



ESP8266 SDK 编程手册

Version 1.4

Espressif Systems IOT Team

Copyright © 2015

免责声明和版权公告

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。

版权归© 2015 乐鑫信息科技（上海）有限公司所有。保留所有权利。



Table of Content

2.	前言.....	14
2.	概述.....	15
3.	应用程序接口 (APIs)	16
3.1.	软件定时器.....	16
1.	os_timer_arm.....	16
2.	os_timer_disarm	17
3.	os_timer_setfn	17
4.	system_timer_reinit	17
5.	os_timer_arm_us	18
3.2.	硬件中断定时器	18
1.	hw_timer_init	18
2.	hw_timer_arm	19
3.	hw_timer_set_func.....	19
4.	硬件定时器示例	20
3.3.	系统接口	20
1.	system_get_sdk_version.....	20
2.	system_restore	21
3.	system_restart	21
4.	system_init_done_cb	21
5.	system_get_chip_id	22
6.	system_get_vdd33	22
7.	system_adc_read	23
8.	system_deep_sleep	23
9.	system_deep_sleep_set_option	24
10.	system_phy_set_rfoption	24
11.	system_phy_set_powerup_option.....	25
12.	system_phy_set_max_tpw.....	25
13.	system_phy_set_tpw_via_vdd33.....	26
14.	system_set_os_print	26
15.	system_print_meminfo.....	26



16. system_get_free_heap_size	27
17. system_os_task	27
18. system_os_post	28
19. system_get_time	29
20. system_get_rtc_time	29
21. system_rtc_clock_calib_proc	30
22. system_rtc_mem_write	30
23. system_rtc_mem_read	31
24. system_uart_swap	31
25. system_uart_de_swap	32
26. system_get_boot_version	32
27. system_get_userbin_addr	32
28. system_get_boot_mode	32
29. system_restart_enhance	33
30. system_update_cpu_freq	33
31. system_get_cpu_freq	34
32. system_get_flash_size_map	34
33. system_get_rst_info	35
34. system_soft_wdt_stop	36
35. system_soft_wdt_restart	36
36. system_soft_wdt_feed	36
37. system_show_malloc	37
38. os_memset	37
39. os_memcpy	38
40. os_strlen	38
41. os_printf	39
42. os_bzero	39
43. os_delay_us	39
44. os_install_putc1	40
3.4. SPI Flash 接口	41
1. spi_flash_get_id	41
2. spi_flash_erase_sector	41
3. spi_flash_write	41



4.	spi_flash_read.....	42
5.	system_param_save_with_protect	43
6.	system_param_load	43
7.	spi_flash_set_read_func.....	44
3.5.	Wi-Fi 接口	46
1.	wifi_get_opmode	46
2.	wifi_get_opmode_default	46
3.	wifi_set_opmode.....	46
4.	wifi_set_opmode_current.....	47
5.	wifi_station_get_config.....	47
6.	wifi_station_get_config_default.....	48
7.	wifi_station_set_config	48
8.	wifi_station_set_config_current	49
9.	wifi_station_connect	50
10.	wifi_station_disconnect.....	50
11.	wifi_station_get_connect_status	50
12.	wifi_station_scan	51
13.	scan_done_cb_t.....	52
14.	wifi_station_ap_number_set.....	52
15.	wifi_station_get_ap_info.....	53
16.	wifi_station_ap_change.....	53
17.	wifi_station_get_current_ap_id.....	53
18.	wifi_station_get_auto_connect.....	54
19.	wifi_station_set_auto_connect	54
20.	wifi_station_dhcpc_start	54
21.	wifi_station_dhcpc_stop	55
22.	wifi_station_dhcpc_status.....	55
23.	wifi_station_dhcpc_set_maxtry	56
24.	wifi_station_set_reconnect_policy	56
25.	wifi_station_get_rssi.....	56
26.	wifi_station_set_hostname	57
27.	wifi_station_get_hostname.....	57
28.	wifi_softap_get_config	57



29. wifi_softap_get_config_default	58
30. wifi_softap_set_config	58
31. wifi_softap_set_config_current	59
32. wifi_softap_get_station_num	59
33. wifi_softap_get_station_info	59
34. wifi_softap_free_station_info	60
35. wifi_softap_dhcps_start	60
36. wifi_softap_dhcps_stop	61
37. wifi_softap_set_dhcps_lease	61
38. wifi_softap_get_dhcps_lease	63
39. wifi_softap_set_dhcps_lease_time	63
40. wifi_softap_get_dhcps_lease_time	63
41. wifi_softap_reset_dhcps_lease_time	64
42. wifi_softap_dhcps_status	64
43. wifi_softap_set_dhcps_offer_option	64
44. wifi_set_phy_mode	65
45. wifi_get_phy_mode	65
46. wifi_get_ip_info	66
47. wifi_set_ip_info	66
48. wifi_set_macaddr	67
49. wifi_get_macaddr	68
50. wifi_set_sleep_type	69
51. wifi_get_sleep_type	69
52. wifi_status_led_install	69
53. wifi_status_led_uninstall	70
54. wifi_set_broadcast_if	70
55. wifi_get_broadcast_if	71
56. wifi_set_event_handler_cb	71
57. wifi_wps_enable	73
58. wifi_wps_disable	73
59. wifi_wps_start	74
60. wifi_set_wps_cb	74
61. wifi_register_send_pkt_freedom_cb	75



62. wifi_unregister_send_pkt_freedom_cb	75
63. wifi_send_pkt_freedom	75
64. wifi_rfid_locp_rcv_open	76
65. wifi_rfid_locp_rcv_close	76
66. wifi_register_rfid_locp_rcv_cb	77
67. wifi_unregister_rfid_locp_rcv_cb	77
3.6. Rate Control 接口	78
1. wifi_set_user_fixed_rate	78
2. wifi_get_user_fixed_rate	79
3. wifi_set_user_sup_rate	79
4. wifi_set_user_rate_limit	80
5. wifi_set_user_limit_rate_mask	81
6. wifi_get_user_limit_rate_mask	82
3.7. 强制休眠接口	83
1. wifi_fpm_open	83
2. wifi_fpm_close	83
3. wifi_fpm_do_wakeup	83
4. wifi_fpm_do_sleep	84
5. wifi_fpm_set_sleep_type	84
6. wifi_fpm_get_sleep_type	85
7. 示例代码	85
3.8. ESP-NOW 接口	87
1. esp_now_init	87
2. esp_now_deinit	87
3. esp_now_register_rcv_cb	88
4. esp_now_unregister_rcv_cb	88
5. esp_now_register_send_cb	88
6. esp_now_unregister_send_cb	89
7. esp_now_send	89
8. esp_now_add_peer	90
9. esp_now_del_peer	91
10. esp_now_set_self_role	91
11. esp_now_get_self_role	91



12. esp_now_set_peer_role	92
13. esp_now_get_peer_role	92
14. esp_now_set_peer_key	93
15. esp_now_get_peer_key	93
16. esp_now_set_peer_channel	93
17. esp_now_get_peer_channel	94
18. esp_now_is_peer_exist	94
19. esp_now_fetch_peer	95
20. esp_now_get_cnt_info	95
21. esp_now_set_kok	95
3.9. 云端升级 (FOTA) 接口	97
1. system_upgrade_userbin_check	97
2. system_upgrade_flag_set	97
3. system_upgrade_flag_check	97
4. system_upgrade_start	98
5. system_upgrade_reboot	98
3.10. Sniffer 相关接口	99
1. wifi_promiscuous_enable	99
2. wifi_promiscuous_set_mac	99
3. wifi_set_promiscuous_rx_cb	100
4. wifi_get_channel	100
5. wifi_set_channel	100
3.11. smart config 接口	101
1. smartconfig_start	101
2. smartconfig_stop	103
3. smartconfig_set_type	103
3.12. SNTP 接口	104
1. sntp_setserver	104
2. sntp_getserver	104
3. sntp_setservername	104
4. sntp_getservername	105
5. sntp_init	105
6. sntp_stop	105



7.	sntp_get_current_timestamp	105
8.	sntp_get_real_time.....	106
9.	SNTP 示例	107
4.	TCP/UDP 接口	109
4.1.	通用接口	109
1.	espconn_delete	109
2.	espconn_gethostbyname	109
3.	espconn_port	110
4.	espconn_regist_sentcb	110
5.	espconn_regist_recvcb	111
6.	espconn_sent_callback	111
7.	espconn_recv_callback.....	112
8.	espconn_send	112
9.	espconn_sent	113
4.2.	TCP 接口	114
1.	espconn_accept	114
2.	espconn_regist_time.....	114
3.	espconn_get_connection_info	115
4.	espconn_connect	115
5.	espconn_regist_connectcb	116
6.	espconn_connect_callback.....	116
7.	espconn_set_opt	116
8.	espconn_clear_opt	117
9.	espconn_set_keepalive	118
10.	espconn_get_keepalive	119
11.	espconn_reconnect_callback.....	120
12.	espconn_regist_reconcb	120
13.	espconn_disconnect.....	121
14.	espconn_regist_disconcb	121
15.	espconn_regist_write_finish	122
16.	espconn_tcp_get_max_con.....	123
17.	espconn_tcp_set_max_con	123
18.	espconn_tcp_get_max_con_allow	123



19. espconn_tcp_set_max_con_allow	123
20. espconn_recv_hold.....	124
21. espconn_recv_unhold.....	124
22. espconn_secure_accept.....	125
23. espconn_secure_set_size	125
24. espconn_secure_get_size.....	126
25. espconn_secure_connect.....	126
26. espconn_secure_send.....	127
27. espconn_secure_sent.....	127
28. espconn_secure_disconnect	128
29. espconn_secure_ca_disable.....	128
30. espconn_secure_ca_enable.....	129
31. espconn_secure_cert_req_enable.....	129
32. espconn_secure_cert_req_disable.....	130
33. espconn_secure_set_default_certificate.....	130
34. espconn_secure_set_default_private_key	131
4.3. UDP 接口	132
1. espconn_create.....	132
2. espconn_sendto.....	132
3. espconn_igmp_join.....	133
3. espconn_igmp_leave	133
4. espconn_dns_setserver	133
4.4. mDNS 接口	135
1. espconn_mdns_init.....	135
2. espconn_mdns_close.....	136
3. espconn_mdns_server_register	136
4. espconn_mdns_server_unregister.....	136
5. espconn_mdns_get_servername	137
6. espconn_mdns_set_servername	137
7. espconn_mdns_set_hostname	137
8. espconn_mdns_get_hostname.....	137
9. espconn_mdns_disable	138
10. espconn_mdns_enable.....	138



5.	应用相关接口	139
5.1.	AT 接口	139
1.	at_response_ok	139
2.	at_response_error	139
3.	at_cmd_array_regist	139
4.	at_get_next_int_dec	140
5.	at_data_str_copy	140
6.	at_init	141
7.	at_port_print	141
8.	at_set_custom_info	141
9.	at_enter_special_state	142
10.	at_leave_special_state	142
11.	at_get_version	142
12.	at_register_uart_rx_intr	143
13.	at_response	143
14.	at_register_response_func	144
5.2.	JSON 接口	145
1.	jsonparse_setup	145
2.	jsonparse_next	145
3.	jsonparse_copy_value	145
4.	jsonparse_get_value_as_int	146
5.	jsonparse_get_value_as_long	146
6.	jsonparse_get_len	146
7.	jsonparse_get_value_as_type	147
8.	jsonparse_strcmp_value	147
9.	jsontree_set_up	147
10.	jsontree_reset	148
11.	jsontree_path_name	148
12.	jsontree_write_int	149
13.	jsontree_write_int_array	149
14.	jsontree_write_string	149
15.	jsontree_print_next	150
16.	jsontree_find_next	150



6.	参数结构体和宏定义	151
6.1.	定时器	151
6.2.	WiFi 参数.....	151
1.	station 参数	151
2.	soft-AP 参数	151
3.	scan 参数	152
4.	WiFi event 结构体	152
5.	smart config 结构体.....	155
6.3.	json 相关结构体	155
1.	json 结构体.....	155
2.	json 宏定义.....	157
6.4.	espconn 参数.....	157
1.	回调函数	157
2.	espconn	157
6.5.	中断相关宏定义.....	159
7.	外围设备驱动接口	162
7.1.	GPIO 接口	162
1.	PIN 相关宏定义	162
2.	gpio_output_set.....	162
3.	GPIO 输入输出相关宏	163
4.	GPIO 中断	163
5.	gpio_pin_intr_state_set	163
6.	GPIO 中断处理函数.....	164
7.2.	UART 接口	165
1.	uart_init.....	165
2.	uart0_tx_buffer.....	165
3.	uart0_rx_intr_handler	166
7.3.	I2C Master 接口	167
1.	i2c_master_gpio_init	167
2.	i2c_master_init.....	167
3.	i2c_master_start	167
4.	i2c_master_stop	168
5.	i2c_master_send_ack	168



