ESP8266 RTOS SDK

Programming Guide

Version 1.0.2

Espressif Systems IOT Team

Disclaimer and Copyright Notice

Information in this document, including URL references, is subject to change without notice.

THIS DOCUMENT IS PROVIDED AS IS WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. All liability, including liability for infringement of any proprietary rights, relating to use of information in this document is disclaimed. No licenses express or implied, by estoppel or otherwise, to any intellectual property rights are granted herein.

Table of Contents

1. Preambles

ESP8266 WiFi SoC offers a complete and self-contained Wi-Fi networking solution; it can be used to host the application or to offload Wi-Fi networking functions from another application processor. When ESP8266 hosts the application, it boots up directly from an external flash. In has integrated cache to improve the performance of the system in such applications. Alternately, serving as a Wi-Fi adapter, wireless internet access can be added to any microcontroller-based design with simple connectivity through UART interface or the CPU AHB bridge interface.

ESP8266EX is amongst the most integrated WiFi chip in the industry; it integrates the antenna switches, RF balun, power amplifier, low noise receive amplifier, filters, power management modules, it requires minimal external circuitry, and the entire solution, including front-end module, is designed to occupy minimal PCB area.

ESP8266EX also integrates an enhanced version of Tensilica's L106 Diamond series 32-bit processor, with on-chip SRAM, besides the Wi-Fi functionalities. ESP8266EX is often integrated with external sensors and other application specific devices through its GPIOs; codes for such applications are provided in examples in the SDK.

Sophisticated system-level features include fast sleep/wake context switching for energy-efficient VoIP, adaptive radio biasing for low-power operation, advance signal processing, and spur cancellation and radio co-existence features for common cellular, Bluetooth, DDR, LVDS, LCD interference mitigation.

The SDK based on ESP8266 IoT platform offers users an easy, fast and efficient way to develop IoT devices. This programming guide provides overview of the SDK as well as details on the API. It is written for embedded software developers to help them program on ESP8266 IoT platform.

2. Overview

The SDK provides a set of interfaces for data receive and transmit functions over the Wi-Fi and TCP/IP layer so programmers can focus on application development on the high level. Users can easily make use of the corresponding interfaces to realize data receive and transmit.

All networking functions on the ESP8266 IoT platform are realized in the library, and are not transparent to users. Instead, users can initialize the interface in `user_main.c`.

`void user_init(void)` is the default method provided. Users can add functions like firmware initialization, network parameters setting, and timer initialization in the interface.

Notice：

- We suggest to use a timer to check periodically.
- There are 15 task priorities in RTOS SDK, call freeRTOS API `xTaskCreate` to create a task. freeRTOS related introduction and APIs, please visit http://www.freertos.org
  - RTOS SDK has already taken some tasks： pp task priority 13， precise ets timer thread priority 12, lwip task priority 10, freeRTOS timer priority 2, idle priority 0
  - User task could use priority 1 ~ 9, please do NOT revise `FreeRTOSConfig.h`, task priorities depend on source code inside RTOS SDK, change `FreeRTOSConfig.h` will not take effect.

3. Software APIs

3.1. Software Timer

Timer APIs can be found: /esp_iot_rtos_sdk/include/espressif/Esp_timer.h.

Please be noted that os_timer APIs listed below are software timer, they are executed in task, so timer callback may not be precisely executed at the right time, it depends on priority.

- For the same timer, os_timer_arm (or os_timer_arm_us) cannot be invoked repeatedly. os_timer_disarm should be invoked first.
- os_timer_setfn can only be invoked when the timer is not enabled, i.e., after os_timer_disarm or before os_timer_arm (or os_timer_arm_us).

1. os_timer_arm

```
Function:
    Enable a millisecond timer.

Prototype:
    void os_timer_arm (
        ETSTimer *ptimer,
        uint32_t milliseconds,
        bool repeat_flag
    )

Parameters:
    ETSTimer *ptimer : Timer structure
    uint32_t milliseconds : Timing, Unit: millisecond, the maximum value allowed
    to input is 0x41893

    bool repeat_flag : Whether the timer will be invoked repeatedly or not

Return:
    null
```

2. os_timer_disarm

```
Function:
    Disarm timer

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

Parameters:

```
ETSTimer *ptimer : Timer structure
```

Return:

```
null
```

3. os_timer_setfn

Function:

Set timer callback function.

Note:

- For enabled timer, timer callback has to be set.
- Operating system scheduling is disabled in timer callback.

Prototype:

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

Parameters:

```
ETSTimer *ptimer : Timer structure
TESTimerFunc *pfunction : timer callback function
void *parg : callback function parameter
```

Return:

```
null
```

4. os_timer_arm_us

Function:

Enable a microsecond timer.

Prototype:

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

Parameters:

ETSTimer *ptimer : Timer structure

uint32_t microseconds : Timing, Unit: microsecond, the minimum value is 0x64, the maximum value allowed to input is 0xFFFFFFF

bool repeat_flag : Whether the timer will be invoked repeatedly or not

Return:

null

3.2. System APIs

1. system_get_sdk_version

Function:

Get SDK version

Prototype:

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

Parameter:

none

Return:

SDK version

Example:

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

2. system_restore

Function:

Reset to default settings of following APIs ： wifi_station_set_auto_connect, wifi_set_phy_mode, wifi_softap_set_config related, wifi_station_set_config related, and wifi_set_opmode.

Prototype:

```
void system_restore(void)
```

Parameters:

null

Return:

null

3. system_restart

Function:

Restart

Prototype:

`void system_restart(void)`

Parameters:

null

Return:

null

4. system_get_chip_id

Function:

Get chip ID

Prototype:

`uint32 system_get_chip_id (void)`

Parameters:

null

Return:

Chip ID

5. system_get_vdd33

Function:

Measure the power voltage of VDD3P3 pin 3 and 4, unit: 1/1024 V

Note:

- system_get_vdd33 can only be called when TOUT pin is suspended
- The 107th byte in esp_init_data_default.bin (0~127byte) is named as "vdd33_const" , when TOUT pin is suspended vdd33_const must be set as 0xFF, that is 255

Prototype:

`uint16 system_get_vdd33(void)`

Parameter:

none

Return:

power voltage of VDD33, unit: 1/1024 V

6. system_adc_read

Function:

Measure the input voltage of TOUT pin 6, unit: 1/1024 V

Note:

- system_adc_read is only available when wire TOUT pin to external circuitry, Input Voltage Range restricted to 0 ~ 1.0V.
- The 107th byte in esp_init_data_default.bin(0~127byte) is named as "vdd33_const", and when wire TOUT pin to external circuitry, the vdd33_const must be set as real power voltage of VDD3P3 pin 3 and 4.
- The range of operating voltage of ESP8266 is 1.8V~3.6V, the unit of vdd33_const is 0.1V, so effective value range of vdd33_const is [18, 36].

Prototype:

```
uint16 system_adc_read(void)
```

Parameter:

none

Return:

input voltage of TOUT pin 6, unit: 1/1024 V

7. system_deep_sleep

Function:

Configures chip for deep-sleep mode. When the device is in deep-sleep, it automatically wakes up periodically; the period is configurable. Upon waking up, the device boots up from user_init.

Note:

- Hardware has to support deep-sleep wake up (XPD_DCDC connects to EXT_RSTB with 0R).
- system_deep_sleep(0): there is no wake up timer; in order to wakeup, connect a GPIO to pin RST, the chip will wake up by a falling-edge on pin RST

Prototype:

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

Parameters:

uint32 time_in_us : during the time (us) device is in deep-sleep

Return:

null

8. system_deep_sleep_set_option

Function:

Call this API before system_deep_sleep to set what the chip will do when next deep-sleep wake up.

Prototype:

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

Parameter:

uint8 option :

deep_sleep_set_option(0): Radio calibration after deep-sleep wake up depends on esp_init_data_default.bin (0~127byte) byte 108.

deep_sleep_set_option(1): Radio calibration is done after deep-sleep wake up; this increases the current consumption.

deep_sleep_set_option(2): No radio calibration after deep-sleep wake up; this reduces the current consumption.

deep_sleep_set_option(4): Disable RF after deep-sleep wake up, just like modem sleep; this has the least current consumption; the device is not able to transmit or receive data after wake up.

Return:

```
true  : succeed
false : fail
```

9. system_phy_set_rfoption

Function:

Enable RF or not when wakeup from deep-sleep.

Note:

- This API can only be called in user_rf_pre_init.
- Function of this API is similar to system_deep_sleep_set_option, if they are both called, it will disregard system_deep_sleep_set_option which is called before deep-sleep, and refer to system_phy_set_rfoption which is called when deep-sleep wake up.
- Before calling this API, system_deep_sleep_set_option should be called once at least.

Prototype:

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

Parameter:

uint8 option :

system_phy_set_rfoption(0) : Radio calibration after deep-sleep wake up depends on esp_init_data_default.bin (0~127byte) byte 108.

system_phy_set_rfoption(1) : Radio calibration is done after deep-sleep wake up; this increases the current consumption.

system_phy_set_rfoption(2) : No radio calibration after deep-sleep wake up; this reduces the current consumption.

system_phy_set_rfoption(4) : Disable RF after deep-sleep wake up, just like modem sleep; this has the least current consumption; the device is not able to transmit or receive data after wake up.

Return:

none

10. system_phy_set_max_tpw

Function:

Set maximum value of RF TX Power, unit : 0.25dBm

Prototype:

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

Parameter:

uint8 max_tpw : maximum value of RF Tx Power, unit : 0.25dBm, range [0, 82]

it can be set refer to the 34th byte (target_power_qdb_0) of esp_init_data_default.bin(0~127byte)

Return:

none

11. system_phy_set_tpw_via_vdd33

Function:

Adjust RF TX Power according to VDD33, unit : 1/1024 V

Note:

When TOUT pin is suspended, VDD33 can be got by system_get_vdd33;

When wire TOUT pin to external circuitry, system_get_vdd33 can not be used.

Prototype:

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

```
Parameter:
    uint16 vdd33 : VDD33, unit : 1/1024V, range [1900, 3300]

Return:
    none
```

12. system_print_meminfo

```
Function:
    Print memory information, including data/rodata/bss/heap

Prototype:
    void system_print_meminfo (void)

Parameters:
    null

Return:
    null
```

13. system_get_free_heap_size

```
Function:
    Get free heap size

Prototype:
    uint32 system_get_free_heap_size(void)

Parameters:
    null

Return:
    uint32 : available heap size
```

14. system_get_time

```
Function:
    Get system time (us).

Prototype:
    uint32 system_get_time(void)

Parameter:
    null

Return:
    System time in microsecond.
```

15. system_get_rtc_time

Function: Get RTC time, as denoted by the number of RTC clock periods.

Example:

If system_get_rtc_time returns 10 (it means 10 RTC cycles), and system_rtc_clock_cali_proc returns 5.75 (means 5.75us per RTC cycle), then the real time is 10 x 5.75 = 57.5 us.

Note:

System time will return to zero because of system_restart, but RTC still goes on.

- reset by pin EXT_RST : RTC memory won't change, RTC timer returns to zero
- watchdog reset : RTC memory won't change, RTC timer won't change
- system_restart : RTC memory won't change, RTC timer won't change
- power on : RTC memory is random value, RTC timer starts from zero
- reset by pin CHIP_EN : RTC memory is random value, RTC timer starts from zero

Prototype:

```
uint32 system_get_rtc_time(void)
```

Parameter:

null

Return:

RTC time

16. system_rtc_clock_cali_proc

Function:

Get RTC clock period.

Note:

RTC clock period has decimal part.

RTC clock period will change according to temperature, so RTC timer is not very precise.

Prototype:

```
uint32 system_rtc_clock_cali_proc(void)
```

Parameter:

null

Return:

RTC clock period (in us), bit11~ bit0 are decimal. ((RTC_CAL * 100)>> 12)

Note:

see RTC demo in Appendix.

17. system_rtc_mem_write

Function:

During deep sleep, only RTC still working, so maybe we need to save some user data in RTC memory. Only user data area can be used by user.

