ESP8266 SDK API Guide

Version 1.0.0

Espressif Systems IOT Team

Table of Content

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.

The SDK provides an API to handle JSON, and users can also use self-defined data types to handle the them.

3. Application Programming Interface (APIs)

3.1. Timer

Found in `/esp_iot_sdk/include/osapi.h`.

1. os_timer_arm

```
Function:
    Arm timer

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

Input Parameters:
    ETSTimer *ptimer : Timer structure
    uint32_t milliseconds : Timing, Unit: millisecond
    bool repeat_flag : Whether to repeat the timing

Return:
    null
```

2. os_timer_disarm

```
Function:
    Disarm timer

Prototype:
    void os_timer_disarm (ETSTimer *ptimer)

Input Parameters:
    ETSTimer *ptimer : Timer structure

Return:
    null
```

3. os_timer_setfn

```
Function:
    Set timer callback function
```

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

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

Return:
    null
```

3.2. System APIs

1. system_restore

```
Function:
    Reset to default settings

Prototype:
    void system_restore(void)

Input Parameters:
    null

Return:
    null
```

2. system_restart

```
Function:
    Restart

Prototype:
    void system_restart(void)

Input Parameters:
    null

Return:
    null
```

3. system_timer_reinit

Function:

Reinitiate the timer when you need to use microsecond timer

Notes:

1. Define `USE_US_TIMER`;
2. Put `system_timer_reinit` at the beginning and user_init in the first sentence.

Prototype:

```
void system_timer_reinit (void)
```

Input Parameters:

null

Return:

null

4. system_init_done_cb

Function:

Call this API in user_init to register a system-init-done callback.

Note:

`wifi_station_scan` has to be called after system init done and station enable.

Prototype:

```
void system_init_done_cb(init_done_cb_t cb)
```

Parameter:

`init_done_cb_t cb` : system-init-done callback

Return:

null

Example:

```
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

Function:

Get chip ID

Prototype:

`uint32 system_get_chip_id (void)`

Input Parameters:

null

Return:

Chip ID

6. 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.

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

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

7. system_deep_sleep_set_option

Function:

Call this API before system_deep_sleep to set what the chip will do when 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 init data 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.

Note:

Init data refers `esp_init_data_default.bin`.

Return:

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

8. system_set_os_print

Function:

Turn on/off print logFunction

Prototype:

```
void system_set_os_print (uint8 onoff)
```

Input Parameters:

`uint8 onoff`

Note:

`onoff==0`: print function off

`onoff==1`: print function on

Default:

print function on

Return:

null

9. system_print_meminfo

Function:

Print memory information, including data/rodata/bss/heap

Prototype:

```
void system_print_meminfo (void)
```

Input Parameters:

    null

Return:

    null

10. system_get_free_heap_size

Function:

    Get free heap size

Prototype:

```
uint32 system_get_free_heap_size(void)
```

Input Parameters:

    null

Return:

    uint32 : available heap size

11. system_os_task

Function:

    Set up tasks

Prototype:

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

Input Parameters:

os_task_t task : task function

uint8 prio : task priority. 3 priorities are supported: 0/1/2; 0 is the lowest priority.

os_event_t *queue : message queue pointer

uint8 qlen : message queue depth

Return:

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

Example:

```
#define SIG_RX      0
#define TEST_QUEUE_LEN  4
os_event_t *testQueue;
void test_task (os_event_t *e) {
    switch (e->sig) {
        case SIG_RX:
            os_printf(sig_rx %c/n, (char)e->par);
            break;
        default:
            break;
    }
}
void task_init(void) {
    testQueue=(os_event_t *)os_malloc(sizeof(os_event_t)*TEST_QUEUE_LEN);
    system_os_task(test_task,USER_TASK_PRIO_0,testQueue,TEST_QUEUE_LEN);
}
```

12. system_os_post

Function: send message to task

Prototype:

```
bool system_os_post (
    uint8 prio,
    os_signal_t sig,
    os_param_t par
)
```

Input Parameters:

```
uint8 prio      : task priority, corresponding to that you set up
os_signal_t sig : message type
os_param_t par  : message parameters
```

Return:

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

Referring to the above example:

```
void task_post(void) {
    system_os_post(USER_TASK_PRIO_0, SIG_RX, ‘a’);
}
```

Printout:

```
sig_rx a
```

13. system_get_time

Function:

Get system time (us).

Prototype:

`uint32 system_get_time(void)`

Parameter:

null

Return:

System time in microsecond.

14. 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 (means 5us per RTC cycle), then the real time is 10 x 5 = 50 us.

Note:

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

Prototype:

`uint32 system_get_rtc_time(void)`

Parameter:

null

Return:

RTC time

15. system_rtc_clock_cali_proc

Function:

Get RTC clock period.

Prototype:

`uint32 system_rtc_clock_cali_proc(void)`

Parameter:

null

Return:

RTC clock period (in us), bit11~ bit0 are decimal.

Note:

see RTC demo in Appendix.

16. 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

17. 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
```

18. 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

19. 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)

20. 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.

21. 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).

22. 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.

23. system_update_cpu_freq

Function:

Set CPU frequency. Default is 80MHz.

Prototype:

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

Parameter:

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

Return:

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

24. system_get_cpu_freq

Function:

Get CPU frequency.

Prototype:

```
uint8 system_get_cpu_freq(void)
```

Parameter:

null

Return:

CPU frequency, unit : MHz.

3.3. SPI Flash Related APIs

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

Note:

More details in document Espressif IOT Flash RW Operation

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.

Note:

More details in document

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.

Note:

More details in document

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;
```

3.4. WIFI Related APIs

1. wifi_get_opmode

```
Function:
    get WiFi current operating mode

Prototype:
    uint8 wifi_get_opmode (void)

Input Parameters:
    null

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

2. wifi_set_opmode

```
Function:
    Sets WiFi working mode as station, soft-AP or station+soft-AP.

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.

