ESP8266 Non-OS SDK

API Reference

Version 2.0.0

About This Guide

This document lists ESP8266_NONOS_SDK APIs.

The document is structured as follows.

Chapter	Title	Subject
Chapter 1	Preambles	An instruction to the ESP8266EX
Chapter 2	Overview	An overview of the ESP8266_NONOS_SDK
Chapter 3	System APIs	Important system APIs for timer control, SPI flash operations, Wi-Fi radio control and OTA firmware upgrade
Chapter 4	TCP/UDP APIs	APIs for TCP/UDP operation
Chapter 5	Mesh APIs	Mesh APIs for the ESP8266
Chapter 6	Application Related	Application specific APIs (AT command set and JSON parser)
Chapter 7	Definitions & Structures	Important definitions and data structures
Chapter 8	Peripheral Related Drivers	APIs for peripheral interfacing (GPIO, UART, I2C, PWM and SDIO)
Chapter 9	Appendix	Other relevant information

Release Notes

Date	Version	Release notes
2016.01	V1.5.2	First Release.
2016.03	V1.5.2	Updated Chapter 3.2, Chapter 9.5 and Chapter 3.3.37.
2016.04	V1.5.3	Added Chapter 3.5.11 and Chapter 3.5.12. Updated Chapter 3.5.67 and Chapter 3.7.9.
2016.06	V1.5.4	Added Chapter 3.3.8 and Chapter 3.7.8. Added Chapter 3.3.46, Chapter 3.3.47, and Chapter 3.3.48 Updated Chapter 3.7
2016.07	V2.0.0	Added Chapter 3.9, Chapter 3.14 , Chapter 3.3.48, Chapter 3.5.72 and Chapter 3.5.73 Updated Chapter 3.8.6, Chapter 3.5.65

Table of Contents

1. Preambles

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

ESP8266EX is amongst the most integrated Wi-Fi chips 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, on top of its Wi-Fi functionalities. ESP8266EX is often integrated with external sensors and other application specific devices through its GPIOs. Codes for such applications are provided as examples in the SDK. The examples also demonstrate the use APIs listed in this document.

Sophisticated system-level features include fast sleep/wake switching for energy-efficient VoIP, adaptive radio biasing for low-power operations, advanced signal processing, 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 detailed description of the APIs. 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 layers so programmers can focus on application development at a higher level. Users can easily make use of the corresponding interfaces to receive and transmit data.

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 it.

void user_rf_pre_init (void) is to be added in user_main.c since ESP8266_NONOS_SDK_V1.1.0 and is provided for RF initialization. User can call system_phy_set_rfoption to set RF option in user_rf_pre_init, or call system_deep_sleep_set_option before deep-sleep. If RF is disabled, ESP8266 station and soft-AP will both be disabled, so the related APIs must not be called, and Wi-Fi function can not be used either.

⚠ **Notice:**

- Using non-OS SDK which is single-threaded, the CPU should not take long to execute tasks:
  - If a task occupies the CPU for too long, and the watchdog cannot be fed, it will cause a watchdog reset.
  - If interrupt is disabled, CPU can only be occupied in us range and the time should not be more than 10 us; if interrupt is not disabled, it is suggested that CPU should not be occupied more than 500 ms.
  - It is suggested that interrupt should not be disabled, if it has to be disabled, the disabled time should be kept to a bare minimum.
- We suggest using a timer to check periodically, if users need to call function os_delay_us or while, or for in timer callback, please do not occupy CPU more than 15 ms.
- Using non-OS SDK, please do not call any function defined with ICACHE_FLASH_ATTR from inside an interrupt handler.
- We suggest using RTOS SDK, RTOS to schedule different tasks.
- Read and write RAM has to be aligned by 4 bytes, so please do not cast pointer directly, for example, please use os_memcpy instead of float temp = *((float*)data);.
- For printing logs inside interrupt handlers, please use API os_printf_plus. Printing excess logging data from inside an interrupt handler may occupy the CPU for too long, causing errors.
- esp_init_data.bin has to be downloaded into flash at least once.
- user_rf_cal_sector_set has to be added in the user application.

3. Application Programming Interface (APIs)

3.1. Software Timer

Timer APIs can be found in: **/ESP8266_NONOS_SDK/include/osapi.h**.

Please note that os_timer APIs listed below are software timers executed in task, hence timer callbacks may not be precisely executed at the right time; it depends on priority. If you need a precise timer, please use a hardware timer which can be executed in hardware interrupt. Please refer to hw_timer.c.

- 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 (
    os_timer_t *ptimer,
    uint32_t milliseconds,
    bool repeat_flag
)
```

Parameters:

os_timer_t *ptimer : Timer structure

uint32_t milliseconds : Timing, Unit: millisecond

- if system_timer_reinit has been called, the timer value allowed range from 100 to 0x689D0.
- if didn't call system_timer_reinit has NOT been called, the timer value allowed range from 5 to 0x68D7A3.

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 (os_timer_t *ptimer)

Parameters:

os_timer_t *ptimer : Timer structure

Return:

null

3. os_timer_setfn

Function:

Set timer callback function. The timer callback function must be set before arming a timer.

Prototype:

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

Parameters:

os_timer_t *ptimer : Timer structure

os_timer_func_t *pfunction : timer callback function, use typecasting to pass function as (os_timer_func_t *) your_function

void *parg : callback function parameter

Return:

null

4. 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 of user_init, in the first sentence.

Prototype:

void system_timer_reinit (void)

Parameters:

null

Return:

null

5. os_timer_arm_us

Function:

Enable a microsecond timer.

Notes:

1. Define USE_US_TIMER, and put system_timer_reinit at the beginning of user_init, in the first sentence.
2. The highest precision is 500 us.

Prototype:

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

Parameters:

os_timer_t *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. Hardware Timer

Hardware timer APIs can be found in **/ESP8266_NONOS_SDK/examples/driver_lib/hw_timer.c**. User can use it according to "readme.txt" which can be found in the folder driver_lib.

Notes:

- If NMI is used as the ISR source for auto-loading the timer, parameter val of hw_timer_arm can not be less than 100.
- When NMI source is used, the timer has the highest priority. It can interrupt other ISRs. FRC1 source should be used to prevent the timer from interrupting other ISRs.
- APIs in hw_timer.c can not be called when PWM APIs are in use, because they all use the same hardware timer.
- The hardware timer callback function must NOT be defined with ICACHE_FLASH_ATTR.
- The system must not be allowed to enter light sleep mode (wifi_set_sleep_type(LIGT_SLEEP)) when hardware timer is enabled. Light sleep stops the CPU and it can not be interrupted by NMI.

1. hw_timer_init

Function:

Initialize the hardware ISR timer

Prototype:

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

Parameters:

FRC1_TIMER_SOURCE_TYPE source_type : ISR source of timer

FRC1_SOURCE, timer uses FRC1 ISR as ISR source.

NMI_SOURCE, timer uses NMI ISR as ISR source.

u8 req : 0, autoload disabled

1, autoload enabled

Return:

none

2. hw_timer_arm

Function:

Set a trigger timer delay to enable this timer.

Prototype:

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

Parameters:

uint32 val : Timing

- in autoload mode :
  - For FRC1_SOURCE, range : 50 ~ 0x199999;
  - For NMI_SOURCE, range : 100 ~ 0x199999;
- in non autoload mode, range : 10 ~ 0x199999;

Return:

none

3. hw_timer_set_func

Function:

Set timer callback function.

The timer callback function must be set before arming a timer.

Notes:

ICACHE_FLASH_ATTR is not allowed to be added before the timer callback.

Prototype:

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

Parameters:

void (* user_hw_timer_cb_set)(void) : Timer callback function, must NOT be defined as ICACHE_FLASH_ATTR.

Return:

none

4. Hardware timer example

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

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

```
        j = 0;
    }
}

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

3.3. System APIs

System APIs can be found in: **/ESP8266_NONOS_SDK/include/user_interface.h**.

os_XXX APIs can be found in: **/ESP8266_NONOS_SDK/include/osapi.h**.