```
|<--------system data--------->|<-----------------user data--------------->|
|          256 bytes           |                  512 bytes                 |
```

Note:

RTC memory is 4 bytes aligned for read and write operations. Parameter des_addr means block number(4 bytes per block). So, if we want to save some data at the beginning of user data area, des_addr will be 256/4 = 64, save_size will be data length.

Prototype:

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

Parameter:

```
uint32 des_addr  :  destination address (block number) in RTC memory,
des_addr >=64
void * src_addr  :  data pointer.
uint32 save_size :  data length ( byte)
```

Return:

```
true:  succeed
false: fail
```

18. system_rtc_mem_read

Function:

Read user data from RTC memory. Only user data area should be accessed by the user.

```
|<--------system data--------->|<-----------------user data--------------->|
|          256 bytes           |                  512 bytes                 |
```

Note:

RTC memory is 4 bytes aligned for read and write operations. Parameter src_addr means block number(4 bytes per block). So, to read data from the beginning of user data area, src_addr will be 256/4=64, save_size will be data length.

Prototype:

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

Parameter:

```
uint32 src_addr  :  source address (block number) in rtc memory, src_addr >=
64
void * des_addr  :  data pointer
uint32 save_size :  data length, byte
```

Return:

```
true:  succeed
false: fail
```

19. system_uart_swap

Function:

UART0 swap. Use MTCK as UART0 RX, MTDO as UART0 TX, so ROM log will not output from this new UART0. We also need to use MTDO (U0CTS) and MTCK (U0RTS) as UART0 in hardware.

Prototype:

```
void system_uart_swap (void)
```

Parameter:

null

Return:

null

20. system_uart_de_swap

Function:

Disable UART0 swap. Use original UART0, not MTCK and MTDO.

Prototype:

```
void system_uart_de_swap (void)
```

Parameter:

null

Return:

null

21. system_get_boot_version

Function:

Get version info of boot

Prototype:

```
uint8 system_get_boot_version (void)
```

Parameter:

null

Return:

Version info of boot.

Note:

If boot version >= 3 , you could enable boot enhance mode (refer to system_restart_enhance)

22. system_get_userbin_addr

Function: Get address of the current running user bin (user1.bin or user2.bin).

Prototype:

```
uint32 system_get_userbin_addr (void)
```

Parameter:

null

Return:

Start address info of the current running user bin.

23. system_get_boot_mode

Function: Get boot mode.

Prototype:

```
uint8 system_get_boot_mode (void)
```

Parameter:

null

Return:

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

Note:

Enhance boot mode: can load and run FW at any address;

Normal boot mode: can only load and run normal user1.bin (or user2.bin).

24. system_restart_enhance

Function:

Restarts system, and enters enhance boot mode.

Prototype:

```
bool system_restart_enhance(
    uint8 bin_type,
    uint32 bin_addr
)
```

Parameter:

```
uint8 bin_type : type of bin
#define SYS_BOOT_NORMAL_BIN  0  // user1.bin or user2.bin
#define SYS_BOOT_TEST_BIN  1    // can only be Espressif test bin
uint32 bin_addr : start address of bin file
```

Return:

```
true:  succeed
false: Fail
```

Note:

SYS_BOOT_TEST_BIN is for factory test during production; you can apply for the test bin from Espressif Systems.

25. system_get_flash_size_map

Function:

Get current flash size and flash map.

Flash map depends on the selection when compiling, more details in document “2A-ESP8266__IOT_SDK_User_Manual”

Structure:

```
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
};
```

Prototype:

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

Parameter:

none

Return:

flash map

26. system_get_rst_info

Function:

Get information about current startup.

Structure:

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

    struct rst_info {
        uint32 reason;   // enum rst_reason
        uint32 exccause;
        uint32 epc1;
        uint32 epc2;
        uint32 epc3;
        uint32 excvaddr;
        uint32 depc;
    };
Prototype:
    struct rst_info* system_get_rst_info(void)

Parameter:
    none

Return:
    Information about startup.
```

27. os_delay_us

```
Function:
    Time delay, max : 65535 us

Prototype:
    void os_delay_us(uint16 us)

Parameter:
    uint16 us – time, unit: us

Return:
    none
```

28. os_install_putc1

```
Function:
    Register print output function.

Prototype:
    void os_install_putc1(void(*p)(char c))

Parameter:
    void(*p)(char c) – pointer of print function

Return:
    none

Example:

    os_install_putc1((void *)uart1_write_char) in uart_init will set printf to
    be output from UART 1, otherwise, printf default output from UART 0.
```

29. os_putc

```
Function:
    Print a char, default output from UART0
```

Prototype:

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

Parameter:

`char c` – character

Return:

none

3.3. SPI Flash Related APIs

More details about flash read/write operation in documentation “99A-SDK-Espressif IOT Flash RW Operation” http://bbs.espressif.com/viewtopic.php?f=21&t=413

1. spi_flash_get_id

Function:

Get ID info of spi flash

Prototype:

```
uint32 spi_flash_get_id (void)
```

Parameters:

null

Return:

SPI flash ID

2. spi_flash_erase_sector

Function:

Erase sector in flash

Prototype:

```
SpiFlashOpResult spi_flash_erase_sector (uint16 sec)
```

Parameters:

`uint16 sec` : Sector number, the count starts at sector 0, 4KB per sector.

Return:

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

3. spi_flash_write

Function:

Write data to flash. Flash read/write has to be 4-bytes aligned.

Prototype:

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

Parameters:

```
uint32 des_addr  : destination address in flash.
uint32 *src_addr : source address of the data.
uint32 size      :length of data
```

Return:

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

4. spi_flash_read

Function:

Read data from flash. Flash read/write has to be 4-bytes aligned.

