验使能：1：enable；0：disable

#define UART_PARITY (BIT(0)) 校验类型设置 1：奇校验；0：偶校验

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

11.3.3. 数据位

#define UART_BIT_NUM 0x00000003 //数据位长度占用两个 bits：

设置这两个 bits 可以配置数据长度 0：5 bits；1：6 bits；2：7 bits；3：8 bits

#define UART_BIT_NUM_S 2 // 寄存器偏移为 2（第 2 bit 开始）

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

11.3.4. 停止位

#define UART_STOP_BIT_NUM 0x00000003 //数据位长度占用两个 bits

设置这两个 bits 可以配置停止位长度 1：1bits；2：1.5 bits；3：2 bits

#define UART_STOP_BIT_NUM_S 4 //寄存器偏移为 4（第 4 bit 开始）

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

11.3.5. 反相

UART 各个信号输入与输出信号，可在内部进行反向配置。

#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))

将对应寄存器置位，可以将对应信号线反向输出 / 输入。

接口：`void UART_SetLineInverse(uint8 uart_no, UART_LineLevelInverse inverse_mask);`

11.3.6. 切换打印函数输出端口

默认情况下，系统打印函数 os_printf 从 uart0 口输出内容，通过以下接口可以设置从 UART0 或者 UART1 口输出打印。

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

11.3.7. 读取 tx/rx 队列内当前剩余的字节数

Tx fifo length：

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

&UART_TXFIFO_CNT;

接口：`TX_FIFO_LEN(uart_no)`

Rx fifo length:

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

&UART_RXFIFO_CNT;

接口：`RF_FIFO_LEN(uart_no)`

11.3.8. 回环操作（loop-back）

在 UART_CONF0 寄存器中,配置后，uart tx/rx 在内部短接。

#define UART_LOOPBACK (BIT(14)) //回环使能位，1：enable；0：disable

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

接口：`ENABLE_LOOP_BACK(uart_no)`

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

接口：`DISABLE_LOOP_BACK(uart_no)`

11.3.9. 线中止信号

要产生线上中止信号，可以将 UART_TXD_BRK 置 1，这样在 UART 发送队列发送完成后，输出一个 break 信号（tx 输出低电平），需要停止输出将该位置 0。

#define UART_TXD_BRK (BIT(8)) //线中止信号，1：enable；0：disable。

11.3.10.流量控制

配置过程：

先配置 UART0 的 pin12、pin13 脚复用为 U0CTS 和 U0RTS 功能。

#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);

接收方向的硬件流控可以配置阈值，当 rx fifo 中的长度大于所设的阈值，U0RTS 脚就会拉高，阻止对方发送。

配置接收流控阈值：

阈值相关的配置一般都在 UART_CONF1 定义的寄存器中。

#define UART_RX_FLOW_EN (BIT(23)) 第 23 bit 使能接收流控 0：disable；1：enable

#define UART_RX_FLOW_THRHD 0x0000007F //门限值，占用 7 bits，范围 0 ～ 127

#define UART_RX_FLOW_THRHD_S 16 //寄存器偏移为 16（第 16 bit 开始）

发送方向的流控只需配置使能，该寄存器在 UART_CONF0 中：

#define UART_TX_FLOW_EN (BIT(15)) 使能发送流控：0：disable；1：enable。

接口：

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

demo 板硬件连接：

需要将 J68（U0CTS）与 J63（U0RTS）的跳线接上。

11.3.11.其他接口

TX_FIFO_LEN(uart_no) //宏定义，发送队列当前长度

RF_FIFO_LEN(uart_no) //宏定义，接收队列当前长度

11.4. 配置中断

由于所有中断事件在发送到中断控制器之前会一起进行“或运算”操作，所以任意时刻 UART 只能向中断产生一个中断请求。通过查询中断状态函数 `UART_INT_ST(uart_no)`，软件可以在同一个中断服务函数里处理多个中断事件（多个并列的 if 语句）。

11.4.1. 中断寄存器

URAT 的中断寄存器有：

UART_INT_RAW 中断原始状态寄存器

UART_INT_ENA 中断使能寄存器：表示当前使能的uart中断。

UART_INT_ST　中断状态寄存器：表示当前有效的中断状态

UART_INT_CLR 清除中断寄存器：置对应位来清除中断状态寄存器

11.4.2. 接口

打开中断使能：`UART_ENABLE_INTR_MASK(uart_no,ena_mask);`

关闭中断使能：`UART_DISABLE_INTR_MASK(uart_no,disable_mask);`

清除中断状态：`UART_CLR_INTR_STATUS_MASK(uart_no,clr_mask);`

获取中断状态：`UART_GET_INTR_STATUS(uart_no);`

中断类型

11.4.3. 接收 full 中断

中断状态位：`UART_RXFIFO_FULL_INT_ST`

定义：当配置阈值并使能中断后，当 rx fifo 中的数据长度大于阈值后，触发该中断。

应用：比较多用于处理 UART 接收的数据，配合流量控制，直接处理或者 post 出消息,或者转存入 buffer。比如，配置阈值为 100，并使能 full 中断，当串口收到 100 字节后，会触发 full 中断。

