ESP8266 RTOS SDK 编程手册

Version 1.0.4

Espressif Systems IOT Team

Table of Content

2. 前言 8

2. 概述 9

3. 应用程序接口 (APIs) 10

3.1. 定时器 10

1. os_timer_arm 10
2. os_timer_disarm 10
3. os_timer_setfn 11
4. os_timer_arm_us 11

3.2. 系统接口 12

1. system_get_sdk_version 12
2. system_restore 12
3. system_restart 12
4. system_get_rst_info 13
5. system_get_chip_id 14
6. system_get_vdd33 14
7. system_adc_read 14
8. system_deep_sleep 15
9. system_deep_sleep_set_option 15
10. system_phy_set_rfoption 16
11. system_phy_set_max_tpw 16
12. system_phy_set_tpw_via_vdd33 17
13. system_print_meminfo 17
14. system_get_free_heap_size 17
15. system_get_time 18
16. system_get_rtc_time 18
17. system_rtc_clock_cali_proc 19
18. system_rtc_mem_write 19
19. system_rtc_mem_read 20
20. system_uart_swap 20
21. system_uart_de_swap 21

22. system_get_boot_version 21
23. system_get_userbin_addr 21
24. system_get_boot_mode 22
25. system_restart_enhance 22
26. system_get_flash_size_map 23
27. os_install_putc1 23
28. os_delay_us 24
29. os_putc 24
3.3. SPI Flash 接口 24
1. spi_flash_get_id 24
2. spi_flash_erase_sector 24
3. spi_flash_write 25
4. spi_flash_read 25
5. system_param_save_with_protect 26
6. system_param_load 27
3.4. WIFI 接口 28
1. wifi_get_opmode 28
2. wifi_get_opmode_default 28
3. wifi_set_opmode 29
4. wifi_set_opmode_current 29
5. wifi_station_get_config 29
6. wifi_station_get_config_default 30
7. wifi_station_set_config 30
8. wifi_station_set_config_current 31
9. wifi_station_connect 32
10. wifi_station_disconnect 32
11. wifi_station_get_connect_status 32
12. wifi_station_scan 33
13. scan_done_cb_t 34
14. wifi_station_ap_number_set 34
15. wifi_station_get_ap_info 35
16. wifi_station_ap_change 35
17. wifi_station_get_current_ap_id 35

18. wifi_station_get_auto_connect 36
19. wifi_station_set_auto_connect 36
20. wifi_station_dhcpc_start 36
21. wifi_station_dhcpc_stop 37
22. wifi_station_dhcpc_status 37
23. wifi_station_set_reconnect_policy 38
24. wifi_station_get_reconnect_policy 38
25. wifi_softap_get_config 38
26. wifi_softap_get_config_default 39
27. wifi_softap_set_config 39
28. wifi_softap_set_config_current 39
29. wifi_softap_get_station_num 40
30. wifi_softap_get_station_info 40
31. wifi_softap_free_station_info 41
32. wifi_softap_dhcps_start 41
33. wifi_softap_dhcps_stop 42
34. wifi_softap_set_dhcps_lease 42
35. wifi_softap_dhcps_status 44
36. wifi_softap_set_dhcps_offer_option 44
37. wifi_set_phy_mode 45
38. wifi_get_phy_mode 45
39. wifi_get_ip_info 45
40. wifi_set_ip_info 46
41. wifi_set_macaddr 47
42. wifi_get_macaddr 48
43. wifi_status_led_install 48
44. wifi_status_led_uninstall 48
45. wifi_set_event_handler_cb 49

3.5. 云端升级 (FOTA) 接口 50

1. system_upgrade_userbin_check 50
2. system_upgrade_flag_set 51
3. system_upgrade_flag_check 51
4. system_upgrade_reboot 51

3.6. Sniffer 相关接口 52
1. wifi_promiscuous_enable 52
2. wifi_promiscuous_set_mac 52
3. wifi_set_promiscuous_rx_cb 53
4. wifi_get_channel 53
5. wifi_set_channel 53
3.7. smart config 接口 54
1. smartconfig_start 54
2. smartconfig_stop 56
3.8. cJSON 接口 57
1. cJSON_Parse 57
2. cJSON_Print 57
3. cJSON_Delete 57
4. cJSON_GetArraySize 57
5. cJSON_GetArrayItem 58
6. cJSON_GetObjectItem 58
7. cJSON_CreateXXX 58
8. cJSON_CreateXXXArray 59
9. cJSON_InitHooks 59
10. cJSON_AddItemToArray 60
11. cJSON_AddItemReferenceToArray 60
12. cJSON_AddItemToObject 60
13. cJSON_AddItemReferenceToObject 61
14. cJSON_DetachItemFromArray 61
15. cJSON_DeleteItemFromArray 61
16. cJSON_DetachItemFromObject 62
17. cJSON_DeleteItemFromObject 62
18. cJSON_InsertItemInArray 62
19. cJSON_ReplaceItemInArray 63
20. cJSON_ReplaceItemInObject 63
21. cJSON_Duplicate 64
22. cJSON_ParseWithOpts 64

4. 参数结构体和宏定义 66

4.1. 定时器 66
4.2. Wi-Fi 参数 66
1. station 参数 66
2. soft-AP 参数 66
3. scan 参数 67
4. Wi-Fi event 结构体 67

5. 附录 71
5.1. RTC APIs 使用示例 71
5.2. Sniffer 结构体说明 73
5.3. ESP8266 soft-AP 和 station 信道定义 76

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。

注意

- 建议使用定时器实现长时间的查询功能，可将定时器设置为循环调用，定时器内部请勿使用 `while(1)` 的方式延时；
- 从 esp_iot_rtos_sdk_v1.0.4 起，无需添加宏 `ICACHE_FLASH_ATTR`，函数将默认存放在 CACHE 区，中断函数也可以存放在 CACHE 区；如需将部分频繁调用的函数定义在 RAM 中，请在函数前添加宏 `IRAM_ATTR`；
- 网络编程使用通用的 socket 编程，网络通信时，socket 请勿绑定在同一端口；
- freeRTOS 操作系统及系统自带的 API 说明请参考 http://www.freertos.org
- RTOS SDK 的系统任务优先级为 15 ，创建任务的接口 `xTaskCreate` 为 freeRTOS 自带接口，使用 `xTaskCreate` 创建任务时，任务堆栈设置范围为 [176, 512]
  - 在任务内部如需使用长度超过 60 的大数组，建议使用 `malloc` 和 `free` 的方式操作，否则，大数组将占用任务的堆空间；
  - SDK 底层已占用部分优先级：pp task 优先级 13，精确 ets timer 线程优先级 12，lwip task 优先级 10，freeRTOS timer 优先级 2，idle 优先级 0；
  - 可供用户线程使用的优先级为 1 ~ 9，请勿修改 `FreeRTOSConfig.h`，优先级由 SDK 底层决定，此处修改头文件并不能生效。

3. 应用程序接口 (APIs)