Prototype:

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

Parameters:

```
uint32  src_addr: source address in flash
uint32 *des_addr: destination address to keep data.
uint32  size:     length of data
```

Return:

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

Example:

```
uint32 value;

uint8 *addr = (uint8 *)&value;

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

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

5. system_param_save_with_protect

Function:

Write data into flash with protection. Flash read/write has to be 4-bytes aligned.

Protection of flash read/write : use 3 sectors (4KBytes per sector) to save 4KB data with protect, sector 0 and sector 1 are data sectors, back up each other, save data alternately, sector 2 is flag sector, point out which sector is keeping the latest data, sector 0 or sector 1.

Note:

More details about protection of flash read/write in document "99A-SDK-Espressif IOT Flash RW Operation" http://bbs.espressif.com/viewtopic.php?f=21&t=413

Prototype:

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

Parameter:

`uint16 start_sec` : start sector (sector 0) of the 3 sectors which used for flash read/write protection.

For example, in IOT_Demo we could use the 3 sectors (3 * 4KB) starts from flash 0x3D000 for flash read/write protection, so the parameter start_sec should be 0x3D

void *param : pointer of data need to save
uint16 len : data length, should less than a sector which is 4 * 1024

Return:

true, succeed;

false, fail

6. system_param_load

Function:

Read data which saved into flash with protection. Flash read/write has to be 4-bytes aligned.

Protection of flash read/write : use 3 sectors (4KBytes per sector) to save 4KB data with protect, sector 0 and sector 1 are data sectors, back up each other, save data alternately, sector 2 is flag sector, point out which sector is keeping the latest data, sector 0 or sector 1.

Note:

More details about protection of flash read/write in document “99A-SDK-Espressif IOT Flash RW Operation” http://bbs.espressif.com/viewtopic.php?f=21&t=413

Prototype:

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

Parameter:

uint16 start_sec : start sector (sector 0) of the 3 sectors which used for flash read/write protection. It can not sector 1 or sector 2.

For example, in IOT_Demo we could use the 3 sectors (3 * 4KB) starts from flash 0x3D000 for flash read/write protection, so the parameter start_sec is 0x3D, can not be 0x3E or 0x3F.

uint16 offset : offset of data saved in sector
void *param : data pointer
uint16 len : data length, offset + len ≤ 4 * 1024

```
Return:
    true, succeed;

    false, fail
```

3.4. Wi-Fi Related APIs

wifi_station APIs and other APIs which set/get configuration of ESP8266 station can only be called if ESP8266 station is enabled.

wifi_softap APIs and other APIs which set/get configuration of ESP8266 soft-AP can only be called if ESP8266 soft-AP is enabled.

Flash system parameter area is the last 16KB of flash.

1. wifi_get_opmode

```
Function:
    get WiFi current operating mode

Prototype:
    uint8 wifi_get_opmode (void)

Parameters:
    null

Return:
    WiFi working modes:
        0x01: station mode
        0x02: soft-AP mode
        0x03: station+soft-AP
```

2. wifi_get_opmode_default

```
Function:
    get WiFi operating mode that saved in flash

Prototype:
    uint8 wifi_get_opmode_default (void)

Parameters:
    null

Return:
    WiFi working modes:
        0x01: station mode
        0x02: soft-AP mode
        0x03: station+soft-AP
```

3. wifi_set_opmode

Function:

Sets WiFi working mode as station, soft-AP or station+soft-AP, and save it to flash. Default is soft-AP mode.

Note:

Versions before esp_iot_sdk_v0.9.2, need to call `system_restart()` after this api; after esp_iot_sdk_v0.9.2, need not to restart.

This configuration will be saved in flash system parameter area if changed.

Prototype:

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

Parameters:

`uint8 opmode`: WiFi operating modes:

```
0x01: station mode
0x02: soft-AP mode
0x03: station+soft-AP
```

Return:

```
true:  succeed
false: fail
```

4. wifi_set_opmode_current

Function:

Sets WiFi working mode as station, soft-AP or station+soft-AP, and won't save to flash

Prototype:

```
bool wifi_set_opmode_current (uint8 opmode)
```

Parameters:

`uint8 opmode`: WiFi operating modes:

```
0x01: station mode
0x02: soft-AP mode
0x03: station+soft-AP
```

Return:

```
true:  succeed
false: fail
```

5. wifi_station_get_config

Function:

Get WiFi station current configuration

Prototype:

`bool wifi_station_get_config (struct station_config *config)`

Parameters:

`struct station_config *config` : WiFi station configuration pointer

Return:

true: succeed

false: fail

6. wifi_station_get_config_default

Function:

Get WiFi station configuration that saved in flash

Prototype:

`bool wifi_station_get_config_default (struct station_config *config)`

Parameters:

`struct station_config *config` : WiFi station configuration pointer

Return:

true: succeed

false: fail

7. wifi_station_set_config

Function:

Set WiFi station configuration, and save it to flash

Note:

- This API can be called only if ESP8266 station is enabled.
- If `wifi_station_set_config` is called in `user_init` , there is no need to call `wifi_station_connect` after that, ESP8266 will connect to router automatically; otherwise, need `wifi_station_connect` to connect.
- In general, `station_config.bssid_set` need to be 0, otherwise it will check bssid which is the MAC address of AP.
- This configuration will be saved in flash system parameter area if changed.

Prototype:

`bool wifi_station_set_config (struct station_config *config)`

Parameters:

struct station_config *config: WiFi station configuration pointer

Return:

true: succeed
false: fail

Example:

```
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

Function:

Set WiFi station configuration, won't save to flash

Note:

- This API can be called only if ESP8266 station is enabled.
- If wifi_station_set_config_current is called in user_init , there is no need to call wifi_station_connect after that, ESP8266 will connect to router automatically; otherwise, need wifi_station_connect to connect.
- In general, station_config.bssid_set need to be 0, otherwise it will check bssid which is the MAC address of AP.

Prototype:

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

Parameters:

struct station_config *config: WiFi station configuration pointer

Return:

true: succeed
false: fail

9. wifi_station_connect

Function:

To connect WiFi station to AP

Note:

- If ESP8266 has already connected to a router, then we need to call wifi_station_disconnect first, before calling wifi_station_connect.
- Do not call this API in user_init. This API need to be called after system initialize done and ESP8266 station enable.

Prototype:

```
bool wifi_station_connect (void)
```

Parameters:

null

Return:

true: succeed
false: fail

10. wifi_station_disconnect

Function:

Disconnects WiFi station from AP

Note:

Do not call this API in user_init. This API need to be called after system initialize done and ESP8266 station enable.