配置阈值：

full中断阈值（或门限值）

在UART_CONF1 寄存器

#define UART_RXFIFO_FULL_THRHD　0x0000007F　//门限值 mask，7 bits 长，范围 0 ~ 127

#define UART_RXFIFO_FULL_THRHD_S　0　//寄存器偏移为 0（第 0 bit 开始）

设置中断使能：

在 UART_INT_ENA 寄存器

#define UART_RXFIFO_FULL_INT_ENA　(BIT(0))　//full 中断使能位，1：enable；0：disable

清除中断状态：

对于 full 中断比较特殊，需要先将接收 fifo 中的数据全部读空，然后写清除中断状态寄存器。否则退出后中断状态位还是会被置上。

详见中断处理实例。

11.4.4. 接收溢出中断

中断状态位：UART_RXFIFO_OVF_INT_ST

定义：当使能接收溢出中断后，当接收队列的长度大于队列总长度（128 Bytes）时，会触发该中断信号。

触发场景：一般只在没有设置流控的情况下，因为有流量控制时候不会发生溢出。区别于 full 中断，full 中断是人为设置阈值并且数据不会丢失。溢出中断触发则一般都会存在数据丢失。可用于程序调试与验错。

设置中断使能：

在 UART_INT_ENA 寄存器

#define UART_RXFIFO_OVF_INT_ENA (BIT(4)) //溢出中断使能位：1：enable；0：disable

清除中断状态：

读取队列值，使队列长度小于 128，然后置清除中断状态寄存器即可。

11.4.5. 接收超时中断 tout

中断状态位：UART_RXFIFO_TOUT_INT_ST

定义：当配置 tout 阈值并使能中断后，当 UART 开始接收数据后，停止传输的时间超过所设定的门限值，就会触发 tout 中断。

应用：较多用于处理串口命令或者数据，直接处理数据或者 post 出消息,或者转存入 buffer。

配置阈值与功能使能：

tout 中断阈值（或门限值）在 UART_CONF1 寄存器中。

Tout 阈值的单位为 8 个 UART 数据比特的时间（近似 1 个 Byte）。

#define UART_RX_TOUT_EN (BIT(31)) //超时功能使能位：1：enable；0：disable

#define UART_RX_TOUT_THRHD 0x0000007F //超时阈值配置位，共 7 位，范围 0 ～ 127

#define UART_RX_TOUT_THRHD_S 24 //寄存器偏移为 24（第 24 bit 开始）

设置中断使能：

在UART_INT_ENA 寄存器

#define UART_RXFIFO_TOUT_INT_ENA (BIT(8)) tout //中断使能位，1：enable；0：disable

清除中断状态：

与 full 中断类似，tout 中断也需要先将接收 fifo 中的数据全部读空，然后写清除中断状态寄存器。否则退出后中断状态位还是会被置上。

详见中断处理实例。

11.4.6. 发送 fifo 空中断

中断状态位：UART_TXFIFO_EMPTY_INT_ST

定义：当配置 empty 阈值并使能中断后，当 UART 发送 fifo 内的数据小于所设阈值时，会触发 empty 中断。

应用：可用于实现自动将 buffer 中的数据转发至 UART。需要中断处理配合。例如，将 empty 门限值设为 5，则 tx fifo 长度小于 5 个字节时，触发 empty 中断，在 empty 中断的中断处理中，从 buffer 取数把 tx fifo 填满（操作 fifo 的速度远大于 tx fifo 发送的速度）。这样继续循环，直到 buffer 的数据都被发送完毕，将 empty 中断关闭即可。