3.1. 定时器

以下软件定时器接口位于 /esp_iot_rtos_sdk/include/espressif/Esp_timer.h

请注意，以下接口使用的定时器由软件实现，定时器的函数在任务中被执行。因为任务可能被中断，或者被其他高优先级的任务延迟，因此以下 os_timer 系列的接口并不能保证定时器精确执行。

- 对于同一个 timer，os_timer_arm 或 os_timer_arm_us 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 或 os_timer_arm_us 之前或者 os_timer_disarm 之后

1. os_timer_arm

功能：

使能毫秒级定时器

函数定义：

```
void os_timer_arm (
    os_timer_t *ptimer,
    uint32_t milliseconds,
    bool repeat_flag
)
```

参数：

`os_timer_t *ptimer` ：定时器结构

`uint32_t milliseconds` ：定时时间，单位：毫秒，最大可输入 0x41893

`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 (os_timer_t *ptimer)
```

参数：

os_timer_t *ptimer ： 定时器结构

返回：

无

3. os_timer_setfn

功能：

设置定时器回调函数。使用定时器，必须设置回调函数。

注意：

在定时器回调函数中已关闭调度。

函数定义：

```
void os_timer_setfn(
    os_timer_t *ptimer,
    os_timer_func_t *pfunction,
    void *parg
)
```

参数：

os_timer_t *ptimer ： 定时器结构

os_timer_func_t *pfunction ： 定时器回调函数

void *parg ： 回调函数的参数

返回：

无

4. os_timer_arm_us

功能：

使能微秒级定时器，

函数定义：

```
void os_timer_arm_us (
    os_timer_t *ptimer,
    uint32_t microseconds,
    bool repeat_flag
)
```

参数：

os_timer_t *ptimer ： 定时器结构

uint32_t microseconds ： 定时时间，单位：微秒，最小定时 0x64 ，最大可输入 0xFFFFFFF

bool repeat_flag ： 定时器是否重复

返回：

无

3.2. 系统接口

1. system_get_sdk_version

功能：

查询 SDK 版本信息

函数定义：

```
const char* system_get_sdk_version(void)
```

参数：

无

返回：

SDK 版本信息

示例：

```
printf("SDK version: %s \n", system_get_sdk_version());
```

2. system_restore

功能：

恢复出厂设置。本接口将清除以下接口的设置，恢复默认值：wifi_station_set_auto_connect，wifi_set_phy_mode，wifi_softap_set_config 相关，wifi_station_set_config 相关，wifi_set_opmode。

函数定义：

```
void system_restore(void)
```

参数：

无

返回：

无

3. system_restart

功能：

系统重启

函数定义：

```
void system_restart(void)
```

参数：

无

返回：

无

4. system_get_rst_info

功能：

查询当前启动的信息。

结构体：

```
enum rst_reason {
    REANSON_DEFAULT_RST       = 0,   // normal startup by power on
    REANSON_WDT_RST           = 1,   // hardware watch dog reset
    // exception reset, GPIO status won't change
    REANSON_EXCEPTION_RST     = 2,
    // software watch dog reset, GPIO status won't change
    REANSON_SOFT_WDT_RST      = 3,
    // software restart ,system_restart , GPIO status won't change
    REANSON_SOFT_RESTART      = 4,
    REANSON_DEEP_SLEEP_AWAKE  = 5,   // wake up from deep-sleep
    };

struct rst_info {
    uint32 reason;   // enum rst_reason
    uint32 exccause;
    uint32 epc1;
    uint32 epc2;
    uint32 epc3;
    uint32 excvaddr;
    uint32 depc;
};
```

函数定义：

```
struct rst_info* system_get_rst_info(void)
```

参数：

无

返回：

启动的信息。

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_get_vdd33

功能：

测量 VDD3P3 管脚 3 和 4 的电压值，单位：1/1024 V

注意：

- system_get_vdd33 必须在 TOUT 管脚悬空的情况下使用。
- TOUT 管脚悬空的情况下，esp_init_data_default.bin（0～127byte）中的第 107 byte 为“vdd33_const“ ，必须设为 0xFF，即 255；

函数定义：

```
uint16 system_get_vdd33(void)
```

参数：

无

返回：

VDD33 电压值。单位：1/1024 V

7. system_adc_read

功能：

测量 TOUT 管脚 6 的输入电压，单位：1/1024 V

注意：

- system_adc_read 必须在 TOUT 管脚接外部电路情况下使用， 且TOUT 管脚输入电压范围限定为 0 ~ 1.0V。
- TOUT 管脚接外部电路的情况下，esp_init_data_default.bin(0～127byte)中的第 107 byte (vdd33_const)，必须设为 VDD3P3 管脚 3 和 4 上真实的电源电压
- 第 107 byte (vdd33_const) 的单位是 0.1V，有效取值范围是 [18, 36]；当 vdd33_const 处于无效范围 [0, 18) 或者 (36, 255) 时，使用默认值 3.3V 来优化 RF 电路工作状态。

函数定义：

```
uint16 system_adc_read(void)
```

参数：

无

返回：

TOUT 管脚 6 的输入电压，单位：1/1024 V

8. 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。

9. system_deep_sleep_set_option

功能：

设置下一次 deep-sleep 唤醒后的行为，如需调用此 API，必须在 system_deep_sleep 之前调用。如果用户不设置，默认为 system_deep_sleep_set_option(1) 的行为。

函数定义：

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

参数：

uint8 option ：

0 ：由 esp_init_data_default.bin(0~127byte) 的 byte 108 控制 deep-sleep 唤醒后的是否进行 RF_CAL；

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；

2 ：deep-sleep 唤醒后不进行 RF_CAL，这样电流较小；

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。

返回：

true ：成功

false ：失败

10. system_phy_set_rfoption

功能：

设置此次 ESP8266 deep-sleep 醒来，是否打开 RF。

注意：

- 本接口只允许在 user_rf_pre_init 中调用。
- 本接口与 system_deep_sleep_set_option 功能相似，system_deep_sleep_set_option 在 deep-sleep 前调用，本接口在 deep-sleep 醒来初始化时调用，以本接口设置为准。
- 调用本接口前，要求至少调用过一次 system_deep_sleep_set_option

函数定义：

```
void system_phy_set_rfoption(uint8 option)
```

参数：

uint8 option ：

system_phy_set_rfoption(0) ：由 esp_init_data_default.bin(0~127byte) 的 byte 108 控制 deep-sleep 醒来后的是否进行 RF_CAL；

system_phy_set_rfoption(1) ：deep-sleep 醒来后的初始化和上电一样，要作 RF_CAL，电流较大。

system_phy_set_rfoption(2) ：deep-sleep 醒来后的初始化，不作 RF_CAL，电流较小。

system_phy_set_rfoption(4) ：deep-sleep 醒来后的初始化，不打开 RF，和 modem-sleep 一样，电流最小，但是设备唤醒后无法发送和接收数据。

返回：

无

11. system_phy_set_max_tpw

功能：

设置 RF TX Power 最大值，单位：0.25dBm

函数定义：

