ESP8266 SDK 编程手册

Version 1.0.1

Espressif Systems IOT Team

Table of Content

1. 前言

ESP8266EX 提供完整且自成体系的 Wi-Fi 网络解决方案；它能够搭载软件应用，或者通过另一个应用处理器卸载所有 Wi-Fi 网络功能。当ESP8266作为设备中唯一的处理器搭载应用时，它能够直接从外接闪存（Flash）中启动，内置的高速缓冲存储器（cache）有利于提高系统性能，并减少内存需求。另一种情况，ESP8266可作为 Wi-Fi 适配器，通过 UART 或者 CPU AHB 桥接口连接到任何基于微控制器的设计中，为其提供无线上网服务，简单易行。

ESP8266EX 高度片内集成，包括：天线开关，RF balun，功率放大器，低噪放大器，过滤器，电源管理模块，因此它仅需很少的外围电路，且包括前端模块在内的整个解决方案在设计时就将所占 PCB 空间降到最低。

ESP8266EX 集成了增强版的 Tensilica's L106 钻石系列32位内核处理器，带片上 SRAM。ESP8266EX 通常通过 GPIO 外接传感器和其他功能的应用，SDK 中提供相关应用的示例软件。

ESP8266EX 系统级的领先特征有：节能VoIP在睡眠/唤醒之间快速切换，配合低功率操作的自适应无线电偏置，前端信号处理，故障排除和无线电系统共存特性为消除蜂窝/蓝牙/DDR/LVDS/LCD干扰。

基于ESP8266EX 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. 概述

SDK 为用户提供了一套数据接收、发送的函数接口，用户不必关心底层网络，如Wi-Fi、TCP/IP等的具体实现，只需要专注于物联网上层应用的开发，利用相应接口完成网络数据的收发即可。

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。用户应用的初始化功能可以在 `user_main.c` 中实现。

`void user_init(void)`是上层程序的入口函数，给用户提供一个初始化接口，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. 应用程序接口 (APIs)

3.1. 定时器

位于 /esp_iot_sdk/include/osapi.h.

注意：

- os_timer_arm 不能在中断内调用
- 对于同一个 timer，os_timer_arm 不能重复调用，必须先 os_timer_disarm
- os_timer_setfn 必须在 timer 未使能的情况下调用，在 os_timer_arm 之前或者 os_timer_disarm 之后

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
)
```

参数：

`ETSTimer *ptimer` ： 定时器结构

`uint32_t milliseconds` ： 定时时间，单位：毫秒，最大值 6871947 ms

`bool repeat_flag` ： 定时器是否重复

返回：

无

2. os_timer_disarm

功能：

取消定时器定时

函数定义：

```
void os_timer_disarm (ETSTimer *ptimer)
```

参数：

`ETSTimer *ptimer` ： 定时器结构

返回：

无

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
)
```

参数：

`ETSTimer *ptimer` ： 定时器结构

`TESTimerFunc *pfunction` ： 定时器回调函数

`void *parg` ： 回调函数的参数

返回：

无

3.2. 系统接口

1. system_restore

功能：

恢复出厂设置

函数定义：

```
void system_restore(void)
```

参数：

无

返回：

无

2. system_restart

功能：

系统重启

函数定义：

```
void system_restart(void)
```

参数：

无

返回：

无

3. system_timer_reinit

功能：

重新初始化定时器，当需要使用微秒级定时器时调用

注意：

1. 同时定义 USE_US_TIMER;
2. system_timer_reinit 在程序最开始调用，user_init 的第一句。

函数定义：

```
void system_timer_reinit (void)
```

参数：

无

返回：

无

4. system_init_done_cb

功能：

在 user_init 中调用，注册系统初始化完成的回调函数。

注意：

接口 wifi_station_scan 必须在系统初始化完成后，并且 station 模式使能的情况下调用。

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);
}
```

5. system_get_chip_id

功能：

查询芯片 ID

函数定义：

```
uint32 system_get_chip_id (void)
```

参数：

无

返回：

芯片 ID

6. system_deep_sleep

功能：

设置芯片进入 deep-sleep 模式，休眠设定时间后自动唤醒，唤醒后程序从 user_init 重新运行。

函数定义：

```
void system_deep_sleep(uint32 time_in_us)
```

参数：

uint32 time_in_us ：休眠时间，单位：微秒

返回：

无

注意：

硬件需要将 XPD_DCDC 通过 0R 连接到 EXT_RSTB，用作 deep-sleep 唤醒。

system_deep_sleep(0) 未设置唤醒定时器，可通过外部 GPIO 拉低 RST 脚唤醒。

7. system_deep_sleep_set_option

功能：

设置 deep-sleep 唤醒后的行为，如需调用此 API，必须在 system_deep_sleep 之前调用。

函数定义：

```
bool system_deep_sleep_set_option(uint8 option)
```

参数：

uint8 option :

deep_sleep_set_option(0): 由 init 参数的 byte 108 控制 deep-sleep 唤醒后的是否进行 RF_CAL;

deep_sleep_set_option(1): deep-sleep 唤醒后和重新上电的行为一致，会进行 RF_CAL，这样导致电流较大;

deep_sleep_set_option(2): deep-sleep 唤醒后不进行 RF_CAL，这样电流较小;

deep_sleep_set_option(4): deep-sleep 唤醒后不打开 RF，与 modem-sleep 行为一致，这样电流最小，但是设备唤醒后无法发送和接收数据。

注意：

此处 init 参数指 esp_init_data_default.bin.

返回：

true : 成功

false : 失败

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)
```

参数：

uint8 onoff

注意：

onoff==0: 打印功能关

onoff==1: 打印功能开

默认值：

打印功能开

返回：

无

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)
```

参数：

无

返回：

无

10. system_get_free_heap_size

功能：

查询系统剩余可用 heap 区空间大小

函数定义：

```
uint32 system_get_free_heap_size(void)
```

参数：

无

返回：

uint32 ：可用 heap 空间大小

11. system_os_task

功能：

创建系统任务

函数定义：

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

参数：

os_task_t task ：任务函数

uint8 prio ：任务优先级，当前支持 3 个优先级的任务：0/1/2；0 为最低优先级

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);
}
```

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
)
```

参数：

```
uint8 prio      : 任务优先级，与建立时的任务优先级对应。
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
```

13. system_get_time

功能：

查询系统时间，单位：微秒

函数定义：

```
uint32 system_get_time(void)
```

参数：

无

返回：

系统时间，单位：微秒。

14. system_get_rtc_time

功能：

查询 RTC 时间，单位：RTC 时钟周期

示例：

例如 system_get_rtc_time 返回 10 (表示 10 个 RTC 周期),
system_rtc_clock_cali_proc 返回 5 (表示 1 个 RTC 周期为 5 微秒),

则实际时间为 10 x 5 = 50 微秒。

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)
```

参数：

无

返回：

RTC 时间

15. system_rtc_clock_cali_proc

功能：

查询 RTC 时钟周期.

函数定义：

```
uint32 system_rtc_clock_cali_proc(void)
```

参数：

无

返回：

RTC 时钟周期，单位：微秒，bit11 ~ bit0 为小数部分。

注意：

RTC 示例见附录。

16. system_rtc_mem_write

功能：

由于 deep-sleep 时，仅 RTC 仍在工作，用户如有需要，可将数据存入 RTC memory 中。提供如下图中的 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，每 block 4字节，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
)
```

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：字节。

返回：

true： 成功

false: 失败

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
)
```

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：字节

返回：

true： 成功

false: 失败

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：

无

返回：

无

19. system_get_boot_version

功能：

读取 boot 版本信息

函数定义：

```
uint8 system_get_boot_version (void)
```

参数：

无

返回：

boot 版本信息.

