Espressif IoT SDK 编程手册

Status	Released
Current version	V1.0.0
Author	Fei Yu
Completion Date	2015.03.14
Reviewer	JG Wu
Completion Date	2015.03.14

[√] CONFIDENTIAL

[ ] INTERNAL

[ ] PUBLIC

版本信息

日期	版本	撰写人	修改说明
2014.8.13	0.9	Fei Yu	1.修改 espconn 接口； 2.增加 sniffer 接口； 3.增加 chip 查询接口； 4.增加获取、修改 mac&ip 接口；
2014.9.23	0.9.1	Fei Yu	1、增加休眠接口 2、修改 flash 读写接口 3、记录连接过 AP 4、修改 UDP 接口
2014.11.07	0.9.3	Fei Yu	1、新增 DHCP 接口 2、新增 RTC 接口 3、新增 ADC 接口 4、修改 station、softAP 结构体
2014.12.19	0.9.4	Fei Yu	1、新增省电控制接口； 2、新增 igmp 接口；
2015.01.22	0.9.5	Fei Yu	新增如下接口 1、Upgrade 接口； 2、DHCP 相关接口； 3、连接过的 ap 查询 4、smart config 接口 5、TCP 收包阻塞 6、AT 接口
2015.01.22	0.9.5	Fei Yu	新增如下接口 1、Upgrade 接口； 2、DHCP 相关接口； 3、连接过的 ap 查询 4、smart config 接口 5、TCP 收包阻塞 6、AT 接口
2015.03.14	1.0.0	Fei Yu	新增如下 API 1、CPU 频率设置； 2、设置 sniffer mac 过滤； 3、设置 UDP 广播包从哪个接口发送； 4、更新 smart config 接口 5、更新 AT 接口

免责申明和版权公告

本文中的信息，包括供参考的 URL 地址，如有变更，恕不另行通知。

文档“按现状”提供，不负任何担保责任，包括对适销性、适用于特定用途或非侵权性的任何担保，和任何提案、规格或样品在他处提到的任何担保。本文档不负任何责任，包括使用本文档内信息产生的侵犯任何专利权行为的责任。本文档在此未以禁止反言或其他方式授予任何知识产权使用许可，不管是明示许可还是暗示许可。

Wi-Fi 联盟成员标志归 Wi-Fi 联盟所有。

文中提到的所有商标名称、商标和注册商标均属其各自所有者的财产，特此声明。

目录

1. 前言

基于 ESP8266 物联网平台的 SDK 为用户提供了一个简单、快速、高效开发物联网产品的软件平台。

本文旨在介绍该 SDK 的基本框架，以及相关的 API 接口。主要的阅读对象为需要在 ESP8266 物联网平台进行软件开发的嵌入式软件开发人员。

2. 软件框架

为了让用户不用关心底层网络，如 WIFI、TCP/IP 等的具体实现，仅专注于物联网应用的开发，SDK 为用户提供了一套数据接收、发送函数接口，用户只需利用相应接口即可完成网络数据的收发。

ESP8266 物联网平台的所有网络功能均在库中实现，对用户不透明，用户初始化功能在 user_main.c 文件中实现。

函数 void usre_init(void)的作用是给用户提供一个初始化接口，也是上层程序入口，用户可在该函数内增加硬件初始化、网络参数设置、定时器初始化等功能。

SDK 中提供了对 json 包的处理 API，用户也可以采用自定义数据包格式，自行对数据进行处理。

3. SDK 提供的 API 接口

3.1. 定时器接口

说明：定时器接口接口函数或宏以及所使用的参数结构体定义在（工程目录\include\osapi.h）。

3.1.1. os_timer_arm

功能：初始化定时器

函数定义：

Void os_timer_arm(ETSTimer *ptimer,uint32_t milliseconds, bool repeat_flag)

输入参数：

ETSTimer*ptimer——定时器结构（该结构体参见 4.1 说明）

uint32_t milliseconds——定时时间，单位毫秒

bool repeat_flag——该定时是否重复

返回：

无

3.1.2. os_timer_disarm

功能：取消定时器定时

函数定义：

Void os_timer_disarm(ETSTimer *ptimer)

输入参数：

ETSTimer *ptimer——定时器结构（该结构体参见 4.1 说明）

返回：

无

3.1.3. os_timer_setfn

功能：设置定时器回调函数

函数定义：

Void os_timer_setfn(ETSTimer *ptimer, ETSTimerFunc *pfunction, void *parg)

输入参数：

ETSTimer *ptimer——定时器结构（该结构体参见 4.1 说明）

TESTimerFunc *pfunction——定时器回调函数

void*parg——回调函数参数

返回：

无

3.2. 底层用户接口

3.2.1. system_restore

功能：恢复出厂设置

函数定义：

void system_restore(void)

输入参数：

无

返回：

无

3.2.2. system_restart

功能：重启

函数定义：

void system_restart(void)

输入参数：

无

返回：

无

3.2.3. system_timer_reinit

功能：当需要使用 us 级 timer 时，需要重新初始化 timer。

注意：1、同时定义 USE_US_TIMER；

2、system_timer_reinit 需放在最开始，user_init 第一句。

函数定义：

Void system_timer_reinit (void)

输入参数：

无

返回：

无

3.2.4. system_init_done_cb

功能：在 user_init 中调用，注册系统完成初始化的回调函数。

注意：wifi_station_scan 需在系统初始化完成，且 station 使能的条件下调用。

函数定义：

void system_init_done_cb(init_done_cb_t cb)

参数：

init_done_cb_t cb - 系统初始化完成后的回调函数

返回：

无

示例：

```
void to_scan(void)
{
  wifi_station_scan(NULL,scan_done);
```

```
}


void user_init(void)
{
wifi_set_opmode(STATION_MODE);
system_init_done_cb(to_scan);
}
```

3.2.5. system_get_chip_id

功能：获取芯片 id

函数定义：

uint32 system_get_chip_id(void)

输入参数：

无

返回：

芯片 id 值

3.2.6. system_deep_sleep

功能：设置进入 deep sleep 模式，每休眠多长时间（us）唤醒一次。唤醒以后整个系统重新跑，程序从 user_init 开始。

函数定义：

void system_deep_sleep(uint32 time_in_us)

输入参数：

uint32 time_in_us – 设置休眠时间，单位：us

返回：

无

注意：

硬件上需将 XPD_DCDC 通过 0R 连接到 EXT_RSTB，用作 Deep sleep

唤醒。

system_deep_sleep(0) 则永远不醒，可通过外部 GPIO 拉低 RST 脚唤醒

3.2.7. system_deep_sleep_set_option

功能：在 deep sleep 前调用，用以在 deep sleep 醒来后判断作什么操作。

说明：后述 init 参数指 esp_init_data_default.bin

函数定义：

bool system_deep_sleep_set_option(uint8 option)

输入参数：

uint8 option - option=0 时，init 参数 byte 108 有效，
option>0 时，init 参数的 byte 108 无效。

具体如下：

deep_sleep_set_option(0) 表示由 init 参数 byte 108 控制 deep sleep 醒来后的是否作 RF_CAL。

deep_sleep_set_option(1) 表示 deep sleep 醒来后和上电一样，要作 RF_CAL，电流较大。

deep_sleep_set_option(2) 表示 deep sleep 醒来后不作 RF_cal，电流较小。

deep_sleep_set_option(4) 表示 deep sleep 醒来后，不打开 RF，和 modem sleep 一样，电流最小。

返回：

True，成功；False，失败

3.2.8. system_set_os_print

功能：开关打印 log 功能。

函数定义：

Void system_set_os_print(uint8 onoff)

输入参数：

Uint8 onoff —— 打开/关闭打印功能；

0x00 代表关闭打印功能

0x01 代表打开打印功能

默认为打开打印功能。

返回：

无

3.2.9. system_print_meminfo

功能：打印系统内存空间分配，打印信息包括 data/rodata/bss/heap

函数定义：

Void system_print_meminfo (void)

输入参数：

无

返回：

无

3.2.10. system_get_free_heap_size

功能：获取系统可用 heap 区空间大小

函数定义：

Uint32 system_get_free_heap_size(void)

输入参数：

无

返回：

Uint32 ——可用 heap 区大小

3.2.11. system_os_task

功能：建立系统任务

函数定义：

bool system_os_task(os_task_t task, uint8 prio, os_event_t *queue, uint8 qlen)

输入参数：

Os_task_t task——任务函数

Uint8 prio——任务优先级，当前支持 3 个邮件级的任务 0/1/2，0 最低

Os_event_t *queue——消息队列指针

Uint8 qlen——消息队列深度

返回：

True，成功；False，失败

示例：

```
#define SIG_RX     0
#define TEST_QUEUE_LEN     4
os_event_t *testQueue;
void test_task (os_event_t *e)
{
    switch (e->sig) {
        case SIG_RX:
            os_printf("sig_rx %c\n", (char)e->par);
            break;
        default:
            break;
    }
}
void task_init(void)
{
testQueue=(os_event_t*)os_malloc(sizeof(os_event_t)*TEST_QUEUE_LEN);
system_os_task(test_task,USER_TASK_PRIO_0,testQueue,TEST_QUEUE_
LEN);
}
```

3.2.12. system_os_post

功能：向任务发送消息

函数定义：

bool system_os_post (uint8 prio, os_signal_t sig, os_param_t par)

输入参数：

Uint8 prio——任务优先级，与建立时的优先级对应

Os_signal_t sig——消息类型

Os_param_t par——消息参数

返回：

True，成功；False，失败

结合上一节的示例：

```
void task_post(void)
{
system_os_post(USER_TASK_PRIO_0, SIG_RX, 'a');
}
```

打印输出：sig_rx a

3.2.13. system_get_time

功能：获得系统时间，单位：微秒 us。

函数定义：

uint32 system_get_time(void)

输入参数：

Null

返回：

系统时间，单位：微秒。如果计时满，则归零重新计。

3.2.14. system_get_rtc_time

功能：获得 RTC 时间，单位：RTC 时钟周期。

举例：system_get_rtc_time() 返回 10，表示当前走了 10 个 RTC 时钟周期；system_rtc_clock_cali_proc 返回 5，表示一个 RTC 时钟周期为 5 us，则实际时间为 10 x 5 = 50 us。