```
void system_phy_set_max_tpw(uint8 max_tpw)
```

参数：

uint8 max_tpw ：RF Tx Power 的最大值，可参考 esp_init_data_default.bin（0～127byte）的第 34 byte（target_power_qdb_0）设置，单位：0.25dBm，参数范围 [0, 82]

返回：

无

12. system_phy_set_tpw_via_vdd33

功能：

根据改变的 VDD33 电压值，重新调整 RF TX Power，单位：1/1024 V

注意：

在 TOUT 管脚悬空的情况下，VDD33 电压值可通过 system_get_vdd33 测量获得。

在 TOUT 管脚接外部电路情况下，不可使用 system_get_vdd33 测量 VDD33 电压值。

函数定义：

```
void system_phy_set_tpw_via_vdd33(uint16 vdd33)
```

参数：

uint16 vdd33 ：重新测量的 VDD33 值，单位：1/1024V，有效值范围：[1900, 3300]

返回：

无

13. 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

14. 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

打印输出：

```
sig_rx a
```

15. 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。

16. 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 5.75（表示 1 个 RTC 周期为 5.75 微秒），

则实际时间为 10 x 5.75 = 57.5 微秒。

注意：

system_restart 时，系统时间归零，但是 RTC 时间仍然继续。但是如果外部硬件通过 EXT_RST 脚或者 CHIP_EN 脚，将芯片复位后（包括 deep-sleep 定时唤醒的情况），RTC 时钟会把复位。具体如下：

- 外部复位（EXT_RST）：RTC memory 不变，RTC timer 寄存器从零计数
- watchdog reset ：RTC memory 不变，RTC timer 寄存器不变
- system_restart ：RTC memory 不变，RTC timer 寄存器不变
- 电源上电 ：RTC memory 随机值，RTC timer 寄存器从零计数
- CHIP_EN 复位 ：RTC memory 随机值，RTC timer 寄存器从零计数

函数定义：

```
uint32 system_get_rtc_time(void)
```

参数：

无

返回：

RTC 时间

17. system_rtc_clock_cali_proc

功能：

查询 RTC 时钟周期.

注意：

RTC 时钟周期含有小数部分。

RTC 时钟周期会随温度变化发生偏移，因此 RTC 时钟适用于在精度可接受的范围内进行计时。

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（取 2 位小数并 * 100：（RTC_CAL*100）>> 12 ）

注意：

RTC 示例见附录.

18. 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

19. 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

20. 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

21. system_uart_de_swap

功能：

取消 UART0 转换，仍然使用原有 UART0，而不是将 MTCK、MTDO 作为 UART0。

函数定义：

```
void system_uart_de_swap (void)
```

参数：

无

返回：

无

22. 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)

23. 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。

24. 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。

25. 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。

26. system_get_flash_size_map

功能：

查询当前的 flash size 和 flash map。

flash map 对应编译时的选项，详细介绍请参考文档“2A-ESP8266__IOT_SDK_User_Manual”

结构体：

```
enum flash_size_map {
    FLASH_SIZE_4M_MAP_256_256 = 0,
    FLASH_SIZE_2M,
    FLASH_SIZE_8M_MAP_512_512,
    FLASH_SIZE_16M_MAP_512_512,
    FLASH_SIZE_32M_MAP_512_512,
    FLASH_SIZE_16M_MAP_1024_1024,
    FLASH_SIZE_32M_MAP_1024_1024
};
```

函数定义：

```
enum flash_size_map system_get_flash_size_map(void)
```

参数：

无

返回：

flash map

27. os_install_putc1

功能：

注册打印接口函数

函数定义：

```
void os_install_putc1(void(*p)(char c))
```

参数：

void(*p)(char c) – 打印函数指针

返回：

无

示例：

参考 UART.c 中的 os_install_putc1((void *)uart1_write_char) 将 printf 改为从 UART 1 打印。否则，printf 默认从 UART 0 打印。

28. os_delay_us

功能：

延时函数。最大值 65535 us

函数定义：

```
void os_delay_us(uint16 us)
```

参数：

uint16 us – 延时时间

返回：

无

29. os_putc

功能：

打印一个字符，默认从 UART0 打印

函数定义：

```
void os_putc(char c)
```

参数：

char c – 要打印的字符

返回：

无

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

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

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;
```

示例：

```
uint32 value;

uint8 *addr = (uint8 *)&value;

spi_flash_read(0x3E * SPI_FLASH_SEC_SIZE, (uint32 *)addr, 4);

printf("0x3E sec:%02x%02x%02x%02x\r\n", addr[0], addr[1], addr[2], addr[3]);
```

5. system_param_save_with_protect

功能：

使用带读写保护机制的方式，写入数据到 flash。flash 读写必须 4 字节对齐。

flash 读写保护机制：使用 3 个 sector（4KB 每 sector）保存 1 个 sector 的数据，sector 0 和 sector 1 互相为备份，交替保存数据，sector 2 作为 flag sector，指示最新的数据保存在 sector 0 还是 sector 1。

注意：

flash 读写保护机制的详细介绍，请参考文档“99A-SDK-Espressif IOT Flash RW Operation”
http://bbs.espressif.com/viewtopic.php?f=21&t=413

函数定义：

```
bool system_param_save_with_protect (
    uint16 start_sec,
    void *param,
    uint16 len
)
```

参数：

uint16 start_sec ：读写保护机制使用的 3 个 sector 的起始 sector 0 值。

例如，IOT_Demo 中可使用 0x3D000 开始的 3 个 sector（3 * 4 KB）建立读写保护机制，则参数 start_sec 传 0x3D 。

void *param ：写入数据的指针。

uint16 len ：数据长度，不能超过 1 个 sector 大小，即 4 * 1024

返回：

true，成功；

false，失败

6. system_param_load

功能：

读取使用读写保护机制的方式写入 flash 的数据。flash 读写必须 4 字节对齐。

flash 读写保护机制：使用 3 个 sector（ 4KB 每 sector ）保存 1 个 sector 的数据，sector 0 和 sector 1 互相为备份，交替保存数据，sector 2 作为 flag sector，指示最新的数据保存在 sector 0 还是 sector 1。

注意：

flash 读写保护机制的详细介绍，请参考文档“99A-SDK-Espressif IOT Flash RW Operation”
http://bbs.espressif.com/viewtopic.php?f=21&t=413

函数定义：

```
bool system_param_load (
    uint16 start_sec,
    uint16 offset,
    void *param,
    uint16 len
)
```

参数：

uint16 start_sec ：读写保护机制使用的 3 个 sector 的起始 sector 0 值，请勿填入 sector 1 或者 sector 2 的值。