Prototype:

```
bool wifi_station_disconnect (void)
```

Parameters:

null

Return:

true: succeed
false: fail

11. wifi_station_get_connect_status

Function:

Get connection status of WiFi station to AP

Prototype:

```
uint8 wifi_station_get_connect_status (void)
```

Parameters:

null

Return:

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

12. wifi_station_scan

Function:

Scan all available APs

Note:

Do not call this API in user_init. This API need to be called after system initialize done and ESP8266 station enable.

Prototype:

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

Structure:

```
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.
};
```

Parameters:

struct scan_config *config: AP config for scan

    if config==null: scan all APs
    if config.ssid==null && config.bssid==null && config.channel!=null:
        ESP8266 will scan the specific channel.
    scan_done_cb_t cb: callback function after scan

Return:

true: succeed

false: fail

13. scan_done_cb_t

Function:

Callback function for wifi_station_scan

Prototype:

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

Parameters:

void *arg: information of APs that be found, refer to struct bss_info

STATUS status: get status

Return:

null

Example:

```
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

Function:

Sets the number of APs that will be cached for ESP8266 station mode. Whenever ESP8266 station connects to an AP, it keeps caches a record of this AP's SSID and password. The cached ID index starts from 0.

Note:

This configuration will be saved in flash system parameter area if changed.

Prototype:

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

Parameters:

uint8 ap_number: the number of APs can be recorded (MAX: 5)

Return:

true: succeed
false: fail

15. wifi_station_get_ap_info

Function:

Get information of APs recorded by ESP8266 station.

Prototype:

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

Parameters:

`struct station_config config[]`: information of APs, array size has to be 5.

Return:

The number of APs recorded.

Example:

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

16. wifi_station_ap_change

Function:

Switch ESP8266 station connection to AP as specified

Prototype:

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

Parameters:

`uint8 new_ap_id` : AP’s record id, start counting from 0.

Return:

true: succeed
false: fail

17. wifi_station_get_current_ap_id

Function:

Get the current record id of AP.

Prototype:

```
uint8 wifi_station_get_current_ap_id ();
```

Parameter:

null

Return:

The index of the AP, which ESP8266 is currently connected to, in the cached AP list.

18. wifi_station_get_auto_connect

Function:

Checks if ESP8266 station mode will connect to AP (which is cached) automatically or not when it is powered on.

Prototype:

```
uint8 wifi_station_get_auto_connect(void)
```

Parameter:

null

Return:

```
0:     wil not connect to AP automatically;
Non-0: will connect to AP automatically.
```

19. wifi_station_set_auto_connect

Function:

Set whether ESP8266 station will connect to AP (which is recorded) automatically or not when power on. Default to enable auto-connect.

Note:

Call this API in user_init, it is effective in this current power on; call it in other place, it will be effective in next power on.

This configuration will be saved in flash system parameter area if changed.

Prototype:

```
bool wifi_station_set_auto_connect(uint8 set)
```

Parameter:

uint8 set: Automatically connect or not:

    0: will not connect automatically
    1: to connect automatically

Return:

```
true:  succeed
false: fail
```

20. wifi_station_dhcpc_start

Function:

Enable ESP8266 station DHCP client.

```
Note:
    DHCP default enable.

    This configuration interacts with static IP API (wifi_set_ip_info):

        If enable DHCP, static IP will be disabled;

        If enable static IP, DHCP will be disabled;

    This will depend on the last configuration.

Prototype:
    bool wifi_station_dhcpc_start(void)

Parameter:
    null

Return:
    true:  succeed
    false: fail
```

21. wifi_station_dhcpc_stop

```
Function:
    Disable ESP8266 station DHCP client.

Note:
    DHCP default enable.

Prototype:
    bool wifi_station_dhcpc_stop(void)

Parameter:
    null

Return:
    true:  succeed
    false: fail
```

22. wifi_station_dhcpc_status

```
Function: Get ESP8266 station DHCP client status.

Prototype:
    enum dhcp_status wifi_station_dhcpc_status(void)

Parameter:
    null
```

Return:

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

23. wifi_station_set_reconnect_policy

Function:

Set whether reconnect or not when ESP8266 station disconnected from AP, will reconnect by default.

Note:

We suggest to call this API in `user_init`

This API can only be called when ESP8266 station enable.

Prototype:

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

Parameter:

`bool set` - true, enable reconnect; false, disable reconnect

Return:

true: succeed

false: fail

24. wifi_station_get_reconnect_policy

Function:

Get whether reconnect or not when ESP8266 station disconnected from AP

Note:

This API can only be called when ESP8266 station enable.

Prototype:

```
bool wifi_station_get_reconnect_policy(void)
```

Parameter:

none

Return:

true: enable reconnect

false: disable reconnect

25. wifi_softap_get_config

Function:

Get WiFi soft-AP current configuration

Prototype:

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

Parameter:

`struct softap_config *config` : ESP8266 soft-AP config

Return:

true: succeed
false: fail

26. wifi_softap_get_config_default

Function:

Get WiFi soft-AP configuration that saved in flash

Prototype:

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

Parameter:

`struct softap_config *config` : ESP8266 soft-AP config

Return:

true: succeed
false: fail

27. wifi_softap_set_config

Function:

Set WiFi soft-AP configuration and save it to flash

Note:

- This API can be called only if ESP8266 soft-AP is enabled.
- This configuration will be saved in flash system parameter area if changed.
- In soft-AP + station mode, ESP8266 soft-AP will adjust its channel configuration to be as same as ESP8266. More details in appendix or BBS http://bbs.espressif.com/viewtopic.php?f=10&t=324

Prototype:

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

Parameter:

```
struct softap_config *config :  WiFi soft-AP configuration pointer
```

Return:

```
true:  succeed
false: fail
```

28. wifi_softap_set_config_current

Function:

Set WiFi soft-AP configuration, won't save it to flash

Note:

- This API can be called only if ESP8266 soft-AP is enabled.
- In soft-AP + station mode, ESP8266 soft-AP will adjust its channel configuration to be as same as ESP8266. More details in appendix or BBS http://bbs.espressif.com/viewtopic.php?f=10&t=324

Prototype:

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

Parameter:

```
struct softap_config *config :  WiFi soft-AP configuration pointer
```

Return:

```
true:  succeed
false: fail
```

29. wifi_softap_get_station_num

Function:

Number count of stations which connected to ESP8266 soft-AP

Prototype:

```
uint8 wifi_softap_get_station_num(void)
```

Parameter:

none

Return:

how many stations connected to ESP8266 soft-AP

30. wifi_softap_get_station_info

Function:

Get connected station devices under soft-AP mode, including MAC and IP

Note:

This API can not get static IP, it can only be used when DHCP enabled.

Prototype:

```
struct station_info ∗ wifi_softap_get_station_info(void)
```

Input Parameters:

null

Return:

`struct station_info∗` : station information structure

31. wifi_softap_free_station_info

Function:

Frees the struct `station_info` by calling the `wifi_softap_get_station_info` function

Prototype:

```
void wifi_softap_free_station_info(void)
```

Input Parameters:

null

Return:

null

Examples 1 (Getting MAC and IP information):

```
struct station_info ∗ station = wifi_softap_get_station_info();
struct station_info ∗ next_station;
while(station) {
    printf(bssid : MACSTR, ip : IPSTR/n,
            MAC2STR(station->bssid), IP2STR(&station->ip));
    next_station = STAILQ_NEXT(station, next);
    os_free(station);    // Free it directly
    station = next_station;
}
```

Examples 2 (Getting MAC and IP information):

```
struct station_info ∗ station = wifi_softap_get_station_info();
while(station){
    printf(bssid : MACSTR, ip : IPSTR/n,
            MAC2STR(station->bssid), IP2STR(&station->ip));
    station = STAILQ_NEXT(station, next);
}
wifi_softap_free_station_info();    // Free it by calling functions
```

32. wifi_softap_dhcps_start

Function: Enable ESP8266 soft-AP DHCP server.

Note:

DHCP default enable.

This configuration interacts with static IP API (wifi_set_ip_info):

If enable DHCP, static IP will be disabled;

If enable static IP, DHCP will be disabled;

This will depend on the last configuration.

Prototype:

```
bool wifi_softap_dhcps_start(void)
```

Parameter:

null

Return:

true: succeed
false: fail

33. wifi_softap_dhcps_stop

Function: Disable ESP8266 soft-AP DHCP server.

Note: DHCP default enable.

Prototype:

```
bool wifi_softap_dhcps_stop(void)
```

Parameter:

null

Return:

true: succeed
false: fail

34. wifi_softap_set_dhcps_lease

Function:

Set the IP range that can be got from ESP8266 soft-AP DHCP server.

Note:

- IP range has to be in the same sub-net with ESP8266 soft-AP IP address
- This API can only be called during DHCP server disable (wifi_softap_dhcps_stop)

- This configuration only take effect on next wifi_softap_dhcps_start, if then wifi_softap_dhcps_stop is called; user needs to call this API to set IP range again if needed, then call wifi_softap_dhcps_start to take effect.

Prototype:

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

Parameter:

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

Return:

true: succeed
false: fail

Example:

```
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);
}
```

or

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

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

        dhcps_lease_test();

        wifi_softap_dhcps_start();
    }
```

35. wifi_softap_dhcps_status

Function: Get ESP8266 soft-AP DHCP server status.

Prototype:

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

Parameter:

null

Return:

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

36. wifi_softap_set_dhcps_offer_option

Function:

Set ESP8266 soft-AP DHCP server option.

Structure:

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

Prototype:

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

Parameter:

uint8 level -   OFFER_ROUTER set router option

void* optarg -   default to be enable

```
        bit0, 0 disable router information from ESP8266 softAP DHCP server;
        bit0, 1 enable router information from ESP8266 softAP DHCP server;
```

Return:

```
    true  : succeed
    false : fail
```

Example:

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

37. wifi_set_phy_mode

Fuction: Set ESP8266 physical mode (802.11b/g/n).

Note: ESP8266 soft-AP only support bg.

Prototype:

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

Parameter:

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

Return:

```
    true  : succeed
    false : fail
```

38. wifi_get_phy_mode

Function:

Get ESP8266 physical mode (802.11b/g/n)

Prototype:

```
    enum phy_mode wifi_get_phy_mode(void)
```

Parameter:

null

Return:

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

39. wifi_get_ip_info

Function:

Get IP info of WiFi station or soft-AP interface

Prototype:

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

Parameters:

uint8 if_index : the interface to get IP info: 0x00 for STATION_IF, 0x01 for SOFTAP_IF.

struct ip_info *info : pointer to get IP info of a certain interface

Return:

true: succeed
false: fail

40. wifi_set_ip_info

Function:

Set IP address of ESP8266 station or soft-AP

Note:

To set static IP, please disable DHCP first (wifi_station_dhcpc_stop or wifi_softap_dhcps_stop):

If enable static IP, DHCP will be disabled;

If enable DHCP, static IP will be disabled;

Prototype:

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

Prototype:

```
uint8 if_index  : set station IP or soft-AP IP
    #define STATION_IF      0x00
    #define SOFTAP_IF       0x01
struct ip_info *info  :  IP information
```

Example:

```
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();
```

Return:

```
true:  succeed
false: fail
```

41. wifi_set_macaddr

Function:

Sets MAC address

Note:

- This API can only be called in user_init.
- ESP8266 soft-AP and station have different MAC address, please don’t set them to be the same one.

Prototype:

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

Parameter:

```
uint8 if_index  : set station MAC or soft-AP mac
    #define STATION_IF      0x00
    #define SOFTAP_IF       0x01
uint8 *macaddr  :  MAC address
```

Example:

```
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);
```

Return:

```
true:  succeed
false: fail
```

42. wifi_get_macaddr

Function: get MAC address

Prototype:

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

Parameter:

```
uint8 if_index   :  set station MAC or soft-AP mac
    #define STATION_IF      0x00
    #define SOFTAP_IF       0x01
uint8 *macaddr :  MAC address
```

Return:

```
true:  succeed
false: fail
```

43. wifi_status_led_install

Function:

Installs WiFi status LED

Prototype:

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

Parameter:

```
uint8 gpio_id   : GPIO number
uint8 gpio_name : GPIO mux name
uint8 gpio_func : GPIO function
```

Return:

null

44. wifi_status_led_uninstall

Function: Uninstall WiFi status LED

Prototype:

```
void wifi_status_led_uninstall ()
```

Parameter:

null

Return:

null

45. wifi_set_event_handler_cb

Function:

Register Wi-Fi event handler

Prototype:

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

Parameter:

`wifi_event_handler_cb_t cb` - callback

Return:

none

Example:

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

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

3.5. Upgrade (FOTA) APIs

1. system_upgrade_userbin_check

Function:

Checks user bin

Prototype:

```
uint8 system_upgrade_userbin_check()
```

Parameter:

none

Return:

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

2. system_upgrade_flag_set

Function:

Sets upgrade status flag.