1. system_get_sdk_version

Function:

Get SDK version

Prototype:

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

Parameter:

none

Return:

SDK version

Example:

```
os_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, wifi_set_opmode, and APs information recorded by #define AP_CACHE

Note:

Call system_restart to restart after reset by system_restore.

Prototype:

void system_restore(void)

Parameters:

null

Return:

null

3. system_restart

Function:

Restart

Note:

The ESP8266 will not restart immediately, please do not call other functions after calling this API.

Prototype:

void system_restart(void)

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 initialization is done and station is enabled.

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)

Parameters:

null

Return:

Chip ID

6. 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.
- The return value of system_get_vdd33 may be a little different in different WiFi mode, for example, in modem sleep mode or in normal WiFi working mode.

Prototype:

uint16 system_get_vdd33(void)

Parameter:

none

Return:

power voltage of VDD33, unit:1/1024 V

7. system_adc_read

Function:

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

Note:

- system_adc_read is only available when TOUT pin is wired 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 TOUT pin is wired to external circuitry, the vdd33_const must be set as real power voltage of VDD3P3 pin 3 and 4, and has to be less than 0xFF.
- 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]. If vdd33_const is an ineffective value in the range of [0, 18) or (36, 255), ESP8266 RF calibration will use 3.3V by default.
- The return value of system_adc_read may be a little different in different WiFi mode, for example, in modem sleep mode or in normal WiFi working mode.
- If high precision is needed, please use system_adc_read_fast instead.

Prototype:

uint16 system_adc_read(void)

Parameter:

none

Return:

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

8. system_adc_read_fast

Function:

Fast and high-precision sampling of ADC.

Note:

- system_adc_read_fast is only available when TOUT pin is wired 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 TOUT pin is wired to external circuitry, the vdd33_const must be set as real power voltage of VDD3P3 pin 3 and 4, and has to be less than 0xFF.
- 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]. If vdd33_const is an ineffective value in the range of [0, 18) or (36, 255), ESP8266 RF calibration will use 3.3V by default.

- To use system_adc_read_fast, WiFi has to be disabled. And if ADC continuously sampling is needed, all interrupts have to be disabled, so PWM or NMI hardware timer can not be used when system_adc_read_fast is calling.

Prototype:

void system_adc_read_fast (uint16 *adc_addr, uint16 adc_num, uint8 adc_clk_div)

Parameter:

uint16 *adc_addr : point to the address of ADC continuously fast sampling output.

uint16 adc_num : sampling number of ADC continuously fast sampling, range [1, 65535].

uint8 adc_clk_div : ADC working clock = 80M/adc_clk_div, range [8, 32], the recommended value is 8.

Return:

none

Example:

```
extern void system_adc_read_fast(uint16 *adc_addr, uint16 adc_num, uint8 adc_clk_div);

os_timer_t timer;


void ICACHE_FLASH_ATTR ADC_TEST(void *p)
{
        wifi_set_opmode(NULL_MODE);
        ets_intr_lock( );    //close interrupt

        uint16 adc_addr[10];
        uint16 adc_num = 10;
        uint8 adc_clk_div = 8;
        uint32 i;
        system_adc_read_fast(adc_addr, adc_num, adc_clk_div);

        for(i=0; i<adc_num; i++)
                os_printf("i=%d, adc_v=%d\n", i, adc_addr[i]);

        ets_intr_unlock();              //open interrupt

        os_timer_disarm(&timer);
        os_timer_setfn(&timer, ADC_TEST, NULL);
```

```
        os_timer_arm(&timer,1000,1);
}
```

9. 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 0 ohm resistor).
- 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)

P**arameters**:

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

Return:

null

10. system_deep_sleep_set_option

Function:

Call this API before system_deep_sleep to set whether the chip will do RF calibration or not when it wakes up from deep-sleep again. The option is 1 by default.

Prototype:

bool system_deep_sleep_set_option(uint8 option)

Parameter:

uint8 option :

0 : RF calibration after deep-sleep wake up depends on both the times of entering deep-sleep (deep_sleep_number, returns to 0 in every power up) and the byte 108 of esp_init_data_default.bin(0～127byte).

- if deep_sleep_number <= byte 108, no RF calibration after deep-sleep wake up; this reduces the current consumption.
- if deep_sleep_number = byte 108 +1, the behavior after deep-sleep wake up will be the same as power-up, and deep_sleep_number returns to 0.

1 : the behavior after deep-sleep wake up will be the same as power-up.

2 : No RF calibration after deep-sleep wake up; this reduces the current consumption.

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 : Success

false: Failure

11. 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 at least once.

Prototype:

void system_phy_set_rfoption(uint8 option)

Parameter:

uint8 option :

0 : RF calibration after deep-sleep wake up depends on both the times of entering deep-sleep (deep_sleep_number, returns to 0 in every power up) and the byte 108 of esp_init_data_default.bin(0～127byte).

- if deep_sleep_number <= byte 108, no RF calibration after deep-sleep wake up; this reduces the current consumption.
- if deep_sleep_number = byte 108 + 1, the behavior after deep-sleep wake up will be the same as power-up, and deep_sleep_number returns to 0.

1 : the behavior after deep-sleep wake up will be the same as power-up.

2 : No RF calibration after deep-sleep wake up; this reduces the current consumption.

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

12. system_phy_set_powerup_option

Function:

Set whether the chip will do RF calibration or not when power up. The option is 0 by default.

Prototype:

void system_phy_set_powerup_option(uint8 option)

Parameter:

uint8 option : RF initialization when power up.

0 : RF initialization when power up depends on esp_init_data_default.bin(0～127byte) byte 114. More details in ESP8266_Getting_Started_Guide.

1 : RF initialization only calibrate VDD33 and TX power which will take about 18 ms; this reduces the current consumption.

2 : RF initialization only calibrate VDD33 which will take about 2 ms; this has the least current consumption.

3 : RF initialization will do the whole RF calibration which will take about 200 ms; this increases the current consumption.

Return:

none

13. 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

14. 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 TOUT pin is wired 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

15. system_set_os_print

Function:

Turn log printing on or off.

Prototype:

void system_set_os_print (uint8 onoff)

Parameters:

uint8 onoff

Note:

onoff==0: print function off

onoff==1: print function on

Default:

print function on

Return:

none

16. system_print_meminfo

Function:

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

Prototype:

void system_print_meminfo (void)

Parameters:

none

Return:

none

17. system_get_free_heap_size

Function:

Get free heap size

Prototype:

```
uint32 system_get_free_heap_size(void)
```

Parameters:

none

Return:

uint32 : available heap size

18. system_os_task

Function:

Set up tasks

Prototype:

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

Parameters:

os_task_t task : task function

uint8 prio : task priority. 3 priorities are supported: 0/1/2; 0 is the lowest priority. This means only 3 tasks are allowed to set up.

os_event_t *queue : message queue pointer

uint8 qlen : message queue depth

Return:

true: Success

false: Failure

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);
}
```

19. system_os_post

Function: send message to task

Prototype:

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

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: Success

false: Failure

Referring to the above example:

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

Printout:

sig_rx a

20. system_get_time

Function:

Get system time (us).

Prototype:

uint32 system_get_time(void)

Parameter:

none

Return:

System time in microsecond.

21. 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 contains random value, RTC timer starts from zero
- reset by pin CHIP_EN : RTC memory contains random value, RTC timer starts from zero

Prototype:

uint32 system_get_rtc_time(void)

Parameter:

none

Return:

RTC time

22. system_rtc_clock_cali_proc

Function:

Get RTC clock period.

Note:

RTC clock period has decimal part.

RTC clock period tends to drift with change in temperature, so RTC timer is not very precise.

Prototype:

```
uint32 system_rtc_clock_cali_proc(void)
```

Parameter:

none

Return:

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

Note:

see RTC demo in Appendix.

Example:

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

23. system_rtc_mem_write

Function:

Writes data to the RTC memory.

During deep sleep mode, the RTC is still operational and can store user data in the defined user-data area.

|<------system data (256 bytes)------->|<-----------------user data (512 bytes)---------->|

Note:

Data read/write accesses to the RTC memory must be word aligned (4 bytes boundary aligned). Parameter des_addr means block number(4 bytes per block). For example, to save 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: Success

false: Failure

24. system_rtc_mem_read

Function:

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

|<------system data (256 bytes)------->|<----------------user data (512 bytes)---------->|

Note:

Data read/write accesses to the RTC memory must be word aligned (4 bytes boundary aligned). Parameter src_addr means block number(4 bytes per block). For example, 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: Success

false: Failure

25. 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. MTDO (U0RTS) and MTCK (U0CTS) also needs to be used as UART0 in hardware.

Prototype:

```
void system_uart_swap (void)
```

Parameter:

none

Return:

none

26. system_uart_de_swap

Function:

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

Prototype:

void system_uart_de_swap (void)

Parameter:

none

Return:

none

27. system_get_boot_version

Function:

Get version info of boot

Prototype:

uint8 system_get_boot_version (void)

Parameter:

none

Return:

Version info of boot.

Note:

If boot version >= 3, it is possible to enable enhanced boot mode (refer to system_restart_enhance)

28. 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:

none

Return:

Start address info of the current running user bin.

29. system_get_boot_mode

Function: Get boot mode.

Prototype:

```
uint8 system_get_boot_mode (void)
```

Parameter:

none

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).

30. 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: Success

false: Failure

Note:

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

31. system_update_cpu_freq

Function:

Set CPU frequency. Default is 80MHz.

Note:

System bus frequency is 80MHz, and it is not affected by CPU frequency. The frequency of UART, SPI, or other peripheral devices, are divided from system bus frequency, so they will not be affected by CPU frequency either.

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: Success

false: Failure

32. system_get_cpu_freq

Function:

Get CPU frequency.

Prototype:

```
uint8 system_get_cpu_freq(void)
```

Parameter:

none

Return:

CPU frequency, unit : MHz.

33. system_get_flash_size_map

Function:

Get current flash size and flash map.