例如，IOT_Demo 中可使用 0x3D000 开始的 3 个 sector（3 * 4 KB）建立读写保护机制，则参数 start_sec 传 0x3D，请勿传入 0x3E 或者 0x3F。

uint16 offset ：需读取数据，在 sector 中的偏移地址

void *param ：读取数据的指针。

uint16 len ：数据长度，读取不能超过 1 个 sector 大小，即 offset+len ≤ 4*1024

返回：

true，成功；

false，失败

3.4. WIFI 接口

wifi_station 系列接口以及 ESP8266 station 相关的设置、查询接口，请在 ESP8266 station 使能的情况下调用；

wifi_softap 系列接口以及 ESP8266 soft-AP 相关的设置、查询接口，请在 ESP8266 soft-AP 使能的情况下调用。

后文的 " flash 系统参数区" 位于 flash 的最后 16KB。

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_get_opmode_default

功能：

查询保存在 flash 中的 WiFi 工作模式设置

函数定义：

```
uint8 wifi_get_opmode_default (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

3. wifi_set_opmode

功能：

设置 WiFi 工作模式（station，soft-AP 或者 station+soft-AP），并保存到 flash。

默认为 soft-AP 模式

注意：

本设置如果与原设置不同，会更新保存到 flash 系统参数区。

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

4. wifi_set_opmode_current

功能：

设置 WiFi 工作模式（station，soft-AP 或者 station+soft-AP），不保存到 flash

函数定义：

```
bool wifi_set_opmode_current (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

5. wifi_station_get_config

功能：

查询 WiFi station 接口的当前配置参数。

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

6. wifi_station_get_config_default

功能：

查询 WiFi station 接口保存在 flash 中的配置参数。

函数定义：

```
bool wifi_station_get_config_default (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

7. wifi_station_set_config

功能：

设置 WiFi station 接口的配置参数，并保存到 flash

注意：

- 请在 ESP8266 station 使能的情况下，调用本接口。
- 如果 wifi_station_set_config 在 user_init 中调用，则 ESP8266 station 接口会在系统初始化完成后，自动连接 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）。
- 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。
- 本设置如果与原设置不同，会更新保存到 flash 系统参数区。

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

示例：

```
void ICACHE_FLASH_ATTR
user_set_station_config(void)
{
    char ssid[32] = SSID;
    char password[64] = PASSWORD;
    struct station_config stationConf;

    stationConf.bssid_set = 0;  //need not check MAC address of AP

    os_memcpy(&stationConf.ssid, ssid, 32);
    os_memcpy(&stationConf.password, password, 64);
    wifi_station_set_config(&stationConf);
}
void user_init(void)
{
    wifi_set_opmode(STATIONAP_MODE); //Set softAP + station mode
    user_set_station_config();
}
```

8. wifi_station_set_config_current

功能：

设置 WiFi station 接口的配置参数，不保存到 flash

注意：

- 请在 ESP8266 station 使能的情况下，调用本接口。
- 如果 wifi_station_set_config_current 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）。
- 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_current (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

9. wifi_station_connect

功能：

ESP8266 WiFi station 接口连接 AP

注意：

- 请勿在 user_init 中调用本接口，请在 ESP8266 station 使能并初始化完成后调用；
- 如果 ESP8266 已经连接某个 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

10. wifi_station_disconnect

功能：

ESP8266 WiFi station 接口从 AP 断开连接

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_disconnect (void)
```

参数：

无

返回：

true: 成功

false: 失败

11. 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
};
```

12. 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

13. 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
        ...
    }
}
```

14. 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。

本设置如果与原设置不同，会更新保存到 flash 系统参数区。

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

15. 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);
```

16. 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

17. wifi_station_get_current_ap_id

功能：

获取当前连接的 AP 保存记录 id 值。ESP8266 可记录每一个配置连接的 AP，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。

18. 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 。

19. 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），默认为自动连接。

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。

本设置如果与原设置不同，会更新保存到 flash 系统参数区。

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

20. wifi_station_dhcpc_start

功能：

开启 ESP8266 station DHCP client.

注意：

（1）DHCP 默认开启。

（2）DHCP 与静态 IP 功能（wifi_set_ip_info）互相影响，以最后设置的为准：

DHCP 开启，则静态 IP 失效；设置静态 IP，则关闭 DHCP。

函数定义：

```
bool wifi_station_dhcpc_start(void)
```

参数：

无

返回：

true：成功

false：失败

21. wifi_station_dhcpc_stop

功能：

关闭 ESP8266 station DHCP client.

注意：

（1）DHCP 默认开启。

（2）DHCP 与静态 IP 功能（wifi_set_ip_info）互相影响：

DHCP 开启，则静态 IP 失效；设置静态 IP，则 DHCP 关闭。

函数定义：

```
bool wifi_station_dhcpc_stop(void)
```

参数：

无

返回：

true：成功

false：失败

22. 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
};
```

23. wifi_station_set_reconnect_policy

功能：

设置 ESP8266 station 从 AP 断开后是否重连。默认重连。

注意：

建议在 user_init 中调用本接口；

函数定义：

```
bool wifi_station_set_reconnect_policy(bool set)
```

参数：

bool set – true，断开则重连；false，断开不重连

返回：

true：成功

false：失败

24. wifi_station_get_reconnect_policy

功能：

查询 ESP8266 station 从 AP 断开后是否重连。默认重连。

函数定义：

```
bool wifi_station_get_reconnect_policy(void)
```

参数：

无

返回：

true：断开则重连

false：断开不重连

25. wifi_softap_get_config

功能：

查询 ESP8266 WiFi soft-AP 接口的当前配置

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

26. wifi_softap_get_config_default

功能：

查询 ESP8266 WiFi soft-AP 接口保存在 flash 中的配置

函数定义：

```
bool wifi_softap_get_config_default(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

27. wifi_softap_set_config

功能：

设置 WiFi soft-AP 接口配置，并保存到 flash

注意：

- 请在 ESP8266 soft-AP 使能的情况下，调用本接口。
- 本设置如果与原设置不同，将更新保存到 flash 系统参数区。
- 因为 ESP8266 只有一个信道，因此 soft-AP + station 共存模式时，ESP8266 soft-AP 接口会自动调节信道与 ESP8266 station 一致，详细说明请参考附录。

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

28. wifi_softap_set_config_current

功能：

设置 WiFi soft-AP 接口配置，不保存到 flash

注意：

- 请在 ESP8266 soft-AP 使能的情况下，调用本接口。
- 因为 ESP8266 只有一个信道，因此 soft-AP + station 共存模式时，ESP8266 soft-AP 接口会自动调节信道与 ESP8266 station 一致，详细说明请参考附录。

函数定义：

```
bool wifi_softap_set_config_current (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

29. wifi_softap_get_station_num

功能：

获取 ESP8266 soft-AP 下连接的 station 个数

函数定义：

```
uint8 wifi_softap_get_station_num(void)
```

参数：

无

返回：

ESP8266 soft-AP 下连接的 station 个数

30. 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

注意：

本接口不支持获取静态 IP，仅支持在 DHCP 使能的情况下使用。

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

31. 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) {
    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){
    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
```

32. wifi_softap_dhcps_start

功能：

开启 ESP8266 soft-AP DHCP server.

注意：

- DHCP 默认开启。
- DHCP 与静态 IP 功能（wifi_set_ip_info）互相影响，以最后设置的为准：

  DHCP 开启，则静态 IP 失效；设置静态 IP，则关闭 DHCP。

函数定义：