注意：deep sleep（或者 system_restart）时，系统时间归零，但是 RTC 时间仍然继续。

函数定义：

uint32 system_get_rtc_time(void)

输入参数：

Null

返回：

RTC 时间，单位：微秒。如果计时满，则归零重新计。

3.2.15. system_rtc_clock_cali_proc

功能：RTC 校准函数。

函数定义：

uint32 system_rtc_clock_cali_proc(void)

输入参数：

Null

返回：

返回 RTC 时钟周期。单位 us，bit11 到 bit0 为小数部分。

注意：RTC 系列接口的使用示例，见附录。

3.2.16. system_rtc_mem_write

功能：由于 deep sleep 时，仅 RTC 在工作，用户如有需要，可将数据存入 RTC memory 中。提供如下图中 user data 段共 512 bytes 供用户存储数据。

|_ _ _ _ _system data _ _ _ _ _|_ _ _ _ _ _ _ _ _ user data _ _ _ _ _ _ _ _ _|
| 256 bytes | 512 bytes |

注意： RTC memory 只能 4 字节整存整取，函数中参数 des_addr 为 block number，4 字节每 block，因此若写入上图 user data 区起始位置，des_addr 为 256/4 = 64，save size 为存入数据的字节数。

函数定义：

bool system_rtc_mem_write (uint32 des_addr, void * src_addr, uint32 save_size)

输入参数：

uint32 des_addr —— 写入 rtc memory 的位置，des_addr >=64

void * src_addr —— 数据指针

uint32 save_size ——数据长度，单位 byte

返回：

True， 成功；False，失败。

3.2.17. system_rtc_mem_read

功能：读取 RTC memory 中的数据，提供如下图中 user data 段共 512 bytes 给用户存储数据。

|_ _ _ _ _system data _ _ _ _ _|_ _ _ _ _ _ _ _ _ user data _ _ _ _ _ _ _ _ _|
| 256 bytes | 512 bytes |

注意： RTC memory 只能 4 字节整存整取，函数中的参数 src_addr 为 block number，4 字节每 block，因此若读取上图 user data 区起始位置，src_addr 为 256/4 = 64，save size 为存入数据的字节数。

函数定义：

bool system_rtc_mem_read (uint32 src_addr, void * des_addr, uint32 save_size)

输入参数：

uint32 src_addr —— 读取 rtc memory 的位置，src_addr >=64

void * des_addr —— 数据指针

uint32 save_size ——数据长度，单位 byte

返回：

True， 成功；False，失败。

3.2.18. system_uart_swap

功能：UART0 转换。将 MTCK 作为 UART0 RX，MTDO 作为 UART0 TX。硬件上也从 MTDO(U0CTS) 和 MTCK(U0RTS) 连出 UART0，从而避免上电时从 UART0 打印出 ROM LOG。

函数定义：

void system_uart_swap (void)

参数：

NULL

返回：

NULL

3.2.19. system_get_boot_version

功能：读取 boot 版本。

函数定义：

Uint8 system_get_boot_version (void)

参数：

NULL

返回：

Boot 版本号。

说明：

Boot 版本号 >= 3 时，支持 boot 增强模式（详见 system_restart_enhance）

3.2.20. system_get_userbin_addr

功能：读取当前正在运行的 user1.bin（或者 user2.bin）的存放地址。

函数定义：

uint32 system_get_userbin_addr (void)

参数：

NULL

返回：

User bin 的存放地址。

3.2.21. system_get_boot_mode

功能：读取 boot 模式。

函数定义：

Uint8 system_get_boot_mode (void)

参数：

NULL

返回：

#define SYS_BOOT_ENHANCE_MODE 0

#define SYS_BOOT_NORMAL_MODE 1

说明：

Boot 增强模式：支持跳转到任意位置运行程序；

Boot 普通模式：仅能跳转到固定的 user1.bin（或 user2.bin）位置运行。

3.2.22. system_restart_enhance

功能：重启系统，进入 Boot 增强模式。

函数定义：

bool system_restart_enhance(uint8 bin_type, uint32 bin_addr)

参数：

uint8 bin_type – bin 的类型

#define SYS_BOOT_NORMAL_BIN 0

#define SYS_BOOT_TEST_BIN 1 // 仅支持 Espressif test bin

uint32 bin_addr – bin 的存放地址

返回：

TRUE，成功；FALSE，失败。

说明：

SYS_BOOT_TEST_BIN 用于产测，客户可以向 Espressif 申请产测方案。

3.2.23. system_update_cpu_freq

功能：调整 CPU 频率。默认为 80MHz。

函数定义：

bool system_update_cpu_freq(uint8 freq)

参数：

uint8 freq – CPU 频率

#define SYS_CPU_80MHz 80

#define SYS_CPU_160MHz 160

返回：

TRUE，成功；FALSE，失败。

3.2.24. system_get_cpu_freq

功能：查询 CPU 频率。

函数定义：

Uint8 system_get_cpu_freq(void)

参数：

NULL

返回：

CPU 频率，单位 MHz。

3.3. SPI Flash 相关接口

3.3.1. spi_flash_get_id

功能：获取 spi flash id。

函数定义：

Uint32 spi_flash_get_id (void)

参数：

Null

返回：

SPI Flash id

3.3.2. spi_flash_erase_sector

功能：擦除 Flash 的某个扇区。

说明：flash 读写操作的介绍，详见文档“Espressif IOT Flash 读写说明”。

函数定义：

SpiFlashOpResult spi_flash_erase_sector (uint16 sec)

参数：

uint16 sec – 扇区号，从扇区 0 开始计数，每扇区 4KB

返回：

```
Typedef enum{
    SPI_FLASH_RESULT_OK,
    SPI_FLASH_RESULT_ERR,
    SPI_FLASH_RESULT_TIMEOUT
}SpiFlashOpResult;
```

3.3.3. spi_flash_write

功能：将数据存到 Flash。

说明：flash 读写操作的介绍，详见文档“Espressif IOT Flash 读写说明”。

函数定义：

SpiFlashOpResult spi_flash_write (uint32 des_addr, uint32 *src_addr, uint32 size)

参数：

uint32 des_addr - 写入 Flash 的地址，起始位置。

uint32 *src_addr - 写入 Flash 的数据指针。

Uint32 size - 写入数据长度

返回：

```
Typedef enum{
    SPI_FLASH_RESULT_OK,
    SPI_FLASH_RESULT_ERR,
    SPI_FLASH_RESULT_TIMEOUT
}SpiFlashOpResult;
```

3.3.4. spi_flash_read

功能： 从 flash 读取数据。

说明：flash 读写操作的介绍，详见文档“Espressif IOT Flash 读写说明”。

函数定义：

SpiFlashOpResult spi_flash_read(uint32 src_addr, uint32 * des_addr, uint32 size)

参数：

uint32 src_addr- 读取 Flash 的地址，起始位置。

uint32 * des_addr – 存储读取到数据的指针。

Uint32 size - 读取数据长度

返回：

```
Typedef enum{
    SPI_FLASH_RESULT_OK,
    SPI_FLASH_RESULT_ERR,
    SPI_FLASH_RESULT_TIMEOUT
}SpiFlashOpResult;
```

3.4. WIFI 相关接口

3.4.1. wifi_get_opmode

功能：获取 wifi 工作模式

函数定义：

uint8 wifi_get_opmode (void)

输入参数：

无

返回：

wifi 工作模式。

#define STATION_MODE 0x01

#define SOFTAP_MODE 0x02

#define STATIONAP_MODE 0x03

3.4.2. wifi_set_opmode

功能：设置 wifi 工作模式为 STATION、SOFTAP、STATION+SOFTAP

说明：在 esp_iot_sdk_v0.9.2 之前版本，本接口调用后，需要重启生效；v0.9.2（含）之后版本，无需重启即可生效。

函数定义：

bool wifi_set_opmode (uint8 opmode)

输入参数：

uint8 opmode——wifi 工作模式

具体如下：

#define STATION_MODE 0x01

#define SOFTAP_MODE 0x02

#define STATIONAP_MODE 0x03

返回：

True ， 成功；False ， 失败

3.4.3. wifi_station_get_config

功能：获取 wifi 的 station 接口参数

函数定义：

bool wifi_station_get_config (struct station_config *config)

输入参数：

struct station_config *config——wifi 的 station 接口参数指针

返回：

True ， 成功；False ， 失败

3.4.4. wifi_station_set_config

功能：设置 wifi 的 station 接口参数。

注意：如果在 user_init 中调用 wifi_station_set_config，底层会自动连接对应路由，不需要调用 wifi_station_connect 来进行连接。

请注意 station_config.bssid_set 参数一般需初始化为 0。

函数定义：

bool wifi_station_set_config (struct station_config *config)

输入参数：

struct station_config *config——wifi 的 station 接口参数指针

返回：

True ， 成功；False ， 失败

3.4.5. wifi_station_connect

功能：wifi 的 station 接口连接所配置的路由

注意：如果连接过路由，请先 wifi_station_disconnect，再调用 wifi_station_connect。

函数定义：

bool wifi_station_connect(void)

输入参数：

无

返回：

True ， 成功；False ， 失败

3.4.6. wifi_station_disconnect

功能：wifi 的 station 接口断开所连接的路由

函数定义：

```
bool wifi_station_disconnect(void)
```

输入参数：

无

返回：

True ， 成功；False ， 失败

3.4.7. wifi_station_get_connect_status

功能：获取 wifi station 接口连接 AP 的状态

函数定义：

```
uint8 wifi_station_get_connect_status (void)
```

输入参数：

无

返回：

```
enum{
    STATION_IDLE = 0,
    STATION_CONNECTING,
    STATION_WRONG_PASSWORD,
    STATION_NO_AP_FOUND,
    STATION_CONNECT_FAIL,
    STATION_GOT_IP
};
```

3.4.8. wifi_station_scan

功能：获取 AP 热点信息

注意：请勿在 user_init 中调用本接口，因为此时 station 接口尚未完成初始化。

函数定义：

```
bool wifi_station_scan (struct scan_config *config, scan_done_cb_t cb);
```