注意：

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

20. system_get_userbin_addr

功能：

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

函数定义：

```
uint32 system_get_userbin_addr (void)
```

参数：

无

返回：

正在运行的 user bin 的存放地址。

21. system_get_boot_mode

功能：

查询 boot 模式。

函数定义：

```
uint8 system_get_boot_mode (void)
```

参数：

无

返回：

```
#define SYS_BOOT_ENHANCE_MODE 0
#define SYS_BOOT_NORMAL_MODE  1
```

注意：

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。

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 类型
   #define SYS_BOOT_NORMAL_BIN  0  // user1.bin 或者 user2.bin
   #define SYS_BOOT_TEST_BIN  1    // 向 Espressif 申请的 test bin
uint32 bin_addr : bin 的起始地址
```

返回：

true: 成功

false: 失败

注意：

SYS_BOOT_TEST_BIN 用于量产测试，用户可以向 Espressif Systems 申请获得。

23. system_update_cpu_freq

功能：

设置 CPU 频率。默认为 80MHz。

函数定义：

```
bool system_update_cpu_freq(uint8 freq)
```

参数：

```
uint8 freq : CPU frequency
   #define SYS_CPU_80MHz  80
   #define SYS_CPU_160MHz 160
```

返回：

true: 成功

false: 失败

24. system_get_cpu_freq

功能：

查询 CPU 频率。

函数定义：

```
uint8 system_get_cpu_freq(void)
```

参数：

无

返回：

CPU 频率，单位 ：MHz.

3.3. SPI Flash 接口

1. spi_flash_get_id

功能：

查询 spi flash 的 id

函数定义：

```
uint32 spi_flash_get_id (void)
```

参数：

无

返回：

spi flash id

2. spi_flash_erase_sector

功能：

擦除 flash 扇区

注意：

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. spi_flash_write

功能：

写入数据到 flash

注意：

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 目的地址
uint32 *src_addr : 写入数据的指针.
uint32 size      : 数据长度
```

返回：

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

4. spi_flash_read

功能：

从 flash 读取数据

函数定义：

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

参数：

```
uint32  src_addr: 读取 flash 数据的地址
uint32 *des_addr: 存放读取到数据的指针
uint32  size:     数据长度
```

返回：

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

3.4. WIFI 接口

1. wifi_get_opmode

功能：

查询 WiFi 当前工作模式

函数定义：

```
uint8 wifi_get_opmode (void)
```

参数：

无

返回：

WiFi 工作模式：

0x01: station 模式

0x02: soft-AP 模式

0x03: station + soft-AP 模式

2. wifi_set_opmode

功能：

设置 WiFi 工作模式（station，soft-AP or station+soft-AP）

注意：

esp_iot_sdk_v0.9.2 以及之前版本，设置之后需要调用 system_restart() 重启生效；

esp_iot_sdk_v0.9.2 之后的版本，不需要重启，即时生效。

函数定义：

```
bool wifi_set_opmode (uint8 opmode)
```

参数：

uint8 opmode: WiFi 工作模式：

0x01: station 模式

0x02: soft-AP 模式

0x03: station+soft-AP

返回：

true: 成功

false: 失败

3. wifi_station_get_config

功能：

查询 WiFi station 接口的配置参数

函数定义：

```
bool wifi_station_get_config (struct station_config *config)
```

参数：

struct station_config *config : WiFi station 接口参数指针

返回：

true: 成功

false: 失败

4. wifi_station_set_config

功能：

设置 WiFi station 接口的配置参数

注意：

（1）如果 wifi_station_set_config 是在 user_init 中调用，则 ESP8266 station 接口会在系统初始化完成后，自动按照配置参数连接 AP（路由），无需再调用 wifi_station_connect ；

否则，需要调用 wifi_station_connect 连接 AP（路由）。

（2）station_config.bssid_set 一般设置为 0 ，仅当需要检查 AP 的 MAC 地址时（多用于有重名 AP 的情况下）设置为 1。

函数定义：

```
bool wifi_station_set_config (struct station_config *config)
```

参数：

struct station_config *config: WiFi station 接口配置参数指针

返回：

true: 成功

false: 失败

5. wifi_station_connect

功能：

ESP8266 WiFi station 接口连接 AP

注意：

如果 ESP8266 已经连接到某个 AP，请先调用 wifi_station_disconnect 断开上一次连接。

函数定义：

```
bool wifi_station_connect (void)
```

参数：

无

返回：

true： 成功

false：失败

6. wifi_station_disconnect

功能：

ESP8266 WiFi station 接口从 AP 断开连接

函数定义：

```
bool wifi_station_disconnect (void)
```

参数：

无

返回：

true： 成功

false：失败

7. wifi_station_get_connect_status

功能：

查询 ESP8266 WiFi station 接口连接 AP 的状态

函数定义：

```
uint8 wifi_station_get_connect_status (void)
```

参数：

无

返回：

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

8. wifi_station_scan

功能：

获取 AP 的信息

注意：

请勿在 user_init 中调用本接口，本接口必须在系统初始化完成后，并且 ESP8266 station 接口使能的情况下调用。

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's ssid
    uint8 *bssid;      // AP's bssid
    uint8 channel;     //scan a specific channel
    uint8 show_hidden; //scan APs of which ssid is hidden.
};
```

参数：

struct scan_config *config: 扫描 AP 的配置参数

若 config==null: 扫描获取所有可用 AP 的信息

若 config.ssid==null && config.bssid==null && config.channel!=null:

ESP8266 station 接口扫描获取特定信道上的 AP 信息.

若 config.ssid!=null && config.bssid==null && config.channel==null:

ESP8266 station 接口扫描获取所有信道上的某特定名称 AP 的信息.

scan_done_cb_t cb: 扫描完成的 callback

返回：

true: 成功

false: 失败

9. scan_done_cb_t

功能：

wifi_station_scan 的回调函数

函数定义：

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

参数：

void *arg: 扫描获取到的 AP 信息指针，以链表形式存储，数据结构 struct bss_info

STATUS status: 扫描结果

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
        ...
    }
}
```

10. wifi_station_ap_number_set

功能：

设置 ESP8266 station 最多可记录几个 AP 的信息。

ESP8266 station 成功连入一个 AP 时，可以保存 AP 的 SSID 和 password 记录。

函数定义：

```
bool wifi_station_ap_number_set (uint8 ap_number)
```

参数：

uint8 ap_number：记录 AP 信息的最大数目（最大值为 5）

返回：

true： 成功

false：失败

11. wifi_station_get_ap_info

功能：

获取 ESP8266 station 保存的 AP 信息，最多记录 5 个。

函数定义：

```
uint8 wifi_station_get_ap_info(struct station_config config[])
```

参数：

struct station_config config[]：AP 的信息，数组大小必须为 5

返回：

记录 AP 的数目.

示例：

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

12. wifi_station_ap_change

功能：

ESP8266 station 切换到已记录的某号 AP 配置连接

函数定义：

```
bool wifi_station_ap_change (uint8 new_ap_id)
```

参数：

uint8 new_ap_id : AP 记录的 id 值，从 0 开始计数

返回：

true: 成功

false: 失败

13. wifi_station_get_current_ap_id

功能：

获取当前连接的 AP 保存记录的 id 值。ESP8266 每配置连接一个 AP，会进行记录，从 0 开始计数。

函数定义：

```
uint8 wifi_station_get_current_ap_id ();
```

参数：

无

返回：

当前连接的 AP 保存记录的 id 值。

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)
```

参数：

无

返回：

0: 不自动连接 AP ；

Non-0: 自动连接 AP 。

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：

本接口如果在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)
```

参数：

uint8 set：上电是否自动连接 AP

0：不自动连接 AP

1：自动连接 AP

返回：

true：成功

false：失败

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)
```

参数：

无

返回：

true：成功

false：失败

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)
```

参数：

无

返回:

true: 成功

false: 失败

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)
```

参数:

无

返回:

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

19. wifi_softap_get_config

功能:

查询 ESP8266 WiFi soft-AP 接口配置

函数定义:

```
bool wifi_softap_get_config(struct softap_config *config)
```

参数:

`struct softap_config *config` : ESP8266 soft-AP 配置参数

返回:

true: 成功

false: 失败

20. wifi_softap_set_config

功能:

设置 WiFi soft-AP 接口配置

函数定义:

```
bool wifi_softap_set_config (struct softap_config *config)
```

参数:

`struct softap_config *config` : ESP8266 WiFi soft-AP 配置参数

返回：

true： 成功

false：失败

21. wifi_softap_get_station_info

功能：

获取 ESP8266 soft-AP 接口下连入的 station 的信息，包括 MAC 和 IP

函数定义：

```
struct station_info * wifi_softap_get_station_info(void)
```

参数：

无

返回：

struct station_info* ：station 信息的结构体

22. wifi_softap_free_station_info

功能：

释放调用 wifi_softap_get_station_info 时结构体 station_info 占用的空间

函数定义：

```
void wifi_softap_free_station_info(void)
```

参数：

无

返回：

无

获取 **MAC** 和 **IP** 信息示例，注意释放资源：

示例 **1** ：

```
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);    // Free it directly
    station = next_station;
}
```

示例 2:

```
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();    // Free it by calling functions
```

23. wifi_softap_dhcps_start

功能:

开启 ESP8266 soft-AP DHCP server.

注意:

DHCP 默认开启。

函数定义:

```
bool wifi_softap_dhcps_start(void)
```

参数:

无

返回:

true: 成功

false: 失败

24. wifi_softap_dhcps_stop

功能:

关闭 ESP8266 soft-AP DHCP server.

注意:

DHCP 默认开启。

函数定义:

```
bool wifi_softap_dhcps_stop(void)
```

参数:

无

返回:

true: 成功

false: 失败

25. wifi_softap_set_dhcps_lease

功能：

设置 ESP8266 soft-AP DHCP server 分配 IP 地址的范围

注意：

本接口必须在 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

26. wifi_softap_dhcps_status

功能：

获取 ESP8266 soft-AP DHCP server 状态.

函数定义：

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

参数：

无

返回：

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

27. wifi_set_phy_mode

功能：

设置 ESP8266 物理层模式 (802.11b/g/n).

注意：

ESP8266 soft-AP 仅支持 bg.

函数定义：

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

参数：

```
enum phy_mode mode : 物理层模式
enum phy_mode {
    PHY_MODE_11B = 1,
    PHY_MODE_11G = 2,
    PHY_MODE_11N = 3
};
```

返回：

```
true  : 成功
false : 失败
```

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)
```

参数：

无

返回：

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

29. wifi_get_ip_info

功能：

查询 WiFi station 接口或者 soft-AP 接口的 IP 地址

函数定义：

```
bool wifi_get_ip_info(
   uint8 if_index,
   struct ip_info *info
)
```

参数：

uint8 if_index : 获取 station 或者 soft-AP 接口的信息

```
#define STATION_IF      0x00
#define SOFTAP_IF       0x01
```

struct ip_info *info : 获取到的 IP 信息

返回：

true: 成功

false: 失败

30. wifi_set_ip_info

功能：

设置 ESP8266 station 或者 soft-AP 的 IP 地址

注意：

本接口必须在 user_init 中调用。

函数定义：

```
bool wifi_set_ip_info(
   uint8 if_index,
   struct ip_info *info
)
```

参数：

uint8 if_index : 设置 station 或者 soft-AP 接口

```
#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

31. wifi_set_macaddr

功能：

设置 MAC 地址

注意：

本接口必须在 user_init 中调用

函数定义：

```
bool wifi_set_macaddr(
   uint8 if_index,
   uint8 *macaddr
)
```

参数：

```
uint8 if_index  : 设置 station 或者 soft-AP 接口
    #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

32. wifi_get_macaddr

功能：

查询 MAC 地址

函数定义：

```
bool wifi_get_macaddr(
   uint8 if_index,
   uint8 *macaddr
)
```

参数：

```
uint8 if_index  :  查询 station 或者 soft-AP 接口
    #define STATION_IF      0x00
    #define SOFTAP_IF       0x01
uint8 *macaddr :  MAC 地址
```

返回：

true：成功

false：失败

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：

默认为 modem-sleep 模式。

函数定义：

```
bool wifi_set_sleep_type(enum sleep_type type)
```

参数：

enum sleep_type type ： 省电模式

返回：

true：成功

false：失败

34. wifi_get_sleep_type

功能：

查询省电模式。

函数定义：

```
enum sleep_type wifi_get_sleep_type(void)
```

参数：

无

返回：

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

35. wifi_status_led_install

功能：

注册 WiFi 状态 LED。

函数定义：

```
void wifi_status_led_install (
    uint8 gpio_id,
    uint32 gpio_name,
    uint8 gpio_func
)
```

参数：

```
uint8 gpio_id   : gpio id
uint8 gpio_name : gpio mux 名称
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)
```

36. wifi_status_led_uninstall

功能：

注销 WiFi 状态 LED。

函数定义：

```
void wifi_status_led_uninstall ()
```

参数：

无

返回：

无

37. wifi_set_broadcast_if

功能：

设置 ESP8266 发送 UDP 广播包时，从 station 接口还是 soft-AP 接口发送。

默认从 soft-AP 接口发送。

注意：

如果设置仅从 station 接口发 UDP 广播包，会影响 ESP8266 softAP 的功能，DHCP server 无法使用。需要使能 softAP 的广播包功能，才可正常使用 ESP8266 softAP。

函数定义：

```
bool wifi_set_broadcast_if (uint8 interface)
```

参数：

uint8 interface : 1:station； 2:soft-AP； 3:station 和 soft-AP 均发送

返回：

true： 成功

false: 失败

38. wifi_get_broadcast _if

功能：

查询 ESP8266 发送 UDP 广播包时，从 station 接口还是 soft-AP 接口发送。

函数定义：

```
uint8 wifi_get_broadcast_if (void)
```

参数：

无

返回：

1: station

2: soft-AP

3: station 和 soft-AP 接口均发送

3.5. 云端升级 (FOTA) 接口

1. system_upgrade_userbin_check

功能：

查询 user bin

函数定义：

```
uint8 system_upgrade_userbin_check()
```

参数：

无

返回：

```
0x00 : UPGRADE_FW_BIN1, i.e. user1.bin
0x01 : UPGRADE_FW_BIN2, i.e. user2.bin
```

2. system_upgrade_flag_set

功能：

设置升级状态标志。

注意：

若调用 system_upgrade_start 升级，本接口无需调用；

若用户调用 spi_flash_write 自行写 flash 实现升级，新软件写入完成后，将 flag 置为 UPGRADE_FLAG_FINISH， 再调用 system_upgrade_reboot 重启运行新软件。

函数定义：

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

参数：

```
uint8 flag:
#define UPGRADE_FLAG_IDLE      0x00
#define UPGRADE_FLAG_START     0x01
#define UPGRADE_FLAG_FINISH    0x02
```

返回：

无

3. system_upgrade_flag_check

功能：

查询升级状态标志。

函数定义：

```
uint8 system_upgrade_flag_check()
```

参数：

无

返回：

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

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` ：升级服务器的相关参数

返回：

`true`：开始升级

`false`：已经在升级过程中，无法开始升级

5. system_upgrade_reboot

功能：

重启系统，运行新软件

函数定义：

```
void system_upgrade_reboot (void)
```

参数：

无

返回：

无

3.6. Sniffer 相关接口

1. wifi_promiscuous_enable

功能：

开启混杂模式（sniffer）

注意：

若开启混杂模式，请先关闭自动连接 wifi_station_set_auto_connect(0)

sniffer 过程中请勿调用其他 API，请先调用 wifi_promiscuous_enable(0) 退出 sniffer

函数定义：

```
void wifi_promiscuous_enable(uint8 promiscuous)
```

参数：

uint8 promiscuous ：

0：关闭混杂模式；

1：开启混杂模式

返回：

无

示例：

用户可以向 Espressif Systems 申请 sniffer demo

2. wifi_promiscuous_set_mac

功能：

设置 sniffer 模式时的 MAC 地址过滤

注意：

MAC 地址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 地址过滤。

函数定义：

```
void wifi_promiscuous_set_mac(const uint8_t *address)
```

参数：

const uint8_t *address ： MAC 地址

返回：

无

3. wifi_set_promiscuous_rx_cb

功能：

注册混杂模式下的接收数据回调函数，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)
```

参数：

`wifi_promiscuous_cb_t cb` ： 回调函数

返回：

无

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)
```