Prototype:
    bool wifi_set_opmode (uint8 opmode)

Input Parameters:
    uint8 opmode: WiFi operating modes:
        0x01: station mode
        0x02: soft-AP mode
        0x03: station+soft-AP

Return:
    true:  succeed
    false: fail
```

3. wifi_station_get_config

```
Function:
    Get WiFi station configuration
```

Prototype:

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

Input Parameters:

`struct station_config *config` : WiFi station configuration pointer

Return:

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

4. wifi_station_set_config

Function:

Set WiFi station configuration

Note:

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.

Prototype:

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

Input Parameters:

`struct station_config *config`: WiFi station configuration pointer

Return:

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

5. 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.

Prototype:

```
bool wifi_station_connect (void)
```

Input Parameters:

null

Return:

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

6. wifi_station_disconnect

Function:

Disconnects WiFi station from AP

Prototype:

```
bool wifi_station_disconnect (void)
```

Input Parameters:

null

Return:

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

7. wifi_station_get_connect_status

Function:

Get connection status of WiFi station to AP

Prototype:

```
uint8 wifi_station_get_connect_status (void)
```

Input Parameters:

null

Return:

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

8. 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 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
```

9. scan_done_cb_t

```
Function:
    Callback function for wifi_station_scan

Prototype:
    void scan_done_cb_t (void *arg, STATUS status)

Input 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
        ...
    }
}
```

10. 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.

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

11. 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);
```

12. 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
```

13. 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.

14. 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.
```

15. wifi_station_set_auto_connect

Function:

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

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.

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
```

16. wifi_station_dhcpc_start

Function:

Enable ESP8266 station DHCP client.

Note:

DHCP default enable.

Prototype:

```
bool wifi_station_dhcpc_start(void)
```

Parameter:

null

Return:

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

17. 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
```

18. 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
};
```

19. wifi_softap_get_config

Function:

Get WiFi soft-AP 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
```

20. wifi_softap_set_config

Function:

Set WiFi soft-AP configuration

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
```

21. wifi_softap_get_station_info

```
Function:
    get connected station devices under soft-AP mode, including MAC and ip

Prototype:
    struct station_info * wifi_softap_get_station_info(void)

Input Parameters:
    null

Return:
    struct station_info* : station information structure
```

22. 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) {
        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;
    }
```

Examples 2 (Getting MAC and IP information):

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

23. wifi_softap_dhcps_start

Function: Enable ESP8266 soft-AP DHCP server.

Note:
DHCP default enable.

Prototype:

```
bool wifi_softap_dhcps_start(void)
```

Parameter:
null

Return:
true: succeed
false: fail

24. 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

25. wifi_softap_set_dhcps_lease

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

Note:

```
    This API need to be called during DHCP server disable.
```

Prototype:

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

Parameter:

```
    struct dhcps_lease {
        uint32 start_ip;
        uint32 end_ip;
    };
```

Return:

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

26. 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
    };
```

27. 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
```

28. 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
    };
```

29. 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
```

30. wifi_set_ip_info

```
Function:
    Set IP address of ESP8266 station or soft-AP
```

Note:

Can only be used in `user_init`.

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;
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);
```

Return:

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

31. wifi_set_macaddr

Function:

Sets MAC address

Note:

Can only be used in `user_init`.

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
```

32. 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
```

33. wifi_set_sleep_type

```
Function:
    Sets sleep type for power saving. Set NONE_SLEEP_T to disable power saving.

Note: Default to be Modem sleep.

Prototype:
    bool wifi_set_sleep_type(enum sleep_type type)

Parameters:
    enum sleep_type type  :  sleep type
Return:
    true:  succeed
    false: fail
```

34. wifi_get_sleep_type

```
Function:
    Gets sleep type.
Prototype:
    enum sleep_type wifi_get_sleep_type(void)
Parameters:
    null
Return:
    enum sleep_type {
        NONE_SLEEP_T = 0;
        LIGHT_SLEEP_T,
        MODEM_SLEEP_T
    };
```

35. 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
```

Example:

Use GPIO0 as WiFi status LED

```
#define HUMITURE_WIFI_LED_IO_MUX     PERIPHS_IO_MUX_GPIO0_U
#define HUMITURE_WIFI_LED_IO_NUM     0
#define HUMITURE_WIFI_LED_IO_FUNC    FUNC_GPIO0
wifi_status_led_install(HUMITURE_WIFI_LED_IO_NUM,
        HUMITURE_WIFI_LED_IO_MUX, HUMITURE_WIFI_LED_IO_FUNC)
```

36. wifi_status_led_uninstall

Function: Uninstall WiFi status LED

Prototype:

```
void wifi_status_led_uninstall ()
```

Parameter:

null

Return:

null

37. wifi_set_broadcast_if

Function:

Set ESP8266 send UDP broadcast from station interface or soft-AP interface. Default to be soft-AP.

Prototype:

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

Parameter:

`uint8 interface` : 1:station; 2:soft-AP, 3:station+soft-AP

Return:

true: succeed
false: fail

38. wifi_get_broadcast _if

Function:

Get interface which ESP8266 sent UDP broadcast from. This is usually used when you have STA+soft-AP mode to avoid ambiguity.

Prototype:

```
uint8 wifi_get_broadcast_if (void)
```

Parameter:

null

```
    Return:
        1: station
        2: soft-AP
        3: both station and soft-AP
```

3.5. Upgrade (FOTA) APIs

1. system_upgrade_userbin_check

Function:

Checks user bin

Prototype:

```
uint8 system_upgrade_userbin_check()
```

Input Parameters:

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:

If you using system_upgrade_start to upgrade, this API need not be called.
If you using spi_flash_write to upgrade firmware yourself, this flag need to be set to UPGRADE_FLAG_FINISH, then 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_start

Function:

Configures parameters and start upgrade

Prototype:

```
bool system_upgrade_start (struct upgrade_server_info *server)
```

Parameters:

`struct upgrade_server_info *server` : server related parameters

Return:

true: start upgrade

false: upgrade can't be started.

5. system_upgrade_reboot

Function: reboot system and use new version

Prototype:

```
void system_upgrade_reboot (void)
```

Input Parameters:

none

Return:

none

3.6. Sniffer Related APIs

1. wifi_promiscuous_enable

```
Function:
    Enable promiscuous mode for sniffer

Prototype:
    void wifi_promiscuous_enable(uint8 promiscuous)

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

Return:
    null

Example:
    Apply for a demo of sniffer function from Espressif
```

2. wifi_promiscuous_set_mac

```
Function:
    Set MAC address filter for sniffer.