结构体：

```
struct scan_config{
    uint8 *ssid;          // AP 的 ssid
    uint8 *bssid;         // AP 的 bssid
    uint8 channel;        //扫描某特定信道
    uint8 show_hidden;    //是否扫描隐藏 ssid 的 AP
};
```

输入参数：

struct scan_config *config – 扫描 AP 的相关参数，传 NULL 表示此项不设限。

例如，若 config 传 NULL，则扫描获取所有 AP 的信息；

若 config 中 ssid、bssid 传 NULL，仅设置 channel，则扫描该特定 channel 上的 AP 信息。

若 config 中 ssid 指定，bssid 为 NULL，channel 为 0，则在所有信道上扫描某指定 AP。

scan_done_cb_t cb - 获取 AP 热点信息回调 function

返回：

True ，成功；False ，失败

3.4.9. scan_done_cb_t

功能：scan 回调 function

函数定义：

```
void scan_done_cb_t (void *arg, STATUS status);
```

输入参数：

void *arg——获取的 AP 热点信息入口参数，arg 指针需要转换为 struct bss_info 结构体指针来解析所扫描的 AP 信息，AP 热点信息以链表形式储存（参见 struct bss_info 结构体定义）

STATUS status——获取结果

返回：

无

示例：

```
wifi_station_scan(&config, scan_done);
static void ICACHE_FLASH_ATTR
scan_done(void *arg, STATUS status)
{
  if (status == OK)
  {
    struct bss_info *bss_link = (struct bss_info *)arg;
    bss_link = bss_link->next.stqe_next;//ignore first
    ……
    }
}
```

3.4.10. wifi_station_ap_number_set

功能：设置 ESP8266 station 最多可记录几个 AP 的信息

函数定义：

bool wifi_station_ap_number_set (uint8 ap_number);

输入参数：

uint8 ap_number—— 最多可记录 AP 的数目（MAX: 5）。

返回：

True ， 成功；False ， 失败

3.4.11. wifi_station_get_ap_info

功能：获取 ESP8266 station 曾经连接过的 AP 的信息，最多纪录 5 个。

函数定义：

uint8 wifi_station_get_ap_info(struct station_config config[])

参数：

struct station_config config[] — 获取记录的所有 AP 连接信息，最多 5 个，因此请传入数组大小为 5。

返回：

实际纪录的 AP 信息个数。

示例：

```
struct station_config config[5];
int i = wifi_station_get_ap_info(config);
```

3.4.12. wifi_station_ap_change

功能：ESP8266 station 切换到第几号 AP 配置进行连接

函数定义：

bool wifi_station_ap_change (uint8 current_ap_id);

输入参数：

uint8 current_ap_id——第几号 AP 配置。从 0 开始计数。

返回：

True ， 成功；False ， 失败

3.4.13. wifi_station_get_current_ap_id

功能：当前正在使用第几号记录的 AP。ESP8266 每配置连接一个 AP，会进行记录，从 0 开始计数，这样，如果当前 AP 无法成功联网时，ESP8266 会自动切换到下一个 AP 进行连接。

函数定义：

Uint8 wifi_station_get_current_ap_id ();

输入参数：

无

返回：

当前使用的 AP id，即当前 AP 为 ESP8266 记录的第几号配置信息。

3.4.14. wifi_station_get_auto_connect

功能：查询 ESP8266 station 上电是否会自动连接已记录的 AP (路由)

函数定义：

uint8 wifi_station_get_auto_connect(void)

输入参数：

Null

返回：

0 ，不自动连接； 非 0 ，自动连接

3.4.15. wifi_station_set_auto_connect

功能：设置 ESP8266 station 上电是否自动连接已记录的 AP (路由)

注意：此 api 如果在 user_init 中调用，则当前这次上电就生效；如果在其他地方调用，则下一次上电生效。

函数定义：

bool wifi_station_set_auto_connect(uint8 set)

输入参数：

uint8 set — 是否自动连接；0，关闭自动连接；1，开启自动连接。

返回：

True ，成功；False ，失败

3.4.16. wifi_station_dhcpc_start

功能：开启 ESP8266 station dhcp client.

注意：dhcp 默认开启。

函数定义：

bool wifi_station_dhcpc_start(void)

输入参数：

Null

返回：

True ， 成功；False ， 失败

3.4.17. wifi_station_dhcpc_stop

功能：关闭 ESP8266 station dhcp client.

注意：dhcp 默认开启。

函数定义：

bool wifi_station_dhcpc_stop(void)

输入参数：

Null

返回：

True ， 成功；False ， 失败

3.4.18. wifi_station_dhcpc_status

功能：查询 ESP8266 station dhcp client 状态

函数定义：

enum dhcp_status wifi_station_dhcpc_status(void)

输入参数：

Null

返回：

```
enum dhcp_status {
    DHCP_STOPPED,
    DHCP_STARTED
};
```

3.4.19. wifi_softap_get_config

功能：设置 wifi 的 softap 接口参数

函数定义：

bool wifi_softap_get_config(struct softap_config *config)

输入参数：

struct softap_config *config——wifi 的 softap 接口参数指针（详见结构体 softap_config 说明）

返回：

True ，成功；False ，失败

3.4.20. wifi_softap_set_config

功能：设置 wifi 的 softap 接口参数

函数定义：

bool wifi_softap_set_config (struct softap_config *config)

输入参数：

struct softap_config *config——wifi 的 softap 接口参数指针（详见结构体 softap_config 说明）

返回：

True ，成功；False ，失败

3.4.21. wifi_softap_get_station_info

功能：获取 softap 模式下连接的 station 设备信息，包括 mac 和 ip

函数定义：

struct station_info * wifi_softap_get_station_info(void)

输入参数：

无

返回：

struct station_info *——所连接的 station 信息链表

3.4.22. wifi_softap_free_station_info

功能：释放由于调用 wifi_softap_get_station_info 函数生成的 struct station_info 空间

函数定义：

void wifi_softap_free_station_info (void)

输入参数：

无

返回：

无

获取 mac、ip 信息示例，注意释放资源：

方法一：

```
struct station_info * station = wifi_softap_get_station_info();
struct station_info * next_station;
while(station){
        os_printf("bssid : "MACSTR", ip : "IPSTR"\n",
MAC2STR(station->bssid), IP2STR(&station->ip));
        next_station = STAILQ_NEXT(station, next);
        os_free(station);  //直接释放
        station = next_station;
}
```

方法二：

```
struct station_info * station = wifi_softap_get_station_info();
while(station){
  os_printf("bssid : "MACSTR", ip : "IPSTR"\n", MAC2STR(station->bssid),
IP2STR(&station->ip));
  station = STAILQ_NEXT(station, next);
}
wifi_softap_free_station_info(); //调用函数释放
```

3.4.23. wifi_softap_dhcps_start

功能：开启 ESP8266 softAP dhcp server.

注意：dhcp 默认开启。

函数定义：

bool wifi_softap_dhcps_start(void)

输入参数：

Null

返回：

True ，成功；False ，失败

3.4.24. wifi_softap_dhcps_stop

功能：关闭 ESP8266 softAP dhcp server.

注意：dhcp 默认开启。

函数定义：

bool wifi_softap_dhcps_stop(void)

输入参数：

Null

返回：

True ，成功；False ，失败

3.4.25. wifi_softap_set_dhcps_lease

功能：设置 ESP8266 softAP dhcp server 分配 IP 地址的范围

注意：必须在 dhcp server 关闭的情况下设置。

函数定义：

bool wifi_softap_set_dhcps_lease(struct dhcps_lease *please)

参数：

struct dhcps_lease {

uint32 start_ip;

```
    uint32 end_ip;
    };
```

返回：

True ， 成功；False ， 失败

3.4.26. wifi_softap_dhcps_status

功能：查询 ESP8266 softAP dhcp server 状态

函数定义：

```
    enum dhcp_status wifi_softap_dhcps_status(void)
```

参数：

NULL

返回：

```
    enum dhcp_status {
        DHCP_STOPPED,
        DHCP_STARTED
    };
```

3.4.27. wifi_set_phy_mode

功能：设置 ESP8266 物理层模式（802.11b/g/n）.

注意：ESP8266 softAP 只支持 bg。

函数定义：

```
    bool wifi_set_phy_mode(enum phy_mode mode)
```

输入参数：

```
    enum phy_mode mode – 设置的模式
    enum phy_mode{
        PHY_MODE_11B = 1,
        PHY_MODE_11G = 2,
        PHY_MODE_11N = 3
    };
```

返回：

True ， 成功；False ， 失败

3.4.28. wifi_get_phy_mode

功能：查询 ESP8266 物理层模式（802.11b/g/n）

函数定义：

Enum phy_mode wifi_get_phy_mode(void)

输入参数：

Null

返回：

```
enum phy_mode{
    PHY_MODE_11B = 1,
    PHY_MODE_11G = 2,
    PHY_MODE_11N = 3
};
```

3.4.29. wifi_get_ip_info

功能：获取 wifi 的 station 或 softap 接口 ip 信息

注意：默认未连接情况下 station 模式的 ip 地址都为 0，因此可以使用此函数来判断本地是否成功连接 DHCP 模式的 AP(路由)，另外 softap 模式下默认的 ip 地址为：192.168.4.1

函数定义：

bool wifi_get_ip_info(uint8 if_index, struct ip_info *info)

输入参数：

uint8 if_index——获取 ip 信息的接口，其中 STATION_IF 为 0x00，SOFTAP_IF 为 0x01。

struct ip_info *info——获取的指定接口的 ip 信息指针

返回：

True ， 成功；False ， 失败

3.4.30. wifi_set_ip_info

功能：修改 ip 地址

函数定义：

bool wifi_set_ip_info(uint8 if_index, struct ip_info *info)

输入参数：

uint8 if_index – 设置 station ip 还是 softAP ip

#define STATION_IF 0x00

#define SOFTAP_IF 0x01

struct ip_info *info – ip 信息

示例：

```
struct ip_info info;

IP4_ADDR(&info.ip, 192, 168, 3, 200);
IP4_ADDR(&info.gw, 192, 168, 3, 1);
IP4_ADDR(&info.netmask, 255, 255, 255, 0);
wifi_set_ip_info(STATION_IF, &info);

IP4_ADDR(&info.ip, 10, 10, 10, 1);
IP4_ADDR(&info.gw, 10, 10, 10, 1);
IP4_ADDR(&info.netmask, 255, 255, 255, 0);
wifi_set_ip_info(SOFTAP_IF, &info);
```