Flash map depends on the selection when compiling, more details in documentation "2A-ESP8266-SDK_Getting_Started_Guide"

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

34. 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
        REANSON_EXT_SYS_RST= 6,         // external system reset
};
```

```
struct rst_info {
        uint32 reason;      // enum rst_reason
        uint32 exccause;
        uint32 epc1;        // the address that error occurred
        uint32 epc2;
        uint32 epc3;
        uint32 excvaddr;
        uint32 depc;
};
```

Prototype:

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

Parameter:

none

Return:

Information about startup.

35. system_soft_wdt_stop

Function:

Stop software watchdog

Note:

The software watchdog must not be stopped for too long (over 6 seconds), otherwise it will trigger hardware watchdog reset.

Prototype:

```
void system_soft_wdt_stop(void)
```

Parameter:

none

Return:

none

36. system_soft_wdt_restart

Function:

Restart software watchdog

Note:

This API can only be called if software watchdog is stopped (system_soft_wdt_stop)

Prototype:

void system_soft_wdt_restart(void)

Parameter:

none

Return:

none

37. system_soft_wdt_feed

Function:

Feed software watchdog

Note:

This API can only be called if software watchdog is enabled.

Prototype:

void system_soft_wdt_feed(void)

Parameter:

none

Return:

none

38. system_show_malloc

Function:

For debugging memory leak issue, to print the memory usage.

Note:

- To use this API, users need to enable #define MEMLEAK_DEBUG in user_config.h, then refer to the note which is at the beginning of ESP8266_NONOS_SDK\included\mem.h.
- The memory usage which cause memory leak issue may be in the logs, not ensure, just for reference.
- This API is only for debugging. After calling this API, the program may go wrong, so please do not call it in normal usage.

Prototype:

void system_show_malloc(void)

Parameter:

none

Return:

none

39. os_memset

Function:

Set value of memory

Prototype:

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

Parameter:

void *s : pointer of memory

int ch : set value

size_t n : size

Return:

none

Example:

```
uint8 buffer[32];

os_memset(buffer, 0, sizeof(buffer));
```

40. os_memcpy

Function:

Standard function for copying memory content.

Prototype:

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

Parameter:

void *des : pointer of destination

void *src : pointer of source

size_t n : memory size

Return:

none

Example:

```
uint8 buffer[4] = {0};

os_memcpy(buffer, "abcd", 4);
```

41. os_strlen

Function:

Get string length

Prototype:

os_strlen(char *s)

Parameter:

char *s : string

Return:

string length

Example:

```
char *ssid = "ESP8266";

os_memcpy(softAP_config.ssid, ssid, os_strlen(ssid));
```

42. os_printf

Function:

print format

Note:

- Default to be output from UART 0. uart_init in IOT_Demo can set baud rate of UART, and os_install_putc1((void *)uart1_write_char) in it will set os_printf to be output from UART 1.
- Continuously printing more than 125 bytes or repeated calls to this API may cause loss of print data.

Prototype:

os_printf(const char *s)

Parameter:

const char *s : string

Return:

none

Example:

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

43. os_bzero

Function:

Set the first n bytes of string p to be 0, include '\0'

Prototype:

void os_bzero(void *p, size_t n)

Parameter:

void *p : pointer of memory need to be set 0

size_t n : length

Return:

none

44. os_delay_us

Function:

Time delay, max : 65535 us

Prototype:

void os_delay_us(uint16 us)

Parameter:

uint16 us : time, unit: us

Return:

none

45. 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 os_printf to be output from UART 1, otherwise, os_printf default output from UART 0.

46. os_random

Function:

Get a random number.

Prototype:

unsigned long os_random(void)

Parameter:

none.

Return:

the random number

47. os_get_random

Function:

Get a random number of bits specified.

Prototype:

int os_get_random(unsigned char *buf, size_t len)

Parameter:

unsigned char *buf : pointer of the random number it gets

size_t len : specified bits of the random number

Return:

0 : Success

others : Failure

Example:

```
int ret = os_get_random((unsigned char *)temp, 7);
os_printf("ret %d, value 0x%08x%08x\n\r", ret, temp[1], temp[0]);
```

48. user_rf_cal_sector_set

Function:

Set the target flash sector to store RF_CAL parameters.

Note:

The user_rf_cal_sector_set has to be added in application, but need NOT to be called. It will be called inside the SDK.

The system parameter area (4 flash sectors) has already been used, so the RF_CAL parameters will be stored in the target sector set by user_rf_cal_sector_set. Since we do not know which sector is available in user data area, users need to set an available sector in the user_rf_cal_sector_set for the SDK to store RF_CAL parameter.

If the user_rf_cal_sector_set is not added in the application, the compilation will fail in link stage.

Download blank.bin to initialize the sector stored RF_CAL parameter, and download esp_init_data.bin into flash, when the system needs to be initialized, or RF needs to be calibrated again.

Prototype:

uint32 user_rf_cal_sector_set(void)

Parameter:

none

Return:

the target flash sector to store RF_CAL parameters

Example:

Set the 5th sector from the end of the flash to store the RF_CAL parameter.

```
uint32 user_rf_cal_sector_set(void)
{
    enum flash_size_map size_map = system_get_flash_size_map();
    uint32 rf_cal_sec = 0;

    switch (size_map) {
        case FLASH_SIZE_4M_MAP_256_256:
            rf_cal_sec = 128 - 5;
            break;
        case FLASH_SIZE_8M_MAP_512_512:
            rf_cal_sec = 256 - 5;
            break;
        case FLASH_SIZE_16M_MAP_512_512:
        case FLASH_SIZE_16M_MAP_1024_1024:
            rf_cal_sec = 512 - 5;
            break;
        case FLASH_SIZE_32M_MAP_512_512:
        case FLASH_SIZE_32M_MAP_1024_1024:
            rf_cal_sec = 512 - 5;
            break;
```

```
        default:
            rf_cal_sec = 0;
            break;
    }
    return rf_cal_sec;
}
```

3.4. SPI Flash Related APIs

SPI flash APIs can be found in: **/ESP8266_NONOS_SDK/include/spi_flash.h**.

system_param_xxx APIs can be found in: **/ESP8266_NONOS_SDK/include/user_interface.h**.

More details about flash read/write operation in documentation "99A-SDK-Espressif IOT Flash RW Operation"

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 aligned to the 4-bytes boundary.

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, uint: byte, has to be aligned to the 4-bytes boundary.

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 aligned to the 4-bytes boundary.

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, uint : byte, has to be aligned to the 4-bytes boundary.

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);

os_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 aligned to the 4-bytes boundary.

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 documentation "99A-SDK-Espressif IOT Flash RW Operation".

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: Success

false: Failure

6. system_param_load

Function:

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

Protection of flash read/write: use 3 sectors (4KBytes per sector) to save 4KB data with protection, 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 documentation "99A-SDK-Espressif IOT Flash RW Operation".

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 are used for flash read/write protection. It cannot be sectors 1 or 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: Success

false: Failure

7. spi_flash_set_read_func

Function:

Register user-defined SPI flash read API.

Note:

This API can be only used in SPI overlap mode, please refer to ESP8266_NONOS_SDK\driver_lib\driver\spi_overlap.c

Prototype:

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

Parameter:

user_spi_flash_read read : user-defined SPI flash read API

Parameter Definition:

```
typedef SpiFlashOpResult (*user_spi_flash_read)(
    SpiFlashChip *spi,
    uint32 src_addr,
    uint32 * des_addr,
    uint32 size
)
```

Return:

none

3.5. Wi-Fi Related APIs

Wi-Fi APIs can be found in: **/ESP8266_NONOS_SDK/include/user_interface.h**.

wifi_station_xxx APIs and other APIs which set/get configurations of the ESP8266 station can only be called if the ESP8266 station is enabled.

wifi_softap_xxx APIs and other APIs which set/get configurations of the ESP8266 soft-AP can only be called if the 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:

none

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:

none

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 ESP8266_NONOS_SDK_V0.9.2, need to call system_restart() after this api; after ESP8266_NONOS_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: Success

false: Failure

4. wifi_set_opmode_current

Function:

Sets WiFi working mode as station, soft-AP or station+soft-AP, and does not update 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: Success

false: Failure

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: Success

false: Failure

6. wifi_station_get_config_default

Function:

Get WiFi station configuration from flash memory

Prototype:

bool wifi_station_get_config_default (struct station_config *config)

Parameters:

struct station_config *config : WiFi station configuration pointer

Return:

true: Success

false: Failure

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: Success

false: Failure

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, setting in flash is not updated.

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: Success

false: Failure

9. wifi_station_set_cert_key

Function:

This API is deprecated, please use wifi_station_set_enterprise_cert_key instead.

Set certificate and private key for connecting to WPA2-ENTERPRISE AP.

Note:

- Connecting to WPA2-ENTERPRISE AP needs more than 26 KB memory, please ensure enough space (system_get_free_heap_size).

- So far, WPA2-ENTERPRISE can only support unencrypted certificate and private key, and only in PEM format.
  - Header of certificate: - - - - - BEGIN CERTIFICATE - - - - -
  - Header of private key: - - - - - BEGIN RSA PRIVATE KEY - - - - - or - - - - - BEGIN PRIVATE KEY - - - - -
- Please call this API to set certificate and private key before connecting to WPA2-ENTERPRISE AP and the application needs to hold the certificate and private key. Call wifi_station_clear_cert_key to release resources and clear status after connected to the target AP, and then the application can release the certificate and private key.
- If the private key is encrypted, please use openssl pkey command to change it to unencrypted file to use, or use openssl rsa related commands to change it (or change the start TAG).

Prototype:

```
bool wifi_station_set_cert_key (
        uint8 *client_cert, int client_cert_len,
        uint8 *private_key, int private_key_len,
        uint8 *private_key_passwd, int private_key_passwd_len,)
```

Parameter:

uint8 *client_cert : certificate, HEX array
int client_cert_len : length of certificate

uint8 *private_key : private key, HEX array, can NOT be longer than 2048 bits
int private_key_len : length of private key, less than 2048

uint8 *private_key_passwd : password for private key, to be supported, can only be NULL now.
int private_key_passwd_len : length of password, to be supported, can only be 0 now.

Return:

0 : Success
non-0 : Failure

Example:

For example, the private key is - - - - - BEGIN PRIVATE KEY - - - - - … … … …

Then then array should be uint8 key[]={0x2d, 0x2d, 0x2d, 0x2d, 0x2d, 0x42, 0x45, 0x47, … … 0x00 };

It is the ASCII of the characters, and the array needs to terminate with 0x00.

10. wifi_station_clear_cert_key

Function:

This API is deprecated, please use wifi_station_clear_enterprise_cert_key instead.

Release certificate and private key resources and clear related status after connected to the WPA2-ENTERPRISE AP.

Prototype:

void wifi_station_clear_cert_key (void)

Parameter:

none

Return:

none

11. wifi_station_set_username

Function:

This API is deprecated, please use wifi_station_set_enterprise_username instead.

Set ESP8266 station's user name for connecting to WPA2-ENTERPRISE AP.

Prototype:

int wifi_station_set_username (uint8 *username, int len)

Parameter:

uint8 *username : the user name

int len : length of user name

Return:

0 : Success

non-0 : Failure

12. wifi_station_clear_username

Function:

This API is deprecated, please use wifi_station_clear_enterprise_username instead.

Release the user name resources and clear related status after connected to the WPA2-ENTERPRISE AP.

Prototype:

void wifi_station_clear_username (void)

Parameter:

none

Return:

none

13. wifi_station_connect

Function:

To connect WiFi station to AP

Note:

- If the ESP8266 is already connected to a router, wifi_station_disconnect must be called first, before calling wifi_station_connect.
- Do not call this API in user_init. This API need to be called after system initializes and the ESP8266 station mode is enabled.

Prototype:

bool wifi_station_connect (void)

Parameters:

none

Return:

true: Success

false: Failure

14. 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 initializes and the ESP8266 station mode is enabled.

Prototype:

bool wifi_station_disconnect (void)

Parameters:

none

Return:

true: Success

false: Failure

15. wifi_station_get_connect_status

Function:

Get WiFi connection status of ESP8266 station to AP.

Notice:

If in a special case, called wifi_station_set_reconnect_policy to disable reconnect, and did not call wifi_set_event_handler_cb to register WiFi event handler, wifi_station_get_connect_status becomes invalid and can not get the right status.

Prototype:

```
uint8 wifi_station_get_connect_status (void)
```

Parameters:

none

Return:

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

16. 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 initializes and the ESP8266 station mode is enabled.

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: Success

false: Failure

17. 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 were found, refer to struct bss_info

STATUS status: get status

Return:

none

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;
        ...
    }
}
```

18. 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 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 that can be recorded (MAX: 5)

Return:

true: Success

false: Failure

19. 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);
```

20. 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: Success

false: Failure

21. 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.

22. 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:

none

Return:

0: wil not connect to AP automatically;

Non-0: will connect to AP automatically.

23. wifi_station_set_auto_connect

Function:

Setting the ESP8266 station to connect to the AP (which is recorded) automatically or not when powered on. Enable auto-connect by default.

Note:

When called from user_init, the feature is effective from the current session. When called from elsewhere, the changes take affect after the subsequent power cycle.

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: Success

false: Failure

24. wifi_station_dhcpc_start

Function:

Enable ESP8266 station DHCP client.

Note:

DHCP is enabled by default.

This configuration interacts with static IP API (wifi_set_ip_info):

If DHCP is enabled, static IP will be disabled;

If static IP is enabled, DHCP will be disabled;

These settings depend on the last configuration.

Prototype:

bool wifi_station_dhcpc_start(void)

Parameter:

none

Return:

true: Success

false: Failure

25. wifi_station_dhcpc_stop

Function:

Disable ESP8266 station DHCP client.

Note:

DHCP is enabled by default.

Prototype:

bool wifi_station_dhcpc_stop(void)

Parameter:

none

Return:

true: Success

false: Failure

26. wifi_station_dhcpc_status

Function: Get ESP8266 station DHCP client status.

Prototype:

```
enum dhcp_status wifi_station_dhcpc_status(void)
```

Parameter:

none

Return:

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

27. wifi_station_dhcpc_set_maxtry

Function:

Set the maximum number that ESP8266 station DHCP client will try to reconnect to the AP.

Prototype:

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

Parameter:

uint8 num - the maximum number count

Return:

true: Success

false: Failure

28. wifi_station_set_reconnect_policy

Function:

Set whether the ESP8266 will attempt to reconnect to an AP if disconnected.

Note:

It is recommended that the API be called from user_init

This API can only be called when the ESP8266 station is enabled.

Prototype:

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

Parameter:

bool set : true, enable reconnect; false, disable reconnect

Return:

true: Success

false: Failure

29. wifi_station_get_rssi

Function:

Get rssi of the AP to which the ESP8266 is connected.

Prototype:

sint8 wifi_station_get_rssi(void)

Parameter:

none

Return:

31 : Failure, invalid value.

others : Success, value of rssi, in general, rssi value < 10

30. wifi_station_set_hostname

Function:

Set ESP8266 station DHCP hostname

Prototype:

bool wifi_station_get_hostname(char* hostname)

Parameter:

char* hostname :hostname, max length:32

Return:

true: Success

false: Failure

31. wifi_station_get_hostname

Function:

Get ESP8266 station DHCP hostname

Prototype:

char* wifi_station_get_hostname(void)

Parameter:

none

Return:

hostname

32. 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 configuration information

Return:

true: Success

false: Failure

33. wifi_softap_get_config_default

Function:

Get WiFi soft-AP configurations saved in flash

Prototype:

bool wifi_softap_get_config_default(struct softap_config *config)

Parameter:

struct softap_config *config : ESP8266 soft-AP configuration information

Return:

true: Success

false: Failure

34. wifi_softap_set_config

Function:

Set WiFi soft-AP configuration and save it to flash

Note:

- This API can be called only if the ESP8266 soft-AP is enabled.
- This configuration will be saved in flash system parameter area if changed.

- In soft-AP + station mode, the ESP8266 soft-AP will adjust its channel configuration to be the as same as the 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: Success

false: Failure

35. wifi_softap_set_config_current

Function:

Set WiFi soft-AP configuration, settings are not updated in flash memory.

Note:

- This API can be called only if the ESP8266 soft-AP is enabled.
- In the soft-AP + station mode, ESP8266 soft-AP will adjust its channel configuration to be as same as the 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: Success

false: Failure

36. wifi_softap_get_station_num

Function:

count the number of stations connected to the ESP8266 soft-AP

Prototype:

uint8 wifi_softap_get_station_num(void)

Parameter:

none

Return:

Number of stations connected to ESP8266 soft-AP

37. wifi_softap_get_station_info

Function:

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

Note:

This API depends on DHCP, so it cannot get static IP, etc in case DHCP is not used.

Prototype:

```
struct station_info * wifi_softap_get_station_info(void)
```

Input Parameters:

none

Return:

struct station_info* : station information structure

38. 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:

none

Return:

none

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
```

39. wifi_softap_dhcps_start

Function: Enable ESP8266 soft-AP DHCP server.

Note:

DHCP is enabled by default.

This configuration interacts with static IP API (wifi_set_ip_info):

If DHCP is enabled, static IP will be disabled;

If static IP is enabled, DHCP will be disabled;

This will depend on the last configuration.

Prototype:

```
bool wifi_softap_dhcps_start(void)
```

Parameter:

none

Return:

true: Success

false: Failure

40. wifi_softap_dhcps_stop

Function: Disable ESP8266 soft-AP DHCP server.

Note: DHCP is enabled by default.

Prototype:

```
bool wifi_softap_dhcps_stop(void)
```

Parameter:

none

Return:

true: Success

false: Failure

41. wifi_softap_set_dhcps_lease

Function:

Set the IP range that can be allocated by the ESP8266 soft-AP DHCP server.