配置阈值：

empty 中断阈值（或门限值）在 UART_CONF1 寄存器中

#define UART_TXFIFO_EMPTY_THRHD 0x0000007F //发送队列空中断阈值配置位，共 7 位，范围 0 ～ 127

#define UART_TXFIFO_EMPTY_THRHD_S 8 //寄存器偏移为 8（第 8 bit 开始）

设置中断使能：

在 UART_INT_ENA 寄存器

#define UART_TXFIFO_EMPTY_INT_ENA (BIT(1)) //empty 中断使能位，1：enable；0：disable

清除中断状态：

向发送队列填数，高于门限值，并清除对应的中断状态位。如果没有数据需要发送，需要关闭此中断使能位。

详见中断处理实例。

11.4.7. 错误检测类中断

中断状态位：

奇偶校验错误中断：UART_PARITY_ERR_INT_ST

线终止错误中断（line-break）：UART_BRK_DET_INT_ST

接收帧错误中断：UART_FRM_ERR_INT_ST

定义：

奇偶校验错误中断（parity_err）：接收到的字节存在奇偶校验错误。

线终止错误中断（BRK_DET）：接收到 break 信号，或者接收到错误的起始条件（rx线一直为低电平）。

接收帧错误中断（frm_err）：停止位不为 1。

应用：一般都用于错误检测。

设置中断使能：

在UART_INT_ENA 寄存器，

#define UART_PARITY_ERR_INT_ENA (BIT(2)) //奇偶校验错误中断使能位，1：enable；0：disable

#define UART_BRK_DET_INT_ENA (BIT(7)) //线终止错误中断使能位，

1：enable；0：disable

#define UART_FRM_ERR_INT_ENA (BIT(3)) //接收帧错误中断使能位，

1：enable；0：disable

清除中断状态：

对错误进行相应处理后，清除中断状态位即可。

11.4.8. 流量控制状态中断

中断状态位：

UART_CTS_CHG_INT_ST

UART_DSR_CHG_INT_ST

定义：当 CTS、DSR 引脚线上电平改变时触发该中断。

应用：一般配合流量控制使用，当触发该中断后，检查对应流控线状态，如为高电平，则停止向 tx 队列写数。

#define UART_CTS_CHG_INT_ST (BIT(6))

#define UART_DSR_CHG_INT_ST (BIT(5))

设置中断使能：

在UART_INT_ENA 寄存器，

#define UART_CTS_CHG_INT_ENA (BIT(6)) CTS //线状态中断使能位，1：enable；0：disable

#define UART_DSR_CHG_INT_ENA (BIT(5)) DSR //线状态中断使能位，1：enable；0：disable

清除中断状态：

对错误进行相应处理后，清除中断状态位即可。

11.5. 中断处理函数示例流程

```
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. 关于屏蔽上电打印

ESP8266 在上电时候，UART0 默认会输出一些打印信息，如果对此敏感的应用，可以使用 UART 的内部引脚交换功能，在初始化的时候，将 U0TXD、U0RXD 分别与 U0RTS、U0CTS 交换.

调用接口：`void system_uart_swap(void);`

初始化前：

UART0:

U0TXD: pin26(u0txd)

U0RXD:pin25(u0rxd)

U0CTS:pin12(mtck)

U0RTS: pin13(mtdo)

在初始化执行 pin 脚交换后，

U0TXD:pin13(mtdo)

U0RXD:pin12(mtck)

U0CTS: pin25(u0rxd)

U0RTS: pin26(u0txd)

硬件上 pin13 与 pin12 作为 UART0 的收发脚，在上电启动阶段不会有打印输出，但要注意保证 pin13（mtdo）在 ESP8266 启动阶段不能被外部拉高。

12. PWM 接口说明

12.1. 功能综述

12.1.1. 特性描述

ESP8266 系统的 PWM（Pulse Width Modulation）由 FRC1 在软件上实现，可实现同频率、不同占空比的多路 PWM，可用来控制彩灯、蜂鸣器和电机等设备。

> 📖 **说明：**
> *FRC1* 是一个 *23 bits* 的硬件定时器。

PWM 的特性如下所示。

- 使用 NMI（Non Maskable Interrupt）中断，更加精确。
- 可扩展最多 8 路 PWM 信号。
- >14 bit 分辨率，最小分辨率 45 ns。
- 无需配置寄存器，调用函数接口即可完成配置。

> ⚠ **注意：**
> - *PWM* 驱动接口不能跟硬件定时器 *(hw_timer)* 接口函数同时使用，因为二者共用同一个硬件定时器。
> - 如需使用 *PWM* 驱动，请勿调用 `wifi_set_sleep_type(LIGT_SLEEP);` 将自动睡眠模式设置为 *Light Sleep*。因为 *Light Sleep* 在睡眠期间会停 *CPU*，停 *CPU* 期间不能响应 *NMI* 中断。
> - 如需进入 *Deep Sleep*，请先将 *PWM* 关闭，再进行休眠。

12.1.2. 实现方式

ESP8266 系统提供了一种经过优化的软件算法，通过在 FRC1 定时器上挂载 NMI，实现在 GPIO 端口输出多组 PWM 信号。

PWM 的时钟源由高速系统时钟提供，其频率高达 80 MHz。PWM 通过预分频器将时钟源 16 分频，其输入时钟频率为 5 MHz。PWM 通过 FRC1 来产生粗调定时，结合高速系统时钟的微调，可将分辨率提高到 45 ns。

> 📖 **说明：**
> *NMI* 拥有最高中断优先级，可以保证 *PWM* 输出波形的准确度。

12.1.3. 配置说明

- 在定时中断内，为尽快退出程序只在每次 PWM 周期开始时载入下一个周期的定时参数。
- 设置完各个通道的占空比后，系统会调用 `pwm_start()` 函数来计算定时周期。在此之前系统会进行保护操作，即保存当前的各通道参数，并清除计算完成标志，PWM 周期到来会使用保存后的参数。
- PWM 周期中断后会使用新的参数，因此计算完成后需要设置标志位。这样在实现占空比渐变（如控制 RGB 彩灯）的过程中，能保证颜色平滑过渡。
- 可在 user_light.h 中配置采用的 GPIO。SDK 代码示例使用 5 路 PWM，实际可以自行扩展，最多扩展至 8 路 PWM，具体参见“第 12.3 节 自定义通道”。最小分辨率 45 ns，频率在 1KHz 时，占空比最小可以达到 1/22222。

12.1.4. 参数说明

- 最小分辨率：45 ns（近似对应于硬件 PWM 的输入时钟频率为 22.72 MHz）：>14 bit PWM @ 1 KHz
- PWM 周期：1000 µs（1 KHz） ~ 10000 µs（100 Hz）

12.2. pwm.h 详解

12.2.1. 代码示例