Note:
    This filter only 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
```

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
```

3.7. smart config APIs

1. smartconfig_start

Function:

Start smart configuration mode, to connect ESP8266 station to AP, by sniffing for special packets from the air, containing SSID and password of desired AP. You need to broadcast the SSID and password (e.g. from mobile device or computer) with the SSID and password encoded.

Note:

In this API, ESP8266 is set to station mode. Can not call smartconfig_start twice before it finish.

Prototype:

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

Parameter:

sc_type type : smart config protocol type: AirKiss or ESP-TOUCH.
sc_callback_t cb : pointer to struct station_config; smart config callback; executed when ESP8266 successfully obtains SSID and password of target AP; for parameter of this callback, more information in example.
uint8 log : 1: UART output logs; otherwise: UART only outputs the result.

Return:

true: succeed
false: fail

Example:

```
void ICACHE_FLASH_ATTR
smartconfig_done(void *data) {
    struct station_config *sta_conf = data;
    wifi_station_set_config(sta_conf);
    wifi_station_disconnect();
    wifi_station_connect();
    user_devicefind_init();
    user_esp_platform_init();
}
smartconfig_start(SC_TYPE_ESPTOUCH,smartconfig_done);
```

2. smartconfig_stop

Function:

stop smart config, free the buffer taken by smartconfig_start.

Note:

When connect to AP succeed, this API should be called to free memory taken by smartconfig_start.

Prototype:

```
bool smartconfig_stop(void)
```

Parameter:

null

Return:

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

3. get_smartconfig_status

Function:

Get smart config status

Note:

Can not call this API after smartconfig_stop, because smartconfig_stop will try to free (again) the memory which contains this smart config status.

Prototype:

```
sc_status get_smartconfig_status(void)
```

Parameter:

null

Return:

```
typedef enum  {
    SC_STATUS_WAIT = 0,
    SC_STATUS_FIND_CHANNEL,
    SC_STATUS_GETTING_SSID_PSWD,
    SC_STATUS_GOT_SSID_PSWD,
    SC_STATUS_LINK,
    SC_STATUS_LINK_OVER,
} sc_status;
```

Note:

Use APP to start connection when get_smartconfig_status is SC_STATUS_FIND_CHANNEL.

4. TCP/UDP APIs

Found in esp_iot_sdk/include/espconn.h. The network APIs can be grouped into the following types:

- **General APIs**: APIs can be used for both TCP and UDP .
- **TCP APIs**: APIs that are only used for TCP.
- **UDP APIs**: APIs that are only used for UDP.

4.1. Generic TCP/UDP APIs

1. espconn_delete

```
Function:
    Delete a transmission.

Note:
    Corresponding creation API :
        TCP: espconn_accept,
        UDP: espconn_create

Prototype:
    sint8 espconn_delete(struct espconn *espconn)

Parameter:
    struct espconn *espconn : corresponding connected control block structure

Return:
    0      : succeed
    Non-0  : error (please refer to espconn.h for details.)
```

2. espconn_gethostbyname

```
Function:
    DNS

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

Parameters:

```
struct espconn *espconn  : corresponding connected control block structure
const char *hostname     : domain name string pointer
ip_addr_t *addr          : IP address
dns_found_callback found : callback
```

Return:

```
err_t : ESPCONN_OK
        ESPCONN_INPROGRESS
        ESPCONN_ARG
```

Example as follows. Pls refer to source code of IoT_Demo:

```
ip_addr_t esp_server_ip;
LOCAL void ICACHE_FLASH_ATTR
user_esp_platform_dns_found(const char *name, ip_addr_t *ipaddr, void *arg)
{
    struct espconn *pespconn = (struct espconn *)arg;
    os_printf(user_esp_platform_dns_found %d.%d.%d.%d/n,
        *((uint8 *)&ipaddr->addr), *((uint8 *)&ipaddr->addr + 1),
        *((uint8 *)&ipaddr->addr + 2), *((uint8 *)&ipaddr->addr + 3));
}
void dns_test(void) {
    espconn_gethostbyname(pespconn,iot.espressif.cn, &esp_server_ip,
            user_esp_platform_dns_found);
}
```

3. espconn_port

Function: get void ports

Prototype:

```
uint32 espconn_port(void)
```

Input Parameters:

null

Return:

```
uint32 : id of the port you get
```

4. espconn_regist_sentcb

Function:

Register data sent function which will be called back when data are successfully sent.

Prototype:

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

Parameters:

`struct espconn *espconn` : corresponding connected control block structure
`espconn_sent_callback sent_cb` : registered callback function

Return:

```
0     : succeed
Non-0 : error (please refer to espconn.h)
```

5. espconn_regist_recvcb

Function:

register data receive function which will be called back when data are received

Prototype:

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

Input Parameters:

`struct espconn *espconn` : corresponding connected control block structure
`espconn_connect_callback connect_cb` : registered callback function

Return:

```
0     : succeed
Non-0 : error (please refer to espconn.h)
```

6. espconn_sent_callback

Function:

Callback after the data are sent

Prototype:

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

Input Parameters:

`void *arg` : call back function parameters

Return:

null

7. espconn_recv_callback

```
Function:
    callback after data are received

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

Input Parameters:
    void *arg : callback function parameters
    char *pdata : received data entry parameters
    unsigned short len : received data length

Return:
    null
```

8. espconn_sent

```
Function:
    Send data through WiFi

Note:
    Please call espconn_sent after espconn_sent_callback of the pre-packet.