```
bool wifi_softap_dhcps_start(void)
```

参数：

无

返回：

true：成功

false：失败

33. wifi_softap_dhcps_stop

功能：

关闭 ESP8266 soft-AP DHCP server。默认开启 DHCP。

函数定义：

```
bool wifi_softap_dhcps_stop(void)
```

参数：

无

返回：

true：成功

false：失败

34. wifi_softap_set_dhcps_lease

功能：

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

注意：

- 设置的 IP 分配范围必须与 ESP8266 soft-AP IP 在同一网段。
- 本接口必须在 ESP8266 soft-AP DHCP server 关闭（ wifi_softap_dhcps_stop ）的情况下设置。
- 本设置仅对下一次使能的 DHCP server 生效（ wifi_softap_dhcps_start ），如果 DHCP server 再次被关闭，则需要重新调用本接口设置 IP 范围；否则之后 DHCP server 重新使能，会使用默认的 IP 地址分配范围。

函数定义：

```
bool wifi_softap_set_dhcps_lease(struct dhcps_lease *please)
```

参数：

```
struct dhcps_lease {
    struct ip_addr start_ip;
    struct ip_addr end_ip;
};
```

返回：

true: 成功

false: 失败

示例：

```
void dhcps_lease_test(void)
{
    struct dhcps_lease dhcp_lease;
    const char* start_ip = "192.168.5.100";
    const char* end_ip = "192.168.5.105";
    dhcp_lease.start_ip.addr = ipaddr_addr(start_ip);
    dhcp_lease.end_ip.addr = ipaddr_addr(end_ip);
    wifi_softap_set_dhcps_lease(&dhcp_lease);
}
```

或者

```
void dhcps_lease_test(void)
{
    struct dhcps_lease dhcp_lease;
    IP4_ADDR(&dhcp_lease.start_ip, 192, 168, 5, 100);
    IP4_ADDR(&dhcp_lease.end_ip, 192, 168, 5, 105);
    wifi_softap_set_dhcps_lease(&dhcp_lease);
}
void user_init(void)
{
    struct ip_info info;
    wifi_set_opmode(STATIONAP_MODE); //Set softAP + station mode
    wifi_softap_dhcps_stop();

    IP4_ADDR(&info.ip, 192, 168, 5, 1);
    IP4_ADDR(&info.gw, 192, 168, 5, 1);
    IP4_ADDR(&info.netmask, 255, 255, 255, 0);
    wifi_set_ip_info(SOFTAP_IF, &info);
    dhcps_lease_test();
    wifi_softap_dhcps_start();
}
```

35. 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
};
```

36. wifi_softap_set_dhcps_offer_option

功能：

设置 ESP8266 soft-AP DHCP server 属性.

结构体：

```
enum dhcps_offer_option{
    OFFER_START = 0x00,
    OFFER_ROUTER = 0x01,
    OFFER_END
};
```

函数定义：

```
bool wifi_softap_set_dhcps_offer_option(uint8 level, void* optarg)
```

参数：

uint8 level - OFFER_ROUTER 设置 router 信息

void* optarg - bit0, 0 禁用 router 信息; bit0, 1 启用 router 信息; 默认为 1

返回：

true : 成功
false : 失败

示例：

```
uint8 mode = 0;
wifi_softap_set_dhcps_offer_option(OFFER_ROUTER, &mode);
```

37. 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
```

38. 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
};
```

39. 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
```

40. 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

41. 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_opmode(STATIONAP_MODE);
wifi_set_macaddr(SOFTAP_IF, sofap_mac);
wifi_set_macaddr(STATION_IF, sta_mac);
```

返回：

true: 成功

false: 失败

42. 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

43. 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

44. 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

45. wifi_set_event_handler_cb

功能：

注册 WiFi event 处理回调

函数定义：

```
void wifi_set_event_handler_cb(wifi_event_handler_cb_t cb)
```

参数：

wifi_event_handler_cb_t cb – 回调函数

返回：

无

示例：

```
void wifi_handle_event_cb(System_Event_t *evt)
{
    printf("event %x\n", evt->event);
    switch (evt->event) {
        case EVENT_STAMODE_CONNECTED:
              printf("connect to ssid %s, channel %d\n",
                           evt->event_info.connected.ssid,
                           evt->event_info.connected.channel);
              break;
        case EVENT_STAMODE_DISCONNECTED:
              printf("disconnect from ssid %s, reason %d\n",
                           evt->event_info.disconnected.ssid,
                           evt->event_info.disconnected.reason);
              break;
        case EVENT_STAMODE_AUTHMODE_CHANGE:
            printf("mode: %d -> %d\n",
                           evt->event_info.auth_change.old_mode,
                           evt->event_info.auth_change.new_mode);
            break;
        case EVENT_STAMODE_GOT_IP:
```

```
                printf("ip:" IPSTR ",mask:" IPSTR ",gw:" IPSTR,
                                IP2STR(&evt->event_info.got_ip.ip),
                                IP2STR(&evt->event_info.got_ip.mask),
                                IP2STR(&evt->event_info.got_ip.gw));
                printf("\n");
                break;
        case EVENT_SOFTAPMODE_STACONNECTED:
            printf("station: " MACSTR "join, AID = %d\n",
                    MAC2STR(evt->event_info.sta_connected.mac),
                    evt->event_info.sta_connected.aid);
            break;
         case EVENT_SOFTAPMODE_STADISCONNECTED:
             printf("station: " MACSTR "leave, AID = %d\n",
                    MAC2STR(evt->event_info.sta_disconnected.mac),
                    evt->event_info.sta_disconnected.aid);
            break;
        default:
                break;
    }
}
void user_init(void)
{
    // TODO: add your own code here....
    wifi_set_event_handler_cb(wifi_handle_event_cb);
}
```

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：

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_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：

（1）仅支持在 ESP8266 单 station 模式下，开启混杂模式

（2）混杂模式中，ESP8266 station 和 soft-AP 接口均失效

（3）若开启混杂模式，请先调用 wifi_station_disconnect 确保没有连接

（4）混杂模式中请勿调用其他 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

示例：

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

wifi_promiscuous_set_mac(ap_mac);
```

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

以下介绍 smart config 软件接口的说明，用户可向乐鑫申请详细介绍的文档。开启 smart config 功能前，请先确保目标 AP 已经开启。

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：

（1）仅支持在单 station 模式下调用本接口；

（2）smartconfig 过程中，ESP8266 station 和 soft-AP 失效；

（3）smartconfig_start 未完成之前不可重复执行 smartconfig_start ,请先调用 smartconfig_stop 结束本次快连。

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

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_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;
```