```
#ifndef __PWM_H__
#define __PWM_H__
#define PWM_CHANNEL_NUM_MAX 8                   //最多 8 路PWM。
struct pwm_single_param {                       //定义单个 PWM 通道参数结构体。
    uint16 gpio_set;                            //需要置位的 GPIO。
    uint16 gpio_clear;                          //需要清零的 GPIO。
    uint32 h_time;                              //需要写入 FRC1_LOAD 寄存器的
计数值。
};
struct pwm_param {                              //定义 PWM 参数结构体。
    Uint32 period;                              //PWM 周期。
    Uint32 freq;                                //PWM 频率。
```

```
    uint32 duty[PWM_CHANNEL_NUM_MAX];     //PWM 占空比。
};
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. 接口说明

1. pwm_init

名称	pwm_init
含义	PWM 初始化。
代码示例	`pwm_init (uint32 freq, uint32 *duty, uint32 pwm_channel_num,uint32 (*pin_info_list)[3]);`
描述	PWM GPIO，参数和定时器初始化。

参数	• uint32 freq：PWM 的周期。
	• uint32 *duty：各通道占空比参数。
	• uint32 pwm_channel_num：PWM 通道数。
	• uint32 (*pin_info_list)[3]：PWM 各通道的 GPIO 硬件参数，该参数是一个 n x 3 的数组指针。数组中定义了 GPIO 的寄存器，对应 PIN 脚的 IO 复用值，和 GPIO 对应的序号。
	例如：初始化一个 3 通道的 PWM。
	`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_init(light_param.pwm_period,light_param.pwm_duty,3,io_info);`

调用	系统初始化时调用。目前只能调用一次。
返回值	无

2. pwm_set_period

名称	pwm_set_period
含义	设置 PWM 周期。
代码示例	`pwm_set_period (uint32 period)`

描述	设置 PWM 周期，单位 µs。 例如：1KHz PWM，参数为 1000 µs。
参数说明	uint32 period：PWM 周期。
调用	设置完成后需要调用 `pwm_start()` 才起作用。
返回值	无

3. pwm_set_duty

名称	pwm_set_duty
含义	设置 PWM 某个通道信号的占空比。
代码示例	`pwm_set_duty (uint32 duty, uint8 channel)`
描述	设置 PWM 占空比。设置各路 PWM 信号高电平所占的时间，duty 的范围随PWM周期改变。最大值为：period*1000/45 (以1kHz为例：duty 范围是 0～22222)。
参数说明	• uint32 duty：设置高电平时间参数，占空比的值为 (duty*45)/ (period*1000)。 • uint8 channel：当前要设置的 PWM 通道，在 PWM_CHANNEL 定义的范围内。
调用	设置完成后需要调用 `pwm_start()` 才起作用。
返回值	无

4. pwm_get_period

名称	pwm_get_period
描述	获取当前 PWM 周期。
代码示例	`pwm_get_period (void)`
参数说明	无
返回值	PWM 周期，单位 µs。

5. pwm_get_duty

名称	pwm_get_duty
描述	获取对应 channel 的当前 PWM 信号的 duty 参数。
代码示例	`pwm_get_duty (uint8 channel)`
参数说明	uint8 channel：当前要获取的 PWM 通道，在 PWM_CHANNEL 定义的范围内。
调用	设置完成后需要调用 `pwm_start()` 才起作用。
返回值	channel 对应的通道的占空比，占空比的值为 (duty*45)/ (period*1000)。

6. pwm_start

名称	pwm_start
描述	PWM 更新参数。
代码示例	`pwm_start (void)`
参数说明	无
调用	PWM 相关参数设置完成后，需要调用 `pwm_start()` 才起作用。
返回值	无

12.3. 自定义通道

用户还可以增加 PWM 通道，如需增加 GPIO4 为 PWM 输出的第四通道，设置步骤如下所示。

1. 修改初始化参数。

```
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. 修改 *user_light.h* 文件。

```
#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 红外例程及使用说明

13.1. 红外发送与接收使用说明

用户可向乐鑫申请红外的示例代码。

本文档以 32 bits NEC 发送与接收协议为例，现实红外遥控功能。

发送：

用于发送的载波可以采用以下几种方式：

- I2S 的 BCK
- WS 脚产生 38 KHz 载波
- 由 GPIO 中的 sigma-delta 功能在任意 GPIO 口产生载波，但 sigma-delta 产生的载波占空比约为 20%，推荐使用 MTMS 脚（GPIO14），可产生准确的 38 KHz 且占空比 50% 的标准方波。