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

Input Parameters:
    struct espconn *espconn : corresponding connected control block structure
    uint8 *psent   : sent data pointer
    uint16 length  : sent data length

Return:
    0      : succeed
    Non-0  : error (please refer to espconn.h)
```

4.2. TCP APIs

TCP APIs act only on TCP connections and do not affect nor apply to UDP connections.

1. espconn_accept

Function:

Creates a TCP server (i.e. accepts connections.)

Prototype:

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

Input Parameters:

`struct espconn *espconn` : corresponding connected control block structure

Return:

```
0       :  succeed
Not  0  :  error (please refer to espconn.h)
```

2. espconn_secure_accept

Function:

Creates an SSL TCP server.

Prototype:

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

Input Parameters:

`struct espconn *espconn` : corresponding connected control block structure

Return:

```
0      :  succeed
Non-0  :  error (please refer to espconn.h)
```

3. espconn_regist_time

Function:

register timeout interval of ESP8266 TCP server.

Note:

Call this API after `espconn_accept`.

If timeout is set to 0, timeout will be disable and ESP8266 TCP server will not disconnect TCP clients has stopped communication. This usage of `timeout=0`, is deprecated.

Prototype:

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

```
Input Parameters:
    struct espconn *espconn : corresponding connected control block structure
    uint32 interval : timeout interval, unit: second, maximum: 7200 seconds
    uint8 type_flag : 0, set all connections; 1, set a single connection

Return:
    0      :  succeed
    Non-0  :  error (please refer to espconn.h)
```

4. espconn_get_connection_info

```
Function:
    Get a connection's info in TCP multi-connection case

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

Input Parameters:
    struct espconn *espconn : corresponding connected control block structure
    remot_info **pcon_info  : connect to client info
    uint8 typeflags         : 0, regular server;1, ssl server

Return:
    0      :  succeed
    Non-0  :  error (please refer to espconn.h)
```

5. espconn_connect

```
Function:
    Connect to a TCP server (ESP8266 acting as TCP client).

Prototype:
    sint8 espconn_connect(struct espconn *espconn)

Input Parameters:
    struct espconn *espconn : corresponding connected control block structure

Return:
    0      :  succeed
    Non-0  :  error (please refer to espconn.h)
```

6. espconn_connect_callback

Function: successful listening (ESP8266 as TCP server) or connection (ESP8266 as TCP client) callback

Prototype:

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

Input Parameters:

`void *arg` : callback function parameters

Return:

null

7. espconn_set_opt

Function: Set option of TCP connection

Prototype:

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

Parameter:

`struct espconn *espconn` : corresponding connected control structure
`uint8 opt` : Option of TCP connection
bit 1: 1: free memory after TCP disconnection happen need not wait 2 minutes;
bit 2: 1: disable nalgo algorithm during TCP data transmission, quiken the data transmission.
bit 3: 1: use 2920 bytes write buffer for the data espconn_sent sending.

Return:

```
0      : succeed
Non-0  : error (please refer to espconn.h)
```

Note:

In general, we need not call this API;
If call espconn_set_opt(espconn, 0), please call it in connected callback.
If call espconn_set_opt(espconn, 1), please call it before disconnect.

8. espconn_disconnect

Function:

disconnect a TCP connection

Prototype:

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

Input Parameters:

`struct espconn *espconn` : corresponding connected control structure

Return:

```
0      : succeed
Non-0  : error (please refer to espconn.h)
```

9. espconn_regist_connectcb

Function:

Register connection function which will be called back under successful TCP connection

Prototype:

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

Input Parameters:

`struct espconn *espconn` : corresponding connected control block structure

`espconn_connect_callback connect_cb` : registered callback function

Return:

```
0      : succeed
Non-0  : error (please refer to espconn.h)
```

10. espconn_regist_reconcb

Function:

Register reconnect callback

Note:

Reconnect callback is more like a network error handler; it handles errors that occurred in any phase of the connection. For instance, if `espconn_sent` fails, reconnect callback will be called because the network is broken.

Prototype:

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

Input Parameters:

```
struct espconn *espconn : corresponding connected control block structure
espconn_connect_callback connect_cb : registered callback function
```

Return:

```
0      : succeed
Non-0  : error (please refer to espconn.h)
```

11. espconn_regist_disconcb

Function: register disconnection function which will be called back under successful TCP disconnection

Prototype:

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

Input Parameters:

```
struct espconn *espconn : corresponding connected control block structure
espconn_connect_callback connect_cb : registered callback function
```

Return:

```
0      : succeed
Non-0  : error (please refer to espconn.h)
```

12. espconn_regist_write_finish

Function:

Register a callback which will be called when all sending data is completely write into write buffer or sent.

Prototype:

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

Input Parameters:

```
struct espconn *espconn  :  corresponding connected control block structure
espconn_connect_callback write_finish_fn :  registered callback function
```

Return:

```
0      : succeed
Non-0  : error (please refer to espconn.h)
```

13. espconn_secure_connect

Function:

Secure connect (SSL) to a TCP server (ESP8266 is acting as TCP client.)

Prototype:

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

Input Parameters:

`struct espconn *espconn` : corresponding connected control block structure

Return:

```
0       : succeed
Non-0   : error (please refer to espconn.h)
```

14. espconn_secure_sent

Function: send encrypted data (SSL)

Prototype:

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

Input Parameters:

`struct espconn *espconn` : corresponding connected control block structure

`uint8 *psent` : sent data pointer

`uint16 length` : sent data length

Return:

```
0       : succeed
Non-0   : error (please refer to espconn.h)
```

15. espconn_secure_disconnect

Function: secure TCP disconnection(SSL)

Prototype:

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

Input Parameters:

`struct espconn *espconn` : corresponding connected control block structure

Return:

```
0       : succeed
Non-0   : error (please refer to espconn.h)
```

16. espconn_tcp_get_max_con

Function:

Get maximum number of how many TCP connection is allowed.

Prototype:

```
uint8 espconn_tcp_get_max_con(void)
```

Parameter:

null

Return:

Maximum number of how many TCP connection is allowed.

17. espconn_tcp_set_max_con

Function:

Set the maximum number of how many TCP connection is allowed.

Prototype:

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

Parameter:

`uint8 num` : Maximum number of how many TCP connection is allowed.

Return:

```
0       : succeed
Non-0   : error (please refer to espconn.h)
```

18. espconn_tcp_get_max_con_allow

Function:

Get the maximum number of TCP clients which are allowed to connect to ESP8266 TCP server.

Prototype:

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

Parameter:

`struct espconn *espconn` : corresponding connected control structure

Return:

Maximum number of TCP clients which are allowed.

19. espconn_tcp_set_max_con_allow

Function:

Set the maximum number of TCP clients allowed to connect to ESP8266 TCP server.

Prototype:

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

Parameter:

```
struct espconn *espconn : corresponding connected control structure
uint8 num : Maximum number of TCP clients which are allowed.
```

Return:

```
0      : succeed
Non-0  : error (please refer to espconn.h)
```

20. espconn_recv_hold

Function:

Puts in a request to block the TCP receive function.

Note:

The function does not act immediately; we recommend calling it while reserving 5*1460 bytes of memory.
This API can be called more than once.

Prototype:

```
sint8 espconn_recv_hold(struct espconn *espconn)
```

Parameter:

```
struct espconn *espconn : corresponding connected control structure
```

Return:

```
0           : succeed
Non-0       : error (please refer to espconn.h).
ESPCONN_ARG : could not find the TCP connection according to parameter
espconn
```

21. espconn_recv_unhold

Function:

Unblock TCP receiving data (i.e. undo espconn_recv_hold).

Note:

This API takes effect immediately.

Prototype:

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

Parameter:

```
struct espconn *espconn : corresponding connected control structure
```

```
Return:
    0           : succeed
    Non-0       : error (please refer to espconn.h).
    ESPCONN_ARG : could not find the TCP connection according to parameter
    espconn
```

4.3. UDP APIs

1. espconn_create

```
Function: create UDP transmission.

Prototype:
    sint8 espconn_create(struct espconn *espconn)

Parameter:
    struct espconn *espconn : corresponding connected control block structure

Return:
    0      : succeed
    Non-0  : error (please refer to espconn.h).
```

2. espconn_igmp_join

```
Function:
    Join a multicast group

Prototype:
    sint8 espconn_igmp_join(ip_addr_t *host_ip, ip_addr_t *multicast_ip)

Parameters:
    ip_addr_t *host_ip      :  IP of host
    ip_addr_t *multicast_ip :  IP of multicast group

Return:
    0      : succeed
    Non-0  : error (please refer to espconn.h).
```

3. espconn_igmp_leave

```
Function:
    Quit a multicast group

Prototype:
    sint8 espconn_igmp_leave(ip_addr_t *host_ip, ip_addr_t *multicast_ip)
```

```
Parameters:
    ip_addr_t *host_ip      : IP of host
    ip_addr_t *multicast_ip : IP of multicast group

Return:
    0       : succeed
    Non-0   : error (please refer to espconn.h).
```

5. Application Related

5.1. AT APIs

for AT APIs examples, refer to `esp_iot_sdk/examples/5.at/user/user_main.c`.

1. at_response_ok

```
Function:
    Output OK to AT Port (UART0)

Prototype:
    void at_response_ok(void)

Parameter:
    null

Return:
    null
```

2. at_response_error

```
Function:
    output ERROR to AT Port (UART0)

Prototype:
    void at_response_error(void)

Parameter:
    null

Return:
    null
```

3. at_cmd_array_regist

```
Function:
    register user-define AT commands

Prototype:
    void at_cmd_array_regist (
            at_function * custom_at_cmd_arrar,
            uint32 cmd_num
    )

Parameter:
    at_function * custom_at_cmd_arrar : Array of user-define AT commands
    uint32 cmd_num : Number counts of user-define AT commands
```

Return:

```
null
```

Example:

refer to `esp_iot_sdk/examples/5.at/user/user_main.c`

4. at_get_next_int_dec

Function:

parse int from AT command

Prototype:

```
bool at_get_next_int_dec (char **p_src,int* result,int* err)
```

Parameter:

```
char **p_src : *p_src is the AT command that need to be parsed
int* result  : int number parsed from the AT command
int* err     : 1: no number is found; 3: only '-' is found.
```

Return:

true: parser succeeds (NOTE: if no number is found, it will return True, but returns error code 1)

false: parser is unable to parse string; some probable causes are: int number more than 10 bytes; string contains termination characters '/r'; string contains only '-'.

Example:

refer to `esp_iot_sdk/examples/5.at/user/user_main.c`

5. at_data_str_copy

Function: parse string from AT command

Prototype:

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

Parameter:

```
char * p_dest : string parsed from the AT command
char ** p_src : *p_src is the AT command that need to be parsed
int32 max_len : max string length that allowed
```

Return:

length of string:

```
    >=0: succeed and returns the length of the string
    <0 : fail and returns -1
```

Example:

refer to `esp_iot_sdk/examples/5.at/user/user_main.c`

6. at_init

Function:

AT initialize

Prototype:

`void at_init (void)`

Parameter:

null

Return:

null

Example:

refer to `esp_iot_sdk/examples/5.at/user/user_main.c`

7. at_port_print

Function:

output string to AT PORT(UART0)

Prototype:

`void at_port_print(const char *str)`

Parameter:

`const char *str` : string that need to output

Return:

null

Example:

refer to `esp_iot_sdk/examples/5.at/user/user_main.c`

8. at_set_custom_info

Function:

User-define version info of AT which can be got by AT+GMR.

Prototype:

`void at_set_custom_info (char *info)`

Parameter:

`char *info` : version info

Return:

null

9. at_enter_special_state

Function:

Enter processing state. In processing state, AT core will return busy for any further AT commands.

Prototype:

```
void at_enter_special_state (void)
```

Parameter:

null

Return:

null

10. at_leave_special_state

Function:

Exit from AT processing state.

Prototype:

```
void at_leave_special_state (void)
```

Parameter:

null

Return:

null

11. at_get_version

Function:

Get Espressif AT lib version.

Prototype:

```
uint32 at_get_version (void)
```

Parameter:

null

Return:

Espressif AT lib version

5.2. Related JSON APIs

Found in : `esp_iot_sdk/include/json/jsonparse.h & jsontree.h`

1. jsonparse_setup

```
Function:
    json initialize parsing

Prototype:
    void jsonparse_setup(
            struct jsonparse_state *state,
            const char *json,
            int len
    )

Input Parameters:
    struct jsonparse_state *state : json parsing pointer
    const char *json : json parsing character string
    int len : character string length