参数：

无

返回：

信道号

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)
```

参数：

`uint8 channel` ： 信道号

返回：

true: 成功

false: 失败

3.7. smart config 接口

1. smartconfig_start

功能：

开启快连模式，快速连接 ESP8266 station 到 AP。ESP8266 抓取空中特殊的数据包，包含目标 AP 的 SSID 和 password 信息，同时，用户需要通过手机或者电脑广播加密的 SSID 和 password 信息。

注意：

smartconfig_start 未完成之前不可重复执行 smartconfig_start 函数。

smartconfig 过程中，请勿调用其他 API；请先调用 smartconfig_stop，再使用其他 API。

函数定义：

```
bool smartconfig_start(
   sc_type type,
   sc_callback_t cb,
   uint8 log
)
```

参数：

sc_type type : 快连协议类型：AirKiss 或者 ESP-TOUCH；

sc_callback_t cb : ESP8266 监听到目标 AP 的 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。

uint8 log : 1：UART 打印连接过程；否则：UART 仅打印连接结果。

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);
```

2. smartconfig_stop

功能：

关闭快连模式，释放 smartconfig_start 占用的内存。

注意：

快连成功，连上目标 AP 后，调用本接口释放 smartconfig_start 占用的内存。

函数定义：

```
bool smartconfig_stop(void)
```

参数：

无

返回：

true: 成功

false: 失败

3. get_smartconfig_status

功能：

查询快连状态

注意：

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：

无

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;
```

注意：

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

4. TCP/UDP 接口

位于 esp_iot_sdk/include/espconn.h

网络相关接口可分为以下三类:

- **通用接口**: TCP 和 UDP 均可以调用的接口。
- **TCP APIs**: 仅建立 TCP 连接时，使用的接口。
- **UDP APIs**: 仅收发 UDP 包时，使用的接口。

espconn 接口返回值与 lwip 错误码基本一致，具体如下:

```
/* Definitions for error constants. */
#define ESPCONN_OK          0    /* No error, everything OK. */
#define ESPCONN_MEM        -1    /* Out of memory error.     */
#define ESPCONN_TIMEOUT    -3    /* Timeout.                 */
#define ESPCONN_RTE        -4    /* Routing problem.         */
#define ESPCONN_INPROGRESS -5    /* Operation in progress    */

#define ESPCONN_ABRT       -8    /* Connection aborted.      */
#define ESPCONN_RST        -9    /* Connection reset.        */
#define ESPCONN_CLSD       -10   /* Connection closed.       */
#define ESPCONN_CONN       -11   /* Not connected.           */

#define ESPCONN_ARG        -12   /* Illegal argument.        */
#define ESPCONN_ISCONN     -15   /* Already connected.       */
```

4.1. 通用接口

1. espconn_delete

功能：

删除传输连接。

注意：

对应创建传输的接口如下：

TCP: espconn_accept,

UDP: espconn_create

函数定义：

```
sint8 espconn_delete(struct espconn *espconn)
```

参数：

struct espconn *espconn ： 对应网络传输的结构体

返回：

0 ： 成功

Non-0 ： 失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ： 未找到参数 espconn 对应的网络传输

2. espconn_gethostbyname

功能：

DNS 功能

函数定义：

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

参数：

struct espconn *espconn ： 对应网络传输的结构体

const char *hostname ： 域名字符串的指针

ip_addr_t *addr ： IP 地址

dns_found_callback found ： DNS 回调函数

返回：

err_t : ESPCONN_OK

ESPCONN_INPROGRESS

ESPCONN_ARG

示例如下，请参考 **IoT_Demo**：

```
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_esp_platform_dns_found);
}
```

3. espconn_port

功能：

获取 ESP8266 可用的端口

函数定义：

```
uint32 espconn_port(void)
```

参数：

无

返回：

端口号

4. espconn_regist_sentcb

功能：

注册网络数据发送成功的回调函数

函数定义：

```
sint8 espconn_regist_sentcb(
    struct espconn *espconn,
    espconn_sent_callback sent_cb
)
```

参数：

`struct espconn *espconn` ： 对应网络传输的结构体

`espconn_sent_callback sent_cb` ： 网络数据发送成功的回调函数

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的网络传输

5. espconn_regist_recvcb

功能：

注册成功接收网络数据的回调函数

函数定义：

```
sint8 espconn_regist_recvcb(
    struct espconn *espconn,
    espconn_recv_callback recv_cb
)
```

参数：

struct espconn *espconn ：对应网络传输的结构体

espconn_connect_callback connect_cb ：成功接收网络数据的回调函数

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的网络传输

6. espconn_sent_callback

功能：

网络数据发送成功的回调函数，由 espconn_regist_sentcb 注册

函数定义：

```
void espconn_sent_callback (void *arg)
```

参数：

void *arg ：回调函数的参数，网络传输的结构体指针

返回：

无

7. espconn_recv_callback

功能：

成功接收网络数据的回调函数，由 espconn_regist_recvcb 注册

函数定义：

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

参数：

`void *arg` ： 回调函数的参数，网络传输结构体指针

`char *pdata` ： 接收到的数据

`unsigned short len` ： 接收到的数据长度

返回：

无

8. espconn_sent

功能：

通过 WiFi 发送数据

注意：

一般请在前一包数据发送成功，进入 `espconn_sent_callback` 后，再调用 `espconn_sent` 发送下一包数据。

函数定义：

```
sint8 espconn_sent(
    struct espconn *espconn,
    uint8 *psent,
    uint16 length
)
```

参数：

`struct espconn *espconn` ： 对应网络传输的结构体

`uint8 *psent` ： 发送的数据

`uint16 length` ： 发送的数据长度

返回：

0 ： 成功

Non-0 ： 失败，返回错误码（请参考 `espconn.h`）

其中 `ESPCONN_ARG` ： 未找到参数 `espconn` 对应的网络传输

4.2. TCP 接口

TCP 接口仅用于 TCP 连接，请勿用于 UDP 传输。

1. espconn_accept

功能：

创建 TCP server，建立侦听

函数定义：

```
sint8 espconn_accept(struct espconn *espconn)
```

参数：

struct espconn *espconn ：对应网络连接的结构体

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的 TCP 连接

2. espconn_secure_accept

功能：

创建 SSL TCP server，侦听 SSL 握手

函数定义：

```
sint8 espconn_secure_accept(struct espconn *espconn)
```

参数：

struct espconn *espconn ：对应网络连接的结构体

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的 TCP 连接

3. espconn_regist_time

功能：

注册 ESP8266 TCP server 超时时间

注意：

请在 espconn_accept 之后调用本接口

如果超时时间设置为 0，ESP8266 TCP server 将始终不会断开已经不与它通信的 TCP client，不建议这样使用。

函数定义：

```
sint8 espconn_regist_time(
        struct espconn *espconn,
        uint32 interval,
        uint8 type_flag
)
```

参数：

struct espconn *espconn ：对应网络连接的结构体

uint32 interval ：超时时间，单位：秒，最大值：7200 秒

uint8 type_flag ：0，对所有 TCP 连接生效；1，仅对某一 TCP 连接生效

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的 TCP 连接

4. espconn_get_connection_info

功能：

针对 TCP 多连接的情况，获得某个 ESP8266 TCP server 连接的所有 TCP client 的信息。

函数定义：