示例代码通过系统 FRC2 的 DSR TIMER 接口，产生发送序列并驱动红外发送状态机。由于发送 NEC 红外码需要精确到 μs 级的定时，所以在 IR TX 初始化时，会先调用 `system_timer_reinit` 来提高 FRC2 timer 精度。在 `user_config.h` 中，将 `USE_US_TIMER` 定义打开，则可以使用 `os_timer_arm_us` 接口实现 us 级精度的定时。

接收：

红外接收功能主要通过 GPIO 的边沿中断完成。读取系统时间，将两次时间相减可以得到波形持续时间。由软件状态机 `ir_intr_handler` 进行处理。

> ⚠ **注意：**
>
> - 红外接收通过 *GPIO* 中断实现，而同时，系统只能注册一个 *IO* 中断处理程序，如果有其他 *IO* 口也需要中断的话，请将这些中断在同一个处理程序中处理（判断中断源并相应处理）
> - 在非 *OS* 版本的 *SDK* 中，进入中断处理（*GPIO, UART, FRC* 等）直到退出中断的整个过程中，不可调用带 *ICACHE_FLASH_ATTR* 属性的函数，包括打印函数 *os_printf*（定义在 *FLASH IROM* 区）。

13.2. 参数配置

红外接收、发送相关的参数均在 ***ir_tx_rx.h*** 中进行配置。

发送参数：

```
#define GEN_IR_CLK_FROM_IIS 0        // 配置载波模式
                                     // 1: IIS 时钟信号产生发送载波
                                     // 0: GPIO sigma-delta模式产生发送
载波
                                     // 建议使用 MTMS 脚作为红外发射功能

//设置红外发射 PIN 脚的寄存器与复用功能
#define IR_GPIO_OUT_MUX PERIPHS_IO_MUX_GPIO5_U
#define IR_GPIO_OUT_NUM 5
#define IR_GPIO_OUT_FUNC      FUNC_GPIO5
```

接收参数：

```
//设置红外接收 buffer 的大小
#define RX_RCV_LEN      128

//设置红外接收管脚的 GPIO 寄存器与复用功能
#define IR_GPIO_IN_NUM  14
#define IR_GPIO_IN_MUX  PERIPHS_IO_MUX_MTMS_U
#define IR_GPIO_IN_FUNC FUNC_GPIO14
```

其他设置：

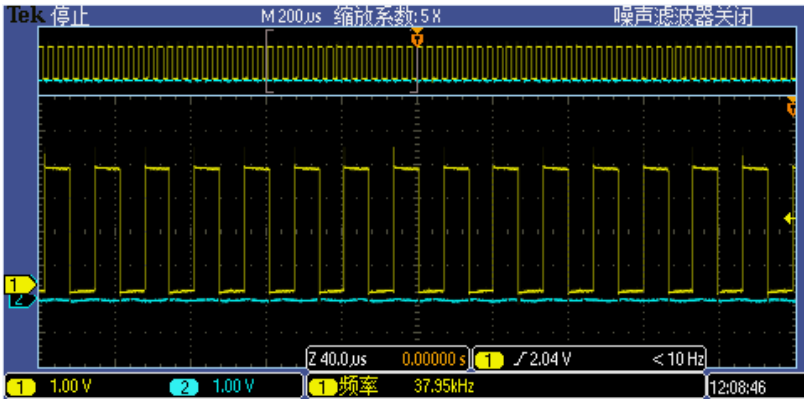
请在 ***user_config.h*** 中定义 #define USE_US_TIMER 。

发射载波波形：

模式1：使用 IIS 时钟（MTMS 脚，GPIO14）

```
#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
```

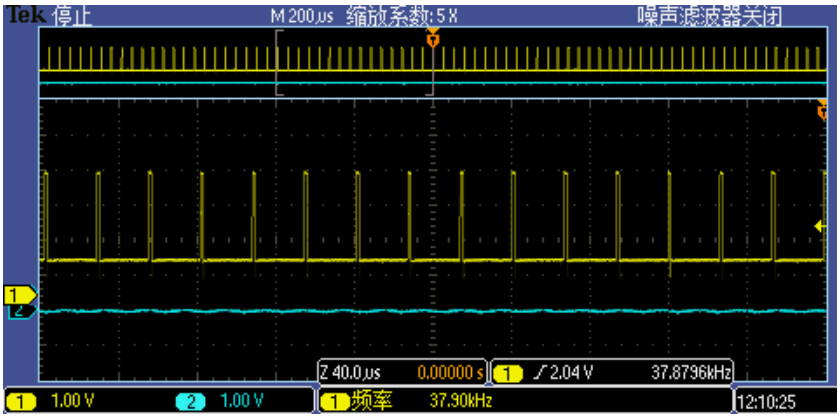
如图：



模式2：使用 sigma-delta 产生载波（任意 GPIO）

```
#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
```