返回：

True ， 成功；False ， 失败

3.4.31. wifi_set_macaddr

功能：设置 mac 地址

注意：建议在 user_init 中调用此接口修改 mac 地址。

函数定义：

bool wifi_set_macaddr(uint8 if_index, uint8 *macaddr)

输入参数：

uint8 if_index – 设置 station mac 还是 softAP mac

#define STATION_IF 0x00

#define SOFTAP_IF 0x01

uint8 *macaddr – mac 地址

示例：

```
char sofap_mac[6] = {0x16, 0x34, 0x56, 0x78, 0x90, 0xab};
char sta_mac[6] = {0x12, 0x34, 0x56, 0x78, 0x90, 0xab};

wifi_set_macaddr(SOFTAP_IF, sofap_mac);
wifi_set_macaddr(STATION_IF, sta_mac);
```

返回：

True ， 成功；False ， 失败

3.4.32. wifi_get_macaddr

功能：获取 wifi 的 station 或 softap 接口 mac 信息

函数定义：

Bool wifi_get_macaddr(uint8 if_index , uint8 *macaddr)

输入参数：

uint8 if_index——获取 mac 信息的接口，其中 STATION_IF 为 0x00，SOFTAP_IF 为 0x01。

uint8 *macaddr——获取的指定接口的 mac 信息指针

返回：

True ， 成功；False ， 失败

3.4.33. wifi_set_sleep_type

功能：设置省电模式。设置为 NONE_SLEEP_T，则关闭省电模式。

注意：ESP8266 仅在单 station 模式支持省电，共支持 Modem sleep，light sleep，deep sleep 三种省电模式，默认为 Modem sleep 模式。Deep sleep 会即时休眠，请调用 system_deep_sleep 设置。

函数定义：

Bool wifi_set_sleep_type(enum sleep_type type)

参数：

enum sleep_type type —— 设置省电模式

返回：

True ，成功；False ，失败

3.4.34. wifi_get_sleep_type

功能：获取当前的省电模式。

函数定义：

Enum sleep_type wifi_get_sleep_type(void)

参数：

NULL

返回：

```
Enum sleep_type{
    NONE_SLEEP_T = 0;
    LIGHT_SLEEP_T,
    MODEM_SLEEP_T
};
```

3.4.35. wifi_status_led_install

功能：注册 wifi led 状态 led

函数定义：

Void wifi_status_led_install (uint8 gpio_id, uint32 gpio_name, uint8 gpio_func)

输入参数：

uint8 gpio_id——gpio 号

uint8 gpio_name——gpio mux 名

uint8 gpio_func——gpio 功能

返回：

无

示例：

使用 GPIO0 作为 wifi 状态 LED

```
#define HUMITURE_WIFI_LED_IO_MUX      PERIPHS_IO_MUX_GPIO0_U
#define HUMITURE_WIFI_LED_IO_NUM      0
#define HUMITURE_WIFI_LED_IO_FUNC     FUNC_GPIO0
wifi_status_led_install(HUMITURE_WIFI_LED_IO_NUM,
HUMITURE_WIFI_LED_IO_MUX, HUMITURE_WIFI_LED_IO_FUNC)
```

3.4.36. wifi_status_led_uninstall

功能：注销 wifi led 状态 led

函数定义：

Void wifi_status_led_uninstall ()

参数：

NULL

返回：

NULL

3.4.37. wifi_set_broadcast_if

功能：设置 ESP8266 发送 UDP 广播包时，从 station 接口还是 softAP 接口发送。默认从 softAP 接口发送。

函数定义：

Bool wifi_set_broadcast_if (uint8 interface)

参数：

uint8 interface ——　1，station 接口

2，softAP 接口

3，station 接口和 softAP 接口均发送

返回：

True ，成功；False ，失败

3.4.38. wifi_get_broadcast _if

功能：获取当前的 UDP 广播包的发送接口。

函数定义：

Uint8 wifi_get_broadcast_if (void)

参数：

NULL

返回：

1，从 station 接口发送

2，从 softAP 接口发送

3，station 接口和 softAP 接口均发送

3.5. 云端升级接口

3.5.1. system_upgrade_userbin_check

功能：检查当前正在使用的 firmware 是 user1 还是 user2。

函数定义：

uint8 system_upgrade_userbin_check()

输入参数：

无

返回：

0x00 : UPGRADE_FW_BIN1 ，即 user1.bin

0x01 : UPGRADE_FW_BIN2 ， 即 user2.bin

3.5.2. system_upgrade_flag_set

功能：设置 upgrade 状态 flag。

说明：如直接调用 system_upgrade_start 采用 Espressif 提供的云端升级方式，SDK 底层会设置 flag，上层无需再调用；

如用户自行调用 spi_flash_write 实现云端升级，则需调用本接口设置为 UPGRADE_FLAG_FINISH，再调用 system_upgrade_reboot 重启切换到新软件运行。

函数定义：

```
void system_upgrade_flag_set(uint8 flag)
```

参数：

uint8 flag – 具体取值如下：

```
#define UPGRADE_FLAG_IDLE      0x00
#define UPGRADE_FLAG_START 0x01
#define UPGRADE_FLAG_FINISH 0x02
```

返回值：

无

3.5.3. system_upgrade_flag_check

功能：查询 upgrade 状态 flag。

函数定义：

```
uint8 system_upgrade_flag_check()
```

参数：

无

返回值：

```
#define UPGRADE_FLAG_IDLE      0x00
#define UPGRADE_FLAG_START 0x01
#define UPGRADE_FLAG_FINISH 0x02
```

3.5.4. system_upgrade_start

功能：配置参数，开始升级。

函数定义：

bool system_upgrade_start (struct upgrade_server_info *server)

参数：

struct upgrade_server_info *server - server 相关的参数。

返回

True ：开始进行升级。

False ：已正在升级过程中，无法执行 upgrade start。

3.5.5. system_upgrade_reboot

功能：重启系统，运行新版本软件。

函数定义：

void system_upgrade_reboot (void)

输入参数：

无

返回：

无

3.6. sniffer 相关接口

3.6.1. wifi_promiscuous_enable

功能：使能混杂模式，用于 sniffer

函数定义：

Void wifi_promiscuous_enable(uint8 promiscuous)

输入参数：

uint8 promiscuous —— 0, disable promiscuous

1, enable promiscuous

返回：

无

示例：可向 Espressif 申请实现 sniffer 的 demo

3.6.2. wifi_promiscuous_set_mac

功能：设置 sniffer 时的 MAC 地址过滤

注意：mac 过滤仅对当前这次的 sniffer 有效，如果停止 sniffer，又再次 sniffer，需要重新设置 mac 过滤。

函数定义：

Void wifi_promiscuous_set_mac(const uint8_t *address)

输入参数：

const uint8_t *address —— MAC 地址

返回：

无

3.6.3. wifi_set_promiscuous_rx_cb

功能：注册 wifi 混杂模式下的 call back 函数，每收到一包数据，都会进入注册的回调函数。

函数定义：

Void wifi_set_promiscuous_rx_cb(wifi_promiscuous_cb_t cb)

输入参数：

wifi_promiscuous_cb_t cb——回调函数，详见附录 C

返回：

无

3.6.4. wifi_get_channel

功能：用于 sniffer 功能，获取信道号

函数定义：

Uint8 wifi_get_channel (void)

输入参数：

无

返回：

信道号

3.6.5. wifi_set_channel

功能：用于 sniffer 功能，设置信道号

函数定义：

bool wifi_set_channel (uint8 channel)

输入参数：

uint8 channel——信道号

返回：

True ，成功；False ，失败

3.7. smart config 接口

3.7.1. smartconfig_start

功能：配置设备连接 AP。

注意：本接口调用后，设备被设置成 station 模式。打开相应的手机 APP 进行操作，设备会监听到需要连接 AP 的 ssid 和 password。smartconfig_start 未完成之前不可重复执行 smartconfig_start 函数。

函数定义：

bool smartconfig_start(sc_type type, sc_callback_t cb, …)

参数：

sc_type type ——选择使用哪种协议连接：AirKiss 或者 ESP-TOUCH。

sc_callback_t cb——设备监听到 ssid 和 password 后的回调函数，传入回调函数的参数为 struct station_config 类型的指针变量。

… —— 可变参数。1，串口打印连接过程；否则，串口仅打印最后结果。

返回：

True，成功； False ，失败

示例：

```
void ICACHE_FLASH_ATTR
smartconfig_done(void *data)
{
    struct station_config *sta_conf = data;

    wifi_station_set_config(sta_conf);
    wifi_station_disconnect();
    wifi_station_connect();
    user_devicefind_init();
    user_esp_platform_init();
}
smartconfig_start(SC_TYPE_ESPTOUCH,smartconfig_done);
```

3.7.2. smartconfig_stop

功能：停止配置设备，并且清空调用 smartconfig_start 函数所申请的内存。

说明：连上 AP 后，可调用本接口释放内存。

函数定义：

```
bool smartconfig_stop(void)
```

参数：

NULL

返回：

True，成功； False ，失败

3.7.3. get_smartconfig_status

功能：获取连接状态

说明：请勿在 smartconfig_stop 后调用本接口。因为 smartconfig_stop 会释放内存，包括本接口的查询状态。

函数定义：

```
sc_status get_smartconfig_status(void)
```

参数：

NULL

返回：

```
typedef enum  {
 SC_STATUS_WAIT = 0,
 SC_STATUS_FIND_CHANNEL,
 SC_STATUS_GETTING_SSID_PSWD,
 SC_STATUS_GOT_SSID_PSWD,
 SC_STATUS_LINK,
 SC_STATUS_LINK_OVER,
} sc_status;
```