6.	i2c_master_send_nack	168
7.	i2c_master_checkAck.....	169
8.	i2c_master_readByte	169
9.	i2c_master_writeByte.....	169
7.4.	PWM 接口	170
1.	pwm_init.....	170
2.	pwm_start	170
3.	pwm_set_duty	171
4.	pwm_get_duty.....	171
5.	pwm_set_period	172
6.	pwm_get_period.....	172
7.	get_pwm_version	172
8.	附录.....	173
8.1.	ESPCONN 编程	173
1.	TCP Client 模式	173
2.	TCP Server 模式	173
3.	espconn callback	174
8.2.	RTC APIs 使用示例	175
8.3.	Sniffer 结构体说明.....	177
8.4.	ESP8266 soft-AP 和 station 信道定义	180
8.5.	ESP8266 启动信息说明	181



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_v1.1.0 及之后版本，请在 `user_main.c` 增加 `void user_rf_pre_init(void)`，可参考 IOT_Demo 的 `user_main.c`。用户可在 `user_rf_pre_init` 中配置 RF 初始化，相关 RF 设置接口为 `system_phy_set_rfoption`，或者在 deep-sleep 前调用 `system_deep_sleep_set_option`。如果设置为 RF 不打开，则 ESP8266 station 及 soft-AP 均无法使用，请勿调用 Wi-Fi 相关接口及网络功能。

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。

注意

- 非 OS SDK 中，由于是单线程，任何 task 都不能长期占用 CPU：
 - ▶ 如果一个 task 占用 CPU 不退出，将导致看门狗的喂狗函数无法执行，系统重启；
 - ▶ 如果关闭中断，请勿占用 CPU 超过 10 微秒；如果不关闭中断，建议不超过 500 毫秒；
- 建议使用定时器 (timer) 实现周期性的查询功能，如需在定时器的执行函数中调用 `os_delay_us` 或者 `while`、`for` 等函数进行延时或循环操作，占用时间请勿超过 15 毫秒；
- 非 OS SDK 在中断处理函数中，请勿使用任何 `ICACHE_FLASH_ATTR` 定义的函数；
- 建议使用 RTOS SDK，OS 会调度不同 task，每个 task 编程可认为独占 CPU；
- 内存必须 4 字节对齐进行读写，请勿直接进行指针转换，例如，语句 `float temp = *((float*)data);` 可能引起异常，建议使用 `os_memcpy`。
- 如需在中断处理函数中打印，请使用 `os_printf_plus`。



3. 应用程序接口 (APIs)

3.1. 软件定时器

以下软件定时器接口位于 `/esp_iot_sdk/include/osapi.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。

如果需要精确的定时，例如，周期性操作某 GPIO，请使用硬件中断定时器，具体可参考 `hw_timer.c`，硬件定时器的执行函数在中断里被执行。

注意：

- 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

如未调用 `system_timer_reinit`，最大可输入 `0xFFFFFFFF`

如调用了 `system_timer_reinit`，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。

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. `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

5. os_timer_arm_us

功能：

使能微秒级定时器，

注意：

1. 请定义 `USE_US_TIMER`
2. 请在 `user_init` 起始第一句，先调用 `system_timer_reinit`

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 `0xFFFFFFFF`

`bool repeat_flag` : 定时器是否重复

返回：

无

3.2. 硬件中断定时器

以下硬件中断定时器接口位于 `/esp_iot_sdk/examples/driver_lib/hw_timer.c`。用户可根据 `driver_lib` 文件夹下的“`readme.txt`”文件使用。

注意：

- 如果使用 NMI 中断源，且为自动填充的定时器，调用 `hw_timer_arm` 时参数 `val` 必须大于 100
- 如果使用 NMI 中断源，那么该定时器将为最高优先级，可打断其他 ISR
- 如果使用 FRC1 中断源，那么该定时器无法打断其他 ISR
- `hw_timer.c` 的接口不能跟 PWM 驱动接口函数同时使用，因为二者共用了同一个硬件定时器。

1. hw_timer_init

功能：

初始化硬件 ISR 定时器



函数定义：

```
void hw_timer_init (  
    FRC1_TIMER_SOURCE_TYPE source_type,  
    u8 req  
)
```

参数：

FRC1_TIMER_SOURCE_TYPE source_type : 定时器的 ISR 源

FRC1_SOURCE, 使用 FRC1 中断源

NMI_SOURCE, 使用 NMI 中断源

u8 req : 0, 不自动填装

1, 自动填装

返回：

无

2. hw_timer_arm

功能：

使能硬件中断定时器

函数定义：

```
void hw_timer_arm (uint32 val)
```

参数：

uint32 val : 定时时间

- 自动填装模式：
 - 使用 FRC1 中断源 (**FRC1_SOURCE**)，取值范围 : 50 ~ 0x7ffffff;
 - 使用 NMI 中断源 (**NMI_SOURCE**)，取值范围 : 100 ~ 0x7ffffff;
- 非自动填装模式，取值范围 : 10 ~ 0x7ffffff;

返回：

无

3. hw_timer_set_func

功能：

设置定时器回调函数。使用定时器，必须设置回调函数。

函数定义：

```
void hw_timer_set_func (void (* user_hw_timer_cb_set)(void) )
```



参数:

`void (* user_hw_timer_cb_set)(void)` : 定时器回调函数

返回:

无

4. 硬件定时器示例

```
#define REG_READ(_r)      (*(volatile uint32 *) (_r))
#define WDEV_NOW()        REG_READ(0x3ff20c00)
uint32 tick_now2 = 0;
void hw_test_timer_cb(void)
{
    static uint16 j = 0;
    j++;

    if( (WDEV_NOW() - tick_now2) >= 1000000 )
    {
        static u32 idx = 1;
        tick_now2 = WDEV_NOW();
        os_printf("b%u:%d\n", idx++, j);
        j = 0;
    }
}

void ICACHE_FLASH_ATTR user_init(void)
{
    hw_timer_init(FRC1_SOURCE, 1);
    hw_timer_set_func(hw_test_timer_cb);
    hw_timer_arm(100);
}
```

3.3. 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`，以及 `#define AP_CACHE` 记录的 AP 信息。

注意：

恢复出厂设置后，请务必重新启动（`system_restart`），再正常使用。

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_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_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。

注意：

- 硬件需要将 `XPB_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。

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



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。默认 option 为 1。

函数定义：

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

参数：

`uint8 option`：设置下一次 deep-sleep 唤醒后的行为。

0：由 `esp_init_data_default.bin` (0~127byte) 的 byte 108 和 deep-sleep 的次数 (`deep_sleep_number`，上电时初始化为 0) 共同控制 deep-sleep 唤醒后的行为，

若 `deep_sleep_number < byte 108`，则 deep-sleep 唤醒后不进行任何 RF_CAL，初始电流较小；

若 `deep_sleep_number = byte 108`，则 deep-sleep 唤醒后的行为与上电的行为一致，且将 `deep_sleep_number` 归零；

1：deep-sleep 唤醒后的行为与上电的行为一致；

2：deep-sleep 唤醒后不进行 RF_CAL，初始电流较小；

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`：



0 : 由 `esp_init_data_default.bin` (0~127byte) 的 byte 108 和 deep-sleep 的次数 (deep_sleep_number, 上电时初始化为 0) 共同控制 deep-sleep 唤醒后的行为,
若 deep_sleep_number < byte 108, 则 deep-sleep 唤醒后不进行任何 RF_CAL, 初始电流较小;
若 deep_sleep_number = byte 108, 则 deep-sleep 唤醒后的行为与上电的行为一致, 且将 deep_sleep_number 归零;
1 : deep-sleep 唤醒后的行为与上电的行为一致;
2 : deep-sleep 唤醒后不进行 RF_CAL, 初始电流较小;
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:

无

11. system_phy_set_powerup_option

功能:

设置上电时 RF 初始化的行为, 默认为 option 0。

函数定义:

```
void system_phy_set_powerup_option(uint8 option)
```

参数:

uint8 option : power up 时, RF 初始化的行为

0 : 由 `esp_init_data_default.bin`(0~127byte) byte 114 控制 RF 初始化行为, 可参考文档 “2A-ESP8266_IOT_SDK_User_Manual_v1.4” 附录一章。

1 : RF 初始化仅做 VDD33 和 TX power CAL, 耗时约 18 ms, 初始电流较小。

2 : RF 初始化仅做 VDD33 校准, 耗时约 2 ms, 初始电流最小。

3 : RF 初始化进行全部 RF CAL, 耗时约 200 ms, 初始电流较大。

返回:

无

12. 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



13. 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

14. 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` : 0, 打印功能关; 1, 打印功能开

默认值：

打印功能开

返回：

无

15. 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



16. 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

17. system_os_task

功能：

创建系统任务，最多支持创建 3 个任务，优先级分别为：0/1/2

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，可为 0/1/2；0 为最低优先级。这表示最多只支持建立 3 个任务

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_PRI0_0,testQueue,TEST_QUEUE_LEN);
}
```

18. 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_PRI0_0, SIG_RX, 'a');
}
```

打印输出：

```
sig_rx a
```



19. 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。

20. 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 $10 \times 5.75 = 57.5$ 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



21. 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 时钟周期会随温度或电源电压变化发生偏移，因此 RTC 时钟适用于在精度可接受的范围内进行计时，建议最多每分钟调用一次即可。

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

示例：

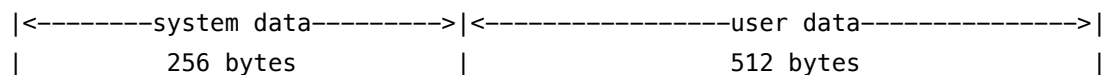
```
os_printf("clk cal : %d \r\n",system_rtc_clock_cali_proc())>>12);
```

详细 RTC 示例见附录。

22. 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。



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。



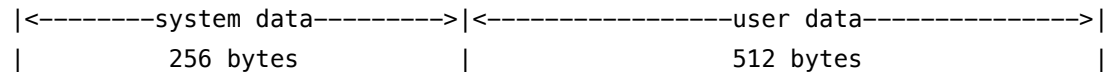
返回：

true: 成功
false: 失败

23. 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。



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

24. system_uart_swap

功能：

UART0 转换。将 MTCK 作为 UART0 RX，MTD0 作为 UART0 TX。硬件上也从 MTD0(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



25. 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

26. 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](#))

27. 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。

28. 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。

29. 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。

30. system_update_cpu_freq

功能：

设置 CPU 频率。默认为 80MHz。

注意：



系统总线时钟频率始终为 80MHz，不受 CPU 频率切换的影响。UART、SPI 等外设频率由系统总线时钟分频而来，因此也不受 CPU 频率切换的影响。

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
```

31. 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。

32. 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

33. 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
    REANSON_EXT_SYS_RST      = 6,    // external system reset
};
```

```
struct rst_info {
    uint32 reason;    // enum rst_reason
    uint32 exccause;
    uint32 epc1;    // the address that error occurred
    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。

34. system_soft_wdt_stop

功能：

关闭软件看门狗

注意：

请勿将软件看门狗关闭太长时间（小于 5 秒），否则将触发硬件看门狗复位

函数定义：

```
void system_soft_wdt_stop(void)
```

参数：

无

返回：

无

35. system_soft_wdt_restart

功能：

重启软件看门狗

注意：

仅支持在软件看门狗关闭（`system_soft_wdt_stop`）的情况下，调用本接口。

函数定义：

```
void system_soft_wdt_restart(void)
```

参数：

无

返回：

无

36. system_soft_wdt_feed

功能：

喂软件看门狗

注意：



仅支持在软件看门狗开启的情况下，调用本接口。

函数定义：

```
void system_soft_wdt_feed(void)
```

参数：

无

返回：

无

37. system_show_malloc

功能：

打印目前所分配的堆空间所有内存块，包括分配该内存块的文件名、行号和分配大小。在怀疑有内存泄露时，可以调用本接口查看当前内存状态。

注意：

- 在 user_config.h 定义 `#define MEMLEAK_DEBUG` 后，可调用本接口查证内存泄露的问题。
- 泄露的内存一般在打印结果中，但打印结果中的内存不保证一定是泄露的内存。
- 本接口仅用于调试，无法确保使用本接口后，程序能继续正常执行，因此请勿在正常运行情况下，调用本接口。

函数定义：

```
void system_show_malloc(void)
```

参数：

无

返回：

无

38. os_memset

功能：

封装 C 语言函数，在一段内存块中填充某个给定值。

函数定义：

```
os_memset(void *s, int ch, size_t n)
```

参数：

`void *s` - 内存块指针

`int ch` - 填充值

`size_t n` - 填充大小

返回：

无



示例:

```
uint8 buffer[32];  
os_memset(buffer, 0, sizeof(buffer));
```

39. os_memcpy

功能:

封装 C 语言函数, 内存拷贝。

函数定义:

```
os_memcpy(void *des, void *src, size_t n)
```

参数:

`void *des` - 目标内存块指针

`void *src` - 源内存块指针

`size_t n` - 拷贝内存大小

返回:

无

示例:

```
uint8 buffer[4] = {0};  
os_memcpy(buffer, "abcd", 4);
```

40. os_strlen

功能:

封装 C 语言函数, 计算字符串长度。

函数定义:

```
os_strlen(char *s)
```

参数:

`char *s` - 字符串

返回:

字符串长度

示例:

```
char *ssid = "ESP8266";  
os_memcpy(softAP_config.ssid, ssid, os_strlen(ssid));
```



41. os_printf

功能：

格式化输出，打印字符串。

注意：

- 本接口默认从 UART 0 打印。IOT_Demo 中的 `uart_init` 可以设置波特率，将 `os_printf` 改为从 UART 1 打印：`os_install_putc1((void *)uart1_write_char);`
- 请勿调用本接口打印超过 125 字节的数据，或者频繁连续调用本接口打印，否则可能会丢失部分待打印数据。

函数定义：

```
void os_printf(const char *s)
```

参数：

`const char *s` - 字符串

返回：

无

示例：

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

42. os_bzero

功能：

置字符串 `p` 的前 `n` 个字节为零且包含 `'\0'`

函数定义：

```
void os_bzero(void *p, size_t n)
```

参数：

`void *p` - 要置零的数据的起始地址

`size_t n` - 要置零的数据字节数

返回：

无

43. 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

44. `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)` - 打印接口函数指针

返回:

无

示例:

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



3.4. 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。flash 读写必须 4 字节对齐。

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。flash 读写必须 4 字节对齐。

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]);
```

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 $offset + len \leq 4 * 1024$

返回：

true, 成功;

false, 失败

7. spi_flash_set_read_func

功能：

注册用户自定义的 SPI flash read 接口函数

注意：

仅支持在 SPI overlap 模式下使用，请用户参考 esp_iot_sdk/examples/driver_lib
/driver/spi_overlap.c

函数定义：

```
void spi_flash_set_read_func (user_spi_flash_read read)
```

参数：

user_spi_flash_read read : 用户自定义 SPI flash read 接口函数

参数定义：

```
typedef SpiFlashOpResult (*user_spi_flash_read)(
```



```
SpiFlashChip *spi,  
uint32 src_addr,  
uint32 * des_addr,  
uint32 size  
)
```

返回:

无



3.5. Wi-Fi 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：

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。

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。



注意：

若为特殊应用场景：调用 `wifi_station_set_reconnect_policy` 关闭重连功能，且未调用 `wifi_set_event_handler_cb` 注册 WiFi 事件回调，则本接口失效，无法准确获得连接状态。

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 dhcpc_status wifi_station_dhcpc_status(void)
```

参数：

无



返回：

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

23. wifi_station_dhcpc_set_maxtry

功能：

设置 ESP8266 station DHCP client 最大重连次数。默认会一直重连。

函数定义：

```
bool wifi_station_dhcpc_set_maxtry(uint8 num)
```

参数：

uint8 num - 最大重连次数

返回：

true: 成功

false: 失败

24. 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

25. wifi_station_get_rssi

功能：

查询 ESP8266 station 已连接的 AP 信号强度



函数定义：

```
sint8 wifi_station_get_rssi(void)
```

参数：

无

返回：

< 10 : 查询成功，返回信号强度

31 : 查询失败，返回错误码

26. wifi_station_set_hostname

功能：

设置 ESP8266 station DHCP 分配的主机名称。

函数定义：

```
bool wifi_station_set_hostname(char* hostname)
```

参数：

char* hostname : 主机名称，最长 32 个字符。

返回：

true: 成功

false: 失败

27. wifi_station_get_hostname

功能：

查询 ESP8266 station DHCP 分配的主机名称。

函数定义：

```
char* wifi_station_get_hostname(void)
```

参数：

无

返回：

主机名称

28. 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

29. 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

30. 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



31. 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

32. 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

33. 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

34. 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
```

35. 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

36. [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

37. [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();
}
```



38. wifi_softap_get_dhcps_lease

功能：

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

注意：

本接口仅支持在 ESP8266 soft-AP DHCP server 使能的情况下查询。

函数定义：

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

返回：

true: 成功

false: 失败

39. wifi_softap_set_dhcps_lease_time

功能：

设置 ESP8266 soft-AP DHCP server 的租约时间。默认为 120 分钟。

注意：

本接口仅支持在 ESP8266 soft-AP DHCP server 使能的情况下查询。

函数定义：

```
bool wifi_softap_set_dhcps_lease_time(uint32 minute)
```

参数：

`uint32 minute` : 租约时间，单位：分钟，取值范围：[1, 2880]

返回：

true: 成功

false: 失败

40. wifi_softap_get_dhcps_lease_time

功能：

查询 ESP8266 soft-AP DHCP server 租约时间

注意：

本接口仅支持在 ESP8266 soft-AP DHCP server 使能的情况下查询。

函数定义：

```
uint32 wifi_softap_get_dhcps_lease_time(void)
```

返回：

租约时间，单位：分钟



41. wifi_softap_reset_dhcps_lease_time

功能：

复位 ESP8266 soft-AP DHCP server 租约时间，恢复到 120 分钟。

注意：

本接口仅支持在 ESP8266 soft-AP DHCP server 使能的情况下查询。

函数定义：

```
bool wifi_softap_reset_dhcps_lease_time(void)
```

返回：

true: 成功

false: 失败

42. 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  
};
```

43. 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);
```

44. `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 802.11b/g
- 如果在意功耗, 建议设置为 802.11g 模式

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

45. `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
};
```

46. 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

47. wifi_set_ip_info

功能:

设置 ESP8266 station 或者 soft-AP 的 IP 地址

注意:

(1) 本接口设置静态 IP，请先关闭对应 DHCP 功能 ([wifi_station_dhcpc_stop](#) 或者 [wifi_softap_dhcps_stop](#))

(2) 设置静态 IP，则关闭 DHCP；DHCP 开启，则静态 IP 失效。



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：

```
wifi_set_opmode(STATIONAP_MODE); //Set softAP + station mode
struct ip_info info;
wifi_station_dhcpc_stop();
wifi_softap_dhcps_stop();

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);
wifi_softap_dhcps_start();
```

返回：

```
true: 成功
false: 失败
```

48. wifi_set_macaddr

功能：

设置 MAC 地址

注意：

- 本接口必须在 `user_init` 中调用
- ESP8266 soft-AP 和 station MAC 地址不同，请勿将两者设置为同一 MAC 地址
- ESP8266 MAC 地址第一个字节的 bit 0 不能为 1。例如，MAC 地址可以设置为 “1a:XX:XX:XX:XX:XX”，但不能设置为 “15:XX:XX:XX:XX:XX”。



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:

```
wifi_set_opmode(STATIONAP_MODE);  
  
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
```

49. 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
```



50. 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

51. 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  
};
```

52. 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)
```

53. 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

54. 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 接口均发送
```

返回:

```
true: 成功  
false: 失败
```

55. 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
```

56. 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)  
{
```



```
os_printf("event %x\n", evt->event);
switch (evt->event) {
    case EVENT_STAMODE_CONNECTED:
        os_printf("connect to ssid %s, channel %d\n",
            evt->event_info.connected.ssid,
            evt->event_info.connected.channel);

        break;
    case EVENT_STAMODE_DISCONNECTED:
        os_printf("disconnect from ssid %s, reason %d\n",
            evt->event_info.disconnected.ssid,
            evt->event_info.disconnected.reason);

        break;
    case EVENT_STAMODE_AUTHMODE_CHANGE:
        os_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:
        os_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));

        os_printf("\n");
        break;
    case EVENT_SOFTAPMODE_STACONNECTED:
        os_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:
        os_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);  
}
```

57. wifi_wps_enable

功能：

使能 Wi-Fi WPS 功能

注意：

WPS 功能必须在 ESP8266 station 使能的情况下调用。

结构体：

```
typedef enum wps_type {  
    WPS_TYPE_DISABLE=0,  
    WPS_TYPE_PBC,  
    WPS_TYPE_PIN,  
    WPS_TYPE_DISPLAY,  
    WPS_TYPE_MAX,  
}WPS_TYPE_t;
```

函数定义：

```
bool wifi_wps_enable(WPS_TYPE_t wps_type)
```

参数：

WPS_TYPE_t wps_type : WPS 的类型，目前仅支持 **WPS_TYPE_PBC**

返回：

true, 成功;

false, 失败

58. wifi_wps_disable

功能：

关闭 Wi-Fi WPS 功能，释放占用的资源。

函数定义：

```
bool wifi_wps_disable(void)
```

参数：

无

返回：

true, 成功;

false, 失败



59. wifi_wps_start

功能：

WPS 开始进行交互

注意：

WPS 功能必须在 ESP8266 station 使能的情况下调用。

函数定义：

```
bool wifi_wps_start(void)
```

参数：

无

返回：

true, 成功开始交互，并不表示 WPS 成功完成

false, 失败

60. wifi_set_wps_cb

功能：

设置 WPS 回调函数，回调函数中将传入 WPS 运行状态。WPS 不支持 WEP 加密方式。

回调及参数结构体：

```
typedef void (*wps_st_cb_t)(int status);

enum wps_cb_status {
    WPS_CB_ST_SUCCESS = 0,
    WPS_CB_ST_FAILED,
    WPS_CB_ST_TIMEOUT,
    WPS_CB_ST_WEP, // WPS failed because that WEP is not supported.
    WPS_CB_ST_SCAN_ERR, // can not find the target WPS AP
};
```

注意：

- 如果回调函数的传入参数状态为 `WPS_CB_ST_SUCCESS`，表示成功获得 AP 密钥，请调用 `wifi_wps_disable` 关闭 WPS 功能释放资源，并调用 `wifi_station_connect` 连接 AP。
- 否则，表示 WPS 失败，可以创建一个定时器，间隔一段时间后调用 `wifi_wps_start` 再次尝试 WPS，或者调用 `wifi_wps_disable` 关闭 WPS 并释放资源。

函数定义：

```
bool wifi_set_wps_cb(wps_st_cb_t cb)
```

参数：

`wps_st_cb_t cb` : 回调函数



返回:

true, 成功;

false, 失败

61. wifi_register_send_pkt_freedom_cb

功能:

注册 freedom 发包的回调函数。freedom 发包功能, 即支持发送用户自定义 802.11 的包。

注意:

freedom 发包必须等前一个包发送完毕, 进入发包回调 `freedom_outside_cb_t` 之后, 才能发下一个包。

回调函数定义:

```
typedef void (*freedom_outside_cb_t)(uint8 status);
```

参数 `status` : 0, 发包成功; 其他值, 发包失败。

函数定义:

```
int wifi_register_send_pkt_freedom_cb(freedom_outside_cb_t cb)
```

参数:

`freedom_outside_cb_t cb` : 回调函数

返回:

0, 注册成功;

-1, 注册失败

62. wifi_unregister_send_pkt_freedom_cb

功能:

注销 freedom 发包的回调函数。

函数定义:

```
void wifi_unregister_send_pkt_freedom_cb(void)
```

参数:

无

返回:

无

63. wifi_send_pkt_freedom

功能:

发包函数。

注意:



- 发送包必须是完整的 802.11 包，长度不包含 FCS。发包长度必须大于最小 802.11 头，即 24 字节，且不能超过 1400 字节，否则返回发包失败。
- duration 域填写无效，由 ESP8266 底层程序决定，自动填充。
- 发包速率限制成管理包速率，与系统的发包速率一致。
- 不支持加密包，即包头中的加密 bit 必须为 0，否则返回发包失败。
- freedom 发包必须等前一个包发送完毕，进入发包回调之后，才能发下一个包。

函数定义：

```
int wifi_send_pkt_freedom(uint8 *buf, int len, bool sys_seq)
```

参数：

uint8 *buf : 数据包指针

int len : 数据包长度

bool sys_seq : 是否跟随系统的 802.11 包 sequence number，如果跟随系统，将会在每次发包后自加 1

返回：

0，成功；

-1，失败

64. wifi_rfid_locp_recv_open

功能：

开启 RFID LOCP (Location Control Protocol) 功能，用于接收 WDS 类型的包。

函数定义：

```
int wifi_rfid_locp_recv_open(void)
```

参数：

无

返回：

0，成功；

其他值，失败

65. wifi_rfid_locp_recv_close

功能：

关闭 RFID LOCP (Location Control Protocol) 功能。

函数定义：

```
void wifi_rfid_locp_recv_close(void)
```

参数：

无



返回:

无

66. wifi_register_rfid_locp_recv_cb

功能:

注册 WDS 收包回调。仅在收到的 WDS 包的第一个 MAC 地址为组播地址时，才会进入回调函数。

回调函数定义:

```
typedef void (*rfid_locp_cb_t)(uint8 *frm, int len, int rssi);
```

参数说明:

`uint8 *frm` : 指向 802.11 包头的指针

`int len` : 数据包长度

`int rssi` : 信号强度

函数定义:

```
int wifi_register_rfid_locp_recv_cb(rfid_locp_cb_t cb)
```

参数:

`rfid_locp_cb_t cb` : 回调函数

返回:

0, 成功;

其他值, 失败

67. wifi_unregister_rfid_locp_recv_cb

功能:

注销 WDS 收包回调。

函数定义:

```
void wifi_unregister_rfid_locp_recv_cb(void)
```

参数:

无

返回:

无



3.6. Rate Control 接口

1. wifi_set_user_fixed_rate

功能：

设置 ESP8266 station 或 soft-AP 发数据的固定 rate 和 mask

参数定义：

```
enum FIXED_RATE {  
    PHY_RATE_48    =    0x8,  
    PHY_RATE_24    =    0x9,  
    PHY_RATE_12    =    0xA,  
    PHY_RATE_6     =    0xB,  
    PHY_RATE_54    =    0xC,  
    PHY_RATE_36    =    0xD,  
    PHY_RATE_18    =    0xE,  
    PHY_RATE_9     =    0xF,  
}  
  
#define FIXED_RATE_MASK_NONE        (0x00)  
#define FIXED_RATE_MASK_STA        (0x01)  
#define FIXED_RATE_MASK_AP        (0x02)  
#define FIXED_RATE_MASK_ALL        (0x03)
```

注意：

- 当 enable_mask 的对应 bit 为 1，ESP8266 station 或 soft-AP 才会以固定 rate 发送数据。
- 如果 enable_mask 设置成 0，则 ESP8266 station 和 soft-AP 均不会以固定 rate 发送数据。
- ESP8266 station 和 soft-AP 共享同一个 rate，不支持分别设置为不同 rate 值。

函数定义：

```
int wifi_set_user_fixed_rate(uint8 enable_mask, uint8 rate)
```

参数：

```
uint8 enable_mask : 0x00 - 禁用固定 rate  
                  0x01 - 固定 rate 用于 ESP8266 station 接口  
                  0x02 - 固定 rate 用于 ESP8266 soft-AP 接口  
                  0x03 - 固定 rate 用于 ESP8266 station + soft-AP  
  
uint8 rate        : 固定 rate 值
```

返回：

0，成功；
其他，失败



2. wifi_get_user_fixed_rate

功能：

获取已经设置的固定 rate 的 mask 和 rate 值

函数定义：

```
int wifi_get_user_fixed_rate(uint8 *enable_mask, uint8 *rate)
```

参数：

`uint8 *enable_mask` : mask 的指针

`uint8 *rate` : rate 的指针

返回：

0, 成功;

其他, 失败

3. wifi_set_user_sup_rate

功能：

设置 ESP8266 beacon、probe req/resp 等包里的 support rate 的 IE 中支持的 rate 范围。

用于将 ESP8266 支持的通信速率告知通信对方，以限制对方设备的发包速率。

注意：

本接口目前仅支持 802.11g 模式，后续会增加支持 802.11b。

参数定义：

```
enum support_rate {  
    RATE_11B5M      = 0,  
    RATE_11B11M     = 1,  
    RATE_11B1M      = 2,  
    RATE_11B2M      = 3,  
    RATE_11G6M      = 4,  
    RATE_11G12M     = 5,  
    RATE_11G24M     = 6,  
    RATE_11G48M     = 7,  
    RATE_11G54M     = 8,  
    RATE_11G9M      = 9,  
    RATE_11G18M     = 10,  
    RATE_11G36M     = 11,  
};
```

函数定义：

```
int wifi_set_user_sup_rate(uint8 min, uint8 max)
```

参数：

`uint8 min` : support rate 下限值，仅支持从 `enum support_rate` 中取值。

`uint8 max` : support rate 上限值，仅支持从 `enum support_rate` 中取值。



返回：

0，成功；

其他，失败

示例：

```
wifi_set_user_sup_rate(RATE_11G6M, RATE_11G24M);
```

4. wifi_set_user_rate_limit

功能：

设置 ESP8266 发包的初始速率范围。重传速率则不受此接口限制。

参数定义：

```
enum RATE_11B_ID {  
    RATE_11B_B11M = 0,  
    RATE_11B_B5M  = 1,  
    RATE_11B_B2M  = 2,  
    RATE_11B_B1M  = 3,  
}
```

```
enum RATE_11G_ID {  
    RATE_11G_G54M = 0,  
    RATE_11G_G48M = 1,  
    RATE_11G_G36M = 2,  
    RATE_11G_G24M = 3,  
    RATE_11G_G18M = 4,  
    RATE_11G_G12M = 5,  
    RATE_11G_G9M  = 6,  
    RATE_11G_G6M  = 7,  
    RATE_11G_B5M  = 8,  
    RATE_11G_B2M  = 9,  
    RATE_11G_B1M  = 10,  
}
```




```
enum RATE_11N_ID {  
    RATE_11N_MCS7S = 0,  
    RATE_11N_MCS7 = 1,  
    RATE_11N_MCS6 = 2,  
    RATE_11N_MCS5 = 3,  
    RATE_11N_MCS4 = 4,  
    RATE_11N_MCS3 = 5,  
    RATE_11N_MCS2 = 6,  
    RATE_11N_MCS1 = 7,  
    RATE_11N_MCS0 = 8,  
    RATE_11N_B5M = 9,  
    RATE_11N_B2M = 10,  
    RATE_11N_B1M = 11  
}
```

函数定义：

```
bool wifi_set_user_rate_limit(uint8 mode, uint8 ifidx, uint8 max, uint8 min)
```

参数：

uint8 mode : 设置模式

```
#define RC_LIMIT_11B 0
```

```
#define RC_LIMIT_11G 1
```

```
#define RC_LIMIT_11N 2
```

uint8 ifidx : 设置接口

0x00 – ESP8266 station 接口

0x01 – ESP8266 soft-AP 接口

uint8 max : 速率上限。请从第一个参数 mode 对应的速率枚举中取值。

uint8 min : 速率下限。请从第一个参数 mode 对应的速率枚举中取值。

返回：

true, 成功;

false, 失败

示例：

设置 11G 模式下的 ESP8266 station 接口的速率，限制为最大 18M，最小 6M。

```
wifi_set_user_rate_limit(RC_LIMIT_11G, 0, RATE_11G_G18M, RATE_11G_G6M);
```

5. wifi_set_user_limit_rate_mask

功能：

设置使能受 wifi_set_user_rate_limit 限制速率的接口。

参数定义：

```
#define LIMIT_RATE_MASK_NONE (0x00)
```



```
#define LIMIT_RATE_MASK_STA    (0x01)
#define LIMIT_RATE_MASK_AP     (0x02)
#define LIMIT_RATE_MASK_ALL    (0x03)
```

函数定义：

```
bool wifi_set_user_limit_rate_mask(uint8 enable_mask)
```

参数：

`uint8 enable_mask` : `0x00` - ESP8266 station + soft-AP 接口均不受限制
`0x01` - ESP8266 station 接口开启限制
`0x02` - ESP8266 soft-AP 接口开启限制
`0x03` - ESP8266 station + soft-AP 接口均开启限制

返回：

`true`, 成功;
`false`, 失败

6. `wifi_get_user_limit_rate_mask`

功能：

查询当前受 `wifi_set_user_rate_limit` 限制速率的接口。

函数定义：

```
uint8 wifi_get_user_limit_rate_mask(void)
```

参数：

无

返回：

`0x00` - ESP8266 station + soft-AP 接口均不受限制
`0x01` - ESP8266 station 接口受限制
`0x01` - ESP8266 soft-AP 接口受限制
`0x00` - ESP8266 station + soft-AP 接口均受限制



3.7. 强制休眠接口

使用强制休眠功能，必须先设置 Wi-Fi 工作模式为 NULL_MODE。退出强制休眠并关闭强制休眠功能后，才能再设置 Wi-Fi 工作模式为 station、soft-AP 或 sta+AP 的正常工作模式，具体可参考本章节后文提供的“示例代码”。

1. wifi_fpm_open

功能：

开启强制休眠功能

函数定义：

```
void wifi_fpm_open (void)
```

参数：

无

默认值：

强制 sleep 功能关闭

返回：

无

2. wifi_fpm_close

功能：

关闭强制休眠功能

函数定义：

```
void wifi_fpm_close (void)
```

参数：

无

返回：

无

3. wifi_fpm_do_wakeup

功能：

唤醒强制休眠

注意：

本接口仅支持在强制休眠功能开启的情况下调用，可在 `wifi_fpm_open` 之后调用；在 `wifi_fpm_close` 之后，不可以调用。

函数定义：

```
void wifi_fpm_do_wakeup (void)
```



参数：

无

返回：

无

4. `wifi_fpm_do_sleep`

功能：

让系统强制休眠，休眠时间到后，系统将自动醒来。

注意：

本接口仅支持在强制休眠功能开启的情况下调用，可在 `wifi_fpm_open` 之后调用；在 `wifi_fpm_close` 之后，不可以调用。

函数定义：

```
int8 wifi_fpm_do_sleep (uint32 sleep_time_in_us)
```

参数：

`uint32 sleep_time_in_us`：休眠时间，单位：us，取值范围：10000 ~ 268435455 (0xFFFFFFFF)

如果参数设置为 0xFFFFFFFF，则系统将一直休眠，直至：

- 若 `wifi_fpm_set_sleep_type` 设置为 `LIGHT_SLEEP_T`，可被 GPIO 唤醒。
- 若 `wifi_fpm_set_sleep_type` 设置为 `MODEM_SLEEP_T`，可被 `wifi_fpm_do_wakeup` 唤醒。

返回：

0，休眠成功；

-1，强制休眠的状态错误，休眠失败；

-2，强制休眠功能未开启，休眠失败。

5. `wifi_fpm_set_sleep_type`

功能：

设置系统强制休眠的休眠类型。

注意：

如需调用本接口，请在 `wifi_fpm_open` 之前调用。

函数定义：

```
void wifi_fpm_set_sleep_type (enum sleep_type type)
```

参数：



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

返回:

无

6. wifi_fpm_get_sleep_type

功能:

查询系统强制休眠的休眠类型。

函数定义:

```
enum sleep_type wifi_fpm_get_sleep_type (void)
```

参数:

无

返回:

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

7. 示例代码

```
#define FPM_SLEEP_MAX_TIME    0xFFFFFFFF

wifi_station_disconnect();

wifi_set_opmode(NULL_MODE);    // set WiFi mode to null mode
wifi_fpm_set_sleep_type(MODEM_SLEEP_T); // set force sleep type
wifi_fpm_open();
wifi_fpm_do_sleep(FPM_SLEEP_MAX_TIME);
.....

// wake up to use WiFi again
wifi_fpm_do_wakeup();
wifi_fpm_close();
```



```
wifi_set_opmode(STATION_MODE);  
wifi_station_connect();
```



3.8. ESP-NOW 接口

ESP-NOW 软件接口使用时的注意事项如下：

- ESP-NOW 目前不支持广播包和组播包；
- ESP-NOW 现阶段主要为智能灯项目实现，建议 slave 角色对应 ESP8266 soft-AP 模式或者 soft-AP +station 共存模式；controller 角色对应 station 模式；
- 当 ESP8266 处于 soft-AP +station 共存模式时，若作为 slave 角色，将从 soft-AP 接口通信；若作为 controller 角色，将从 station 接口通信；
- ESP-NOW 不实现休眠唤醒功能，因此如果通信对方的 ESP8266 station 正处于休眠状态，ESP-NOW 发包将会失败；
- ESP8266 station 模式下，最多可设置 10 个加密的 ESP-NOW peer，加上不加密的设备，总数不超过 20 个；
- ESP8266 soft-AP 模式或者 soft-AP + station 模式下，最多设置 6 个加密的 ESP-NOW peer，加上不加密的设备，总数不超过 20 个。

1. esp_now_init

功能：

初始化 ESP-NOW 功能

函数定义：

```
init esp_now_init(void)
```

参数：

无

返回：

0，成功；

否则，失败

2. esp_now_deinit

功能：

卸载 ESP-NOW 功能

函数定义：

```
int esp_now_deinit(void)
```

参数：

无

返回：

0，成功；



否则，失败

3. esp_now_register_recv_cb

功能：

注册 ESP-NOW 收包的回调函数

注意：

当收到 ESP-NOW 的数据包，进入收包回调函数

```
typedef void (*esp_now_recv_cb_t)(u8 *mac_addr, u8 *data, u8 len)
```