Return:
    null
```

2. jsonparse_next

```
Function:
    Returns jsonparse next object

Prototype:
    int jsonparse_next(struct jsonparse_state *state)

Input Parameters:
    struct jsonparse_state *state : json parsing pointer

Return:
    int : parsing result
```

3. jsonparse_copy_value

```
Function:
    Copies current parsing character string to a certain buffer
```

Prototype:

```
int jsonparse_copy_value(
    struct jsonparse_state *state,
    char *str,
    int size
)
```

Input Parameters:

`struct jsonparse_state *state` : json parsing pointer

`char *str` : buffer pointer

`int size` : buffer size

Return:

`int` : copy result

4. jsonparse_get_value_as_int

Function:

Parses json to get integer

Prototype:

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

Input Parameters:

`struct jsonparse_state *state` : json parsing pointer

Return:

`int` : parsing result

5. jsonparse_get_value_as_long

Function:

Parses json to get long integer

Prototype:

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

Input Parameters:

`struct jsonparse_state *state` : json parsing pointer

Return:

`long` : parsing result

6. jsonparse_get_len

Function:

Gets parsed json length

Prototype:

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

Input Parameters:

```
struct jsonparse_state *state : json parsing pointer
```

Return:

```
int : parsed jason length
```

7. jsonparse_get_value_as_type

Function:

Parses json data type

Prototype:

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

Input Parameters:

```
struct jsonparse_state *state : json parsing pointer
```

Return:

```
int : parsed json data type
```

8. jsonparse_strcmp_value

Function:

Compares parsed json and certain character string

Prototype:

```
int jsonparse_strcmp_value(struct jsonparse_state *state, const char *str)
```

Input Parameters:

```
struct jsonparse_state *state : json parsing pointer
const char *str : character buffer
```

Return:

```
int : comparison result
```

9. jsontree_set_up

Function:

Creates json data tree

Prototype:

```
void jsontree_setup(
        struct jsontree_context *js_ctx,
        struct jsontree_value *root,
        int (* putchar)(int)
)
```

Input Parameters:

`struct jsontree_context *js_ctx` : json tree element pointer

`struct jsontree_value *root` : root element pointer

`int (* putchar)(int)` : input function

Return:

null

10. jsontree_reset

Function:

Resets json tree

Prototype:

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

Input Parameters:

`struct jsontree_context *js_ctx` : json data tree pointer

Return:

null

11. jsontree_path_name

Function:

get json tree parameters

Prototype:

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

Input Parameters:

`struct jsontree_context *js_ctx` : json tree pointer

`int depth` : json tree depth

Return:

`char*` : parameter pointer

12. jsontree_write_int

Function:

write integer to json tree

Prototype:

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

Input Parameters:

`struct jsontree_context *js_ctx` : json tree pointer

`int value` : integer value

Return:

null

13. jsontree_write_int_array

Function:

Writes integer array to json tree

Prototype:

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

Input Parameters:

`struct jsontree_context *js_ctx` : json tree pointer

`int *text` : array entry address

`uint32 length` : array length

Return:

null

14. jsontree_write_string

Function:

Writes string to json tree

Prototype:

```
void jsontree_write_string(
        const struct jsontree_context *js_ctx,
        const char *text
)
```

Input Parameters:

```
struct jsontree_context *js_ctx : json tree pointer
const char* text : character string pointer
```

Return:

```
null
```

15. jsontree_print_next

Function:

```
json tree depth
```

Prototype:

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

Input Parameters:

```
struct jsontree_context *js_ctx : json tree pointer
```

Return:

```
int : json tree depth
```

16. jsontree_find_next

Function:

```
find json tree element
```

Prototype:

```
struct jsontree_value *jsontree_find_next(
        struct jsontree_context *js_ctx,
        int type
)
```

Input Parameters:

```
struct jsontree_context *js_ctx : json tree pointer
int : type
```

Return:

```
struct jsontree_value * : json tree element pointer
```

6. Definition of Structures

6.1. Timer

```
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 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;
    uint8 authmode;
    uint8 ssid_hidden;
    uint8 max_connection;
    uint8 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. smart config structure

```
    typedef enum  {
        SC_STATUS_FIND_CHANNEL = 0,
        SC_STATUS_GETTING_SSID_PSWD,
        SC_STATUS_GOT_SSID_PSWD,
        SC_STATUS_LINK,
    } sc_status;
    typedef enum  {
        SC_TYPE_ESPTOUCH = 0,
        SC_TYPE_AIRKISS,
    } sc_type;
```

6.4. JSON Related Structure

1. json structure

```
    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 macro definition

```
#define JSONTREE_OBJECT(name, ...)                                     /
static struct jsontree_pair jsontree_pair_##name[] = {__VA_ARGS__};     /
static struct jsontree_object name = {                                 /
    JSON_TYPE_OBJECT,                                                  /
sizeof(jsontree_pair_##name)/sizeof(struct jsontree_pair),             /
    jsontree_pair_##name }

#define JSONTREE_PAIR_ARRAY(value) (struct jsontree_value *)(value)
#define JSONTREE_ARRAY(name, ...)                                      /
static struct jsontree_value* jsontree_value_##name[] = {__VA_ARGS__};  /
static struct jsontree_array name = {                                  /
    JSON_TYPE_ARRAY,                                                   /
    sizeof(jsontree_value_##name)/sizeof(struct jsontree_value*),      /
    jsontree_value_##name }
```

6.5. espconn parameters

1. callback function

```
/** callback prototype to inform about events for a espconn */
typedef void (* espconn_recv_callback)(void *arg, char *pdata, unsigned short
len);
typedef void (* espconn_callback)(void *arg, char *pdata, unsigned short len);
typedef void (* espconn_connect_callback)(void *arg);
```

2. espconn

```
typedef void* espconn_handle;
typedef struct _esp_tcp {
    int client_port;
    int server_port;
    char ipaddr[4];
        espconn_connect_callback connect_callback;
espconn_connect_callback reconnect_callback;
    espconn_connect_callback disconnect_callback;
} esp_tcp;

typedef struct _esp_udp {
    int _port;
    char ipaddr[4];
} esp_udp;

/** Protocol family and type of the espconn */
enum espconn_type {
    ESPCONN_INVALID    = 0,
    /* ESPCONN_TCP Group */
    ESPCONN_TCP        = 0x10,
    /* ESPCONN_UDP Group */
    ESPCONN_UDP        = 0x20,
};

/** 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;
    espconn_handle esp_pcb;
    uint8 *ptrbuf;
    uint16 cntr;
};
```

7. Peripheral Related Drivers

7.1. GPIO Related APIs

Please refer to `/user/user_plug.c`.

1. PIN Related Macros

The following macros are used to control the GPIO pins' status.