注：SC_STATUS_FIND_CHANNEL 状态：设备处于扫描信道的过程，这时才可开启 APP 进行配对连接。

3.8. 网络连接相关接口

说明：以下所有接口函数定义在（工程目录\include\espconn.h）

通用接口：TCP 和 UDP 均可以调用的接口。

TCP 连接接口：仅建立 TCP 连接时，使用的接口。

UDP 接口：仅收发 UDP 包时，使用的接口。

3.8.1. 通用接口

3.8.1.1. espconn_delete

功能：删除传输连接（对应 TCP ：espconn_accept， UDP：espconn_create）

函数定义：

Sin8 espconn_delete(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.1.2. espconn_gethostbyname

功能：域名解析

函数定义：

Err_t espconn_gethostbyname(struct espconn *pespconn, const char *hostname, ip_addr_t *addr, dns_found_callback found)

输入参数：

struct espconn *espconn——相应连接的控制块结构

const char *hostname——域名 string 指针

ip_addr_t *addr——ip 地址

dns_found_callback found——回调

返回：

Err_t——ESPCONN_OK

ESPCONN_INPROGRESS

ESPCONN_ARG

示例如下，具体可参考 IoT Demo 代码：

```
ip_addr_t esp_server_ip;
LOCAL void ICACHE_FLASH_ATTR
user_esp_platform_dns_found(const char *name, ip_addr_t *ipaddr, void *arg)
{
    struct espconn *pespconn = (struct espconn *)arg;

    os_printf("user_esp_platform_dns_found %d.%d.%d.%d\n",
            *((uint8 *)&ipaddr->addr), *((uint8 *)&ipaddr->addr + 1),
            *((uint8 *)&ipaddr->addr + 2), *((uint8 *)&ipaddr->addr + 3));
}
Void dns_test(void)
{
espconn_gethostbyname(pespconn,"iot.espressif.cn",&esp_server_ip,user_es
p_platform_dns_found);
}
```

3.8.1.3. espconn_port

功能：获取未使用的端口

函数定义：

```
uint32 espconn_port(void);
```

输入参数：

无

返回：

uint32——获取的端口号

3.8.1.4. espconn_regist_sentcb

功能：注册数据发送完成的回调函数，数据发送成功后回调

函数定义：

Sint8 espconn_regist_sentcb(struct espconn *espconn, espconn_sent_callback sent_cb)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_sent_callback sent_cb——注册的回调函数

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.1.5. espconn_regist_recvcb

功能：注册数据接收函数，收到数据时回调

函数定义：

Sint8 espconn_regist_recvcb(struct espconn *espconn, espconn_recv_callback recv_cb)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_connect_callback connect_cb——注册的回调函数

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.1.6. espconn_sent_callback

功能：数据发送结束回调

函数定义：

void espconn_sent_callback (void *arg)

输入参数：

void *arg——回调函数参数

返回：

无

3.8.1.7. espconn_recv_callback

功能：接收数据回调函数

函数定义：

void espconn_recv_callback (void *arg, char *pdata, unsigned short len)

输入参数：

void *arg——回调函数参数

char *pdata——接收数据入口参数

unsigned short len——接收数据长度

返回：

无

3.8.1.8. espconn_sent

功能：发送数据

注意：必须等到前一包发送的数据发完（进入 espconn_sent_callback），才能调用 espconn_sent 发送下一包数据。

函数定义：

Sint8 espconn_sent(struct espconn *espconn, uint8 *psent, uint16 length)

输入参数：

struct espconn *espconn——相应连接的控制块结构

uint8 *psent——sent 数据指针

uint16 length——sent 数据长度

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.2. TCP 连接接口

3.8.2.1. espconn_accept

功能：当建立了一个 TCP server 时，调用此接口侦听连接。

函数定义：

Sin8 espconn_accept(struct espconn *espconn)

参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.2.2. espconn_secure_accept

功能：当建立了一个 TCP server 时，调用此接口加密侦听连接（SSL）。

函数定义：

Sint8 espconn_secure_accept(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.2.3. espconn_regist_time

功能：ESP8266 作为 TCP Server，设置超时断开连接的超时时长。ESP8266 会与一段时间内都不通信的 TCP client 断开连接。

注意：

请在 espconn_accept 之后调用本接口；

如果设置超时为 0，则永不会超时断开，不建议这样设置。

函数定义：

Sin8 espconn_regist_time(struct espconn *espconn, uint32 interval, uint8 type_flag)

输入参数：

struct espconn *espconn——相应连接的控制块结构

uint32 interval —— 超时时间，单位为秒，最大值为 7200 秒

uint8 type_flag —— 0，设置全部连接；1，设置单个连接

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.2.4. espconn_get_connection_info

功能：TCP 连接，针对多连接的情况，获得某 TCP server 连接的所有 TCP client 的信息。

函数定义：

Sin8 espconn_get_connection_info (struct espconn *espconn, remot_info **pcon_info, uint8 typeflags)

输入参数：

struct espconn *espconn——相应连接的控制块结构

remot_info **pcon_info—— 连接 client 的信息

uint8 typeflags —— 0，普通 server；1，ssl server

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.2.5. espconn_connect

功能：当 ESP8266 作为 TCP client 时，建立 TCP 连接

函数定义：

Sint8 espconn_connect(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.2.6. espconn_connect_callback

功能：TCP 侦听或连接成功回调

函数定义：

Void espconn_connect_callback (void *arg)

输入参数：

void *arg——回调函数参数

返回：

无

3.8.2.7. espconn_set_opt

功能：设置 TCP 连接的属性选择

函数定义：

sint8 espconn_set_opt(struct espconn *espconn, uint8 opt)

输入参数：

struct espconn *espconn——相应连接的控制块结构

uint8 opt – 按 bit 设置 TCP 属性

bit1 置 1，设置 TCP 连接断开时，占用内存无需等待 2min 才释放；

bit2 置 1，设置关闭 TCP 数据传输时的 nalgo 算法，加快传输速度；

bit3 置 1，设置使能 2920 bytes 的 write buffer，用于缓存 espconn_sent 要发送的数据。

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

注意：

此接口一般无需调用；

如需设置 espconn_set_opt(espconn, 0)，请在 connected callback 中调用；

如需设置 espconn_set_opt(espconn, 1)，请在 disconnect 之前调用即可

3.8.2.8. espconn_disconnect

功能：断开 TCP 连接

函数定义：

Sin8 espconn_disconnect(struct espconn *espconn);

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.2.9. espconn_regist_connectcb

功能：注册 TCP 连接函数，成功连接时回调

函数定义：

Sint8 espconn_regist_connectcb(struct espconn *espconn, espconn_connect_callback connect_cb)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_connect_callback connect_cb——注册的回调函数

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.2.10. espconn_regist_reconcb

功能：注册 TCP 重连函数，出错时的回调

说明：reconnect callback 实际为出错处理回调，任何阶段出错时，均会进入 reconnect callback；例如，espconn_sent 失败，则认为网络连接异常，也会进入 reconnect callback；用户可在 reconnect callback 中自定义出错处理。

函数定义：

Sint8 espconn_regist_reconcb(struct espconn *espconn, espconn_connect_callback recon_cb)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_connect_callback connect_cb——注册的回调函数

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.2.11. espconn_regist_disconcb

功能：注册断开 TCP 连接函数，断开连接成功时回调

函数定义：

Sint8 espconn_regist_disconcb(struct espconn *espconn, espconn_connect_callback discon_cb)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_connect_callback discon_cb——注册的回调函数

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.2.12. espconn_regist_write_finish

功能：注册数据写入完成的回调。

函数定义：

Sint8 espconn_regist_write_finish (struct espconn *espconn, espconn_connect_callback write_finish_fn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_connect_callback write_finish_fn——注册的回调函数

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.2.13. espconn_secure_connect

功能：当 ESP8266 作为 TCP client 时，建立加密 TCP 连接（SSL）。

函数定义：

Sint8 espconn_secure_connect (struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.2.14. espconn_secure_sent

功能：TCP 加密发送数据（SSL）

函数定义：

Sint8 espconn_secure_sent (struct espconn *espconn, uint8 *psent, uint16 length)

输入参数：

struct espconn *espconn——相应连接的控制块结构

uint8 *psent——sent 数据指针

uint16 length——sent 数据长度

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.2.15. espconn_secure_disconnect

功能：加密断开 TCP 连接（SSL）

函数定义：

Sint8 espconn_secure_disconnect(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.2.16. espconn_tcp_get_max_con

功能：查询最多允许建立多少个 TCP 连接。

函数定义：

uint8 espconn_tcp_get_max_con(void)

输入参数：

无

返回：

支持建立的 TCP 连接数的上限

3.8.2.17. espconn_tcp_set_max_con

功能：设置最多允许建立的 TCP 连接数的上限

函数定义：

Sint8 espconn_tcp_set_max_con(uint8 num)

输入参数：

uint8 num——允许建立的 TCP 连接数的上限

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.2.18. espconn_tcp_get_max_con_allow

功能：查询 ESP8266 某个 TCP server 最多允许连接的 TCP client 数目

函数定义：

Sint8 espconn_tcp_get_max_con_allow(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

TCP server 最多允许连接的 TCP client 数目

3.8.2.19. espconn_tcp_set_max_con_allow

功能：设置 ESP8266 某个 TCP server 最多允许连接的 TCP client 数目

函数定义：

Sint8 espconn_tcp_set_max_con_allow(struct espconn *espconn, uint8 num)

输入参数：

struct espconn *espconn——相应连接的控制块结构

uint8 num --最多允许连接的 TCP client 数目上限值

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.2.20. espconn_recv_hold

功能：阻塞 TCP 接收数据

说明：调用本接口会逐渐减小 tcp 的窗口，并不是即时阻塞，因此建议预留 1460*5 Byte 左右的空间时候调用，且本接口可以反复调用。

函数定义：

Sint8 espconn_recv_hold(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h，其中返回值为 ESPCONN_ARG 时,表示没有找到 espconn 参数对应的 tcp 对象.

3.8.2.21. espconn_recv_unhold

功能：停止 TCP 收包阻塞。

说明：本接口实时生效。

函数定义：

Sint8 espconn_recv_unhold(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h，其中返回值为 ESPCONN_ARG 时,表示没有找到 espconn 参数对应的 tcp 对象.