函数定义：

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

参数：

sc_callback_t cb ： smartconfig 状态发生改变时，进入回调函数。

传入回调函数的参数 status 表示 smartconfig 状态：

- 当 status 为 SC_STATUS_GETTING_SSID_PSWD 时，参数 void *pdata 为 sc_type * 类型的指针变量，表示此次配置是 AirKiss 还是 ESP-TOUCH；
- 当 status 为 SC_STATUS_LINK 时，参数 void *pdata 为 struct station_config 类型的指针变量；
- 当 status 为 SC_STATUS_LINK_OVER 时，参数 void *pdata 是移动端的 IP 地址的指针， 4 个字节。（仅支持在 ESPTOUCH 方式下，其他方式则为 NULL）
- 当 status 为其他状态时，参数 void *pdata 为 NULL

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 smartconfig_done(sc_status status, void *pdata)
{
    switch(status) {
        case SC_STATUS_WAIT:
            printf("SC_STATUS_WAIT\n");
            break;
        case SC_STATUS_FIND_CHANNEL:
            printf("SC_STATUS_FIND_CHANNEL\n");
            break;
        case SC_STATUS_GETTING_SSID_PSWD:
            printf("SC_STATUS_GETTING_SSID_PSWD\n");
            sc_type *type = pdata;
            if (*type == SC_TYPE_ESPTOUCH) {
                printf("SC_TYPE:SC_TYPE_ESPTOUCH\n");
            } else {
                printf("SC_TYPE:SC_TYPE_AIRKISS\n");
```

```
                }
                break;
            case SC_STATUS_LINK:
                printf("SC_STATUS_LINK\n");
                struct station_config *sta_conf = pdata;
                wifi_station_set_config(sta_conf);
                wifi_station_disconnect();
                     wifi_station_connect();
                break;
            case SC_STATUS_LINK_OVER:
                os_printf("SC_STATUS_LINK_OVER\n");
                    if (pdata != NULL) {
                    uint8 phone_ip[4] = {0};
                    memcpy(phone_ip, (uint8*)pdata, 4);
                    printf("Phone ip: %d.%d.%d.%d
    \n",phone_ip[0],phone_ip[1],phone_ip[2],phone_ip[3]);
                    }
                smartconfig_stop();
                break;
        }
    }
    smartconfig_start(smartconfig_done);
```

2. smartconfig_stop

功能：

关闭快连模式，释放 smartconfig_start 占用的内存。

注意：

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

若快连失败，调用本接口退出快连模式，释放占用的内存

函数定义：

```
bool smartconfig_stop(void)
```

参数：

无

返回：

true: 成功

false: 失败

3.8. cJSON 接口

1. cJSON_Parse

功能：

解析 cJSON 字符串

函数定义：

```
cJSON *cJSON_Parse (const char *value)
```

参数：

`const char *value` ： 传入的字符串

返回：

返回 cJSON 结构体

2. cJSON_Print

功能：

将 cJSON 结构转换为字符串

函数定义：

```
char *cJSON_Print (cJSON *item)
```

参数：

`cJSON *item` ： 传入 cJSON 结构体

返回：

字符串

3. cJSON_Delete

功能：

删除 cJSON 结构体，释放 cJSON 空间

函数定义：

```
void cJSON_Delete (cJSON *c)
```

参数：

`cJSON *c` ： 传入 cJSON

返回：

无

4. cJSON_GetArraySize

功能：

查询 cJSON 数组或对象的大小

函数定义：

```
int cJSON_GetArraySize (cJSON *array)
```

参数：

```
cJSON *array ： 传入 cJSON 数组
```

返回：

cJSON 数组或对象的大小

5. cJSON_GetArrayItem

功能：

以 index 方式获取数组或对象对应的项

函数定义：

```
cJSON *cJSON_GetArrayItem(cJSON *array,int item)
```

参数：

```
cJSON *array ： 传入 cJSON

int item ： 数组或对象对应的项
```

返回：

返回数组或对象中相应 index 的项。如果找不到，则返回 NULL

6. cJSON_GetObjectItem

功能：

以名称方式获取数组或对象对应的项

函数定义：

```
cJSON *cJSON_GetObjectItem (cJSON *object, const char *string)
```

参数：

```
cJSON *object ： 传入 cJSON

const char *string ： 数组或对象对应中的名称
```

返回：

返回数组或对象中相应名称的项。如果找不到，则返回 NULL

7. cJSON_CreateXXX

功能：

创建相应类型的 cJSON 项

函数定义：

```
cJSON *cJSON_CreateNull (void)    // 创建类型为空的 cJSON 结构
```

```
cJSON *cJSON_CreateTrue (void)   // 创建类型为 True 的 cJSON 结构
cJSON *cJSON_CreateFalse (void)  // 创建类型为 False 的 cJSON 结构
cJSON *cJSON_CreateBool (int b)  // 创建类型为 bool 的 cJSON 结构
cJSON *cJSON_CreateNumber (double num)   // 创建类型为双精度数的 cJSON 结构
cJSON *cJSON_CreateString (const char *string) // 创建类型为字符串的 cJSON 结构
cJSON *cJSON_CreateArray (void)   // 创建类型为数组的 cJSON 结构
cJSON *cJSON_CreateObject (void)  // 创建类型为 JSON 树的 cJSON 结构
```

返回：

cJSON

8. cJSON_CreateXXXArray

功能：

创建多个相应类型的数组

函数定义：

```
cJSON *cJSON_CreateIntArray(const int *numbers,int count)
cJSON *cJSON_CreateFloatArray(const float *numbers,int count)
cJSON *cJSON_CreateDoubleArray(const double *numbers,int count)
cJSON *cJSON_CreateStringArray(const char **strings,int count)
```

参数：

numbers ： 数据值

int count ： 创建数目

返回：

cJSON

9. cJSON_InitHooks

功能：

重定义 malloc，realloc，free 等

函数定义：

```
void cJSON_InitHooks (cJSON_Hooks* hooks)
```

参数：

cJSON_Hooks* hooks ： cJSON 回调函数

返回：

无

10. cJSON_AddItemToArray

功能：

将对象加到指定的数组，破坏当前 JSON

函数定义：

```
void cJSON_AddItemToArray (cJSON *array, cJSON *item)
```

参数：

`cJSON *array` ： 指定的数组

`cJSON *item` ： 待添加的对象

返回：

无

11. cJSON_AddItemReferenceToArray

功能：

将对象加到指定的数组，不破坏当前 JSON

函数定义：

```
void cJSON_AddItemReferenceToArray (cJSON *array, cJSON *item)
```

参数：

`cJSON *array` ： 指定的数组

`cJSON *item` ： 待添加的对象

返回：

无

12. cJSON_AddItemToObject

功能：

将对象加到指定的对象，破坏当前 JSON

函数定义：