如图：



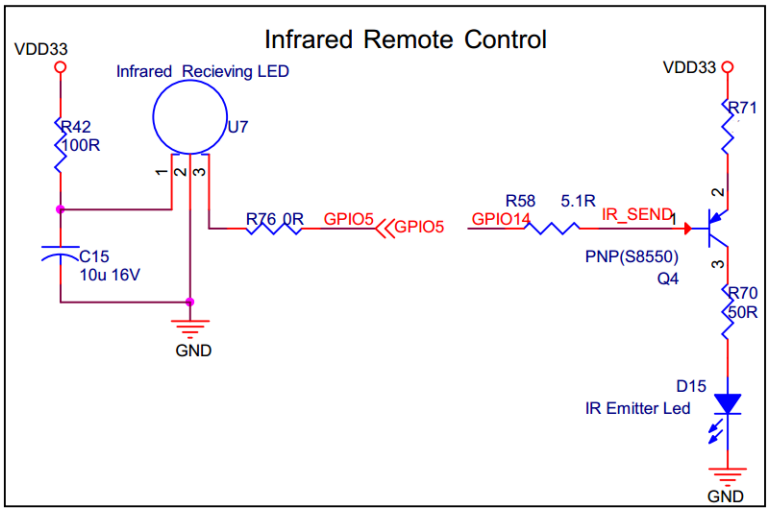
13.3. 例程说明

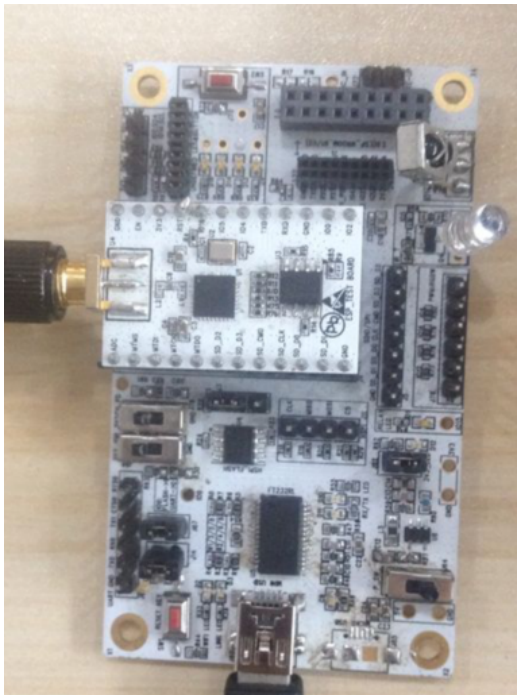
乐鑫提供的红外示例，实现以下功能：

- 系统初始化中，调用接收与发送的初始化函数，并且设置一个 4s 的循环定时器发送红外命令。
- 同时检查红外接收的循环队列，如果有数据，将其出队并打印显示。
- 红外接收的状态机，如果收到符合 NEC 码的波形，会将命令字段存入红外接收的循环队列中。

13.4. 硬件连接

最新版本的 ESP-LAUNCHER 开发板红外硬件部分的原理图如下。（注意，不同版本的开发板可能引脚略有不同，请根据实际情况配置红外收发的 IO 口参数。）







13.5. 实验结果

```
ets Jan  8 2013,rst cause:1, boot mode:(3,2)

load 0x40100000, len 27852, room 16
```

```
tail 12
chksum 0x92
ho 0 tail 12 room 4
load 0x3ffe8000, len 2364, room 12
tail 0
chksum 0xc0
load 0x3ffe8940, len 1192, room 8
tail 0
chksum 0x35
csum 0x35
rSir tx/rx test
mode : softAP(1a:fe:34:9a:c3:81)
dhcp server start:(ip:192.168.4.1,mask:255.255.255.0,gw:192.168.4.1)
add if1
bcn 100
-----------------------
ir rx
==================
ir tx..
addr:55h;cmd:28h;repeat:10;
rep = 0  end
-----------------------
ir rx
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
```

```
==================
ir tx..
addr:55h;cmd:28h;repeat:10;
rep = 0  end
-----------------------
ir rx
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
IR buf pop : 28h
==================
ir tx..
addr:55h;cmd:28h;repeat:10;
rep = 0  end
```

14. Sniffer 应用设计说明

14.1. Sniffer 模式介绍

ESP8266 可以进入混杂模式（sniffer），接收空中的 IEEE802.11 包。可支持如下 HT20 的包：

- 802.11b
- 802.11g
- 802.11n（MCS0 到 MCS7）
- AMPDU

以下类型不支持:

- HT40
- LDPC

尽管有些类型的 IEEE802.11 包是 ESP8266 不能完全接收的，但 ESP8266 可以获得它们的包长。

因此，sniffer 模式下，ESP8266 或者可以接收完整的包，或者可以获得包的长度：

- ESP8266 可完全接收的包，它包含：
  - 一定长度的 MAC 头信息（包含了收发双方的 MAC 地址和加密方式）
  - 整个包的长度
- ESP8266 不可完全接收的包，它包含：
  - 整个包的长度

结构体 RxControl 和 sniffer_buf 分别用于表示了这两种类型的包。其中结构体 sniffer_buf 包含结构体 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
};
```

回调函数 `wifi_promiscuous_rx` 含两个参数（`buf` 和 `len`）。其中 `len` 表示 `buf` 的长度，可分为三种情况：`len` = sizeof (struct sniffer_buf2)，`len` 为 10 的整数倍，以及 `len` = sizeof(struct RxControl)：

LEN == sizeof (struct sniffer_buf2) 的情况

- buf 的数据是结构体 sniffer_buf2，该结构体对应的数据包是管理包，含有 112 字节的数据。
- sniffer_buf2.cnt 为 1。
- sniffer_buf2.len 为管理包的长度。

LEN 为 10 整数倍的情况

- buf 的数据是结构体 sniffer_buf，该结构体是比较可信的，它对应的数据包是通过 CRC 校验正确的。
- sniffer_buf.cnt 表示了该 buf 包含的包的个数，len 的值由 sniffer_buf.cnt 决定。
  - sniffer_buf.cnt==0，此 buf 无效；否则，len = 50 + cnt * 10
- sniffer_buf.buf 表示 IEEE802.11 包的前 36 字节。从成员 sniffer_buf.lenseq[0] 开始，每一个 lenseq 结构体表示一个包长信息。
- 当 sniffer_buf.cnt > 1，由于该包是一个 AMPDU，认为每个 MPDU 的包头基本是相同的，因此没有给出所有的 MPDU 包头，只给出了每个包的长度（从 MAC 包头开始到 FCS）。
- 该结构体中较为有用的信息有：包长、包的发送者和接收者、包头长度。

LEN == sizeof(struct RxControl) 的情况

- buf 的数据是一个结构体 RxControl，该结构体的是不太可信的，它无法表示包所属的发送和接收者，也无法判断该包的包头长度。
- 对于 AMPDU 包，也无法判断子包的个数和每个子包的长度。
- 该结构体中较为有用的信息有：包长，rssi 和 FEC_CODING。
- RSSI 和 FEC_CODING 可以用于评估是否是同一个设备所发。

> 📖 **说明：**
>
> - 对于 *len == sizeof(struct RxControl)* 的情况，包长的计算方法如下：
>   - 如果 *sig_mode == 0*，则包长为 *legacy_length*
>   - 其他情况，包长信息在 *sniffer_buf* 和 *sniffer_buf2* 中，并且更加可信

总结

使用时要加快单个包的处理，否则，可能出现后续的一些包的丢失。

下图展示的是一个完整的 IEEE802.11 数据包的格式：

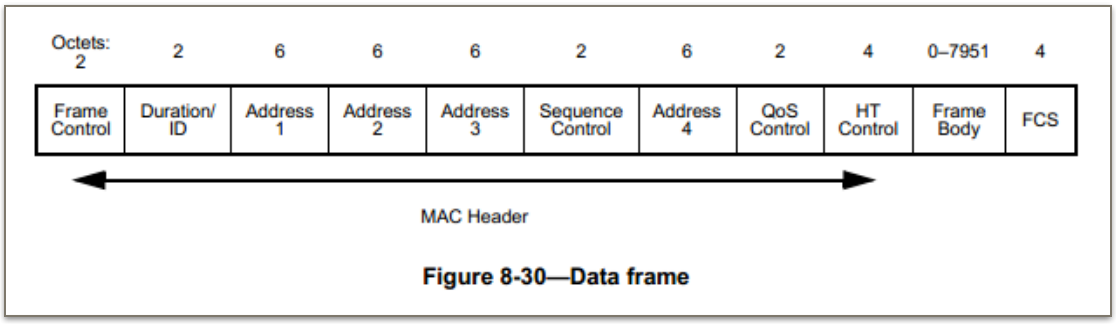


Figure 8-30—Data frame

- Data 帧的 MAC 包头的前 24 字节是必须有的：
  - Address 4 是否存在是由 Frame Control 中的 FromDS 和 ToDS 决定的；
  - QoS Control 是否存在是由 Frame Control 中的 Subtype 决定的；
  - HT Control 域是否存在是由 Frame Control 中的 Order Field 决定的；
  - 具体可参见 IEEE Std 80211-2012。
- 对于 WEP 加密的包，在 MAC 包头后面跟随 4 字节的 IV，在包的结尾（FCS 前）还有 4 字节的 ICV。
- 对于 TKIP 加密的包，在 MAC 包头后面跟随 4 字节的 IV 和 4 字节的 EIV，在包的结尾（FCS 前）还有 8 字节的 MIC 和 4 字节的 ICV。
- 对于 CCMP 加密的包，在 MAC 包头后面跟随 8 字节的 CCMP header，在包的结尾（FCS 前）还有 8 字节的 MIC。

