ESP8266 RTOS SDK 编程手册

Version 0.9.9

Espressif Systems IOT Team

免责申明和版权公告

本文中的信息，包括供参考的URL地址，如有变更，恕不另行通知。

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

Wi-Fi联盟成员标志归Wi-Fi联盟所有。

文中提到的所有商标名称、商标和注册商标均属其各自所有者的财产，特此声明。

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。

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；

3. 应用程序接口 (APIs)

3.1. 定时器

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
- 定时器无法保证定时函数立即执行，系统按照优先级高低执行

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。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

2. system_restart

功能：

系统重启

函数定义：

```
void system_restart(void)
```

参数：

无

返回：

无

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。

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

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：

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

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

3.4. WIFI 接口

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_set_opmode

功能:

设置 WiFi 工作模式（station，soft-AP or station+soft-AP），并保存到 flash 系统参数区。默认为 soft-AP 模式

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，并保存到 flash 系统参数区。

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。

（3）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

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
        ...
    }
}
```

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

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

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
```

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)
```

3.5. 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，请先关闭自动连接 `wifi_station_set_auto_connect(0)`

（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

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

4. 结构体定义

4.1. 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;
```

4.2. 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;          // 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);
```

5. 附录 - 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 (MCS_0 到 MCS_7)
- 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
    };
```

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：

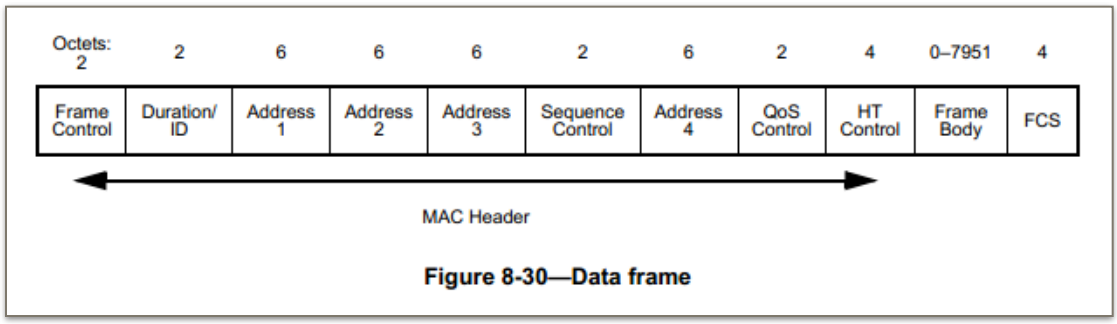

Figure 8-30—Data frame

- 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。