Note:

- IP range has to be in the same sub-net with the ESP8266 soft-AP IP address
- This API can only be called when DHCP server is disabled (wifi_softap_dhcps_stop)
- This configuration only takes effect on next wifi_soft-AP_dhcps_start; if then wifi_softap_dhcps_stop is called, user needs to call this API to set IP range again if needed, and then call wifi_softap_dhcps_start for the configuration 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: Success

false: Failure

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();
}
```

42. wifi_softap_get_dhcps_lease

Function:

Query the IP range that can be allocated by the ESP8266 soft-AP DHCP server.

Note:

This API can only be called when ESP8266 soft-AP DHCP server is enabled.

Prototype:

bool wifi_softap_get_dhcps_lease(struct dhcps_lease *please)

Return:

true: Success

false: Failure

43. wifi_softap_set_dhcps_lease_time

Function:

Set ESP8266 soft-AP DHCP server lease time, default is 120 minutes.

Note:

This API can only be called when ESP8266 soft-AP DHCP server is enabled.

Prototype:

bool wifi_softap_set_dhcps_lease_time(uint32 minute)

Parameter:

uint32 minute : lease time, uint: minute, range:[1, 2880].

Return:

true: Success;

false: Failure

44. wifi_softap_get_dhcps_lease_time

Function:

Get ESP8266 soft-AP DHCP server lease time

Note:

This API can only be called when ESP8266 soft-AP DHCP server is enabled.

Prototype:

uint32 wifi_softap_get_dhcps_lease_time(void)

Return:

lease time, uint: minute.

45. wifi_softap_reset_dhcps_lease_time

Function:

Reset ESP8266 soft-AP DHCP server lease time to its default value, which is 120 minutes.

Note:

This API can only be called when ESP8266 soft-AP DHCP server is enabled.

Prototype:

bool wifi_softap_reset_dhcps_lease_time(void)

Return:

true: Success;

false: Failure

46. wifi_softap_dhcps_status

Function: Get ESP8266 soft-AP DHCP server status.

Prototype:

enum dhcp_status wifi_softap_dhcps_status(void)

Parameter:

none

Return:

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

47. 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 : enabled by default

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

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

Return:

true : Success
false: Failure

Example:

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

48. wifi_set_phy_mode

Function:

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

Note:

- ESP8266 soft-AP only support 802.11b/g.
- Users can set to be 802.11g mode for consumption.

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 : Success

false: Failure

49. wifi_get_phy_mode

Function:

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

Prototype:

enum phy_mode wifi_get_phy_mode(void)

Parameter:

none

Return:

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

50. wifi_get_ip_info

Function:

Get IP info of WiFi station or soft-AP interface

Note:

This API is available after initialization, do not call it in user_init.

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: Success

false: Failure

51. 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 static IP is enabled, DHCP will be disabled;

If DHCP is enabled, 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:

```
wifi_set_opmode(STATIONAP_MODE); //Set softAP + station mode

struct ip_info info;

wifi_station_dhcpc_stop();

wifi_softap_dhcps_stop();
```

```
IP4_ADDR(&info.ip, 192, 168, 3, 200);
IP4_ADDR(&info.gw, 192, 168, 3, 1);
IP4_ADDR(&info.netmask, 255, 255, 255, 0);
wifi_set_ip_info(STATION_IF, &info);

IP4_ADDR(&info.ip, 10, 10, 10, 1);
IP4_ADDR(&info.gw, 10, 10, 10, 1);
IP4_ADDR(&info.netmask, 255, 255, 255, 0);
wifi_set_ip_info(SOFTAP_IF, &info);

wifi_softap_dhcps_start();
```

Return:

true: Success

false: Failure

52. 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 addresses, please do not set them to be the same.
- The bit 0 of the first byte of ESP8266 MAC address can not be 1. For example, MAC address can be "1a:XX:XX:XX:XX:XX", but can not be "15:XX:XX:XX:XX:XX".

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:

```
wifi_set_opmode(STATIONAP_MODE);
```

```
char sofap_mac[6] = {0x16, 0x34, 0x56, 0x78, 0x90, 0xab};
char sta_mac[6] = {0x12, 0x34, 0x56, 0x78, 0x90, 0xab};
wifi_set_macaddr(SOFTAP_IF, sofap_mac);
wifi_set_macaddr(STATION_IF, sta_mac);
```

Return:

true: Success

false: Failure

53. wifi_get_macaddr

Function: Get MAC address

Prototype:

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

Parameter:

uint8 if_index : get station MAC or soft-AP MAC

```
#define STATION_IF      0x00
#define SOFTAP_IF       0x01
```

uint8 *macaddr : MAC address

Return:

true: Success

false: Failure

54. wifi_set_sleep_type

Function:

Sets sleep type for power saving. Set NONE_SLEEP_T to disable power saving.

Note: Default mode: Modem sleep.

Prototype:

```
bool wifi_set_sleep_type(enum sleep_type type)
```

Parameters:

enum sleep_type type : sleep type

Return:

true: Success

false: Failure

55. wifi_get_sleep_type

Function:

Gets sleep type.

Prototype:

```
enum sleep_type wifi_get_sleep_type(void)
```

Parameters:

none

Return:

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

56. 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:

none

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);
```

57. wifi_status_led_uninstall

Function: Uninstall WiFi status LED

Prototype:

void wifi_status_led_uninstall ()

Parameter:

none

Return:

none

58. wifi_set_broadcast_if

Function:

Set ESP8266 to send UDP broadcast from station interface or soft-AP interface, or both station and soft-AP interfaces. Default is soft-AP.

Note:

If broadcast is set to be in station interface only, the soft-AP DHCP server will be disabled.

Prototype:

bool wifi_set_broadcast_if (uint8 interface)

Parameter:

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

Return:

true: Success

false: Failure

59. wifi_get_broadcast _if

Function:

Get interface which ESP8266 sends 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:

none

Return:

1: station

2: soft-AP

3: both station and soft-AP

60. 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)
{
        os_printf("event %x\n", evt->event);
        switch (evt->event) {
        case EVENT_STAMODE_CONNECTED:
                os_printf("connect to ssid %s, channel %d\n",
                                        evt->event_info.connected.ssid,
                                        evt->event_info.connected.channel);
                break;
        case EVENT_STAMODE_DISCONNECTED:
                os_printf("disconnect from ssid %s, reason %d\n",
                                        evt->event_info.disconnected.ssid,
                                        evt->event_info.disconnected.reason);
                break;
        case EVENT_STAMODE_AUTHMODE_CHANGE:
          os_printf("mode: %d -> %d\n",
                                        evt->event_info.auth_change.old_mode,
                                        evt->event_info.auth_change.new_mode);
          break;
        case EVENT_STAMODE_GOT_IP:
                os_printf("ip:" IPSTR ",mask:" IPSTR ",gw:" IPSTR,
                                        IP2STR(&evt->event_info.got_ip.ip),
                                        IP2STR(&evt->event_info.got_ip.mask),
                                        IP2STR(&evt->event_info.got_ip.gw));
                os_printf("\n");
                break;
        case EVENT_SOFTAPMODE_STACONNECTED:
```