14.2. Sniffer 的应用场景和相关问题

因为部分 AP 不转发 UDP 广播包到 WLAN 空口，所以只能监听手机发的 UDP 包。这些 UDP 包是手机发给 AP 并且已加密的。

应用场景 1：IOT-device 能收到手机发出的所有的包。

手机和 AP 的连接工作在 11b、11g、11n HT20 模式时，并且手机到 AP 的距离大于手机到 IOT-device的距离，则 IOT-device 能收到手机发出的所有的包。

IOT-device 的固件可以通过 MAC 地址和 MAC-header（及 MAC-crypt ion-header）来过滤 UDP 包，也可过滤重传包。

同时，对 11n 的 AMPDU 包，也可得到每个子帧的长度和 MAC-header（及 MAC-crypt ion-header）

应用场景 2：IOT-device 不能收到手机发出的所有的包。手机发包信号很强，但格式 IOT-device 不支持。

有两种情况：

案例 1：

手机到 AP 的距离远大于手机到 IOT-device 的距离。这时，手机发的高数据率的包 AP 能收到，但 IOT-device 收不到。

例如：手机发 MCS7 的包，AP 能正确接收，IOT-device 不能正确接收，但可以解出物理层包头（因为物理层包头是用低速率 6 Mbps 编码）。

案例 2：

手机给 AP 发包为 IOT-device 不支持的格式：

- HT40；
- LDPC 编码；
- 11n MCS8 以上即 MIMO 2x2。

同样，IOT-device 不能正确接收，但可以解出物理层包头 HT-SIG。

在以上两种情况下，IOT-device 可收到 HT-SIG，其中包含物理层包长。利用此信息实现，需要注意以下几个问题：

- 当不使用 AMPDU 或 AMPDU 中只有一个子帧时，可以推测出 UDP 包长。如果手机端 APP 发送 UDP 包序列中的间隔较长 20 ms ~ 50 ms，每个 UDP 包就会在不同的物理层包中，可能是只有一个子帧的 AMPDU 包。
- IOT-device 中的固件可以先用 RSSI 过滤其他设备发的包。

- 重传包需要用包序列中隐含的信息来过滤，即要保证连续两个包的长度都是不同的。例如可采用以下方案：
  - 每两个信息包之间用特定包分隔，称为分隔符包。
  - 奇数包长度 0 ~ 511，偶数包的长度大于 512 ~ 1023。

14.3. 手机 APP 设计

对应用场景 2，手机 APP 要注意以下几点：

- 每个 UDP 包发送间隔为 20 ms 或以上。
- 每两个信息包之间用特定包分隔，称为分隔符包。
- 每个信息包的信息有冗余，前后包可相互校验
- 序列开始时，有标志包。这样，APP 是不停地循环发送整个序列。
- AP 的 BSSID（即 MAC 地址）只需发送最低两个字节，IOT-device 可以扫描到。如果 AP 没有使用隐藏 SSID，SSID 也不用加在信息序列中。所以，要分析 AP beacon 来检查是否为“隐藏SSID”。
- UDP 包的长度要乘以 4。因为，有可能手机发送 AMPDU 中只包含一个子帧的情况。此时，包长会补齐为 4 的整数倍。

对应用场景 1，手机 APP 可以尽快发包。所以，考虑手机 APP 在快速发送和间隔发送之间来回切换。手机 APP 不知道当前是手机 Wi-Fi 发的包对 IOT-device 是应用场景 1 还是应用场景 2。

14.4. IOT-device 上固件设计

对应用场景 2，IOT-device 上固件设计要考虑：

- 用 RSSI 搜索 channel。先在最强的 channel 上搜索特征序列。
- 用 RSSI 过滤无用的包。要考虑空气中 10 ~ 15 db 的波动，主要是有些包会下降 10 db 以上。开始时只收最强的包。找到特征序列头后，可以放宽波动范围。
- 可检查 HT-SIG 的 Aggregation bit，即为 AMPDU 包。
- AMPDU 中只能使用 CCMP（AES）加密。
- 分隔符包长度要考虑不同 QoS、加密算法和 AMPDU 还会取 4 整数倍并加 4。
- 使用相对值获得信息，即用信息包长度减去分隔符包长度。这样就不用处理各种加密和 AMPDU 增加的长度。

附录

📖 **说明：**

关于 *GPIO* 寄存器、*SPI* 寄存器、*UART* 寄存器、定时器寄存器，请参考以下附录。

章	标题	内容
附录 1	GPIO 寄存器	关于 GPIO 寄存器的名称、地址、描述等信息。
附录 2	SPI 寄存器	关于 SPI 寄存器的名称、地址、描述等信息。
附录 3	UART 寄存器	关于 UART 寄存器的名称、地址、描述等信息。
附录 4	定时器寄存器	关于定时器寄存器的名称、地址、描述等信息。