回调函数的三个参数分别为：

`u8 *mac_addr` : 发包方的 MAC 地址；

`u8 *data` : 收到的数据；

`u8 len` : 数据长度；

函数定义：

```
int esp_now_register_recv_cb(esp_now_recv_cb_t cb)
```

参数：

`esp_now_recv_cb_t cb` : 回调函数

返回：

0，成功；

否则，失败

4. esp_now_unregister_recv_cb

功能：

注销 ESP-NOW 收包的回调函数

函数定义：

```
int esp_now_unregister_recv_cb(void)
```

参数：

无

返回：

0，成功；

否则，失败

5. esp_now_register_send_cb

功能：

设置 ESP-NOW 发包回调函数



注意：

当发送了 ESP-NOW 的数据包，进入发包回调函数

```
void esp_now_send_cb_t(u8 *mac_addr, u8 status)
```

回调函数的两个参数分别为：

u8 *mac_addr : 发包对方的目标 MAC 地址；

u8 status : 发包状态；0，成功；否则，失败。对应结构体

```
mt_tx_status {  
    T_TX_STATUS_OK = 0,  
    MT_TX_STATUS_FAILED,  
}
```

发包回调函数不判断密钥是否匹配，如果使用密钥加密，请自行确保密钥正确。

函数定义：

```
u8 esp_now_register_send_cb(esp_now_send_cb_t cb)
```

参数：

esp_now_send_cb_t cb : 回调函数

返回：

0，成功；

否则，失败

6. esp_now_unregister_send_cb

功能：

注销 ESP-NOW 发包的回调函数，不再报告发包状态。

函数定义：

```
int esp_now_unregister_send_cb(void)
```

参数：

无

返回：

0，成功；

否则，失败

7. esp_now_send

功能：

发送 ESP-NOW 数据包



函数定义：

```
int esp_now_send(u8 *da, u8 *data, int len)
```

参数：

u8 *da : 目的 MAC 地址；如果为 NULL，则遍历 ESP-NOW 维护的所有 MAC 地址进行发送，否则，向指定 MAC 地址发送。

u8 *data : 要发送的数据；

u8 len : 数据长度；

返回：

0，成功；

否则，失败

8. esp_now_add_peer

功能：

增加 ESP-NOW 匹配设备，将设备 MAC 地址存入 ESP-NOW 维护的列表。

结构体：

```
typedef enum mt_role {  
    MT_ROLE_IDLE = 0,  
    MT_ROLE_CONTROLLER,  
    MT_ROLE_SLAVE,  
    MT_ROLE_MAX,  
}
```

函数定义：

```
int esp_now_add_peer(u8 *mac_addr, u8 role, u8 channel, u8 *key, u8 key_len)
```

参数：

u8 *mac_addr : 匹配设备的 MAC 地址；

u8 role : 该匹配设备的角色；

u8 channel : 匹配设备的信道值；

u8 *key : 与该匹配设备通信时，需使用的密钥，目前仅支持 16 字节的密钥；

u8 key_len : 密钥长度，目前长度仅支持 16 字节

返回：

0，成功；

否则，失败



9. esp_now_del_peer

功能：

删除 ESP-NOW 匹配设备，将设备 MAC 地址从 ESP-NOW 维护的列表中删除。

函数定义：

```
int esp_now_del_peer(u8 *mac_addr)
```

参数：

`u8 *mac_addr` : 要删除设备的 MAC 地址；

返回：

0，成功；

否则，失败

10. esp_now_set_self_role

功能：

设置自身 ESP-NOW 的角色

结构体：

```
typedef enum mt_role {  
    MT_ROLE_IDLE = 0,  
    MT_ROLE_CONTROLLER,  
    MT_ROLE_SLAVE,  
    MT_ROLE_MAX,  
}
```

函数定义：

```
int esp_now_set_self_role(u8 role)
```

参数：

`u8 role` : 角色类型

返回：

0，成功；

否则，失败

11. esp_now_get_self_role

功能：

查询自身 ESP-NOW 的角色

函数定义：

```
u8 esp_now_get_self_role(void)
```

参数：



无

返回：

角色类型

12. esp_now_set_peer_role

功能：

设置指定匹配设备的 ESP-NOW 角色。如果重复设置，新设置会覆盖原有设置。

结构体：

```
typedef enum mt_role {  
    MT_ROLE_IDLE = 0,  
    MT_ROLE_CONTROLLER,  
    MT_ROLE_SLAVE,  
    MT_ROLE_MAX,  
}
```

函数定义：

```
int esp_now_set_peer_role(u8 *mac_addr, u8 role)
```

参数：

`u8 *mac_addr` : 指定设备的 MAC 地址；

`u8 role` : 角色类型

返回：

0，成功；

否则，失败

13. esp_now_get_peer_role

功能：

查询指定匹配设备的 ESP-NOW 角色

函数定义：

```
int esp_now_get_peer_role(u8 *mac_addr)
```

参数：

`u8 *mac_addr` : 指定设备的 MAC 地址；

返回：

`MT_ROLE_CONTROLLER`，角色为 controller；

`MT_ROLE_SLAVE`，角色为 slave；

否则，失败



14. esp_now_set_peer_key

功能：

设置指定匹配设备的 ESP-NOW 密钥。如果重复设置，新设置会覆盖原有设置。

函数定义：

```
int esp_now_set_peer_key(u8 *mac_addr, u8 *key, u8 key_len)
```

参数：

u8 *mac_addr : 指定设备的 MAC 地址；

u8 *key : 密钥指针，目前仅支持 16 字节的密钥；如果传 **NULL**，则清除当前密钥

u8 key_len : 密钥长度，目前仅支持 16 字节

返回：

0，成功；

否则，失败

15. esp_now_get_peer_key

功能：

查询指定匹配设备的 ESP-NOW 密钥

函数定义：

```
int esp_now_get_peer_key(u8 *mac_addr, u8 *key, u8 *key_len)
```

参数：

u8 *mac_addr : 指定设备的 MAC 地址；

u8 *key : 查询到的密钥指针，请使用 16 字节的 buffer 保存密钥

u8 *key_len : 查询到的密钥长度

返回：

0，成功；

> 0，找到目标设备，但未获得 key；

< 0，失败

16. esp_now_set_peer_channel

功能：

记录指定匹配设备的信道值。

当与该指定设备进行 ESP-NOW 通信时，

- 先调用 `esp_now_get_peer_channel` 查询该设备所在信道；



- 再调用 `wifi_set_channel` 与该设备切换到同一信道进行通信；
- 通信完成后，请注意切换回原所在信道。

函数定义：

```
int esp_now_set_peer_channel(u8 *mac_addr, u8 channel)
```

参数：

`u8 *mac_addr` : 指定设备的 MAC 地址；

`u8 channel` : 信道值，一般为 1 ~ 13，部分地区可能用到 14

返回：

0，成功；

否则，失败

17. esp_now_get_peer_channel

功能：

查询指定匹配设备的信道值。ESP-NOW 要求切换到同一信道进行通信。

函数定义：

```
int esp_now_get_peer_channel(u8 *mac_addr)
```

参数：

`u8 *mac_addr` : 指定设备的 MAC 地址；

返回：

1 ~ 13（部分地区可能到 14），成功；

否则，失败

18. esp_now_is_peer_exist

功能：

根据 MAC 地址判断设备是否存在

函数定义：

```
int esp_now_is_peer_exist(u8 *mac_addr)
```

参数：

`u8 *mac_addr` : 指定设备的 MAC 地址；

返回：

0，设备不存在

< 0，出错，查询失败

> 0，设备存在



19. esp_now_fetch_peer

功能：

查询当前指向的 ESP-NOW 配对设备的 MAC 地址，并将内部游标指向 ESP-NOW 维护列表的后一个设备或重新指向 ESP-NOW 维护列表的第一个设备。

注意：

- 本接口不可重入。
- 第一次调用本接口时，参数必须为 `true`，让内部游标指向 ESP-NOW 维护列表的第一个设备。

函数定义：

```
u8 *esp_now_fetch_peer(bool restart)
```

参数：

`bool restart` : `true`，将内部游标重新指向 ESP-NOW 维护列表的第一个设备；
 `false`，将内部游标指向 ESP-NOW 维护列表的后一个设备

返回：

`NULL`，不存在已关联的 ESP-NOW 设备
否则，当前指向的 ESP-NOW 配对设备的 MAC 地址指针

20. esp_now_get_cnt_info

功能：

查询已经匹配的全部设备总数和加密的设备总数。

函数定义：

```
int esp_now_get_cnt_info(u8 *all_cnt, u8 *encryp_cnt)
```

参数：

`u8 *all_cnt` : 已经匹配的全部设备总数
`u8 *encryp_cnt` : 加密的设备总数

返回：

`0`，成功；
否则，失败

21. esp_now_set_kok

功能：

设置用于将通信密钥加密的主密钥（key of key）。所有设备的通信均共享同一主密钥，如不设置，则使用默认主密钥给通信密钥加密。

函数定义：

```
int esp_now_set_kok(u8 *key, u8 len)
```



参数：

`u8 *key` : 主密钥指针

`u8 len` : 主密钥长度，目前长度仅支持 16 字节

返回：

0，成功；

否则，失败



3.9. 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



3.10. 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：

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：

设置信道号，用于混杂模式

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



3.11. smart config 接口

开启 smart config 功能前，请先确保 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；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;
```

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 ICACHE_FLASH_ATTR
smartconfig_done(sc_status status, void *pdata)
{
    switch(status) {
        case SC_STATUS_WAIT:
            os_printf("SC_STATUS_WAIT\n");
            break;
        case SC_STATUS_FIND_CHANNEL:
            os_printf("SC_STATUS_FIND_CHANNEL\n");
            break;
        case SC_STATUS_GETTING_SSID_PSWD:
            os_printf("SC_STATUS_GETTING_SSID_PSWD\n");
            sc_type *type = pdata;
            if (*type == SC_TYPE_ESPTOUCH) {
                os_printf("SC_TYPE:SC_TYPE_ESPTOUCH\n");
            } else {
                os_printf("SC_TYPE:SC_TYPE_AIRKISS\n");
            }
            break;
        case SC_STATUS_LINK:
            os_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);
                os_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. smartconfig_set_type

功能：

设置快连模式的协议类型。

注意：

如需调用本接口，请在 `smartconfig_start` 之前调用。

函数定义：

```
bool smartconfig_set_type(sc_type type)
```

参数：

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

返回：

true: 成功

false: 失败



3.12. SNTP 接口

1. sntp_setserver

功能：

通过 IP 地址设置 SNTP 服务器，一共最多支持设置 3 个 SNTP 服务器

函数定义：

```
void sntp_setserver(unsigned char idx, ip_addr_t *addr)
```

参数：

unsigned char idx : SNTP 服务器编号，最多支持 3 个 SNTP 服务器 (0 ~ 2)；0 号为主服务器，1 号和 2 号为备用服务器。

ip_addr_t *addr : IP 地址；用户需自行确保，传入的是合法 SNTP 服务器

返回：

无

2. sntp_getserver

功能：

查询 SNTP 服务器的 IP 地址，对应的设置接口为：[sntp_setserver](#)

函数定义：

```
ip_addr_t sntp_getserver(unsigned char idx)
```

参数：

unsigned char idx : SNTP 服务器编号，最多支持 3 个 SNTP 服务器 (0 ~ 2)

返回：

IP 地址

3. sntp_setservername

功能：

通过域名设置 SNTP 服务器，一共最多支持设置 3 个 SNTP 服务器

函数定义：

```
void sntp_setservername(unsigned char idx, char *server)
```

参数：

unsigned char idx : SNTP 服务器编号，最多支持 3 个 SNTP 服务器 (0 ~ 2)；0 号为主服务器，1 号和 2 号为备用服务器。

char *server : 域名；用户需自行确保，传入的是合法 SNTP 服务器

返回：

无



4. `sntp_getservername`

功能：

查询 SNTP 服务器的域名，仅支持查询通过 `sntp_setservername` 设置的 SNTP 服务器

函数定义：

```
char * sntp_getservername(unsigned char idx)
```

参数：

`unsigned char idx` : SNTP 服务器编号，最多支持 3 个 SNTP 服务器 (0 ~ 2)

返回：

服务器域名

5. `sntp_init`

功能：

SNTP 初始化

函数定义：

```
void sntp_init(void)
```

参数：

无

返回：

无

6. `sntp_stop`

功能：

SNTP 关闭

函数定义：

```
void sntp_stop(void)
```

参数：

无

返回：

无

7. `sntp_get_current_timestamp`

功能：

查询当前距离基准时间 (1970.01.01 00:00:00 GMT + 8) 的时间戳，单位：秒

函数定义：

```
uint32 sntp_get_current_timestamp()
```



参数:

无

返回:

距离基准时间的时间戳

8. `sntp_get_real_time`

功能:

查询实际时间 (GMT + 8)

函数定义:

```
char* sntp_get_real_time(long t)
```

参数:

`long t` - 与基准时间相距的时间戳

返回:

实际时间

9. `sntp_set_timezone`

功能:

设置时区信息

函数定义:

```
bool sntp_set_timezone (sint8 timezone)
```

注意:

调用本接口前, 请先调用 `sntp_stop`

参数:

`sint8 timezone` - 时区值, 参数范围: -11 ~ 13

返回:

true, 成功;

false, 失败

示例:

```
sntp_stop();  
  
if( true == sntp_set_timezone(-5) ) {  
    sntp_init();  
}
```



10. sntp_get_timezone

功能：

查询时区信息

函数定义：

```
sint8 sntp_get_timezone (void)
```

参数：

无

返回：

时区值，参数范围：-11 ~ 13

11. SNTP 示例

Step 1. enable sntp

```
ip_addr_t *addr = (ip_addr_t *)os_zalloc(sizeof(ip_addr_t));
sntp_setservername(0, "us.pool.ntp.org"); // set server 0 by domain name
sntp_setservername(1, "ntp.sjtu.edu.cn"); // set server 1 by domain name
ipaddr_aton("210.72.145.44", addr);
sntp_setserver(2, addr); // set server 2 by IP address
sntp_init();
os_free(addr);
```

Step 2. set a timer to check sntp timestamp

```
LOCAL os_timer_t sntp_timer;
os_timer_disarm(&sntp_timer);
os_timer_setfn(&sntp_timer, (os_timer_func_t *)user_check_sntp_stamp, NULL);
os_timer_arm(&sntp_timer, 100, 0);
```

Step 3. timer callback

```
void ICACHE_FLASH_ATTR user_check_sntp_stamp(void *arg){
    uint32 current_stamp;
    current_stamp = sntp_get_current_timestamp();
    if(current_stamp == 0){
```



```
        os_timer_arm(&sntp_timer, 100, 0);
    } else{
        os_timer_disarm(&sntp_timer);
        os_printf("sntp: %d, %s \n",current_stamp,
sntp_get_real_time(current_stamp));
    }
}
```



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。
- mDNS APIs: mDNS 相关接口。

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 `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` - 成功
`ESPCONN_ISCONN` - 失败, 错误码含义: 已经连接
`ESPCONN_ARG` - 失败, 错误码含义: 未找到参数 `espconn` 对应的网络传输

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;
    if (ipaddr != NULL)
        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 `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 `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 `espconn` 指针。

注意，本指针为底层维护的指针，不同回调传入的指针地址可能不一样，请勿依此判断网络连接。可根据 `espconn` 结构体中的 `remote_ip`, `remote_port` 判断多连接中的不同网络传输。



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 `espconn` 指针。注意，本指针为底层维护的指针，不同回调传入的指针地址可能不一样，请勿依此判断网络连接。可根据 `espconn` 结构体中的 `remote_ip`, `remote_port` 判断多连接中的不同网络传输。

`char *pdata` : 接收到的数据

`unsigned short len` : 接收到的数据长度

返回：

无

8. `espconn_send`

功能：

通过 WiFi 发送数据

注意：

- 一般情况，请在前一包数据发送成功，进入 `espconn_sent_callback` 后，再调用 `espconn_send` 发送下一包数据。
- 如果是 UDP 传输，请在每次调用 `espconn_send` 前，设置 `espconn->proto.udp->remote_ip` 和 `remote_port` 参数，因为 UDP 无连接，远端信息可能被更改。

函数定义：

```
sint8 espconn_send(  
    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
`ESPCONN_ARG` - 未找到参数 `espconn` 对应的网络传输;
`ESPCONN_MEM` - 空间不足
`ESPCONN_MAXNUM` - 底层发包缓存已满, 发包失败
`ESPCONN_IF` - UDP 发包失败

9. espconn_sent

[@deprecated] 本接口不建议使用, 建议使用 `espconn_send` 代替。

功能:

通过 WiFi 发送数据

注意:

- 一般情况, 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。
- 如果是 UDP 传输, 请在每次调用 `espconn_sent` 前, 设置 `espconn->proto.udp->remote_ip` 和 `remote_port` 参数, 因为 UDP 无连接, 远端信息可能被更改。

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
`ESPCONN_ARG` - 未找到参数 `espconn` 对应的网络传输;
`ESPCONN_MEM` - 空间不足
`ESPCONN_MAXNUM` - 底层发包缓存已满, 发包失败



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

`ESPCONN_ARG` - 未找到参数 `espconn` 对应的 TCP 连接

`ESPCONN_MEM` - 空间不足

`ESPCONN_ISCONN` - 连接已经建立

2. `espconn_regist_time`

功能：

注册 ESP8266 TCP server 超时时间，时间值仅作参考，并不精确。

注意：

请在 `espconn_accept` 之后，连接未建立之前，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 `ESPCONN_ARG` - 未找到参数 `espconn` 对应的 TCP 连接

3. `espconn_get_connection_info`

功能：

查询某个 TCP 连接或者 UDP 传输的远端信息。

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 `ESPCONN_ARG` - 未找到参数 `espconn` 对应的 TCP 连接

4. `espconn_connect`

功能：

连接 TCP server (ESP8266 作为 TCP client)。

注意：

如果 `espconn_connect` 失败，返回非零值，连接未建立，不会进入任何 `espconn` callback。

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
`ESPCONN_ARG` - 未找到参数 `espconn` 对应的 TCP 连接
`ESPCONN_MEM` - 空间不足
`ESPCONN_ISCONN` - 连接已经建立



5. `espconn_regist_connectcb`

功能：

注册 TCP 连接成功建立后的回调函数。

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 `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，对应网络连接的结构体 `espconn` 指针。

注意，本指针为底层维护的指针，不同回调传入的指针地址可能不一样，请勿依此判断网络连接。可根据 `espconn` 结构体中的 `remote_ip`, `remote_port` 判断多连接中的不同网络传输。

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_KEEPAIVE = 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 nagle 算法；

bit 2: 1: 使能 write finish callback，进入此回调表示 `espconn_send` 要发送的数据已经写入 2920 字节的 write buffer 等待发送或已经发送；

bit 3: 1: 使能 keep alive；

返回：

0 : 成功

Non-0 : 失败，返回错误码 `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_KEEPAIVE = 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 [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_KEEPIIDLE,  
    ESPCONN_KEEPIINTVL,  
    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`。

注意, 时间间隔设置并不可靠精准, 仅供参考, 受其他高优先级任务执行的影响。

参数说明如下:

`ESPCONN_KEEPIDLE` - 设置进行 keep alive 探查的时间间隔, 单位: 秒

`ESPCONN_KEEPINTVL` - keep alive 探查过程中, 报文的时间间隔, 单位: 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 `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: 秒

`ESPCONN_KEEPINTVL` - keep alive 探查过程中, 报文的时间间隔, 单位: 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 `ESPCONN_ARG` - 未找到参数 `espconn` 对应的 TCP 连接

11. espconn_reconnect_callback

功能:

TCP 连接异常断开时的回调函数, 相当于出错处理回调, 由 `espconn_regist_reconcb` 注册。

函数定义:

```
void espconn_reconnect_callback (void *arg, sint8 err)
```

参数:

`void *arg` : 回调函数的参数, 对应网络连接的结构体 `espconn` 指针。

注意, 本指针为底层维护的指针, 不同回调传入的指针地址可能不一样, 请勿依此判断网络连接。可根据 `espconn` 结构体中的 `remote_ip`, `remote_port` 判断多连接中的不同网络传输。

`sint8 err` : 异常断开的错误码。

`ESCONN_TIMEOUT` - 超时出错断开

`ESPCONN_ABRT` - TCP 连接异常断开

`ESPCONN_RST` - TCP 连接复位断开

`ESPCONN_CLSD` - TCP 连接在断开过程中出错, 异常断开

`ESPCONN_CONN` - TCP 未连接成功

`ESPCONN_HANDSHAKE` - TCP SSL 握手失败

`ESPCONN_PROTO_MSG` - SSL 应用数据处理异常

返回:

无

12. 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：**

`espconn_reconnect_callback` 功能类似于出错处理回调，任何阶段出错时，均会进入此回调；

例如，`espconn_sent` 失败，则认为网络连接异常，也会进入 `espconn_reconnect_callback`；

用户可在 `espconn_reconnect_callback` 中自行定义出错处理。

函数定义：

```
sint8 espconn_regist_reconcb(  
    struct espconn *espconn,  
    espconn_reconnect_callback recon_cb  
)
```

参数：

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

`espconn_reconnect_callback recon_cb` ： 回调函数

返回：

0 ： 成功

Non-0 ： 失败，返回错误码 `ESPCONN_ARG` - 未找到参数 `espconn` 对应的 TCP 连接

13. `espconn_disconnect`

功能：

断开 TCP 连接

注意：

请勿在 `espconn` 的任何 `callback` 中调用本接口断开连接。如有需要，可以在 `callback` 中使用任务触发调用本接口断开连接。

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 `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 `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。

请先调用 `espconn_set_opt` 使能 `write buffer`。

注意：

- `write buffer` 用于缓存 `espconn_send` 将发送的数据，最多缓存 8 包数据，`write buffer` 的容量为 2920 字节。
- 由 `espconn_set_opt` 设置使能 `write_finish_callback` 回调。
- 对发送速度有要求时，可以在 `write_finish_callback` 中调用 `espconn_send` 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 `ESPCONN_ARG` - 未找到参数 `espconn` 对应的 TCP 连接



16. 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

17. 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 `ESPCONN_ARG` - 未找到参数 `espconn` 对应的 TCP 连接

18. espconn_tcp_get_max_con_allow

功能：

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

函数定义：

```
sint8 espconn_tcp_get_max_con_allow(struct espconn *espconn)
```

参数：

`struct espconn *espconn` : 对应的 TCP server 结构体

返回：

> 0 : 最多允许连接的 TCP client 数目

< 0 : 失败，返回错误码 `ESPCONN_ARG` - 未找到参数 `espconn` 对应的 TCP 连接

19. espconn_tcp_set_max_con_allow

功能：

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



函数定义：

```
sint8 espconn_tcp_set_max_con_allow(struct espconn *espconn, uint8 num)
```

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 `ESPCONN_ARG` - 未找到参数 `espconn` 对应的 TCP 连接

20. 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 `ESPCONN_ARG` - 未找到参数 `espconn` 对应的 TCP 连接

21. 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 `ESPCONN_ARG` - 未找到参数 `espconn` 对应的 TCP 连接



22. espconn_secure_accept

功能：

创建 SSL TCP server，侦听 SSL 握手

注意：

- 目前仅支持建立一个 SSL server，本接口只能调用一次，并且仅支持连入一个 SSL client。
- 如果 SSL 加密一包数据大于 `espconn_secure_set_size` 设置的缓存空间，ESP8266 无法处理，SSL 连接断开，进入 `espconn_reconnect_callback`。
- SSL 相关接口与普通 TCP 接口底层处理不一致，请不要混用。SSL 连接时，仅支持使用 `espconn_secure_XXX` 系列接口和 `espconn_regist_XXX` 系列注册接口，以及 `espconn_port` 获得一个空闲端口。
- 如需创建 SSL server，必须先调用 `espconn_secure_set_default_certificate` 和 `espconn_secure_set_default_private_key` 传入证书和密钥。

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

`ESPCONN_ARG` - 未找到参数 `espconn` 对应的 TCP 连接

`ESPCONN_MEM` - 空间不足

`ESPCONN_ISCONN` - 连接已经建立

23. espconn_secure_set_size

功能：

设置加密（SSL）数据缓存空间的大小

注意：

默认缓存大小为 2KBytes；如需更改，请在加密（SSL）连接建立前调用：

在 `espconn_secure_accept`（ESP8266 作为 TCP SSL server）之前调用；

或者 `espconn_secure_connect`（ESP8266 作为 TCP SSL client）之前调用

函数定义：

```
bool espconn_secure_set_size (uint8 level, uint16 size)
```

参数：

`uint8 level` : 设置 ESP8266 SSL server/client:



0x01 SSL client; 0x02 SSL server; 0x03 SSL client 和 SSL server

uint16 size : 加密数据缓存的空间大小, 取值范围: 1 ~ 8192, 单位: 字节, 默认值为 2048

返回:

true : 成功

false : 失败

24. espconn_secure_get_size

功能:

查询加密 (SSL) 数据缓存空间的大小

函数定义:

```
sint16 espconn_secure_get_size (uint8 level)
```

参数:

uint8 level : 设置 ESP8266 SSL server/client:

0x01 SSL client; 0x02 SSL server; 0x03 SSL client 和 SSL server

返回:

加密 (SSL) 数据缓存空间的大小

25. espconn_secure_connect

功能:

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

注意:

- 如果 `espconn_secure_connect` 失败, 返回非零值, 连接未建立, 不会进入任何 `espconn` callback。
- 目前 ESP8266 作为 SSL client 仅支持一个连接, 本接口只能调用一次, 或者调用 `espconn_secure_disconnect` 断开前一次连接, 才可以再次调用本接口建立 SSL 连接;
- 如果 SSL 加密一包数据大于 `espconn_secure_set_size` 设置的缓存空间, ESP8266 无法处理, SSL 连接断开, 进入 `espconn_reconnect_callback`
- SSL 相关接口与普通 TCP 接口底层处理不一致, 请不要混用。SSL 连接时, 仅支持使用 `espconn_secure_XXX` 系列接口和 `espconn_regist_XXX` 系列注册接口, 以及 `espconn_port` 获得一个空闲端口。

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

ESPCONN_ARG - 未找到参数 `espconn` 对应的 TCP 连接

ESPCONN_MEM - 空间不足

ESPCONN_ISCONN - 传输已经建立

26. espconn_secure_send

功能：

发送加密数据（SSL）

注意：

请在上一包数据发送完成，进入 `espconn_sent_callback` 后，再发下一包数据。

函数定义：

```
sint8 espconn_secure_send (  
    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 `ESPCONN_ARG` - 未找到参数 `espconn` 对应的 TCP 连接

27. espconn_secure_send

[@deprecated] 本接口不建议使用，建议使用 `espconn_secure_send` 代替。

功能：

发送加密数据（SSL）

注意：

请在上一包数据发送完成，进入 `espconn_sent_callback` 后，再发下一包数据。



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 `ESPCONN_ARG` - 未找到参数 `espconn` 对应的 TCP 连接

28. espconn_secure_disconnect

功能：

断开加密 TCP 连接(SSL)

注意：

请勿在 `espconn` 的任何 `callback` 中调用本接口断开连接。如有需要，可以在 `callback` 中使用任务触发调用本接口断开连接。

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 `ESPCONN_ARG` - 未找到参数 `espconn` 对应的 TCP 连接

29. espconn_secure_ca_disable

功能：

关闭 SSL CA 认证功能

注意：

- CA 认证功能，默认关闭，详细介绍可参考文档“ESP8266__SDK__SSL_User_Manual”
- 如需调用本接口，请在加密（SSL）连接建立前调用：
在 `espconn_secure_accept`（ESP8266 作为 TCP SSL server）之前调用；



或者 `espconn_secure_connect` (ESP8266 作为 TCP SSL client) 之前调用

函数定义:

```
bool espconn_secure_ca_disable (uint8 level)
```

参数:

`uint8 level` : 设置 ESP8266 SSL server/client:

0x01 SSL client; 0x02 SSL server; 0x03 SSL client 和 SSL server

返回:

true : 成功

false : 失败

30. espconn_secure_ca_enable

功能:

开启 SSL CA 认证功能

注意:

- CA 认证功能, 默认关闭, 详细介绍可参考文档“ESP8266__SDK__SSL_User_Manual”
- 如需调用本接口, 请在加密 (SSL) 连接建立前调用:

在 `espconn_secure_accept` (ESP8266 作为 TCP SSL server) 之前调用;

或者 `espconn_secure_connect` (ESP8266 作为 TCP SSL client) 之前调用

函数定义:

```
bool espconn_secure_ca_enable (uint8 level, uint16 flash_sector)
```

参数:

`uint8 level` : 设置 ESP8266 SSL server/client:

0x01 SSL client; 0x02 SSL server; 0x03 SSL client 和 SSL server

`uint16 flash_sector` : 设置 CA 证书 (esp_ca_cert.bin) 烧录到 flash 的位置

例如, 参数传入 0x3B, 则对应烧录到 flash 0x3B000

返回:

true : 成功

false : 失败

31. espconn_secure_cert_req_enable

功能:

使能 ESP8266 作为 SSL client 时的证书认证功能

注意:



- 证书认证功能，默认关闭
- 如需调用本接口，请在 `espconn_secure_connect` 之前调用

函数定义：

```
bool espconn_secure_cert_req_enable (uint8 level, uint8 flash_sector)
```

参数：

`uint8 level` : 仅支持设置为 `0x01` ESP8266 作为 SSL client;

`uint8 flash_sector` : 设置密钥 (`esp_cert_private_key.bin`) 烧录到 Flash 的位置，例如，参数传入 `0x3A`，则对应烧录到 Flash `0x3A000`。请注意，不要覆盖了代码或系统参数区域。

返回：

`true` : 成功
`false` : 失败

32. espconn_secure_cert_req_disable

功能：

关闭 ESP8266 作为 SSL client 时的证书认证功能

注意：

- 证书认证功能，默认关闭

函数定义：

```
bool espconn_secure_ca_disable (uint8 level)
```

参数：

`uint8 level` : 仅支持设置为 `0x01` ESP8266 作为 SSL client;

返回：

`true` : 成功
`false` : 失败

33. espconn_secure_set_default_certificate

功能：

设置 ESP8266 作为 SSL server 时的证书

注意：

- `esp-iot-sdk/examples/IoT_Demo` 中提供使用示例
- 本接口必须在 `espconn_secure_accept` 之前调用，传入证书信息

函数定义：

```
bool espconn_secure_set_default_certificate (const uint8_t* certificate,  
uint16_t length)
```



参数:

`const uint8_t* certificate` : 证书指针

`uint16_t length` : 证书长度

返回:

`true` : 成功

`false` : 失败

34. `espconn_secure_set_default_private_key`

功能:

设置 ESP8266 作为 SSL server 时的密钥

注意:

- `esp_iot_sdk/examples/IoT_Demo` 中提供使用示例
- 本接口必须在 `espconn_secure_accept` 之前调用, 传入密钥信息

函数定义:

```
bool espconn_secure_set_default_private_key (const uint8_t* key, uint16_t length)
```

参数:

`const uint8_t* key` : 密钥指针

`uint16_t length` : 密钥长度

返回:

`true` : 成功

`false` : 失败



4.3. UDP 接口

1. espconn_create

功能：

建立 UDP 传输。

注意：

请注意设置 remote_ip 和 remote_port 参数，请勿设置为 0。

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

ESPCONN_ARG - 未找到参数 `espconn` 对应的 UDP 传输

ESPCONN_MEM - 空间不足

ESPCONN_ISCONN - 传输已经建立

2. espconn_sendto

功能：

UDP 发包接口

函数定义：

```
sin16 espconn_sendto(struct espconn *espconn, uint8 *psent, uint16 length)
```

参数：

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

`uint8 *psent` : 待发送的数据

`uint16 length` : 数据长度

返回：

0 : 成功

Non-0 : 失败，返回错误码

ESPCONN_ARG - 未找到参数 `espconn` 对应的 UDP 传输

ESPCONN_MEM - 空间不足

ESPCONN_IF - UDP 发包失败



3. espconn_igmp_join

功能：

加入多播组。

注意：

请在 ESP8266 station 已连入路由的情况下调用。

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 `ESPCONN_MEM` - 空间不足

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 `ESPCONN_MEM` - 空间不足

4. espconn_dns_setserver

功能：

设置默认 DNS server

注意：

本接口必须在 ESP8266 DHCP client 关闭 (`wifi_station_dhcpc_stop`) 的情况下使用。

函数定义：

```
void espconn_dns_setserver(char numdns, ip_addr_t *dnsserver)
```



参数:

`char numdns` : DNS server ID, 支持设置两个 DNS server, ID 分别为 0 和 1
`ip_addr_t *dnsserver` : DNS server IP

返回:

无



4.4. mDNS 接口

1. espconn_mdns_init

功能：

mDNS 初始化

注意：

(1) 若为 soft-AP+station 模式，请先调用 `wifi_set_broadcast_if(STATIONAP_MODE);`

若使用 ESP8266 station 接口，请获得 IP 后，再调用本接口初始化 mDNS；

(2) `txt_data` 必须为 `key = value` 的形式，如示例；

结构体：

```
struct mdns_info{
    char *host_name;
    char *server_name;
    uint16 server_port;
    unsigned long ipAddr;
    char *txt_data[10];
};
```

函数定义：

```
void espconn_mdns_init(struct mdns_info *info)
```

参数：

`struct mdns_info *info` : mDNS 结构体

返回：

无

示例：

```
struct mdns_info *info = (struct mdns_info *)os_zalloc(sizeof(struct
mdns_info));
info->host_name = "espressif";
info->ipAddr = station_ipconfig.ip.addr; //ESP8266 station IP
info->server_name = "iot";
info->server_port = 8080;
info->txt_data[0] = "version = now";
info->txt_data[1] = "user1 = data1";
```



```
info->txt_data[2] = "user2 = data2";  
espconn_mdns_init(info);
```

2. espconn_mdns_close

功能：

关闭 mDNS，对应开启 mDNS 的 API：`espconn_mdns_init`

函数定义：

```
void espconn_mdns_close(void)
```

参数：

无

返回：

无

3. espconn_mdns_server_register

功能：

注册 mDNS 服务器

函数定义：

```
void espconn_mdns_server_register(void)
```

参数：

无

返回：

无

4. espconn_mdns_server_unregister

功能：

注销 mDNS 服务器

函数定义：

```
void espconn_mdns_server_unregister(void)
```

参数：

无

返回：

无



5. espconn_mdns_get_servername

功能：

查询 mDNS 服务器名称

函数定义：

```
char* espconn_mdns_get_servername(void)
```

参数：

无

返回：

服务器名称

6. espconn_mdns_set_servername

功能：

设置 mDNS 服务器名称

函数定义：

```
void espconn_mdns_set_servername(const char *name)
```

参数：

`const char *name` - 服务器名称

返回：

无

7. espconn_mdns_set_hostname

功能：

设置 mDNS 主机名称

函数定义：

```
void espconn_mdns_set_hostname(char *name)
```

参数：

`char *name` - 主机名称

返回：

无

8. espconn_mdns_get_hostname

功能：

查询 mDNS 主机名称

函数定义：

```
char* espconn_mdns_get_hostname(void)
```



参数:

无

返回:

主机名称

9. espconn_mdns_disable

功能:

去能 mDNS , 对应使能 API : [espconn_mdns_enable](#)

函数定义:

```
void espconn_mdns_disable(void)
```

参数:

无

返回:

无

10. espconn_mdns_enable

功能:

使能 mDNS

函数定义:

```
void espconn_mdns_enable(void)
```

参数:

无

返回:

无



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



12. at_register_uart_rx_intr

功能：

设置 UART0 RX 是由用户使用，还是由 AT 使用。

注意：

本接口可以重复调用。

运行 AT BIN, UART0 RX 默认供 AT 使用。

函数定义：

```
void at_register_uart_rx_intr (at_custom_uart_rx_intr rx_func)
```

参数：

at_custom_uart_rx_intr : 注册用户使用 UART0 的 RX 中断处理函数；如果传 NULL，则切换为 AT 使用 UART0

返回：

无

示例：

```
void user_uart_rx_intr (uint8* data, int32 len)
{
    // UART0 rx for user
    os_printf("len=%d \r\n",len);
    os_printf(data);

    // change UART0 for AT
    at_register_uart_rx_intr(NULL);
}

void user_init(void){ at_register_uart_rx_intr(user_uart_rx_intr); }
```

13. at_response

功能：

设置 AT 响应

注意：

默认情况下，**at_response** 从 UART0 TX 输出，与 **at_port_print** 功能相同。

如果调用了 **at_register_response_func**，**at_response** 的字符串成为 **response_func** 的参数，由用户自行处理。

函数定义：

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



参数:

`const char *str` : 字符串

返回:

无

14. `at_register_response_func`

功能:

注册 `at_response` 的回调函数。调用了 `at_register_response_func`, `at_response` 的字符串将传入 `response_func`, 由用户自行处理。

函数定义:

```
void at_register_response_func (at_custom_response_func_type response_func)
```

参数:

`at_custom_response_func_type` : `at_response` 的回调函数

返回:

无



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. 参数结构体和宏定义

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;           // 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.
    sint16 freq_offset; // AP's frequency offset
};

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

4. WiFi event 结构体

```
enum {
    EVENT_STAMODE_CONNECTED = 0,
    EVENT_STAMODE_DISCONNECTED,
    EVENT_STAMODE_AUTHMODE_CHANGE,
    EVENT_STAMODE_GOT_IP,
    EVENT_STAMODE_DHCP_TIMEOUT,
    EVENT_SOFTAPMODE_STACONNECTED,
    EVENT_SOFTAPMODE_STADISCONNECTED,
```




```
EVENT_SOFTAPMODE_PROBEREQRECVED,
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,
    REASON_AUTH_FAIL             = 202,
    REASON_ASSOC_FAIL            = 203,
    REASON_HANDSHAKE_TIMEOUT     = 204,
};

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 struct {
    int rssi;
    uint8 mac[6];
} Event_SoftAPMode_ProbeReqRecved_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_SoftAPMode_ProbeReqRecved_t   ap_probereqrecved;
    } Event_Info_u;

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

```

5. 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_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_ESPTOUCH_AIRKISS,
} sc_type;