```
sint8 espconn_get_connection_info(
        struct espconn *espconn,
        remot_info **pcon_info,
        uint8 typeflags
)
```

参数：

struct espconn *espconn ：对应网络连接的结构体

remot_info **pcon_info ：connect to client info

uint8 typeflags ：0, regular server;1, ssl server

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的 TCP 连接

5. espconn_connect

功能：

连接 TCP server（ESP8266 作为 TCP client）.

函数定义：

```
sint8 espconn_connect(struct espconn *espconn)
```

参数：

struct espconn *espconn ：对应网络连接的结构体

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的 TCP 连接

6. espconn_connect_callback

功能：

成功建立 TCP 连接的回调函数，由 espconn_regist_connectcb 注册。ESP8266 作为 TCP server 侦听到 TCP client 连入；或者 ESP8266 作为 TCP client 成功与 TCP server 建立连接。

函数定义：

```
void espconn_connect_callback (void *arg)
```

参数：

void *arg ：回调函数的参数

返回：

无

7. espconn_set_opt

功能：

设置 TCP 连接的相关配置，对应清除配置标志位的接口为 espconn_clear_opt

函数定义：

```
sint8 espconn_set_opt(
        struct espconn *espconn,
        uint8 opt
)
```

结构体：

```
enum espconn_option{
       ESPCONN_START = 0x00,
       ESPCONN_REUSEADDR = 0x01,
       ESPCONN_NODELAY = 0x02,
       ESPCONN_COPY = 0x04,
       ESPCONN_KEEPALIVE = 0x08,
       ESPCONN_END
}
```

参数：

struct espconn *espconn ：对应网络连接的结构体

uint8 opt ：TCP 连接的相关配置，参考 espconn_option

bit 0: 1: TCP 连接断开时，及时释放内存，无需等待 2 分钟才释放占用内存；

bit 1: 1: 关闭 TCP 数据传输时的 nalge 算法，加快传输速度；

bit 2: 1: 使能 2920 字节的 write buffer，用于缓存 espconn_sent 要发送的数据

bit 3: 1: 使能 keep alive；

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的 TCP 连接

注意：

一般情况下，不需要调用本接口；

如果设置 espconn_set_opt，请在 espconn_connect_callback 中调用

8. espconn_clear_opt

功能：

清除 TCP 连接的相关配置

函数定义：

```
sint8 espconn_clear_opt(
        struct espconn *espconn,
        uint8 opt
)
```

结构体：

```
enum espconn_option{
    ESPCONN_START = 0x00,
    ESPCONN_REUSEADDR = 0x01,
    ESPCONN_NODELAY = 0x02,
    ESPCONN_COPY = 0x04,
    ESPCONN_KEEPALIVE = 0x08,
    ESPCONN_END
}
```

参数：

struct espconn *espconn ：对应网络连接的结构体

uint8 opt ：清除 TCP 连接的相关配置，配置参数可参考 espconn_option

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的 TCP 连接

9. espconn_set_keepalive

功能：

设置 TCP keep alive 的参数

函数定义：

```
sint8 espconn_set_keepalive(struct espconn *espconn, uint8 level, void*
optarg)
```

结构体：

```
enum espconn_level{
    ESPCONN_KEEPIDLE,
    ESPCONN_KEEPINTVL,
    ESPCONN_KEEPCNT
}
```

参数：

struct espconn *espconn ：对应网络连接的结构体

uint8 level ：默认设置为每隔 ESPCONN_KEEPIDLE 时长进行一次 keep alive 探查，如果报文无响应，则每隔 ESPCONN_KEEPINTVL 时长探查一次，最多探查 ESPCONN_KEEPCNT 次；若始终无响应，则认为网络连接断开，释放本地连接相关资源，进入 espconn_reconnect_callback。

参数说明如下：

ESPCONN_KEEPIDLE – 设置进行 keep alive 探查的时间间隔，单位：500 毫秒

ESPCONN_KEEPINTVL – keep alive 探查过程中，报文的时间间隔，单位：500 毫秒

ESPCONN_KEEPCNT – 每次 keep alive 探查，发送报文的最大次数

void* optarg ：设置参数值

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的 TCP 连接

注意：

一般情况下，不需要调用本接口；

如果设置，请在 espconn_connect_callback 中调用，并先设置 espconn_set_opt 使能 keep alive；

10. espconn_get_keepalive

功能：

查询 TCP keep alive 的参数

函数定义：

```
sint8 espconn_set_keepalive(struct espconn *espconn, uint8 level, void*
optarg)
```

结构体：

```
enum espconn_level{
    ESPCONN_KEEPIDLE,
    ESPCONN_KEEPINTVL,
    ESPCONN_KEEPCNT
}
```

参数：

struct espconn *espconn ：对应网络连接的结构体

uint8 level ：

ESPCONN_KEEPIDLE – 设置进行 keep alive 探查的时间间隔，单位：500 毫秒

ESPCONN_KEEPINTVL – keep alive 探查过程中，报文的时间间隔，单位：500 毫秒

ESPCONN_KEEPCNT – 每次 keep alive 探查，发送报文的最大次数

void* optarg ：参数值

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的 TCP 连接

11. espconn_disconnect

功能：

断开 TCP 连接

函数定义：

```
sint8 espconn_disconnect(struct espconn *espconn)
```

参数：

```
struct espconn *espconn : 对应网络连接的结构体
```

返回：

```
0          : 成功
Non-0      : 失败，返回错误码（请参考 espconn.h）
             其中 ESPCONN_ARG : 未找到参数 espconn 对应的 TCP 连接
```

12. espconn_regist_connectcb

功能：

注册成功建立 TCP 连接后的回调函数

函数定义：

```
sint8 espconn_regist_connectcb(
        struct espconn *espconn,
        espconn_connect_callback connect_cb
)
```

参数：

```
struct espconn *espconn : 对应网络连接的结构体
espconn_connect_callback connect_cb : 成功建立 TCP 连接后的回调函数
```

返回：

```
0          : 成功
Non-0      : 失败，返回错误码（请参考 espconn.h）
             其中 ESPCONN_ARG : 未找到参数 espconn 对应的 TCP 连接
```

13. espconn_regist_reconcb

功能：

注册 TCP 连接发生异常断开时的回调函数，可以在回调函数中进行重连。

注意：

reconnect callback 功能类似于出错处理回调，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
)
```

参数：

struct espconn *espconn ：对应网络连接的结构体

espconn_connect_callback connect_cb ：回调函数

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的 TCP 连接

14. espconn_regist_disconcb

功能：

注册 TCP 连接正常断开成功的回调函数

函数定义：

```
sint8 espconn_regist_disconcb(
        struct espconn *espconn,
        espconn_connect_callback discon_cb
)
```

参数：

struct espconn *espconn ：对应网络连接的结构体

espconn_connect_callback connect_cb ：回调函数

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的 TCP 连接

15. espconn_regist_write_finish

功能：

注册所有需发送的数据均成功写入 write buffer 后的回调函数。

write buffer 用于缓存 espconn_sent 将发送的数据，由 espconn_set_opt 设置，对发送速度有要求时，可以在 write finish callback 中调用 espconn_sent 发送下一包，无需等到 espconn_sent_callback

函数定义：

```
sint8 espconn_regist_write_finish (
        struct espconn *espconn,
        espconn_connect_callback write_finish_fn
)
```

参数：

```
struct espconn *espconn   :  对应网络连接的结构体
espconn_connect_callback write_finish_fn :  回调函数
```

返回：

```
0          : 成功
Non-0      : 失败，返回错误码（请参考 espconn.h）
             其中 ESPCONN_ARG : 未找到参数 espconn 对应的 TCP 连接
```