```
void cJSON_AddItemToObject (cJSON *object, const char *string, cJSON *item)
void cJSON_AddItemToObjectCS (cJSON *object, const char *string, cJSON
*item)
```

参数：

`cJSON *object` ： 当前对象

`const char *string` ： 待添加的对象

`cJSON *item` ： 指定的对象

返回：

无

13. cJSON_AddItemReferenceToObject

功能：

将对象加到指定的对象，不破坏当前 JSON

函数定义：

```
void cJSON_AddItemReferenceToObject (
    cJSON *object,
    const char *string,
    cJSON *item)
```

参数：

cJSON *object ： 当前对象

const char *string ： 待添加的对象

cJSON *item ： 指定的对象

返回：

无

14. cJSON_DetachItemFromArray

功能：

从指定的数组中移除对象

函数定义：

```
cJSON *cJSON_DetachItemFromArray (cJSON *array, int which)
```

参数：

cJSON *array ： 指定的数组

int which ： 待移除的对象位置

返回：

cJSON

15. cJSON_DeleteItemFromArray

功能：

从指定的数组中删除对象

函数定义：

```
void cJSON_DeleteItemFromArray (cJSON *array, int which)
```

参数：

`cJSON *array` ： 指定的数组

`int which` ： 待删除的对象位置

返回：

无

16. cJSON_DetachItemFromObject

功能：

从指定的对象中移除对象

函数定义：

```
cJSON *cJSON_DetachItemFromObject (cJSON *object, const char *string)
```

参数：

`cJSON *object` ： 指定的对象

`const char *string` ： 待移除的对象

返回：

cJSON 结构

17. cJSON_DeleteItemFromObject

功能：

从指定的对象中删除对象

函数定义：

```
void cJSON_DetachItemFromObject (cJSON *object, const char *string)
```

参数：

`cJSON *object` ： 指定的对象

`const char *string` ： 待删除的对象

返回：

无

18. cJSON_InsertItemInArray

功能：

新增数组对象

函数定义：

```
void cJSON_InsertItemInArray (cJSON *array, int which, cJSON *newitem)
```

参数：

`cJSON *array` ：指定的数组

`int which` ：新增对象的数组位置

`cJSON *newitem` ：新增的对象

返回：

无

19. cJSON_ReplaceItemInArray

功能：

更新数组中的某对象

函数定义：

```
void cJSON_ReplaceItemInArray (cJSON *array, int which, cJSON *newitem)
```

参数：

`cJSON *array` ：指定的数组

`int which` ：待更新的对象在数组中的位置

`cJSON *newitem` ：新对象

返回：

无

20. cJSON_ReplaceItemInObject

功能：

更新对象中的某对象

函数定义：

```
void cJSON_ReplaceItemInObject (
    cJSON *object,
    const char *string,
    cJSON *newitem)
```

参数：

`cJSON *object` ：指定的对象

`const char *string` ：待更新的对象

`cJSON *newitem` ：新对象

返回：

无

21. cJSON_Duplicate

功能：

复制创建一个相同的新 cJSON

函数定义：

```
cJSON *cJSON_Duplicate (cJSON *item, int recurse)
```

参数：

cJSON *item ： 复制的对象

int recurse ： 复制创建的深度

返回：

cJSON

22. cJSON_ParseWithOpts

功能：

解析 cJSON

函数定义：

```
cJSON *cJSON_ParseWithOpts (
    const char *value,
    const char **return_parse_end,
    int require_null_terminated)
```

参数：

const char *value ： 输入字符串

const char **return_parse_end ： 解析字符串末尾地址

int require_null_terminated ： 终止解析的位置

返回：

cJSON

4. 参数结构体和宏定义

4.1. 定时器

```
typedef void os_timer_func_t(void *timer_arg);
typedef struct _os_timer_t {
    struct _os_timer_t     *timer_next;
    void                   *timer_handle;
    uint32                 timer_expire;
    uint32                 timer_period;
    os_timer_func_t        *timer_func;
    bool                   timer_repeat_flag;
    void                   *timer_arg;
} os_timer_t;
```

4.2. Wi-Fi 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;          // support 1 ~ 13
    uint8 authmode;         // Don't support AUTH_WEP in soft-AP mode
    uint8 ssid_hidden;      // default 0
    uint8 max_connection;   // default 4, max 4
    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. Wi-Fi event 结构体

```
enum {
    EVENT_STAMODE_CONNECTED = 0,
    EVENT_STAMODE_DISCONNECTED,
    EVENT_STAMODE_AUTHMODE_CHANGE,
```

```
        EVENT_STAMODE_GOT_IP,
        EVENT_SOFTAPMODE_STACONNECTED,
           EVENT_SOFTAPMODE_STADISCONNECTED,
        EVENT_MAX
    };

    enum {
           REASON_UNSPECIFIED              = 1,
           REASON_AUTH_EXPIRE              = 2,
           REASON_AUTH_LEAVE               = 3,
           REASON_ASSOC_EXPIRE             = 4,
           REASON_ASSOC_TOOMANY            = 5,
           REASON_NOT_AUTHED               = 6,
           REASON_NOT_ASSOCED              = 7,
           REASON_ASSOC_LEAVE              = 8,
           REASON_ASSOC_NOT_AUTHED         = 9,
           REASON_DISASSOC_PWRCAP_BAD      = 10,  /* 11h */
           REASON_DISASSOC_SUPCHAN_BAD     = 11,  /* 11h */
           REASON_IE_INVALID               = 13,  /* 11i */
           REASON_MIC_FAILURE              = 14,  /* 11i */
           REASON_4WAY_HANDSHAKE_TIMEOUT   = 15,  /* 11i */
           REASON_GROUP_KEY_UPDATE_TIMEOUT = 16,  /* 11i */
           REASON_IE_IN_4WAY_DIFFERS       = 17,  /* 11i */
           REASON_GROUP_CIPHER_INVALID     = 18,  /* 11i */
           REASON_PAIRWISE_CIPHER_INVALID  = 19,  /* 11i */
           REASON_AKMP_INVALID             = 20,  /* 11i */
           REASON_UNSUPP_RSN_IE_VERSION    = 21,  /* 11i */
           REASON_INVALID_RSN_IE_CAP       = 22,  /* 11i */
           REASON_802_1X_AUTH_FAILED       = 23,  /* 11i */
           REASON_CIPHER_SUITE_REJECTED    = 24,  /* 11i */

           REASON_BEACON_TIMEOUT           = 200,
           REASON_NO_AP_FOUND              = 201,
    };

    typedef struct {
           uint8 ssid[32];
           uint8 ssid_len;
           uint8 bssid[6];
```