3.8.3. UDP 接口

3.8.3.1. espconn_create

功能：建立 UDP 传输

函数定义：

Sin8 espconn_create(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.3.2. espconn_igmp_join

功能：加入多播组

函数定义：

Sin8 espconn_igmp_join(ip_addr_t *host_ip, ip_addr_t *multicast_ip)

参数：

ip_addr_t *host_ip —— 主机 ip

ip_addr_t *multicast_ip – 多播组 ip

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.8.3.3. espconn_igmp_leave

功能：退出多播组

函数定义：

Sin8 espconn_igmp_leave(ip_addr_t *host_ip, ip_addr_t *multicast_ip)

参数：

ip_addr_t *host_ip —— 主机 ip

ip_addr_t *multicast_ip – 多播组 ip

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.9. AT 接口

AT 接口使用示例请参考 esp_iot_sdk/examples/5.at/user/user_main.c

3.9.1. at_response_ok

功能：回复“OK”到 AT Port (UART0)

函数定义：

void at_response_ok(void)

参数：

NULL

返回：

NULL

3.9.2. at_response_error

功能：回复“ERROR”到 AT Port (UART0)

函数定义：

void at_response_error(void)

参数：

NULL

返回：

NULL

3.9.3. at_cmd_array_regist

功能：注册客户自定义的 AT 指令集

函数定义：

void at_cmd_array_regist (at_funcation * custom_at_cmd_arrar, uint32 cmd_num)

参数：

at_funcation * custom_at_cmd_arrar - 客户自定义的 AT 指令数组

uint32 cmd_num - 客户自定义的指令数目

返回：

NULL

示例：参考 esp_iot_sdk/examples/5.at/user/user_main.c

3.9.4. at_get_next_int_dec

功能：从 AT 命令行中解析 int 型数字

函数定义：

bool at_get_next_int_dec (char **p_src,int* result,int* err)

参数：

char **p_src - *p_src 为接收到的 at 命令字符串

int* result - 解析出的 int 型数字

int* err - 解析处理时的错误码

1：数字省略时，返回错误码 1

3：只发现’-’ 时，返回错误码 3

返回：

TRUE，正常解析到数字(数字省略时，仍然返回 True，但错误码会为 1)

FALSE，解析异常，返回错误码，异常可能：数字超过 10 bytes、遇到‘\r‘ 结束符、只发现’-’ 字符

示例：参考 esp_iot_sdk/examples/5.at/user/user_main.c

3.9.5. at_data_str_copy

功能：从 AT 命令行中解析字符串

函数定义：

Int32 at_data_str_copy (char * p_dest, char ** p_src,int32 max_len)

参数：

char * p_dest - 保存解析到的字符串

char ** p_src - *p_src 为接收到的 AT 命令字符串

int32 max_len - 允许的最大字符串长度

返回：

成功，则返回解析到的字符串长度；

失败，返回 -1

示例：参考 esp_iot_sdk/examples/5.at/user/user_main.c

3.9.6. at_init

功能：AT 模块初始化

函数定义：

void at_init (void)

参数：

NULL

返回：

NULL

示例：参考 esp_iot_sdk/examples/5.at/user/user_main.c

3.9.7. at_port_print

功能：输出字符串到 AT PORT(UART0)

函数定义：

void at_port_print(const char *str)

参数：

const char *str - 要输出的字符串

返回：

NULL

示例：参考 esp_iot_sdk/examples/5.at/user/user_main.c

3.9.8. at_set_custom_info

功能：开发者自定义 AT 版本信息，可由指令 AT+GMR 查询到。

函数定义：

void at_set_custom_info (char *info)

参数：

char *info - 要输出的字符串

返回：

NULL

3.9.9. at_enter_special_state

功能：进入 AT 指令执行态，此时不响应其他 AT 指令，返回 busy

函数定义：

void at_enter_special_state (void)

参数：

NULL

返回：

NULL

3.9.10. at_leave_special_state

功能：退出 AT 指令执行态。

函数定义：

void at_leave_special_state (void)

参数：

NULL

返回：

NULL

3.9.11. at_get_version

功能：获得 Espressif AT lib version。

函数定义：

uint32 at_get_version (void)

参数：

NULL

返回：

Espressif AT lib version

3.10. json API 接口

说明：jsonparse 相关接口函数或宏定义在（工程目录\include\json\jsonparse.h），另外 jsontree 相关接口函数或宏定义在（工程目录\include\json\jsontree.h）。

3.10.1. jsonparse_setup

功能：json 解析初始化

函数定义：

void jsonparse_setup(struct jsonparse_state *state, const char *json,int len)

输入参数：

struct jsonparse_state *state——json 解析指针

const char *json——json 解析字符串

int len——字符串长度

返回：

无

3.10.2. jsonparse_next

功能：解析 json 格式下一个元素

函数定义：

Int jsonparse_next(struct jsonparse_state *state)

输入参数：

struct jsonparse_state *state——json 解析指针

返回：

int——解析结果

3.10.3. jsonparse_copy_value

功能：复制当前解析字符串到指定缓存

函数定义：

Int jsonparse_copy_value(struct jsonparse_state *state, char *str, int size)

输入参数：

struct jsonparse_state *state——json 解析指针

char *str——缓存指针

int size——缓存大小

返回：

int——复制结果

3.10.4. jsonparse_get_value_as_int

功能：解析 json 格式为整形数据

函数定义：

Int jsonparse_get_value_as_int(struct jsonparse_state *state)

输入参数：

struct jsonparse_state *state——json 解析指针

返回：

int——解析数据

3.10.5. jsonparse_get_value_as_long

功能：解析 json 格式为长整形数据

函数定义：

Long jsonparse_get_value_as_long(struct jsonparse_state *state)

输入参数：

struct jsonparse_state *state——json 解析指针

返回：

long——解析数据

3.10.6. jsonparse_get_len

功能：解析 json 格式数据长度

函数定义：

Int jsonparse_get_value_len(struct jsonparse_state *state)

输入参数：

struct jsonparse_state *state——json 解析指针

返回：

int——解析的 json 格式数据长度

3.10.7. jsonparse_get_value_as_type

功能：解析 json 格式数据类型

函数定义：

Int jsonparse_get_value_as_type(struct jsonparse_state *state)

输入参数：

struct jsonparse_state *state——json 解析指针

返回：

int——json 格式数据类型

3.10.8. jsonparse_strcmp_value

功能：比较解析的 json 数据与特定字符串

函数定义：

Int jsonparse_strcmp_value(struct jsonparse_state *state, const char *str)

输入参数：

struct jsonparse_state *state——json 解析指针

const char *str——字符缓存

返回：

int——比较结果

3.10.9. jsontree_set_up

功能：生成 json 格式数据树

函数定义：

void jsontree_setup(struct jsontree_context *js_ctx,

struct jsontree_value *root, int (* putchar)(int))

输入参数：

struct jsontree_context *js_ctx——json 格式树元素指针

struct jsontree_value *root——根树元素指针

int (* putchar)(int)——输入函数

返回：

无

3.10.10. jsontree_reset

功能：设置 json 树

函数定义：

void jsontree_reset(struct jsontree_context *js_ctx)

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

返回：

无

3.10.11. jsontree_path_name

功能：json 树参数获取

函数定义：

const char *jsontree_path_name(const struct jsontree_cotext *js_ctx,int depth)

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

int depth——json 格式树深度

返回：

char*——参数指针

3.10.12. jsontree_write_int

功能：整形数写入 json 树

函数定义：

void jsontree_write_int(const struct jsontree_context *js_ctx, int value)

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

int value——整形值

返回：

无

3.10.13. jsontree_write_int_array

功能：整形数数组写入 json 树

函数定义：

void jsontree_write_int_array(const struct jsontree_context *js_ctx, const int *text, uint32 length)

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

int *text——数组入口地址

uint32 length——数组长度

返回：

无

3.10.14. jsontree_write_string

功能：字符串写入 json 树

函数定义：

void jsontree_write_string(const struct jsontree_context *js_ctx, const char *text)

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

const char* text——字符串指针

返回：

无

3.10.15. jsontree_print_next

功能：json 树深度

函数定义：

int jsontree_print_next(struct jsontree_context *js_ctx)

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

返回：

int——json 树深度

3.10.16. jsontree_find_next

功能：查找 json 树元素

函数定义：

struct jsontree_value *jsontree_find_next(struct jsontree_context *js_ctx, int type)

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

int——类型

返回：

struct jsontree_value *——json 格式树元素指针

4. 数据结构定义

4.1. 定时器结构

```
typedef void ETSTimerFunc(void *timer_arg);

typedef struct _ETSTIMER_ {
struct _ETSTIMER_       *timer_next;
      uint32_t                  timer_expire;
      uint32_t                  timer_period;
     ETSTimerFunc              *timer_func;
     void                      *timer_arg;
} ETSTimer;
```

4.2. wifi 参数

4.2.1. station 配置参数

```
struct station_config {
uint8 ssid[32];
uint8 password[64];
uint8 bssid_set;
uint8 bssid[6];
};
```

注意：

判断 bssid 用于多个 AP 的 ssid 相同时，引入 bssid 来区分；

bssid_set 为 1 时，必须设置对应的 bssid，否则会连接失败；

因此一般情况下，必须将 bssid_set 初始化为 0，则只判断 ssid。

4.2.2. softap 配置参数

```
typedef enum _auth_mode {
    AUTH_OPEN                = 0,
    AUTH_WEP,
    AUTH_WPA_PSK,
    AUTH_WPA2_PSK,
    AUTH_WPA_WPA2_PSK
} AUTH_MODE;
struct softap_config {
    uint8 ssid[32];
    uint8 password[64];
    uint8 ssid_len;
    uint8 channel;
    uint8 authmode;
    uint8 ssid_hidden;
    uint8 max_connection;
    uint16 beacon_interval;   // 100 ~ 60000 ms, default 100
};
```

注意：

在 struct softap_config 中，当 ssid_len 设为 0 时，读取 ssid 直至遇到结束符；当 ssid_len 不为 0 时，使用 ssid_len 长度作为判断 ssid 长度的依据。

4.2.3. scan 参数

```
struct scan_config {
    uint8 *ssid;
    uint8 *bssid;
    uint8 channel;
    uint8 show_hidden; //  是否扫描隐藏 ssid  的 AP
```

```
};


struct bss_info {
    STAILQ_ENTRY(bss_info)      next;
    u8 bssid[6];
    u8 ssid[32];
    u8 channel;
    s8 rssi;
    u8 authmode;
    uint8 is_hidden; //  当前 AP 的 SSID 是否为隐藏的.
};
typedef void (* scan_done_cb_t)(void *arg, STATUS status);
```