16. espconn_secure_connect

功能：

加密（SSL）连接到 TCP SSL server（ESP8266 作为 TCP SSL client.）

函数定义：

```
sint8 espconn_secure_connect (struct espconn *espconn)
```

参数：

```
struct espconn *espconn : 对应网络连接的结构体
```

返回：

```
0          : 成功
Non-0      : 失败，返回错误码（请参考 espconn.h）
             其中 ESPCONN_ARG : 未找到参数 espconn 对应的 TCP 连接
```

17. espconn_secure_sent

功能：

发送加密数据（SSL）

函数定义：

```
sint8 espconn_secure_sent (
        struct espconn *espconn,
        uint8 *psent,
        uint16 length
)
```

参数：

struct espconn *espconn ：对应网络连接的结构体

uint8 *psent ：发送的数据

uint16 length ：发送的数据长度

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的 TCP 连接

18. espconn_secure_disconnect

功能：

断开加密 TCP 连接(SSL)

函数定义：

```
sint8 espconn_secure_disconnect(struct espconn *espconn)
```

参数：

struct espconn *espconn ：对应网络连接的结构体

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的 TCP 连接

19. espconn_tcp_get_max_con

功能：

查询允许的 TCP 最大连接数。

函数定义：

```
uint8 espconn_tcp_get_max_con(void)
```

参数：

无

返回：

允许的 TCP 最大连接数

20. espconn_tcp_set_max_con

功能：

设置允许的 TCP 最大连接数。在内存足够的情况下，建议不超过 10。默认值为 5。

函数定义：

```
sint8 espconn_tcp_set_max_con(uint8 num)
```

参数：

uint8 num ： 允许的 TCP 最大连接数

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的 TCP 连接

21. 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)
```

参数：

struct espconn *espconn ：对应的 TCP server 结构体

返回：

最多允许连接的 TCP client 数目

22. 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)
```

参数：

struct espconn *espconn ：对应的 TCP server 结构体

uint8 num ：最多允许连接的 TCP client 数目

返回：

0 ：成功

Non-0 ：失败，返回错误码（请参考 espconn.h）

其中 ESPCONN_ARG ：未找到参数 espconn 对应的 TCP 连接

23. espconn_recv_hold

功能：

阻塞 TCP 接收数据

注意：

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 字节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)
```

参数：

```
struct espconn *espconn : 对应网络连接的结构体
```

返回：

```
0          : 成功
Non-0      : 失败，返回错误码（请参考 espconn.h）
             其中 ESPCONN_ARG : 未找到参数 espconn 对应的 TCP 连接
```

24. espconn_recv_unhold

功能：

解除 TCP 收包阻塞（i.e. 对应的阻塞接口 espconn_recv_hold).

注意：

本接口实时生效。

函数定义：

```
sint8 espconn_recv_unhold(struct espconn *espconn)
```

参数：

```
struct espconn *espconn : 对应网络连接的结构体
```

返回：

```
0          : 成功
Non-0      : 失败，返回错误码（请参考 espconn.h）
             其中 ESPCONN_ARG : 未找到参数 espconn 对应的 TCP 连接
```

4.3. UDP 接口

1. espconn_create

功能：

建立 UDP 传输。

函数定义：

```
sin8 espconn_create(struct espconn *espconn)
```

参数：

```
struct espconn *espconn : 对应网络连接的结构体
```

返回:

0 : 成功

Non-0 : 失败，返回错误码 (请参考 espconn.h).

其中 ESPCONN_ARG : 未找到参数 espconn 对应的 UDP 传输

2. espconn_igmp_join

功能:

加入多播组。

函数定义:

```
sint8 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 : 成功

Non-0 : 失败，返回错误码 (请参考 espconn.h).

其中 ESPCONN_ARG : 未找到参数 espconn 对应的 UDP 传输

3. espconn_igmp_leave

功能:

退出多播组。

函数定义:

```
sint8 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 : 成功

Non-0 : 失败，返回错误码 (请参考 espconn.h).

其中 ESPCONN_ARG : 未找到参数 espconn 对应的 UDP 传输

5. 应用相关接口

5.1. AT 接口

AT 接口的使用示例，请参考 esp_iot_sdk/examples/at/user/user_main.c.

1. at_response_ok

功能：

AT 串口（UART0）输出 OK

函数定义：

```
void at_response_ok(void)
```

参数：

无

返回：

无

2. at_response_error

功能：

AT 串口（UART0）输出 ERROR

函数定义：

```
void at_response_error(void)
```

参数：

无

返回：

无

3. at_cmd_array_regist

功能：

注册用户自定义的 AT 指令。请仅调用一次，将所有用户自定义 AT 指令一并注册。

函数定义：

```
void at_cmd_array_regist (
        at_function * custom_at_cmd_arrar,
        uint32 cmd_num
)
```

参数：

at_function * custom_at_cmd_arrar ：用户自定义的 AT 指令数组

uint32 cmd_num ：用户自定义的 AT 指令数目

返回：

无

示例：

请参考 esp_iot_sdk/examples/at/user/user_main.c

4. at_get_next_int_dec

功能：

从 AT 指令行中解析 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 指令字符串

int* result : 从 AT 指令中解析出的 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：

请参考 esp_iot_sdk/examples/at/user/user_main.c

5. at_data_str_copy

功能：

从 AT 指令行中解析字符串

函数定义：

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

参数：

char * p_dest : 从 AT 指令行中解析到的字符串

char ** p_src : *p_src 为接收到的 AT 指令字符串

int32 max_len : 允许的最大字符串长度

返回：

解析到的字符串长度：

>=0：成功，则返回解析到的字符串长度

<0 : 失败，返回 -1

示例：

请参考 esp_iot_sdk/examples/at/user/user_main.c

6. at_init

功能：

AT 初始化

函数定义：

```
void at_init (void)
```

参数：

无

返回：

无

示例：

请参考 esp_iot_sdk/examples/at/user/user_main.c

7. at_port_print

功能：

从 AT 串口(UART0) 输出字符串

函数定义：

```
void at_port_print(const char *str)
```

参数：

`const char *str` : 字符串

返回：

无

示例：

请参考 esp_iot_sdk/examples/at/user/user_main.c

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` : 版本信息

返回：

无

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：

无

返回：

无

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：

无

返回：

无

11. at_get_version

功能：

查询 Espressif Systems 提供的 AT lib 版本号.

函数定义：

```
uint32 at_get_version (void)
```

参数：

无

返回：

Espressif AT lib 版本号

5.2. JSON 接口

位于 : esp_iot_sdk/include/json/jsonparse.h & jsontree.h

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
)
```

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

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)
```

参数:

struct jsonparse_state *state : json 解析指针

返回:

int : 解析结果

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
)
```

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
```

4. jsonparse_get_value_as_int

功能：

解析 json 格式为整型数据

函数定义：

```
int jsonparse_get_value_as_int(struct jsonparse_state *state)
```

参数：

```
struct jsonparse_state *state : json 解析指针
```

返回：

```
int : 解析结果
```

5. jsonparse_get_value_as_long

功能：

解析 json 格式为长整型数据

函数定义：

```
long jsonparse_get_value_as_long(struct jsonparse_state *state)
```

参数：

```
struct jsonparse_state *state : json 解析指针
```

返回：

```
long : 解析结果
```

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)
```

参数：

```
struct jsonparse_state *state : json 解析指针
```

返回：

```
int : 数据长度
```

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)
```

参数：

```
struct jsonparse_state *state : json 解析指针
```

返回：

```
int : json 格式数据类型
```

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)
```

参数：

```
struct jsonparse_state *state : json 解析指针
const char *str : 字符串缓存
```

返回：

```
int : 比较结果
```

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)
)
```

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

10. jsontree_reset

功能：

设置 json 树

函数定义：