```
PIN_PULLUP_DIS(PIN_NAME)
    Disable pin pull up

PIN_PULLUP_EN(PIN_NAME)
    Enable pin pull up

PIN_PULLDWN_DIS(PIN_NAME)
    Disable pin pull down

PIN_PULLDWN_EN(PIN_NAME)
    Enable pin pull down

PIN_FUNC_SELECT(PIN_NAME, FUNC)
    Select pin function

Example:
    PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U, FUNC_GPIO12); // Use MTDI pin as
    GPIO12.
```

2. gpio_output_set

```
Function: set gpio property

Prototype:
    void gpio_output_set(
        uint32 set_mask,
        uint32 clear_mask,
        uint32 enable_mask,
        uint32 disable_mask
    )

Input Parameters:
    uint32 set_mask : set high output; 1:high output; 0:no status change
    uint32 clear_mask : set low output; 1:low output; 0:no status change
    uint32 enable_mask : enable outpout bit
    uint32 disable_mask : enable input bit
```

```
Return:
    null

Example:
    gpio_output_set(BIT12, 0, BIT12, 0):
        Set GPIO12 as high-level output;
    gpio_output_set(0, BIT12, BIT12, 0):
        Set GPIO12 as low-level output
    gpio_output_set(BIT12, BIT13, BIT12|BIT13, 0):
        Set GPIO12 as high-level output, GPIO13 as low-level output.
    gpio_output_set(0, 0, 0, BIT12):
        Set GPIO12 as input
```

3. GPIO input and output macro

```
GPIO_OUTPUT_SET(gpio_no, bit_value)
    Set gpio_no as output bit_value, the same as the output example in 5.1.2

GPIO_DIS_OUTPUT(gpio_no)
    Set gpio_no as input, the same as the input example in 5.1.2.

GPIO_INPUT_GET(gpio_no)
    Get the level status of gpio_no.
```

4. GPIO interrupt

```
ETS_GPIO_INTR_ATTACH(func, arg)
    Register GPIO interrupt control function

ETS_GPIO_INTR_DISABLE()
    Disable GPIO interrupt

ETS_GPIO_INTR_ENABLE()
    Enable GPIO interrupt
```

5. gpio_pin_intr_state_set

```
Function:
    set GPIO interrupt state

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

```
Input Parameters:
    uint32 i : GPIO pin ID, if you want to set GPIO14, pls use GPIO_ID_PIN(14);
    GPIO_INT_TYPE intr_state : interrupt type as the following:
    typedef enum {
        GPIO_PIN_INTR_DISABLE = 0,
        GPIO_PIN_INTR_POSEDGE = 1,
        GPIO_PIN_INTR_NEGEDGE = 2,
        GPIO_PIN_INTR_ANYEGDE = 3,
        GPIO_PIN_INTR_LOLEVEL = 4,
        GPIO_PIN_INTR_HILEVEL = 5
    } GPIO_INT_TYPE;

Return:
    null
```

6. GPIO Interrupt Handler

Follow the steps below to clear interrupt status in GPIO interrupt processing function:

```
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 Related APIs

By default, UART0 is debug output interface. In the case of dual Uart, UART0 works as data receive and transmit interface, and UART1as debug output interface.

Please make sure all hardware are correctly connected.

1. uart_init

```
Function:
    Initializes baud rates of the two uarts

Prototype:
    void uart_init(
        UartBautRate uart0_br,
        UartBautRate uart1_br
    )

Parameters:
    UartBautRate uart0_br : uart0 baud rate
    UartBautRate uart1_br : uart1 baud rate
```

Baud Rates:

```
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;
```

Return:

null

2. uart0_tx_buffer

Function:

Sends user-defined data through UART0

Prototype:

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

Parameter:

`uint8 *buf` : data to send later

`uint16 len` : the length of data to send later

Return:

null

3. uart0_rx_intr_handler

Function:

UART0 interrupt processing function. Users can add the processing of received data in this function. (Receive buffer size: `0x100`; if the received data are more than `0x100`, pls handle them yourselves.)

Prototype:

```
void uart0_rx_intr_handler(void *para)
```

Parameter:

`void *para` : the pointer pointing to RcvMsgBuff structure

Return:

null

7.3. I2C Master Related APIs

1. i2c_master_gpio_init

Function:
Set GPIO in I2C master mode

Prototype:
`void i2c_master_gpio_init (void)`

Input Parameters:
null

Return:
null

2. i2c_master_init

Function:
Initialize I2C

Prototype:
`void i2c_master_init(void)`

Input Parameters:
null

Return:
null

3. i2c_master_start

Function: configures I2C to start sending data

Prototype:
`void i2c_master_start(void)`

Input Parameters:
null

Return:
null

4. i2c_master_stop

Function:
configures I2C to stop sending data

Prototype:

```
void i2c_master_stop(void)
```

Input Parameters:

null

Return:

null

5. i2c_master_send_ack

Function:

Sends I2C ACK

Prototype:

```
void i2c_master_send_ack (void)
```

Input Parameters:

null

Return:

null

6. i2c_master_send_nack

Function:

Sends I2C NACK

Prototype:

```
void i2c_master_send_nack (void)
```

Input Parameters:

null

Return:

null

7. i2c_master_checkAck

Function:

Checks ACK from slave

Prototype:

```
bool i2c_master_checkAck (void)
```

Input Parameters:

null

```
Return:
    true:  get I2C slave ACK
    false: get I2C slave NACK
```

8. i2c_master_readByte

```
Function:
    Read one byte from I2C slave

Prototype:
    uint8 i2c_master_readByte (void)

Input Parameters:
    null

Return:

    uint8 : the value that was read
```

9. i2c_master_writeByte

```
Function:
    Write one byte to slave

Prototype:
    void i2c_master_writeByte (uint8 wrdata)

Input Parameters:
    uint8 wrdata : data to write

Return:
    null
```

7.4. PWM Related

ESP9266 supports 4x PWM outputs. More details can be found in `pwm.h`. It is possible to increase the number of PWM outputs, but it is beyond the scope of this document.

1. pwm_init

```
Function:
    Initialize PWM function, including GPIO selection, frequency and duty cycle.