4.3. smart config 结构体

```
typedef enum    {
    SC_STATUS_FIND_CHANNEL = 0,
    SC_STATUS_GETTING_SSID_PSWD,
    SC_STATUS_GOT_SSID_PSWD,
    SC_STATUS_LINK,
} sc_status;

typedef enum    {
    SC_TYPE_ESPTOUCH = 0,
    SC_TYPE_AIRKISS,
} sc_type;
```

4.4.json 相关结构

4.3.1. json 结构

```
struct jsontree_value {
uint8_t type;
};

struct jsontree_pair {
const char *name;
struct jsontree_value *value;
};

struct jsontree_context {
struct jsontree_value *values[JSONTREE_MAX_DEPTH];
uint16_t index[JSONTREE_MAX_DEPTH];
int (* putchar)(int);
uint8_t depth;
uint8_t path;
int callback_state;
};

struct jsontree_callback {
uint8_t type;
int (* output)(struct jsontree_context *js_ctx);
int (* set)(struct jsontree_context *js_ctx, struct jsonparse_state *parser);
};

struct jsontree_object {
```

```
uint8_t type;
uint8_t count;
struct jsontree_pair *pairs;
};

struct jsontree_array {
uint8_t type;
uint8_t count;
struct jsontree_value **values;
};

struct jsonparse_state {
const char *json;
int pos;
int len;
int depth;
int vstart;
int vlen;
char vtype;
char error;
char stack[JSONPARSE_MAX_DEPTH];
};
```


4.3.2. json 宏定义

```
#define                    JSONTREE_OBJECT(name,                    ...)
\
static struct jsontree_pair jsontree_pair_##name[] = {__VA_ARGS__};    \
static struct jsontree_object name = {                                 \
```

```
    JSON_TYPE_OBJECT,                                   \
sizeof(jsontree_pair_##name)/sizeof(struct jsontree_pair),          \
    jsontree_pair_##name }

#define JSONTREE_PAIR_ARRAY(value) (struct jsontree_value *)(value)
#define                JSONTREE_ARRAY(name,                ...)
\
static struct jsontree_value* jsontree_value_##name[] = {__VA_ARGS__};   \
static struct jsontree_array name = {                               \
    JSON_TYPE_ARRAY,                                \
sizeof(jsontree_value_##name)/sizeof(struct jsontree_value*),       \
    jsontree_value_##name }
```

4.5. espconn 参数

4.4.1 回调 function

```
/** callback prototype to inform about events for a espconn */
typedef void (* espconn_recv_callback)(void *arg, char *pdata, unsigned short
len);
typedef void (* espconn_callback)(void *arg, char *pdata, unsigned short len);
typedef void (* espconn_connect_callback)(void *arg);
```

4.4.2 espconn

```
    typedef void* espconn_handle;
    typedef struct _esp_tcp {
        int client_port;
        int server_port;
        char ipaddr[4];
        espconn_connect_callback connect_callback;
```

```
    espconn_connect_callback reconnect_callback;
    espconn_connect_callback disconnect_callback;
} esp_tcp;

typedef struct _esp_udp {
    int _port;
    char ipaddr[4];
} esp_udp;

/** Protocol family and type of the espconn */
enum espconn_type {
    ESPCONN_INVALID     = 0,
    /* ESPCONN_TCP Group */
    ESPCONN_TCP         = 0x10,
    /* ESPCONN_UDP Group */
    ESPCONN_UDP         = 0x20,
};

/** Current state of the espconn. Non-TCP espconn are always in state ESPCONN_NONE! */
enum espconn_state {
    ESPCONN_NONE,
    ESPCONN_WAIT,
    ESPCONN_LISTEN,
    ESPCONN_CONNECT,
    ESPCONN_WRITE,
    ESPCONN_READ,
    ESPCONN_CLOSE
};
```

```
/** A espconn descriptor */
struct espconn {
    /** type of the espconn (TCP, UDP) */
    enum espconn_type type;
    /** current state of the espconn */
    enum espconn_state state;
    union {
        esp_tcp *tcp;
        esp_udp *udp;
    } proto;
    /** A callback function that is informed about events for this espconn */
    espconn_recv_callback recv_callback;
    espconn_sent_callback sent_callback;
    espconn_handle esp_pcb;
    uint8 *ptrbuf;
    uint16 cntr;
};
```

5. 驱动接口

5.1. GPIO 接口 API

关于 gpio 接口 API 操作的具体应用，可参看\user\ user_plug.c。

5.1.1. PIN 脚功能设置宏

- ✓ PIN_PULLUP_DIS(PIN_NAME)

  管脚上拉屏蔽
- ✓ PIN_PULLUP_EN(PIN_NAME)

  管脚上拉使能
- ✓ PIN_PULLDWN_DIS(PIN_NAME)

  管脚下拉屏蔽
- ✓ PIN_PULLDWN_EN(PIN_NAME)

  管脚下拉使能
- ✓ PIN_FUNC_SELECT(PIN_NAME, FUNC)

  管脚功能选择

  例 如 ： PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U, FUNC_GPIO12);

  选择 MTDI 管脚为复用 GPIO12。

5.1.2. gpio_output_set

功能：设置 gpio 口属性

函数定义：

void gpio_output_set(uint32 set_mask, uint32 clear_mask, uint32 enable_mask, uint32 disable_mask)

输入参数：

uint32 set_mask——设置输出为高的位，对应位为 1，输出高，对应位为 0，不改变状态

uint32 clear_mask——设置输出为低的位，对应位为 1，输出低，对应位为 0，不改变状态

uint32 enable_mask——设置使能输出的位

uint32 disable_mask——设置使能输入的位

返回：

无

例子：

- ✓ 设置 GPIO12 输出高电平，则：gpio_output_set(BIT12, 0, BIT12, 0);
- ✓ 设置 GPIO12 输出低电平，则：gpio_output_set(0, BIT12, BIT12, 0);
- ✓ 设置 GPIO12 输出高电平，GPIO13 输出低电平，则：

  gpio_output_set(BIT12, BIT13, BIT12|BIT13, 0);
- ✓ 设置 GPIO12 为输入，则 gpio_output_set(0, 0, 0, BIT12);

5.1.3. GPIO 输入输出相关宏

- ✓ GPIO_OUTPUT_SET(gpio_no, bit_value)

  设置 gpio_no 管脚输出 bit_value，同 5.1.2 例子中输出高低电平的功能。
- ✓ GPIO_DIS_OUTPUT(gpio_no)

  设置 gpio_no 管脚为输入，同 5.1.2 例子中输入。
- ✓ GPIO_INPUT_GET(gpio_no)

  获取 gpio_no 管脚的电平状态。

5.1.4. GPIO 中断控制相关宏

- ✓ ETS_GPIO_INTR_ATTACH(func, arg)

  注册 GPIO 中断处理函数。
- ✓ ETS_GPIO_INTR_DISABLE()

  关 GPIO 中断。
- ✓ ETS_GPIO_INTR_ENABLE()

  开 GPIO 中断。

5.1.5. gpio_pin_intr_state_set

功能：设置 gpio 脚中断触发状态

函数定义：

```
void gpio_pin_intr_state_set(uint32 i, GPIO_INT_TYPE intr_state)
```

输入参数：

uint32 i——GPIO 管脚 ID，如需设置 GPIO14，则为 GPIO_ID_PIN(14)；

GPIO_INT_TYPE intr_state——中断触发状态

其中：

```
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;
```

返回：

无

5.1.6. GPIO 中断处理函数

在 GPIO 中断处理函数内，需要做如下操作来清除响应位的中断状态：

```
uint32 gpio_status;
gpio_status = GPIO_REG_READ(GPIO_STATUS_ADDRESS);
//clear interrupt status
GPIO_REG_WRITE(GPIO_STATUS_W1TC_ADDRESS, gpio_status);
```

5.2. 双 UART 接口 API

默认情况下，UART0 作为系统的 debug 输出接口，当配置为双 UART 时，UART0 作为数据收发接口，UART1 作为 debug 输出接口。

使用时，请确保硬件连接正确。

5.2.1. uart_init

功能：双 uart 模式，两个 uart 波特率初始化

函数定义：

void uart_init(UartBautRate uart0_br, UartBautRate uart1_br)

输入参数：

UartBautRate uart0_br——uart0 波特率

UartBautRate uart1_br——uart1 波特率

其中：

```
typedef enum {
    BIT_RATE_9600       = 9600,
    BIT_RATE_19200      = 19200,
    BIT_RATE_38400      = 38400,
    BIT_RATE_57600      = 57600,
    BIT_RATE_74880      = 74880,
    BIT_RATE_115200 = 115200,
    BIT_RATE_230400 = 230400,
    BIT_RATE_460800 = 460800,
    BIT_RATE_921600 = 921600
} UartBautRate;
```

返回：

无

5.2.2. uart0_tx_buffer

功能：通过 UART0 发送用户自定义数据

函数定义：

Void uart0_tx_buffer(uint8 *buf, uint16 len)

输入参数：

Uint8 *buf——待发送数据

Uint16 len——待发送数据长度

返回：

无

5.2.3. uart0_rx_intr_handler

功能：UART0 中断处理函数，用户可在该函数内添加对接收到数据包的处理。（接收缓冲区大小为 0x100，如果接受数据大于 0x100，请自行处理）

函数定义：

Void uart0_rx_intr_handler(void *para)

输入参数：

Void*para——指向 RcvMsgBuff 结构的指针

返回：

无

5.3. i2c master 接口

ESP8266 不能作为 I2C 从设备，但可以作为 I2C 主设备，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。

5.3.1. i2c_master_gpio_init

功能：i2c master 模式时，初始化，对相应 GPIO 口进行设置

函数定义：

Void i2c_master_gpio_init (void)

输入参数：

无

返回：

无

5.3.2. i2c_master_init

功能：初始化 i2c 操作

函数定义：

Void i2c_master_init(void)