```
void jsontree_reset(struct jsontree_context *js_ctx)
```

参数：

`struct jsontree_context *js_ctx` : json 格式树指针

返回：

无

11. jsontree_path_name

功能：

获取 json 树参数

函数定义：

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

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

12. jsontree_write_int

功能：

整型数写入 json 树

函数定义：

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

参数：

struct jsontree_context *js_ctx : json 树指针

int value : 整型数

返回：

无

13. jsontree_write_int_array

功能：

整型数组写入 json 树

函数定义：

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

参数：

struct jsontree_context *js_ctx : json 树指针

int *text : 数组入口地址

uint32 length : 数组长度

返回：

无

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
)
```

参数：

```
struct jsontree_context *js_ctx : json 格式树指针
const char* text : 字符串指针
```

返回：

无

15. jsontree_print_next

功能：

获取 json 树下一个元素

函数定义：

```
int jsontree_print_next(struct jsontree_context *js_ctx)
```

参数：

```
struct jsontree_context *js_ctx : json 树指针
```

返回：

```
int : json 树深度
```

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
)
```

参数：

```
struct jsontree_context *js_ctx : json 树指针
int : 类型
```

返回：

```
struct jsontree_value * : json 树元素指针
```

6. 结构体定义

6.1. 定时器

```
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;
```

6.2. WiFi 参数

1. station 参数

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

注意：

BSSID 表示 AP 的 MAC 地址，用于多个 AP 的 SSID 相同的情况。

如果 station_config.bssid_set==1 ， station_config.bssid 必须设置，否则连接失败。

一般情况， station_config.bssid_set 设置为 0.

2. soft-AP 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：

如果 softap_config.ssid_len==0， 读取 SSID 直至结束符；

否则，根据 softap_config.ssid_len 设置 SSID 的长度。

3. scan 参数

```
struct scan_config {
    uint8 *ssid;
    uint8 *bssid;
    uint8 channel;
    uint8 show_hidden; // Scan APs which are hiding their ssid or not.
};
struct bss_info {
    STAILQ_ENTRY(bss_info) next;
    u8 bssid[6];
    u8 ssid[32];
    u8 channel;
    s8 rssi;
    u8 authmode;
    uint8 is_hidden; // SSID of current AP is hidden or not.
};
typedef void (* scan_done_cb_t)(void *arg, STATUS status);
```

4. smart config 结构体

```
typedef enum  {
    SC_STATUS_WAIT = 0,     // 连接未开始，请勿在此阶段开始连接
    SC_STATUS_FIND_CHANNEL, // 请在此阶段开启 APP 进行配对连接
    SC_STATUS_GETTING_SSID_PSWD,
    SC_STATUS_GOT_SSID_PSWD,
    SC_STATUS_LINK,
    SC_STATUS_LINK_OVER,  // 获取到 IP，连接路由完成
} sc_status;
typedef enum  {
```

```
    SC_TYPE_ESPTOUCH = 0,
    SC_TYPE_AIRKISS,
} sc_type;
```

6.4. json 相关结构体

1. json 结构体

```
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];
};
```

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 }
```

6.5. espconn 参数

1. 回调函数

```
/** 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);
```

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,
};

enum espconn_option{
       ESPCONN_START = 0x00,
       ESPCONN_REUSEADDR = 0x01,
       ESPCONN_NODELAY = 0x02,
       ESPCONN_COPY = 0x04,
```

```
        ESPCONN_KEEPALIVE = 0x08,
        ESPCONN_END
}

enum espconn_level{
        ESPCONN_KEEPIDLE,
        ESPCONN_KEEPINTVL,
        ESPCONN_KEEPCNT
}

/** 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;
    uint8 link_cnt;
    void *reverse; // reversed for customer use
};
```

7. 外围设备驱动接口

7.1. GPIO 接口

请参考 /user/user_plug.c.

1. PIN 相关宏定义

以下宏定义控制 GPIO 管脚状态

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

示例:

```
// Use MTDI pin as GPIO12.

PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U, FUNC_GPIO12);
```

2. gpio_output_set

功能:

设置 GPIO 属性

函数定义:

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

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

示例:

gpio_output_set(BIT12, 0, BIT12, 0):

设置 GPIO12 输出高电平;

gpio_output_set(0, BIT12, BIT12, 0):

设置 GPIO12 输出低电平;

gpio_output_set(BIT12, BIT13, BIT12|BIT13, 0):

设置 GPIO12 输出高电平， GPIO13 输出低电平;

gpio_output_set(0, 0, 0, BIT12):

设置 GPIO12 为输入

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，与上一节的输出高低电平的示例相同。

GPIO_DIS_OUTPUT(gpio_no)

设置 gpio_no 管脚输入，与上一节的设置输入示例相同。

GPIO_INPUT_GET(gpio_no)

获取 gpio_no 管脚的电平状态。

4. GPIO 中断

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. gpio_pin_intr_state_set

功能:

设置 GPIO 中断触发状态

函数定义:

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

参数：

uint32 i : GPIO pin ID，例如设置 GPIO14，则为 GPIO_ID_PIN(14);

GPIO_INT_TYPE intr_state : 中断触发状态：

```
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

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);
```

7.2. UART 接口

默认情况下，UART0 作为系统的打印信息输出接口，当配置为双 UART 时，UART0 作为数据收发接口，UART1 作为打印信息输出接口。

使用时，请确保硬件连接正确。

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
)
```

参数：

UartBautRate uart0_br : uart0 波特率

UartBautRate uart1_br : uart1 波特率

波特率：

```
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

2. uart0_tx_buffer

功能：

通过 UART0 输出用户数据

函数定义：

```
void uart0_tx_buffer(uint8 *buf, uint16 len)
```

参数：

uint8 *buf ：数据缓存

uint16 len ：数据长度

返回：

无

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)
```

参数：

void *para ：指向数据结构 RcvMsgBuff 的指针

返回：

无

7.3. 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。

1. i2c_master_gpio_init

功能：

设置 GPIO 为 I2C master 模式

函数定义：

```
void i2c_master_gpio_init (void)
```

参数：

无

返回：

无

2. i2c_master_init

功能：

初始化 I2C

函数定义：

```
void i2c_master_init(void)
```

参数：

无

返回：

无

3. i2c_master_start

功能：

设置 I2C 进入发送状态

函数定义：

```
void i2c_master_start(void)
```

参数：

无

返回：

无

4. i2c_master_stop

功能：

设置 I2C 停止发送

函数定义：

```
void i2c_master_stop(void)
```

参数：

无

返回：

无

5. i2c_master_send_ack

功能：

发送 I2C ACK

函数定义：

```
void i2c_master_send_ack (void)
```

参数：

无

返回：

无

6. i2c_master_send_nack

功能：

发送 I2C NACK

函数定义：

```
void i2c_master_send_nack (void)
```

参数：

无

返回：

无

7. i2c_master_checkAck

功能：

检查 I2C slave 的 ACK

函数定义：

```
bool i2c_master_checkAck (void)
```

参数：

无

返回：

true: 获取 I2C slave ACK

false: 获取 I2C slave NACK

8. i2c_master_readByte

功能：

从 I2C slave 读取一个字节

函数定义：

```
uint8 i2c_master_readByte (void)
```

参数：

无

返回：

uint8 : 读取到的值

9. i2c_master_writeByte

功能：

向 I2C slave 写一个字节

函数定义：

```
void i2c_master_writeByte (uint8 wrdata)
```

参数：

uint8 wrdata : 数据

返回：

无

7.4. PWM 接口

ESP8266 目前支持 4 路 PWM，可以在 pwm.h 中对采用的 GPIO 进行配置选择，也可以新增多路 PWM，但这部分内容将不在本文档中介绍。

1. pwm_init

功能：

初始化 PWM，包括 GPIO 选择，频率和占空比

函数定义：

```
void pwm_init(uint16 freq, uint8 *duty)
```

参数：

uint16 freq : PWM 频率;

uint8 *duty : 各路 PWM 的占空比

返回：

无

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)
```

参数：

无

返回：

无

3. pwm_set_duty

功能：

设置某路 PWM 的占空比

函数定义：