```
                os_printf("station: " MACSTR "join, AID = %d\n",
                                MAC2STR(evt->event_info.sta_connected.mac),
                                evt->event_info.sta_connected.aid);
                break;
                case EVENT_SOFTAPMODE_STADISCONNECTED:
                  os_printf("station: " MACSTR "leave, AID = %d\n",
                                MAC2STR(evt->event_info.sta_disconnected.mac),
                                evt->event_info.sta_disconnected.aid);
                break;
            default:
                    break;
            }
}
void user_init(void)
{
   // TODO: add your own code here....
   wifi_set_event_hander_cb(wifi_handle_event_cb);
}
```

61. wifi_wps_enable

Function:

Enable Wi-Fi WPS function

Note:

WPS can only be used when ESP8266 station is enabled.

Structure:

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

Prototype:

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

Parameter:

WPS_TYPE_t wps_type : WPS type, so far only WPS_TYPE_PBC is supported

Return:

true: Success

false: Failure

62. wifi_wps_disable

Function:

Disable Wi-Fi WPS function and release resources allocated to it.

Prototype:

bool wifi_wps_disable(void)

Parameter:

none

Return:

true: Success

false: Failure

63. wifi_wps_start

Function:

WPS starts to work

Note:

WPS can only be used when ESP8266 station is enabled.

Prototype:

bool wifi_wps_start(void)

Parameter:

none

Return:

true: means that WPS starts to work successfully, does not mean WPS succeeded.

false: Failure

64. wifi_set_wps_cb

Function:

Set WPS callback, parameter of the callback is the status of WPS.

Callback and parameter structure:

typedef void (*wps_st_cb_t)(int status);

enum wps_cb_status {

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

Note:

- If parameter status == WPS_CB_ST_SUCCESS in WPS callback, it means WPS got AP's information, user can call wifi_wps_disable to disable WPS and release resource, then call wifi_station_connect to connect to target AP.
- Otherwise, it means that WPS failed, user can create a timer to retry WPS by wifi_wps_start after a while, or call wifi_wps_disable to disable WPS and release resource.

Prototype:

bool wifi_set_wps_cb(wps_st_cb_t cb)

Parameter:

wps_st_cb_t cb : callback

Return:

true: Success

false: Failure

65. wifi_register_send_pkt_freedom_cb

Function:

Register a callback for sending user-defined 802.11 packets.

Note:

Only after the previous packet was sent, entered the freedom_outside_cb_t, the next packet is allowed to be sent.

Callback Definition:

typedef void (*freedom_outside_cb_t)(uint8 status);

parameter status : 0, packet sending succeeded; otherwise, failed.

The send callback can determine the status of a transmitted packet, however, please note the following points:

- For unicast packet:
  - If the status is OK in the freedom_outside_cb_t, but the target device failed to receive the packet, the reasons may be as follows:

    1) it may have been corrupted by other unexpected devices

2) incorrect key for communication

3) the application lost or missed the packet

Solution: handshake mechanism should be used in application to get a high success rate with packet transmission.

- If the status is FAIL in the freedom_outside_cb_t, but the target device does receive the packet, the reason may be as below:

1) busy channel, the sender fails to receive the ACK

Solution: the sender application should re-transmit the packet, and the receiver should detect the retransmitted packet.

- For multicast packet (include broadcast packet):
  - If the status is OK in the freedom_outside_cb_t, it indicates that the packet is sent successfully.
  - If the status is FAIL in the freedom_outside_cb_t, it indicates that sending failed.

Prototype:

int wifi_register_send_pkt_freedom_cb(freedom_outside_cb_t cb)

Parameter:
freedom_outside_cb_t cb : callback

Return:

0: Success

-1: Failure

66. wifi_unregister_send_pkt_freedom_cb

Function:

Unregister the callback for sending packets freedom.

Prototype:

void wifi_unregister_send_pkt_freedom_cb(void)

Parameter:
none

Return:

none

67. wifi_send_pkt_freedom

Function:

Send user-defined 802.11 packets.

Note:

- Packet has to be the whole 802.11 packet, excluding the FCS. The length of the packet has to be longer than the minimum length of the header of 802.11 packet which is 24 bytes, and less than 1400 bytes.
- Duration area is invalid for user, it will be filled in SDK.
- The rate of sending packet is same as the management packet which is the same as the system rate of sending packets.
- Can send: unencrypted data packet, unencrypted beacon/probe req/probe resp.
- Can NOT send: all encrypted packets (the encrypt bit in the packet has to be 0, otherwise it is not supported), control packet, other management packet except unencrypted beacon/probe req/probe resp.
- Only after the previous packet was sent, and the sent callback is entered, the next packet is allowed to send. Otherwise, wifi_send_pkt_freedom will return fail.

Prototype:

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

Parameter:

uint8 *buf : pointer of packet

int len : packet length

bool sys_seq : follow the system's 802.11 packets sequence number or not, if it is true, the sequence number will be increased 1 every time a packet sent.

Return:

0: Success

-1: Failure

68. wifi_rfid_locp_recv_open

Function:

Enable RFID LOCP (Location Control Protocol) to receive WDS packets.

Prototype:

int wifi_rfid_locp_recv_open(void)

Parameter:

none

Return:

0: Success

otherwise: Failure

69. wifi_rfid_locp_recv_close

Function:

Disable RFID LOCP (Location Control Protocol).

Prototype:

void wifi_rfid_locp_recv_close(void)

Parameter:
none

Return:

none

70. wifi_register_rfid_locp_recv_cb

Function:

Register a callback on receiving WDS packets. Only if the first MAC address of the WDS packet is a multicast address.

Callback Definition:

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

Parameter:

uint8 *frm : point to the head of 802.11 packet

int len : packet length

int rssi : signal strength

Prototype:

int wifi_register_rfid_locp_recv_cb(rfid_locp_cb_t cb)

Parameter:
rfid_locp_cb_t cb : callback

Return:

0: Success

otherwise: Failure

71. wifi_unregister_rfid_locp_recv_cb

Function:

Unregister the callback of receiving WDS packets.

Prototype:

void wifi_unregister_rfid_locp_recv_cb(void)

Parameter:
none

Return:

none

72. wifi_enable_gpio_wakeup

Function:

Set a GPIO to wake the ESP8266 up from light-sleep mode.

Note:

If the ESP8266 enters light-sleep automatically (wifi_set_sleep_type(LIGHT_SLEEP_T);), after being waken up by GPIO, when the chip attempts to sleep again, it will check the status of the GPIO:

- if the GPIO is still in the wakeup status, the EP8266 will enter modem-sleep mode instead
- if the GPIO is NOT in the wakeup status, the ESP8266 will enter light-sleep mode.

Prototype:

void wifi_enable_gpio_wakeup(uint32 i, GPIO_INT_TYPE intr_status)

Parameter:
uint32 i : GPIO number, range: [0, 15]

GPIO_INT_TYPE intr_status : status of GPIO interrupt to trigger the wakeup process

Return:

none

Example:

ESP8266 will be wakened from light-sleep, when the GPIO12 is in low-level.

```
GPIO_DIS_OUTPUT(12);

PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U, FUNC_GPIO12);

wifi_enable_gpio_wakeup(12, GPIO_PIN_INTR_LOLEVEL);
```

73. wifi_disable_gpio_wakeup

Function:

Disable the function that the GPIO can wake the ESP8266 up from light-sleep mode.

Prototype:

void wifi_disable_gpio_wakeup(void)

Parameter:
none

Return:

none

3.6. Rate Control APIs

Wi-Fi rate control APIs can be found in: **/ESP8266_NONOS_SDK/include/user_interface.h**.

1. wifi_set_user_fixed_rate

Function:

Set the fixed rate and mask of sending data from ESP8266.

Structure and Definition:

```
enum FIXED_RATE {
        PHY_RATE_48   =   0x8,
        PHY_RATE_24   =   0x9,
        PHY_RATE_12   =   0xA,
        PHY_RATE_6    =   0xB,
        PHY_RATE_54   =   0xC,
        PHY_RATE_36   =   0xD,
        PHY_RATE_18   =   0xE,
        PHY_RATE_9    =   0xF,
}

#define FIXED_RATE_MASK_NONE        (0x00)
#define FIXED_RATE_MASK_STA         (0x01)
#define FIXED_RATE_MASK_AP          (0x02)
#define FIXED_RATE_MASK_ALL         (0x03)
```

Note:

- Only if the corresponding bit in enable_mask is 1, ESP8266 station or soft-AP will send data in the fixed rate.
- If the enable_mask is 0, both ESP8266 station and soft-AP will not send data in the fixed rate.
- ESP8266 station and soft-AP share the same rate, they can not be set to different data rates.

Prototype:

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

Parameter:

uint8 enable_mask : 0x00 - disable the fixed rate
0x01 - use the fixed rate on ESP8266 station
0x02 - use the fixed rate on ESP8266 soft-AP
0x03 - use the fixed rate on ESP8266 station and soft-AP

uint8 rate : value of the fixed rate

Return:

0: Success

otherwise: Failure

2. wifi_get_user_fixed_rate

Function:

Get the fixed rate and mask of ESP8266.

Prototype:

int wifi_get_user_fixed_rate(uint8 *enable_mask, uint8 *rate)

Parameter:

uint8 *enable_mask : pointer of the enable_mask

uint8 *rate : pointer of the fixed rate

Return:

0: Success

otherwise: Failure

3. wifi_set_user_sup_rate

Function:

Set the rate range in the IE of support rate in ESP8266's beacon, probe req/resp and other packets. Tell other devices about the rate range supported by ESP8266 to limit the rate of sending packets from other devices.

Note:

This API can only support 802.11g now, but it will support 802.11b in next version.

Parameter Definition:

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

Prototype:

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

Parameter:

uint8 min : the minimum value of the support rate, according to enum support_rate.

uint8 max : the maximum value of the support rate, according to enum support_rate.

Return:

0: Success

otherwise: Failure

Example:

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

4. wifi_set_user_rate_limit

Function:

Limit the initial rate of sending data from ESP8266. The rate of retransmission is not limited by this API.

Parameter Definition:

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

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

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

Prototype:

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

Parameter:

uint8 mode : WiFi mode

```
#define RC_LIMIT_11B    0
#define RC_LIMIT_11G    1
#define RC_LIMIT_11N    2
```

uint8 ifidx : interface of ESP8266

0x00 - ESP8266 station

0x01 - ESP8266 soft-AP

uint8 max : the maximum value of the rate, according to the enum rate corresponding to the first parameter mode.

uint8 min : the minimum value of the rate, according to the enum rate corresponding to the first parameter mode.

Return:

true: Success

false: Failure

Example:

// Set the rate limitation of ESP8266 station in 11G mode, 6M ~ 18M.

wifi_set_user_rate_limit(RC_LIMIT_11G, 0, RATE_11G_G18M, RATE_11G_G6M);

5. wifi_set_user_limit_rate_mask

Function:

Set the interfaces of ESP8266 whose rate of sending packets is limited by wifi_set_user_rate_limit.

Definition:

```
#define LIMIT_RATE_MASK_NONE (0x00)
#define LIMIT_RATE_MASK_STA  (0x01)
#define LIMIT_RATE_MASK_AP   (0x02)
#define LIMIT_RATE_MASK_ALL  (0x03)
```

Prototype:

bool wifi_set_user_limit_rate_mask(uint8 enable_mask)

Parameter:
uint8 enable_mask :

- 0x00 - disable the limitation on both ESP8266 station and soft-AP
- 0x01 - enable the limitation on ESP8266 station
- 0x02 - enable the limitation on ESP8266 soft-AP
- 0x03 - enable the limitation on both ESP8266 station and soft-AP

Return:

true: Success

false: Failure

6. wifi_get_user_limit_rate_mask

Function:

Get the interfaces of ESP8266 whose rate of sending data is limited by wifi_set_user_rate_limit.

Prototype:

uint8 wifi_get_user_limit_rate_mask(void)

Parameter:
none

Return:

0x00 - both ESP8266 station and soft-AP are not limited

0x01 - ESP8266 station is limited

0x02 - ESP8266 soft-AP is limited

0x03 - both ESP8266 station and soft-AP are limited

3.7. Force Sleep APIs

Force Sleep APIs can be found in: **/ESP8266_NONOS_SDK/include/user_interface.h**.

wifi_set_opmode has to be set to NULL_MODE before entering forced sleep mode. Then users need to wake ESP8266 up from sleep, or wait till the sleep time out and enter the wakeup callback(register by wifi_fpm_set_wakeup_cb) . Disable the force sleep function by wifi_fpm_close before setting Wi-Fi mode back to normal mode.

Timer will prevent the chip from entering light-sleep mode, please disable all timers in application before entering light-sleep.

More details in "Example" below.

1. wifi_fpm_open

Function:

Enable force sleep function.

Prototype:

void wifi_fpm_open (void)

Parameter:

none

Default:

Force sleep function is disabled.

Return:

none

2. wifi_fpm_close

Function:

Disable force sleep function.

Prototype:

void wifi_fpm_close (void)

Parameter:

none

Return:

none

3. wifi_fpm_do_wakeup

Function:

Wake ESP8266 up from MODEM_SLEEP_T force sleep.

Note:

This API can only be called when MODEM_SLEEP_T force sleep function is enabled, after calling wifi_fpm_open. This API can not be called after calling wifi_fpm_close.

Prototype:

void wifi_fpm_do_wakeup (void)

Parameter:

none

Return:

none

4. wifi_fpm_set_wakeup_cb

Function:

Set a wake-up callback function to be called on wake-up from force sleep because of timeout.

Notice:

- This API can only be called when force sleep function is enabled, after calling wifi_fpm_open. This API can not be called after calling wifi_fpm_close.
- fpm_wakeup_cb_func will be called after system wakes up only if the force sleep time out (wifi_fpm_do_sleep and the parameter is not 0xFFFFFFF).
- fpm_wakeup_cb_func will not be called if wake-up is caused by wifi_fpm_do_wakeup from MODEM_SLEEP_T type force sleep.

Prototype:

void wifi_fpm_set_wakeup_cb(void (*fpm_wakeup_cb_func)(void))

Parameter:

void (*fpm_wakeup_cb_func)(void) : callback on wake-up

Return:

none

5. wifi_fpm_do_sleep

Function:

Force ESP8266 to enter sleep mode, and it will wake up automatically on time out.

Note:

- This API can only be called when force sleep function is enabled, after calling wifi_fpm_open. This API can not be called after calling wifi_fpm_close.
- If this API returned 0 means that the configuration is set successfully, but the ESP8266 will not enter sleep mode immediately, it is going to sleep in the system idle task. Please do not call other WiFi related function right after calling this API.

Prototype:

int8 wifi_fpm_do_sleep (uint32 sleep_time_in_us)

Parameter:

uint32 sleep_time_in_us : sleep time, ESP8266 will wake up automatically on time out. Unit: us.

Range: 10000 ~ 268435455(0xFFFFFFF)

If sleep_time_in_us is 0xFFFFFFF, the ESP8266 will sleep till be woke up as below:

- if wifi_fpm_set_sleep_type is set to be LIGHT_SLEEP_T, ESP8266 can wake up by GPIO.
- if wifi_fpm_set_sleep_type is set to be MODEM_SLEEP_T, ESP8266 can wake up by wifi_fpm_do_wakeup.

Return:

0, setting successful

-1, failed to sleep, sleep status error;

-2, failed to sleep, force sleep function is not enabled.

6. wifi_fpm_set_sleep_type

Function:

Set sleep type for force sleep function.

Note:

This API can only be called before wifi_fpm_open.

Prototype:

void wifi_fpm_set_sleep_type (enum sleep_type type)

Parameter:

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

Return:

none

7. wifi_fpm_get_sleep_type

Function:

Get sleep type of force sleep function.

Prototype:

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

Parameter:

none

Return:

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

8. wifi_fpm_auto_sleep_set_in_null_mode

Function:

Set whether enter modem sleep mode automatically or not after disabled Wi-Fi mode (wifi_set_opmode(NULL_MODE)).

Prototype:

```
void wifi_fpm_auto_sleep_set_in_null_mode (uint8 req)
```

Parameter:

uint8 req :

0, disable auto-sleep function;

1, enable auto modem sleep when Wi-Fi mode is NULL_MODE.

Return:

none

9. Example

For example, forced sleep interface can be called, the RF circuit can be closed mandatorily so as to lower the power.

> Note:
> When forced sleep interface is called, the chip will not enter sleep mode instantly, it will enter sleep mode when the system is executing idle task. Please refer to the below sample code.

Example one: Modem-sleep mode (disable RF)

```
#define FPM_SLEEP_MAX_TIME    0xFFFFFFF

void fpm_wakup_cb_func1(void)
{
  wifi_fpm_close();                          // disable force sleep function
  wifi_set_opmode(STATION_MODE);             // set station mode
  wifi_station_connect();                    // connect to AP
}
void user_func(...)
{
  wifi_station_disconnect();
  wifi_set_opmode(NULL_MODE);                // set WiFi mode to null mode.
  wifi_fpm_set_sleep_type(MODEM_SLEEP_T); // modem sleep
  wifi_fpm_open();                           // enable force sleep

#ifdef SLEEP_MAX
 /* For modem sleep, FPM_SLEEP_MAX_TIME can only be wakened by calling
wifi_fpm_do_wakeup. */
  wifi_fpm_do_sleep(FPM_SLEEP_MAX_TIME);
#else
  // wakeup automatically when timeout.
  wifi_fpm_set_wakeup_cb(fpm_wakup_cb_func1);  // Set wakeup callback
  wifi_fpm_do_sleep(50*1000);
#endif
}
```

```
#ifdef SLEEP_MAX
void func1(void)
{
    wifi_fpm_do_wakeup();
    wifi_fpm_close();                       // disable force sleep function
    wifi_set_opmode(STATION_MODE);          // set station mode
    wifi_station_connect();                 // connect to AP
}
#endif
```

Example two: Light-sleep mode (disable RF and CPU)

```
#define FPM_SLEEP_MAX_TIME    0xFFFFFFF

void fpm_wakup_cb_func1(void)
{
    wifi_fpm_close();                       // disable force sleep function
    wifi_set_opmode(STATION_MODE);          // set station mode
    wifi_station_connect();                 // connect to AP
}

#ifndef SLEEP_MAX
// Wakeup till time out.
void user_func(...)
{
    wifi_station_disconnect();
    wifi_set_opmode(NULL_MODE);             // set WiFi mode to null mode.
    wifi_fpm_set_sleep_type(LIGHT_SLEEP_T);         // light sleep
    wifi_fpm_open();                        // enable force sleep
    wifi_fpm_set_wakeup_cb(fpm_wakup_cb_func1); // Set wakeup callback
    wifi_fpm_do_sleep(50*1000);
}

#else

// Or wake up by GPIO
void user_func(...)
{
    wifi_station_disconnect();
    wifi_set_opmode(NULL_MODE);                     // set WiFi mode to null mode.
    wifi_fpm_set_sleep_type(LIGHT_SLEEP_T);         // light sleep
    wifi_fpm_open();                        // enable force sleep
```

```
    PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U, FUNC_GPIO12);

    wifi_enable_gpio_wakeup(12, GPIO_PIN_INTR_LOLEVEL);


    wifi_fpm_set_wakeup_cb(fpm_wakup_cb_func1);   // Set wakeup callback
    wifi_fpm_do_sleep(FPM_SLEEP_MAX_TIME);
    ...
}
#endif
```

3.8. ESP-NOW APIs

ESP-NOW APIs can be found in: **/ESP8266_NONOS_SDK/include/espnow.h**.

More details about ESP-NOW are in "30C_ESP-NOW__User_Guide". Please note the following points carefully:

- ESP-NOW do not support broadcast and multicast.
- It is suggested that slave and combo roles corresponding to soft-AP or soft-AP+station mode, controller role corresponding to station mode.
- When ESP8266 is in soft-AP+station mode, it will communicate through station interface if it is in slave or combo role, and communicate through soft-AP interface if it is in controller role.
- ESP-NOW can not wake ESP8266 up from sleep, so if the target ESP8266 station is in sleep, ESP-NOW communication will fail.
- In station mode, ESP8266 supports 10 encrypt ESP-NOW peers at most, with the unencrypted peers, it can be 20 peers in total at most.
- In the soft-AP mode or soft-AP + station mode, the ESP8266 supports 6 encrypt ESP-NOW peers at most, with the unencrypted peers, it can be 20 peers in total at most.

1. roles of ESP-NOW