输入参数：

无

返回：

无

5.3.3. i2c_master_start

功能：设置 i2c 进入发送状态

函数定义：

Void i2c_master_start(void)

输入参数：

无

返回：

无

5.3.4. i2c_master_stop

功能：设置 i2c 进入停止发送状态

函数定义：

Void i2c_master_stop(void)

输入参数：

无

返回：

无

5.3.5. i2c_master_send_ack

功能：主机发送 i2c ACK

函数定义：

Void i2c_master_send_ack (void)

输入参数：

NULL

返回：

无

5.3.6. i2c_master_send_nack

功能：主机发送 i2c NACK

函数定义：

Void i2c_master_send_nack (void)

输入参数：

NULL

返回：

无

5.3.7. i2c_master_checkAck

功能：检查 slave 的 ACK

函数定义：

bool i2c_master_checkAck (void)

输入参数：

NULL

返回：

TRUE，查到 slave ACK；

FALSE，查到 slave NACK

5.3.8. i2c_master_readByte

功能：从 slave 读取一字节

函数定义：

Uint8 i2c_master_readByte (void)

输入参数：

无

返回：

uint8——读取到的值

5.3.9. i2c_master_writeByte

功能：向 slave 写一字节

函数定义：

Void i2c_master_writeByte (uint8 wrdata)

输入参数：

uint8 wrdata——待写数据

返回：

无

5.4. pwm

当前支持 4 路 PWM，可在 pwm.h 中对采用的 GPIO 口进行配置选择。

5.4.1. pwm_init

功能：pwm 功能初始化，包括 gpio，频率以及占空比

函数定义：

Void pwm_init(uint16 freq, uint8 *duty)

输入参数：

Uint16 freq——pwm 的频率；

uint8 *duty——各路的占空比

返回：

无

5.4.2. pwm_start

功能：pwm 开始，每次更新 pwm 数据后，都需要重新调用本接口进行计算。

函数定义：

Void pwm_start (void)

返回：

无

5.4.3. pwm_set_duty

功能：对某一路设置占空比

函数定义：

Void pwm_set_duty(uint8 duty, uint8 channel)

输入参数：

uint8 duty——占空比

uint8 channel——某路

返回：

无

5.4.4. pwm_set_freq

功能：设置 pwm 频率

函数定义：

Void pwm_set_freq(uint16 freq)

输入参数：

Uint16 freq——pwm 频率

返回：

无

5.4.5. pwm_get_duty

功能：获取某路的占空比

函数定义：

uint8 pwm_get_duty(uint8 channel)

输入参数：

uint8 channel——待获取占空比的 channel

返回：

uint8——占空比

5.4.6. pwm_get_freq

功能：获取 pwm 频率

函数定义：

Uint16 pwm_get_freq(void)

输入参数：

无

返回：

Uint16——频率

6. 附录

A. ESPCONN 编程

介绍 ESP8266 作为 TCP client 和 TCP server 时分别应该怎样编程。

A.1. TCP client 模式

A.1.1. 说明

ESP8266 工作在 Station 模式下，确认 ESP8266 已经连接 AP(路由)分配到 IP 地址时，启用 client 连接。

ESP8266 工作在 softap 模式下，确认连接 ESP8266 的设备已被分配到 ip 地址，启用 client 连接。

A.1.2. 步骤

1) 依据工作协议初始化 espconn 参数。
2) 注册 connect 回调函数，注册 recv 回调函数。
3) 调用 espconn_connect 函数建立与 TCP server 的连接。
4) 连接成功后将调用注册的 connect 函数，该函数中根据应用注册相应的回调函数，建议 disconnect 回调函数必须注册。
5) 在 recv 回调函数或 sent 回调函数执行 disconnect 操作时，建议适当延时一定时间，确保底层函数执行结束。

A.2. TCP server 模式

A.2.1. 说明

ESP8266 工作在 Station 模式下，确认 ESP8266 已经分配到 IP 地址，启用 server 侦听。

ESP8266 工作在 softap 模式下，启用 server 侦听。

A.2.2. 步骤

1) 依据工作协议初始化 espconn 参数。

2) 注册 connect 回调函数，注册 recv 回调函数。

3) 调用 espconn_accept 函数侦听 host 的连接。

连接成功后将调用注册的 connect 函数，该函数中根据应用注册相应的回调函数。

B. RTC 接口使用示例

以下测试示例，可以验证 RTC 时间和系统时间，在 system_restart 时的变化，以及读写 RTC memory。

```
void user_init(void)
{

os_printf("clk cal : %d \n\r",system_rtc_clock_cali_proc()>>12);
uint32 rtc_time = 0, rtc_reg_val = 0,stime = 0,rtc_time2 = 0,stime2 = 0;
rtc_time =  system_get_rtc_time();  //获得 RTC 时间
stime = system_get_time();  //获得系统时间

os_printf("rtc time : %d \n\r",rtc_time);
os_printf("system time : %d \n\r",stime);

//读 RTC memory
if( system_rtc_mem_read(0, &rtc_reg_val, 4) ){
    os_printf("rtc mem val : 0x%08x\n\r",rtc_reg_val);
}else{
    os_printf("rtc mem val error\n\r");
}
```

```
    rtc_reg_val++;
    os_printf("rtc mem val write\n\r");
    system_rtc_mem_write(0, &rtc_reg_val, 4) ; //写 RTC memory

    if( system_rtc_mem_read(0, &rtc_reg_val, 4) ){
        os_printf("rtc mem val : 0x%08x\n\r",rtc_reg_val);
    }else{
        os_printf("rtc mem val error\n\r");
    }

rtc_time2 =   system_get_rtc_time();
stime2 = system_get_time();

os_printf("rtc time : %d \n\r",rtc_time2);
os_printf("system time : %d \n\r",stime2);

os_printf("delta time rtc: %d \n\r",rtc_time2-rtc_time);
os_printf("delta system   time rtc: %d \n\r",stime2-stime);

os_printf("clk cal : %d \n\r",system_rtc_clock_cali_proc()>>12);

os_delay_us(500000);
system_restart();

}
```

C. Sniffer 结构体说明

ESP8266 可以进入 Sniffer 模式，可以接收空中其他 STA 之间相互发送的 ieee80211 包。

ESP8266 的 Sniffer 模式可以完全接收以下包：11b、11g、11n 的 HT20（只支持 MCS0-7，不支持 MCS7-76，不支持 HT40，不支持 LDPC，但支持 AMPDU）。尽管有些类型的 ieee80211 包是 ESP8266 不能完全接收的，但它们的包长可以是 ESP8266 可以接收的。

因此，ESP8266 Sniffer 接收包的格式分为两类：

1、 ESP8266 可完全接收的包，它包含一定长度的 MAC 头(包含了收发双方的 MAC 地址和加密方式)和整个包的长度；

2、 ESP8266 无法完全接收的包，它只包含包长信息。

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;//表示该包的信号强度
    unsigned rate:4;
    unsigned is_group:1;
    unsigned:1;
    unsigned sig_mode:2;//表示该包是否是 11n 的包，0 表示非 11n，非 0 表示 11n
    unsigned legacy_length:12;//如果不是 11n 的包，它表示包的长度
    unsigned damatch0:1;
    unsigned damatch1:1;
    unsigned bssidmatch0:1;
    unsigned bssidmatch1:1;
    unsigned MCS:7;//如果是 11n 的包，它表示包的调制编码序列，有效值：0-76
    unsigned CWB:1;//如果是 11n 的包，它表示是否为 HT40 的包
    unsigned HT_length:16;//如果是 11n 的包，它表示包的长度
    unsigned Smoothing:1;
    unsigned Not_Sounding:1;
    unsigned:1;
    unsigned Aggregation:1;
```

```
    unsigned STBC:2;
    unsigned FEC_CODING:1;//如果是 11n 的包，它表示是否为 LDPC 的包
    unsigned SGI:1;
    unsigned rxend_state:8;
    unsigned ampdu_cnt:8;
    unsigned channel:4;//表示该包所在的信道
    unsigned:12;
};

struct LenSeq{
    u16 len;//包长
    u16 seq;//包的序列号，其中高 12bit 就是序列号，低 4bit 是 Fragment 号(一般是 0)
    u8 addr3[6];//包中的第 3 个地址
};

struct sniffer_buf{
    struct RxControl rx_ctrl;
    u8 buf[36];//包含 ieee80211 包头
    u16 cnt;//包的个数
    struct LenSeq lenseq[1];//包的长度
};
```

回调函数 wifi_promiscuous_rx 有两个参数 buf 和 len

len 表示 buf 的长度，len 的长度可能的值是 len = 12 或 len ≥ 60。

（1） len ≥ 60 时，buf 的数据是一个结构体 sniffer_buf，该结构体是比较可信的，它对应的数据包至少是通过 CRC 校验正确的。其中成员 cnt 表示了该 buf 包含的包的个数，len 的值由 cnt 决定 。cnt 值为 0，则此 buf 无效；否则 len = 50 + cnt * 10。成员 buf 表示 ieee80211 包的前 36 bytes。从成员 len[0]开始，buf 中每两个字节表示一个包长信息。当 cnt 大于 1 时，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。

（2） len = 12 时，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(可以用于评估是否是同一个设备所发)、FEC_CODING(可以用于评估是否是同一个设备所发)。

注意：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。

下面图简单介绍 ieee80211 数据帧的格式。下图展示的是一个完整的 ieee80211 数据包的格式：

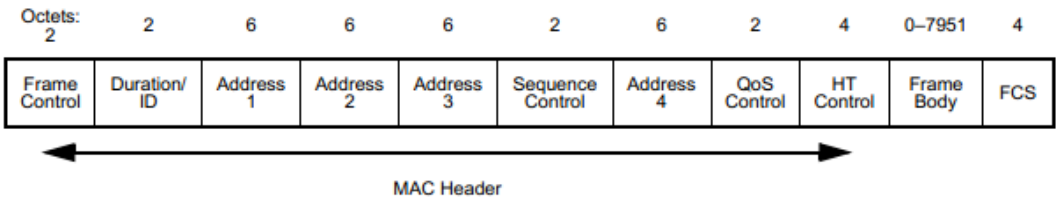

Figure 8-30—Data frame

Data 帧的 MAC 包头的前 24 字节是必须有的，Address4 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。