```
void pwm_set_duty(uint8 duty, uint8 channel)
```

参数：

uint8 duty : 占空比

uint8 channel : 某路 PWM

返回：

无

4. pwm_set_freq

功能：

设置 PWM 频率

函数定义：

```
void pwm_set_freq(uint16 freq)
```

参数：

`uint16 freq` ：PWM 频率

返回：

无

5. pwm_get_duty

功能：

获取某路 PWM 的占空比

函数定义：

```
uint8 pwm_get_duty(uint8 channel)
```

参数：

`uint8 channel` ：待查询的某路 PWM

返回：

PWM 占空比

6. pwm_get_freq

功能：

查询 PWM 频率.

函数定义：

```
uint16 pwm_get_freq(void)
```

参数：

无

返回：

PWM 频率

8. 附录

8.1. ESPCONN 编程

1. TCP Client 模式

注意

- ESP8266 工作在 station 模式下，需确认 ESP8266 已经连接 AP (路由) 分配到 IP 地址，启用 client 连接。
- ESP8266 工作在 softap 模式下，需确认连接 ESP8266 的设备已被分配到 IP 地址，启用 client 连接。

步骤

- 依据工作协议初始化 `espconn` 参数；
- 注册连接成功的回调函数和连接失败重连的回调函数；
  - (调用 `espconn_regist_connectcb` 和 `espconn_regist_reconcb`)
- 调用 `espconn_connect` 建立与 TCP Server 的连接；
- TCP连接建立成功后，在连接成功的回调函数（connect callback）中，注册接收数据的回调函数，发送数据成功的回调函数和断开连接的回调函数。
  - (调用 `espconn_regist_recvcb` , `espconn_regist_sentcb` 和 `espconn_regist_disconcb`)
- 在接收数据的回调函数，或者发送数据成功的回调函数中，执行断开连接操作时，建议适当延时一定时间，确保底层函数执行结束。

2. TCP Server 模式

注意

- ESP8266 工作在 station 模式下，需确认 ESP8266 已经分配到 IP 地址，再启用server侦听。
- ESP8266 工作在 softap 模式下，可以直接启用 server 侦听。

步骤

- 依据工作协议初始化 `espconn` 参数；
- 注册连接成功的回调函数和连接失败重连的回调函数；
  - (调用 `espconn_regist_connectcb` 和 `espconn_regist_reconcb`)
- 调用 `espconn_accept` 侦听 TCP 连接；
- TCP连接建立成功后，在连接成功的回调函数（connect callback）中，注册接收数据的回调函数，发送数据成功的回调函数和断开连接的回调函数。
  - (调用 `espconn_regist_recvcb` , `espconn_regist_sentcb` 和 `espconn_regist_disconcb`)

8.2. RTC APIs 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();
    stime = system_get_time();

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

    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) ;

    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();
}
```

8.3. Sniffer 结构体说明

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 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 11n packet; 1:is not 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
    };
```

回调函数 `wifi_promiscuous_rx` 含两个参数 (`buf` 和 `len`). `len` 表示 `buf`的长度，`len` = 12 或者 `len` ≥ 60:

LEN ≥ 60 的情况

- `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==12 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。

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：

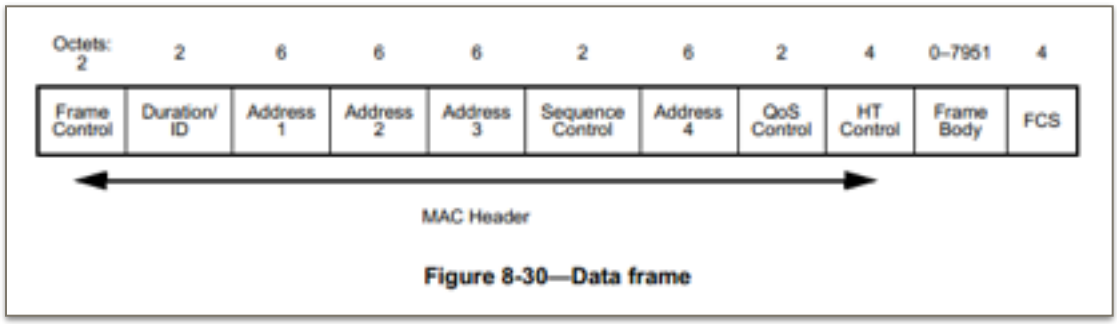

Figure 8-30—Data frame

- 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。

8.4. ESP8266 ADC & VDD3P3

ESP8266 片上的 SARADC 提供以下两种应用，但以下两种应用不可同时使用：

- 测量 VDD3P3 管脚 3 和 4 上的电源电压：

  测量 PA_VDD 管脚电压的函数是：uint16 system_get_vdd33(void)；

- 测量 TOUT 管脚 6 的输入电压：

测量 TOUT 管脚电压的函数是：uint16 system_adc_read(void)。

后文描述的 RF_init 参数，指 esp_init_data_default.bin。

应用场景 1：测量 VDD3P3 管脚 3 和 4 的电源电压

硬件设计: TOUT 管脚必须悬空。

RF_init 参数：

esp_init_data_default.bin（0～127byte）中的第 107 byte 为“vdd33_const“ 必须设为 0xFF，即 255；

RF Calibration 工作过程：

自测 VDD3P3 管脚3和4 上的电源电压，根据测量结果优化 RF 电路工作状态。

用户软件：可使用system_get_vdd33； 不可使用 system_adc_read。

应用场景 2：TOUT管脚 6 的输入电压

硬件设计: TOUT 管脚接 外部电路，输入电压范围限定为 0~1.0V。

RF_init 参数：

esp_init_data_default.bin（0～127byte）中的第 107 byte 为“vdd33_const” 必须设为真实的 VDD3P3 管脚 3 和 4 上的电源电压。

ESP8266 的工作电压范围为1.8V~3.6V，vdd33_const 单位为0.1V，因此 vdd33_const 有效取值范围为 18~36；

若电源电压不稳定，会动态变化，vdd33_const 应输入为电源电压变化的最小值 x 10.

RF Calibration 工作过程：

根据 RF_init 第 107 byte “vdd33_const” 的值来优化 RF 电路工作状态，容许误差约为 + - 0.2V。

用户软件：不可使用 system_get_vdd33； 可使用 system_adc_read。

注意

注1. RF_init参数，即esp_init_data_default.bin（0～127）byte中的第 107 byte，命名为vdd33_const， 此参数的解释如下：

（1）当 vdd33_const = 0xff，ESP8266 RF Calibration 内部自测 VDD3P3 管脚 3 和 4 上的电源电压，根据测量结果优化 RF 电路工作状态。

（2）当 18 =< vdd33_const =< 36，ESP8266 RF Calibration 使用（vdd33_const/10）来 优化 RF 电路工作状态。

（3）当 vdd33_const < 18 或 36 < vdd33_const < 255 时，ESP8266 RF Calibration 使用默认值 3.0V 来优

化 RF 电路工作状态。

注2. 函数 system_get_vdd33用于测量 VDD3P3 管脚 3 和 4 上的电源电压：
（1）Tout 管脚必须悬空，且必须使 RF_init 参数第 107 byte vdd33_const=0xff。
（2）若 RF_init 参数第 107 byte vdd33_const 等于 0xff， system_get_vdd33 函数返回值才为有效的测量值；否则 system_get_vdd33 函数返回 0xffff。
（3）返回值单位：1/1024 V

注3. 函数 system_adc_read 用于测量Tout管脚6 上的输入电压时：
（1）必须 将 RF_init 参数第 107 byte vdd33_const 设置为 真实的电压电源。
（2）若 RF_init 参数第 107 byte vdd33_const 不等于 0xff， system_adc_read 函 数返回值才为有效的测量值；否则 system_adc_read 函数返回0xffff。
（3）返回值单位：1/1024 V