```

6.3. 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 }
```

6.4. espconn 参数

1. 回调函数

```
/** callback prototype to inform about events for a espconn */  
typedef void (* espconn_rcv_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 remote_port;  
    int local_port;  
    uint8 local_ip[4];  
    uint8 remote_ip[4];  
    espconn_connect_callback connect_callback;  
    espconn_reconnect_callback reconnect_callback;  
    espconn_connect_callback disconnect_callback;
```



```
        espconn_connect_callback write_finish_fn;
    } esp_tcp;

typedef struct _esp_udp {
    int remote_port;
    int local_port;
    uint8 local_ip[4];
    uint8 remote_ip[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_KEEPAIVE = 0x08,
    ESPCONN_END
}

enum espconn_level{
    ESPCONN_KEEPIIDLE,
    ESPCONN_KEEPIINTVL,
    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
};
```

6.5. 中断相关宏定义

```
/* interrupt related */
#define ETS_SPI_INUM      2
#define ETS_GPIO_INUM    4
#define ETS_UART_INUM     5
#define ETS_UART1_INUM    5
#define ETS_FRC_TIMER1_INUM 9

/* disable all interrupts */
#define ETS_INTR_LOCK()    ets_intr_lock()
/* enable all interrupts */
#define ETS_INTR_UNLOCK()  ets_intr_unlock()
```



```
/* register interrupt handler of frc timer1 */
#define ETS_FRC_TIMER1_INTR_ATTACH(func, arg) \
ets_isr_attach(ETS_FRC_TIMER1_INUM, (func), (void *)(arg))

/* register interrupt handler of GPIO */
#define ETS_GPIO_INTR_ATTACH(func, arg) \
ets_isr_attach(ETS_GPIO_INUM, (func), (void *)(arg))

/* register interrupt handler of UART */
#define ETS_UART_INTR_ATTACH(func, arg) \
ets_isr_attach(ETS_UART_INUM, (func), (void *)(arg))

/* register interrupt handler of SPI */
#define ETS_SPI_INTR_ATTACH(func, arg) \
ets_isr_attach(ETS_SPI_INUM, (func), (void *)(arg))

/* enable a interrupt */
#define ETS_INTR_ENABLE(inum)    ets_isr_unmask((1<<inum))
/* disable a interrupt */
#define ETS_INTR_DISABLE(inum)  ets_isr_mask((1<<inum))

/* enable SPI interrupt */
#define ETS_SPI_INTR_ENABLE()    ETS_INTR_ENABLE(ETS_SPI_INUM)

/* enable UART interrupt */
#define ETS_UART_INTR_ENABLE()   ETS_INTR_ENABLE(ETS_UART_INUM)
/* disable UART interrupt */
#define ETS_UART_INTR_DISABLE()  ETS_INTR_DISABLE(ETS_UART_INUM)

/* enable frc1 timer interrupt */
#define ETS_FRC1_INTR_ENABLE()   ETS_INTR_ENABLE(ETS_FRC_TIMER1_INUM)
/* disable frc1 timer interrupt */
#define ETS_FRC1_INTR_DISABLE()  ETS_INTR_DISABLE(ETS_FRC_TIMER1_INUM)

/* enable GPIO interrupt */
```




```
#define ETS_GPIO_INTR_ENABLE()    ETS_INTR_ENABLE(ETS_GPIO_INUM)
/* disable GPIO interrupt */
#define ETS_GPIO_INTR_DISABLE()  ETS_INTR_DISABLE(ETS_GPIO_INUM)
```



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_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_ANYEDGE = 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。

用户可向 Espressif Systems 申请详细的 UART 介绍文档。

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。

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。

用户可向 Espressif Systems 申请详细的 I2C 介绍文档。

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

本文档仅简单介绍 `pwm.h` 中的 PWM 相关接口，用户可向 Espressif Systems 申请详细的 PWM 介绍文档。

PWM 驱动接口函数不能跟 `hw_timer.c` 的接口同时使用，因为二者共用了同一个硬件定时器。

1. `pwm_init`

功能：

初始化 PWM，包括 GPIO 选择，周期和占空比。目前仅支持调用一次。

函数定义：

```
void pwm_init(  
    uint32 period,  
    uint8 *duty,  
    uint32 pwm_channel_num,  
    uint32 (*pin_info_list)[3])
```

参数：

`uint32 period` : PWM 周期;

`uint8 *duty` : 各路 PWM 的占空比

`uint32 pwm_channel_num`: PWM 通道数

`uint32 (*pin_info_list)[3]`: PWM 各通道的 GPIO 硬件参数。本参数是一个 $n * 3$ 的数组指针，数组中定义了 GPIO 的寄存器，对应 PIN 脚的 IO 复用值和 GPIO 对应的序号

返回：

无

示例：

初始化一个三通道的PWM：

```
uint32 io_info[][3] =  
    {{PWM_0_OUT_IO_MUX, PWM_0_OUT_IO_FUNC, PWM_0_OUT_IO_NUM},  
     {PWM_1_OUT_IO_MUX, PWM_1_OUT_IO_FUNC, PWM_1_OUT_IO_NUM},  
     {PWM_2_OUT_IO_MUX, PWM_2_OUT_IO_FUNC, PWM_2_OUT_IO_NUM}};  
  
pwm_init(light_param.pwm_period, light_param.pwm_duty, 3, io_info);
```

2. `pwm_start`

功能：

PWM 开始。每次更新 PWM 设置后，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 PWM 某个通道信号的占空比。设置各路 PWM 信号高电平所占的时间，duty 的范围随 PWM 周期改变，最大值为： $\text{Period} * 1000 / 45$ 。例如，1KHz PWM，duty 范围是：0 ~ 22222

注意：

设置完成后，需要调用 `pwm_start` 生效。

函数定义：

```
void pwm_set_duty(uint32 duty, uint8 channel)
```

参数：

uint32 duty：设置高电平时间参数，占空比的值为 $(\text{duty} * 45) / (\text{period} * 1000)$

uint8 channel：当前要设置的 PWM 通道，取值范围依据实际使用了几路 PWM，在 [IOT_Demo](#) 中取值在 `#define PWM_CHANNEL` 定义的范围内。

返回：

无

4. pwm_get_duty

功能：

获取某路 PWM 信号的 duty 参数，占空比的值为 $(\text{duty} * 45) / (\text{period} * 1000)$

函数定义：

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

参数：

uint8 channel：当前要查询的 PWM 通道，取值范围依据实际使用了几路 PWM，在 [IOT_Demo](#) 中取值在 `#define PWM_CHANNEL` 定义的范围内。

返回：

对应某路 PWM 信号的 duty 参数



5. pwm_set_period

功能：

设置 PWM 周期，单位：us。例如，1KHz PWM，参数为 1000 us。

注意：

设置完成后，需要调用 `pwm_start` 生效。

函数定义：

```
void pwm_set_period(uint32 period)
```

参数：

`uint32 period` : PWM 周期，单位：us

返回：

无

6. pwm_get_period

功能：

查询 PWM 周期

函数定义：

```
uint32 pwm_get_period(void)
```

参数：

无

返回：

PWM 周期，单位：us

7. get_pwm_version

功能：

查询 PWM 版本信息

函数定义：

```
uint32 get_pwm_version(void)
```

参数：

无

返回：

PWM 版本信息



8. 附录

8.1. ESPCONN 编程

可参考 Espressif BBS 提供的示例 <http://bbs.espressif.com/viewforum.php?f=21>

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 (`espconn_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 (`espconn_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](#))

3. espconn callback

注册函数	回调函数	说明
espconn_regist_connectcb	espconn_connect_callback	TCP 连接建立成功
espconn_regist_reconcb	espconn_reconnect_callback	TCP 连接发生异常而断开
espconn_regist_sentcb	espconn_sent_callback	TCP 或 UDP 数据发送完成
espconn_regist_recvcb	espconn_recv_callback	TCP 或 UDP 数据接收
espconn_regist_write_finish	espconn_write_finish_callback	数据成功写入 TCP 数据缓存
espconn_regist_disconcb	espconn_disconnect_callback	TCP 连接正常断开

注意

- 回调函数中传入的指针 [arg](#)，对应网络连接的结构体 [espconn](#) 指针。该指针为 SDK 内部维护的指针，不同回调传入的指针地址可能不一样，请勿依此判断网络连接。可根据 [espconn](#) 结构体中的 [remote_ip](#), [remote_port](#) 判断多连接中的不同网络传输。
- 如果 [espconn_connect](#) (或者 [espconn_secure_connect](#)) 失败，返回非零值，连接未建立，不会进入任何 [espconn](#) callback。
- 请勿在 [espconn](#) 任何回调中调用 [espconn_disconnect](#) (或者 [espconn_secure_disconnect](#)) 断开连接。如有需要，可以在 [espconn](#) 回调中使用触发任务的方式 ([system_os_task](#) 和 [system_os_post](#)) 调用 [espconn_disconnect](#) (或者 [espconn_secure_disconnect](#)) 断开连接。



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。

```
#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){
        os_printf("rtc time init...\r\n");
        rtc_time.magic = RTC_MAGIC;
        rtc_time.time_acc= 0;
        rtc_time.time_base = system_get_rtc_time();
        os_printf("time base : %d \r\n",rtc_time.time_base);
    }

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

if(5 == (cnt++)){
    os_printf("system restart\r\n");
    system_restart();
}else{
    os_printf("continue ... \r\n");
}
}

void user_init(void)
{
    rtc_count();
    os_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);
}
```




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 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， $\text{len} = 50 + \text{cnt} * 10$
- `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。

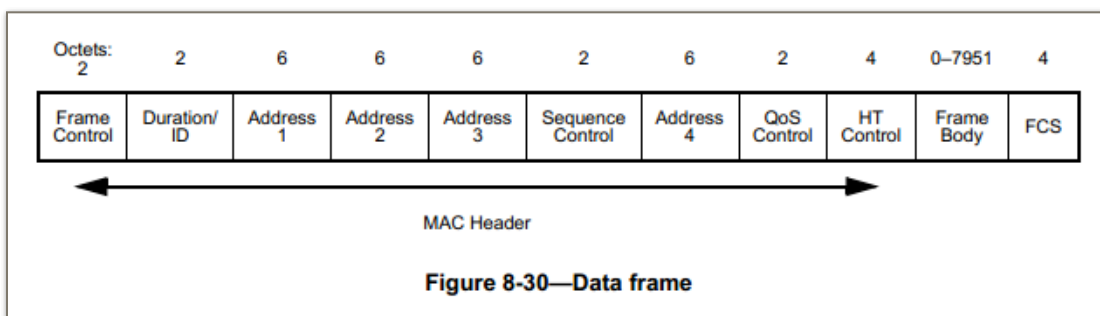
LEN == 12 的情况

- `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：



- 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 soft-AP 和 station 信道定义

虽然 ESP8266 支持 soft-AP + station 共存模式, 但是 ESP8266 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 ESP8266 soft-AP + station 模式时一些行为上的不便, 用户请注意。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) 若设置值合法有效, 该 API 将返回 true, 但信道号仍然会自动调节成与 ESP8266 station 接口一致, 在这个例子里也就是信道号为 6。因为 ESP8266 在硬件上只有一个信道, 由 ESP8266 station 与 soft-AP 接口共用。

情况二

- (1) 调用接口 `wifi_softap_set_config` 设置 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 将自动调整信道号与 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 连接断开。

情况三

- (1) 其他 station 与 ESP8266 soft-AP 建立连接
- (2) 如果 ESP8266 station 一直尝试扫描或连接某路由, 可能导致 ESP8266 softAP 端的连接断开。

因为 ESP8266 station 会遍历各个信道查找目标路由, 意味着 ESP8266 其实在不停切换信道, ESP8266 soft-AP 的信道也因此不停更改。这可能导致 ESP8266 softAP 端的原有连接断开。

这种情况, 用户可以通过设置定时器, 超时后调用 `wifi_station_disconnect` 停止 ESP8266 station 不断连接路由的尝试; 或者在初始配置时, 调用 `wifi_station_set_reconnect_policy` 和 `wifi_station_set_auto_connect` 禁止 ESP8266 station 尝试重连路由。



8.5. ESP8266 启动信息说明

ESP8266 启动时，将从 UART0 以波特率 74880 打印如下启动信息：

```
ets Jan 8 2013,rst cause:2, boot mode:(3,6)

load 0x4010f000, len 1264, room 16

tail 0

chksum 0x42

csum 0x42
```

其中可供用户参考的启动信息说明如下：

启动信息	说明
rst cause	1: power on
	2: external reset
	4: hardware watchdog-reset
boot mode 第一个参数	1：ESP8266 处于 UART-down 模式，可通过 UART 下载固件
	2：ESP8266 处于 Flash-boot 模式，从 Flash 启动运行
chksum	chksum 与 csum 值相等，表示启动过程中 Flash 读取正确