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

```
Input Parameters:
    uint16 freq : PWM frequency;
    uint8 *duty : duty cycle of each output

Return:
    null
```

2. pwm_start

```
Function:
    Starts PWM. This function needs to be called after PWM config is changed.

Prototype:
    void pwm_start (void)

Parameter:
    null

Return:
    null
```

3. pwm_set_duty

```
Function:
    Sets duty cycle of an output

Prototype:
    void pwm_set_duty(uint8 duty, uint8 channel)

Input Parameters:
    uint8 duty : duty cycle
    uint8 channel : an output

Return:
    null
```

4. pwm_set_freq

```
Function:
    Sets PWM frequency

Prototype:
    void pwm_set_freq(uint16 freq)

Input Parameters:
    uint16 freq : PWM frequency

Return:
    null
```

5. pwm_get_duty

Function:

Gets duty cycle of PWM output

Prototype:

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

Input Parameters:

`uint8 channel` : channel of which to get duty cycle

Return:

`uint8` : duty cycle

6. pwm_get_freq

Function:

Gets PWM frequency.

Prototype:

```
uint16 pwm_get_freq(void)
```

Input Parameters:

null

Return:

`uint16` : frequency

8. Appendix

8.1. ESPCONN Programming

1. TCP Client Mode

Notes

- ESP8266, working in Station mode, will start client connection when given an IP address.
- ESP8266, working in soft-AP mode, will start client connection when the devices which are connected to ESP8266 are given an IP address.

Steps

- Initialize `espconn` parameters according to protocols.
- Register connect callback function, and register reconnect callback function.
  - (Call `espconn_regist_connectcb` and `espconn_regist_reconcb`)
- Call `espconn_connect` function and set up the connection with TCP Server.
- Registered connected callback function will be called after successful connection, which will register the corresponding callback function. Recommend to register disconnect callback function.
  - (Call `espconn_regist_recvcb` , `espconn_regist_sentcb` and `espconn_regist_disconcb` in connected callback)
- When using receive callback function or sent callback function to run disconnect, it is recommended to set a time delay to make sure that the all the firmware functions are completed.

2. TCP Server Mode

Notes

- If ESP8266 is in Station mode, it will start server listening when given an IP address.
- If ESP8266 is in soft-AP mode, it will start server listening.

Steps

- Initialize espconn parameters according to protocols.
- Register connect callback and reconnect callback function.
  - (Call espconn_regist_connectcb and espconn_regist_reconcb)
- Call espconn_accept function to listen to the connection with host.

- Registered connect function will be called after successful connection, which will register corresponding callback function.
  - (Call `espconn_regist_recvcb` , `espconn_regist_sentcb` and `espconn_regist_disconcb` in connected callback)

8.2. RTC APIs Example

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

```
void user_init(void) {
    os_printf(clk cal : %d /n/r,system_rtc_clock_cali_proc()>>12);
    uint32 rtc_time = 0, rtc_reg_val = 0,stime = 0,rtc_time2 = 0,stime2 = 0;
    rtc_time =  system_get_rtc_time();
    stime = system_get_time();

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

    if( system_rtc_mem_read(0, &rtc_reg_val, 4)) {
        os_printf(rtc mem val : 0x%08x/n/r,rtc_reg_val);
    } else {
        os_printf(rtc mem val error/n/r);
    }
    rtc_reg_val++;
    os_printf(rtc mem val write/n/r);
    system_rtc_mem_write(0, &rtc_reg_val, 4) ;

    if( system_rtc_mem_read(0, &rtc_reg_val, 4) ){
        os_printf(rtc mem val : 0x%08x/n/r,rtc_reg_val);
    } else {
        os_printf(rtc mem val error/n/r);
    }
    rtc_time2 =  system_get_rtc_time();
    stime2 = system_get_time();

    os_printf(rtc time : %d /n/r,rtc_time2);
    os_printf(system time : %d /n/r,stime2);
    os_printf(delta time rtc: %d /n/r,rtc_time2-rtc_time);
    os_printf(delta system  time rtc: %d /n/r,stime2-stime);
    os_printf(clk cal : %d /n/r,system_rtc_clock_cali_proc()>>12);
```

```
        os_delay_us(500000);
        system_restart();
    }
```

8.3. Sniffer 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
        u16 len[1];  //length of packet
    };
```

Callback `wifi_promiscuous_rx` has two parameters (`buf` and `len`). `len` means the length of `buf`, `len` = 12 or `len` ≥ 60:

Case of LEN ≥ 60

- 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.len[0], every 2 bytes represent a length information of next packet, len[0] represents the length of first packet. If there are two packets where (sniffer_buf.cnt == 2), len[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:

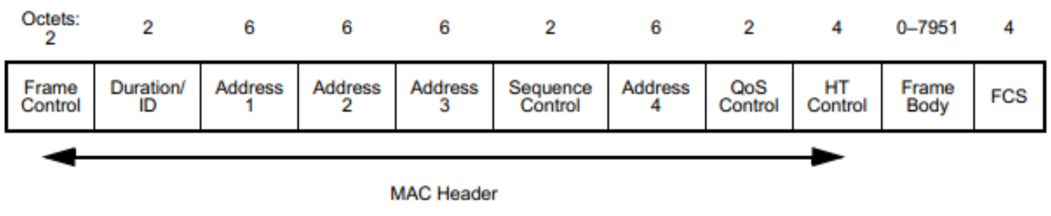

Figure 8-30—Data frame

- 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.