Note:

After downloading new firmware succeed，set flag to be UPGRADE_FLAG_FINISH, call system_upgrade_reboot to reboot to run new firmware.

Prototype:

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

Parameter:

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

Return:

null

3. system_upgrade_flag_check

Function:

Gets upgrade status flag.

Prototype:

```
uint8 system_upgrade_flag_check()
```

Parameter:

null

Return:

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

4. system_upgrade_reboot

Function: reboot system and use new version

Prototype:

```
void system_upgrade_reboot (void)
```

Parameters:

none

Return:

none

3.6. Sniffer Related APIs

1. wifi_promiscuous_enable

Function:

Enable promiscuous mode for sniffer

Note:

(1)promiscuous mode can only be enabled in station mode.

(2)During promiscuous mode (sniffer) , ESP8266 station and soft-AP are disabled.

(3)Before enable promiscuous mode，please call wifi_station_disconnect first

(4)Don’t call any other APIs during sniffer，please call
wifi_promiscuous_enable(0) first.

Prototype:

`void wifi_promiscuous_enable(uint8 promiscuous)`

Parameter:

uint8 promiscuous :
0: disable promiscuous;
1: enable promiscuous

Return:

null

2. wifi_promiscuous_set_mac

Function:

Set MAC address filter for sniffer.

Note:

This filter only be available in the current sniffer phase, if you disable sniffer and then enable sniffer, you need to set filter again if you need it.

Prototype:

`void wifi_promiscuous_set_mac(const uint8_t *address)`

Parameter:

const uint8_t *address : MAC address

Return:

null

Example:

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

```
    wifi_promiscuous_set_mac(ap_mac);
```

3. wifi_set_promiscuous_rx_cb

Function:

Registers an RX callback function in promiscuous mode, which will be called when data packet is received.

Prototype:

```
void wifi_set_promiscuous_rx_cb(wifi_promiscuous_cb_t cb)
```

Parameter:

`wifi_promiscuous_cb_t cb` : callback

Return:

null

4. wifi_get_channel

Function:

Get channel number for sniffer functions

Prototype:

```
uint8 wifi_get_channel(void)
```

Parameters:

null

Return:

Channel number

5. wifi_set_channel

Function:

Set channel number for sniffer functions

Prototype:

```
bool wifi_set_channel (uint8 channel)
```

Parameters:

`uint8 channel` : channel number

Return:

true: succeed

false: fail

4. Definitions & Structures

4.1. Timer

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

4.2. WiFi Related Structures

1. Station Related

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

Note:
   BSSID as MAC address of AP, will be used when several APs have the same
   SSID.
   If station_config.bssid_set==1 , station_config.bssid has to be set,
   otherwise, the connection will fail.
   In general, station_config.bssid_set need to be 0.
```

2. soft-AP related

```
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
};

Note:
   If softap_config.ssid_len==0, check ssid till find a termination characters;
   otherwise, it depends on softap_config.ssid_len.
```

3. scan related

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

4. WiFi event related structure

```
enum {
    EVENT_STAMODE_CONNECTED = 0,
    EVENT_STAMODE_DISCONNECTED,
    EVENT_STAMODE_AUTHMODE_CHANGE,
    EVENT_STAMODE_GOT_IP,
    EVENT_SOFTAPMODE_STACONNECTED,
       EVENT_SOFTAPMODE_STADISCONNECTED,
    EVENT_MAX
};
enum {
       REASON_UNSPECIFIED              = 1,
       REASON_AUTH_EXPIRE              = 2,
       REASON_AUTH_LEAVE               = 3,
       REASON_ASSOC_EXPIRE             = 4,
       REASON_ASSOC_TOOMANY            = 5,
       REASON_NOT_AUTHED               = 6,
       REASON_NOT_ASSOCED              = 7,
       REASON_ASSOC_LEAVE              = 8,
       REASON_ASSOC_NOT_AUTHED         = 9,
       REASON_DISASSOC_PWRCAP_BAD      = 10,  /* 11h */
       REASON_DISASSOC_SUPCHAN_BAD     = 11,  /* 11h */
       REASON_IE_INVALID               = 13,  /* 11i */
       REASON_MIC_FAILURE              = 14,  /* 11i */
       REASON_4WAY_HANDSHAKE_TIMEOUT   = 15,  /* 11i */
       REASON_GROUP_KEY_UPDATE_TIMEOUT = 16,  /* 11i */
       REASON_IE_IN_4WAY_DIFFERS       = 17,  /* 11i */
       REASON_GROUP_CIPHER_INVALID     = 18,  /* 11i */
       REASON_PAIRWISE_CIPHER_INVALID  = 19,  /* 11i */
       REASON_AKMP_INVALID             = 20,  /* 11i */
       REASON_UNSUPP_RSN_IE_VERSION    = 21,  /* 11i */
       REASON_INVALID_RSN_IE_CAP       = 22,  /* 11i */
       REASON_802_1X_AUTH_FAILED       = 23,  /* 11i */
       REASON_CIPHER_SUITE_REJECTED    = 24,  /* 11i */

       REASON_BEACON_TIMEOUT           = 200,
       REASON_NO_AP_FOUND              = 201,
};

typedef struct {
       uint8 ssid[32];
       uint8 ssid_len;
       uint8 bssid[6];
       uint8 channel;
   } Event_StaMode_Connected_t;

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

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

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

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

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

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

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

5. Appendix

5.1. RTC APIs Example

Demo code below shows how to get RTC time and to read and write to RTC memory.