```
        uint8 channel;
} Event_StaMode_Connected_t;

typedef struct {
        uint8 ssid[32];
        uint8 ssid_len;
        uint8 bssid[6];
        uint8 reason;
} Event_StaMode_Disconnected_t;

typedef struct {
        uint8 old_mode;
        uint8 new_mode;
} Event_StaMode_AuthMode_Change_t;

typedef struct {
        struct ip_addr ip;
        struct ip_addr mask;
        struct ip_addr gw;
} Event_StaMode_Got_IP_t;

typedef struct {
        uint8 mac[6];
        uint8 aid;
} Event_SoftAPMode_StaConnected_t;

typedef struct {
        uint8 mac[6];
        uint8 aid;
} Event_SoftAPMode_StaDisconnected_t;

typedef union {
        Event_StaMode_Connected_t               connected;
        Event_StaMode_Disconnected_t            disconnected;
        Event_StaMode_AuthMode_Change_t         auth_change;
        Event_StaMode_Got_IP_t                        got_ip;
        Event_SoftAPMode_StaConnected_t         sta_connected;
        Event_SoftAPMode_StaDisconnected_t      sta_disconnected;
} Event_Info_u;
```

```
typedef struct _esp_event {
    uint32 event;
    Event_Info_u event_info;
} System_Event_t;
```

5. 附录

5.1. 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。

```
#include "ets_sys.h"
#include "osapi.h"
#include "user_interface.h"

os_timer_t rtc_test_t;
#define  RTC_MAGIC  0x55aaaa55

typedef struct {
       uint64 time_acc;
       uint32 magic ;
       uint32 time_base;
}RTC_TIMER_DEMO;

void rtc_count()
{
    RTC_TIMER_DEMO rtc_time;
    static uint8 cnt = 0;
    system_rtc_mem_read(64, &rtc_time, sizeof(rtc_time));

    if(rtc_time.magic!=RTC_MAGIC){
       printf("rtc time init...\r\n");
       rtc_time.magic = RTC_MAGIC;
       rtc_time.time_acc= 0;
       rtc_time.time_base = system_get_rtc_time();
       printf("time base : %d \r\n",rtc_time.time_base);
    }
    printf("RTC time test : \r\n");

    uint32 rtc_t1,rtc_t2;
    uint32 st1,st2;
    uint32 cal1, cal2;
```

```
    rtc_t1 = system_get_rtc_time();
    st1 = system_get_time();
    cal1 = system_rtc_clock_cali_proc();
    os_delay_us(300);

    st2 = system_get_time();
    rtc_t2 = system_get_rtc_time();

    cal2 = system_rtc_clock_cali_proc();
    printf(" rtc_t2-t1 : %d \r\n",rtc_t2-rtc_t1);
    printf(" st2-t2 :  %d  \r\n",st2-st1);
    printf("cal 1  : %d.%d  \r\n", ((cal1*1000)>>12)/1000,
((cal1*1000)>>12)%1000 );
    printf("cal 2  : %d.%d \r\n",((cal2*1000)>>12)/1000,
((cal2*1000)>>12)%1000 );
    printf("==================\r\n\r\n");
    rtc_time.time_acc += (  ((uint64)(rtc_t2 - rtc_time.time_base))  *
( (uint64)((cal2*1000)>>12))  ) ;
    printf("rtc time acc  : %lld \r\n",rtc_time.time_acc);
    printf("power on time :  %lld  us\r\n", rtc_time.time_acc/1000);
    printf("power on time :  %lld.%02lld  S\r\n", (rtc_time.time_acc/10000000)/
100, (rtc_time.time_acc/10000000)%100);

    rtc_time.time_base = rtc_t2;
    system_rtc_mem_write(64, &rtc_time, sizeof(rtc_time));
    printf("-----------------------\r\n");

    if(5== (cnt++)){
       printf("system restart\r\n");
       system_restart();
    }else{
       printf("continue ...\r\n");
    }
}
void user_init(void)
{
    rtc_count();
    printf("SDK version:%s\n", system_get_sdk_version());
```

```
        os_timer_disarm(&rtc_test_t);
        os_timer_setfn(&rtc_test_t,rtc_count,NULL);
        os_timer_arm(&rtc_test_t,10000,1);
    }
```

5.2. 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 不可完全接收的包，它包含：
  - 整个包的长度

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

```
    struct RxControl {
        signed rssi:8;              // signal intensity of packet
        unsigned rate:4;
        unsigned is_group:1;
        unsigned:1;
        unsigned sig_mode:2;        // 0:is 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
    };

    struct sniffer_buf2{
        struct RxControl rx_ctrl;
        u8 buf[112];
        u16 cnt;
        u16 len;  //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，分为三种情况：len = 128 ， len 为 10 的整数倍，len = 12:

LEN == 128 的情况

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

LEN 为 10 整数倍的情况

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

LEN == 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：

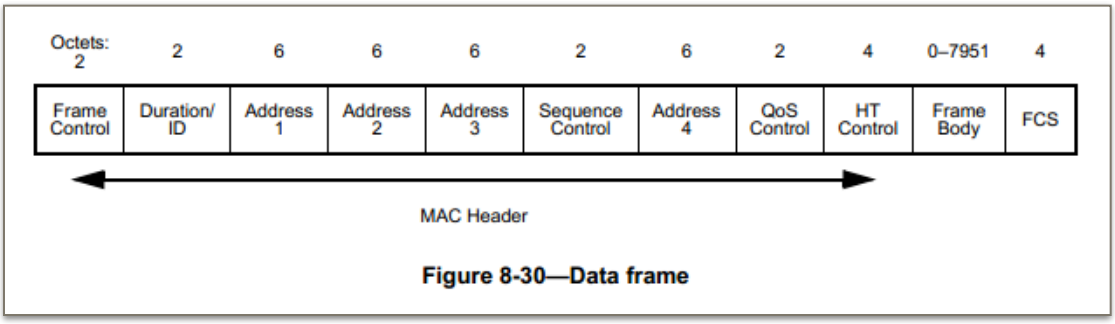

Figure 8-30—Data frame

- 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。

5.3. ESP8266 soft-AP 和 station 信道定义

虽然 ESP8266 支持 soft-AP + station 共存模式，但是它实际只有一个硬件信道。因此在 soft-AP + station 模式时, ESP8266 soft-AP 会动态调整信道值与 ESP8266 station 一致。
这个限制会导致 soft-AP + station 模式时一些行为上的不便，用户需要注意。例如:

情况一

（1）如果 ESP8266 station 连接到一个路由 (假设路由信道号为 6)
（2）通过接口 `wifi_softap_set_config` 设置 ESP8266 soft-AP
（3）如果设置值合法有效， API 将返回 true ，但信道号设置后仍然会被 ESP8266 自动调节成与 ESP8266 station 接口一致，在这个例子里也就是信道号为 6。因为 ESP8266 在硬件上就只有一个信道。

情况二

（1）先使用接口 `wifi_softap_set_config` 设置了 ESP8266 soft-AP(例如信道号为 5)
（2）其他 station 连接到 ESP8266 soft-AP
（3）将 ESP8266 station 连接到路由 (假设路由信道号为 6)
（4） ESP8266 soft-AP 会调整信道号与 ESP8266 station 一致（信道 6）
（5）由于信道改变，之前连接到 ESP8266 soft-AP 的 station 的 WiFi 连接会断开。