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

os_timer_t rtc_test_t;
#define  RTC_MAGIC  0x55aaaa55

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

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

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

    printf("==================\r\n");
    printf("RTC time test : \r\n");

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

    rtc_t1 = system_get_rtc_time();
    st1 = system_get_time();

    cal1 = system_rtc_clock_cali_proc();
    os_delay_us(300);

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

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

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

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

```
void user_init(void)
{
    rtc_count();
    printf("SDK version:%s\n", system_get_sdk_version());

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

5.2. Sniffer Structure Introduction

ESP8266 can enter promiscuous mode (sniffer) and capture IEEE 802.11 packets in the air.

The following HT20 packets are support:

- 802.11b
- 802.11g
- 802.11n (from MCS0 to MCS7)
- AMPDU types of packets

The following are not supported:

- HT40
- LDPC

Although ESP8266 can not completely decipher these kinds of IEEE80211 packets completely, it can still obtain the length of these special packets.

In summary, while in sniffer mode, ESP8266 can either capture completely the packets or obtain the length of the packet:

- Packets that ESP8266 can decipher completely; ESP8266 returns with the
  - MAC address of the both side of communication and encryption type and
  - the length of entire packet.
- Packets that ESP8266 can only partial decipher; ESP8266 returns with
  - the length of packet.

Structure `RxControl` and `sniffer_buf` are used to represent these two kinds of packets. Structure `sniffer_buf` contains structure `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
};
```

Callback `wifi_promiscuous_rx` has two parameters (`buf` and `len`). `len` means the length of `buf`, it can be: `len` = 128, `len` = X * 10, `len` = 12 :

Case of LEN == 128

- `buf` contains structure `sniffer_buf2`: it is the management packet, it has 112 bytes data.
- `sniffer_buf2.cnt` is 1.
- `sniffer_buf2.len` is the length of packet.

Case of LEN == X ∗ 10

- `buf` contains structure `sniffer_buf`: this structure is reliable, data packets represented by it has been verified by CRC.
- `sniffer_buf.cnt` means the count of packets in `buf`. The value of `len` depends on `sniffer_buf.cnt`.
    - `sniffer_buf.cnt==0`, invalid buf; otherwise, `len = 50 + cnt ∗ 10`
- `sniffer_buf.buf` contains the first 36 bytes of ieee80211 packet. Starting from `sniffer_buf.lenseq[0]`, each structure `lenseq` represent a length information of packet. `lenseq[0]` represents the length of first packet. If there are two packets where `(sniffer_buf.cnt == 2)`, `lenseq[1]` represents the length of second packet.
- If `sniffer_buf.cnt > 1`, it is a AMPDU packet, head of each MPDU packets are similar, so we only provide the length of each packet (from head of MAC packet to FCS)
- This structure contains: length of packet, MAC address of both sides of communication, length of the head of packet.

Case of LEN == 12

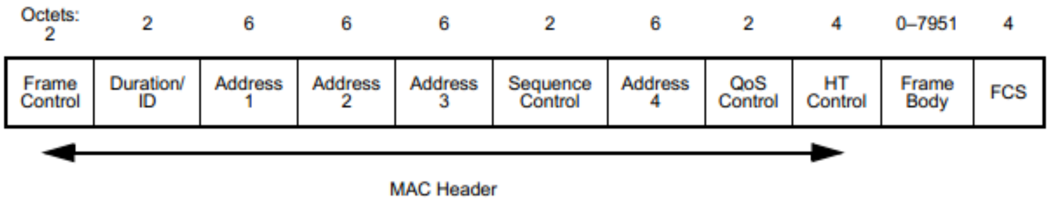
- `buf` contains structure `RxControl`; but this structure is not reliable, we can not get neither MAC address of both sides of communication nor length of the head of packet.
- For AMPDU packet, we can not get the count of packets or the length of packet.
- This structure contains: length of packet, `rssi` and `FEC_CODING`.

- `RSSI` and `FEC_CODING` are used to guess if the packets are sent from same device.

Summary

We should not take too long to process the packets. Otherwise, other packets may be lost.

The diagram below shows the format of a ieee80211 packet:



Octets: 2	2	6	6	6	2	6	2	4	0–7951	4
Frame Control	Duration/ ID	Address 1	Address 2	Address 3	Sequence Control	Address 4	QoS Control	HT Control	Frame Body	FCS

MAC Header

Figure 8-30—Data frame

- The first 24 bytes of MAC Header of data packet are needed:
  - `Address 4` field depends on `FromDS` and `ToDS` which is in `Frame Control`;
  - `QoS Control` field depends on `Subtype` which is in `Frame Control`;
  - `HT Control` field depends on `Order Field` which is in `Frame Control`;
  - More details are found in IEEE Std 80211-2012.
- For WEP packets, MAC Header is followed by 4 bytes IV and before FCS there are 4 bytes ICV.
- For TKIP packet, MAC Header is followed by 4 bytes IV and 4 bytes EIV, and before FCS there are 8 bytes MIC and 4 bytes ICV.
- For CCMP packet, MAC Header is followed by 8 bytes CCMP header, and before FCS there are 8 bytes MIC.

5.3. ESP8266 soft-AP and station channel configuration

Even though ESP8266 can be in soft-AP + station mode，it actually has only one hardware channel.

So in soft-AP + station mode, ESP8266 soft-AP will adjust its channel configuration to be as same as ESP8266 station.

This limitation may cause some inconvenience in softAP + station mode users need to pay attention，for example:

Case 1.

（1） If user connect ESP8266 station to a router(e.g. router is in channel 6)

（2） Then set ESP8266 softAP by `wifi_softap_set_config`

（3） The API may return true, but channel will always be channel 6. Because we have only one hardware channel.

Case 2.

（1）If user set ESP8266 softAP a channel number(e.g. channel 5) by `wifi_softap_set_config`

（2）Some stations connected to ESP8266 softAP.

（3）Then connect ESP8266 station to a router of which channel number is different (e.g. channel 6) .

（4）ESP8266 softAP has to adjust its channel to be as same as ESP8266 station , in this case, is channel 6.

（5）So the stations that connected to ESP8266 softAP in step 2 will be disconnected because of the channel change.