```
enum esp_now_role {

        ESP_NOW_ROLE_IDLE = 0,

        ESP_NOW_ROLE_CONTROLLER,

        ESP_NOW_ROLE_SLAVE,

        ESP_NOW_ROLE_COMBO,          // both slave and controller

        ESP_NOW_ROLE_MAX,

};
```

2. esp_now_init

Function:

ESP-NOW initialization

Prototype:

init esp_now_init(void)

Parameter:

none

Return:

0: Success

otherwise: Failure

3. esp_now_deinit

Function:

Deinitialize ESP-NOW

Prototype:

int esp_now_deinit(void)

Parameter:

none

Return:

0: Success

otherwise: Failure

4. esp_now_register_recv_cb

Function:

Register ESP-NOW receive callback

Note:

When received an ESP-NOW packet, enter receive callback:

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

Parameters of ESP-NOW receive callback:

u8 *mac_addr	: MAC address of the sender
u8 *data	: data received
u8 len	: data length

Prototype:

int esp_now_register_recv_cb(esp_now_recv_cb_t cb)

Parameter:

esp_now_recv_cb_t cb : receive callback

Return:

0: Success

otherwise: Failure

5. esp_now_unregister_recv_cb

Function:

Unregister ESP-NOW receive callback

Prototype:

int esp_now_unregister_recv_cb(void)

Parameter:

none

Return:

0: Success

otherwise: Failure

6. esp_now_register_send_cb

Function:

Register ESP-NOW send callback

Notice:

ESP-NOW send callback:

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

Parameter:

u8 *mac_addr : MAC address of target device

u8 status : status of ESP-NOW sending packet

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

The status will be updated to MT_TX_STATUS_OK, if ESP-NOW send the packet successfully.

Users must explicitly make sure that the key for communication is correct.

The send callback can determine the status of a transmitted packet, however, please note the following points:

- For unicast packet:
  - If the status is OK in the esp_now_send_cb_t, but the target device failed to receive the packet, the reasons may be as follows:

    1) it may have been corrupted by other unexpected devices

    2) incorrect key for communication

    3) the application lost or missed the packet

    Solution: handshake mechanism should be used in application to get a high success rate with packet transmission.

  - If the status is FAIL in the esp_now_send_cb_t, but the target device does receive the packet, the reason may be as below:

    1) busy channel, the sender fails to receive the ACK

    Solution: the sender application should re-transmit the packet, and the receiver should detect the retransmitted packet.

- For multicast packet (include broadcast packet):
  - If the status is OK in the esp_now_send_cb_t, it indicates that the packet is sent successfully.
  - If the status is FAIL in the esp_now_send_cb_t, it indicates that sending failed.

Prototype:

u8 esp_now_register_send_cb(esp_now_send_cb_t cb)

Parameter:

esp_now_send_cb_t cb : callback

Return:

0: Success

otherwise: Failure

7. esp_now_unregister_send_cb

Function:

Unregister ESP-NOW send callback

Prototype:

int esp_now_unregister_send_cb(void)

Parameter:

none

Return:

0: Success

otherwise: Failure

8. esp_now_send

Function:

Send ESP-NOW packet

Prototype:

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

Parameter:

u8 *da : Destination MAC address. If it's NULL, the packet is sent to all MAC addresses recorded by ESP-NOW; otherwise, packet is sent to the target MAC address only.

u8 *data : data need to send

u8 len : data length

Return:

0: Success

otherwise: Failure

9. esp_now_add_peer

Function:

Add an ESP-NOW peer, store MAC address of target device into ESP-NOW MAC list.

Prototype:

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

Parameter:

u8 *mac_addr : MAC address of device

u8 role : role type of device, refer to esp_now_role

u8 channel : channel of device

u8 *key : 16 bytes key which is needed for ESP-NOW communication

u8 key_len : length of key, has to be 16 bytes now

Return:

0: Success

otherwise: Failure

10. esp_now_del_peer

Function:

Delete an ESP-NOW peer, delete MAC address of the device from ESP-NOW MAC list.

Prototype:

int esp_now_del_peer(u8 *mac_addr)

Parameter:

u8 *mac_addr : MAC address of device

Return:

0: Success

otherwise: Failure

11. esp_now_set_self_role

Function:

Set ESP-NOW role of device itself

Prototype:

int esp_now_set_self_role(u8 role)

Parameter:

u8 role : role type, refer to esp_now_role

Return:

0: Success

otherwise: Failure

12. esp_now_get_self_role

Function:

Get ESP-NOW role of device itself

Prototype:

u8 esp_now_get_self_role(void)

Parameter:

none

Return:

role type, refer to esp_now_role

13. esp_now_set_peer_role

Function:

Set ESP-NOW role for a target device. If it is set multiple times, the new role setting will override the old role.

Prototype:

int esp_now_set_peer_role(u8 *mac_addr,u8 role)

Parameter:

u8 *mac_addr : MAC address of target device

u8 role : role type, refer to esp_now_role

Return:

0: Success

otherwise: Failure

14. esp_now_get_peer_role

Function:

Get ESP-NOW role of a target device

Prototype:

int esp_now_get_peer_role(u8 *mac_addr)

Parameter:

u8 *mac_addr : MAC address of target device

Return:

esp_now_role: the role type

otherwise: Failure

15. esp_now_set_peer_key

Function:

Set ESP-NOW key for a target device. If it is set multiple times, the latest setting will be valid.

Prototype:

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

Parameter:

u8 *mac_addr : MAC address of target device

u8 *key : 16 bytes key which is needed for ESP-NOW communication,

if it is NULL, current key will be reset to be none.

u8 key_len : key length, has to be 16 bytes now

Return:

0: Success

otherwise: Failure

16. esp_now_get_peer_key

Function:

Get ESP-NOW key of a target device.

Prototype:

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

Parameter:

u8 *mac_addr : MAC address of target device

u8 *key : pointer to the key, buffer size has to be 16 bytes at least

u8 *key_len : key length

Return:

0: Success

> 0: Found target device but cannot get key

< 0: Failure

17. esp_now_set_peer_channel

Function:

Record channel information of a ESP-NOW device.

To communicate with a device,

- call esp_now_get_peer_channel to get its channel first,
- then call wifi_set_channel to be on the same channel and continue further communication.

Prototype:

int esp_now_set_peer_channel(u8 *mac_addr,u8 channel)

Parameter:

u8 *mac_addr : MAC address of target device

u8 channel : channel, usually to be 1 ~ 13, some area may use channel 14

Return:

0: Success

otherwise: Failure

18. esp_now_get_peer_channel

Function:

Get channel information of a ESP-NOW device. ESP-NOW communication needs to be at the same channel.

Prototype:

int esp_now_get_peer_channel(u8 *mac_addr)

Parameter:

u8 *mac_addr : MAC address of target device

Return:

1 ~ 13 (some area may get 14): Success

otherwise: Failure

19. esp_now_is_peer_exist

Function:

Check if target device exists or not.

Prototype:

int esp_now_is_peer_exist(u8 *mac_addr)

Parameter:

u8 *mac_addr : MAC address of target device

Return:

0: Device does not exist

< 0: Error, check for device failed

> 0: Device exists

20. esp_now_fetch_peer

Function:

Get MAC address of ESP-NOW device which is pointed now, and move the pointer to next one in ESP-NOW MAC list or move the pointer to the first one in ESP-NOW MAC list

Note:

- This API can not re-entry
- Parameter has to be true when you call it the first time.

Prototype:

u8 *esp_now_fetch_peer(bool restart)

Parameter:

bool restart : true, move pointer to the first one in ESP-NOW MAC list

false, move pointer to the next one in ESP-NOW MAC list

Return:

NULL, no ESP-NOW devices exist

Otherwise, MAC address of ESP-NOW device which is pointed now

21. esp_now_get_cnt_info

Function:

Get the total number of ESP-NOW devices which are associated, and the number count of encrypted devices.

Prototype:

int esp_now_get_cnt_info(u8 *all_cnt, u8 *encryp_cnt)

Parameter:

u8 *all_cnt : total number of ESP-NOW devices which are associated

u8 *encryp_cnt : number count of encrypted devices

Return:

0: Success

otherwise: Failure

22. esp_now_set_kok

Function:

Set the encryption key for the communication key. All ESP-NOW devices share the same encrypt key. If users do not set the encrypt key, ESP-NOW communication key will be encrypted by a default key.

If this API needs to be called, please call it before esp_now_add_peer and esp_now_set_peer_key.

Prototype:

int esp_now_set_kok(u8 *key, u8 len)

Parameter:

u8 *key : pointer of encryption key

u8 len : key length, has to be 16 bytes now

Return:

0: Success

otherwise: Failure

3.9. Simple Pair APIs

Simple Pair APIs can be found in: **/ESP8266_NONOS_SDK/include/simple_pair.h.**

1. status of simple pair

```
typedef enum {
    SP_ST_STA_FINISH = 0,    // station finished negotiation
    SP_ST_AP_FINISH = 0,     // AP finished negotiation
    SP_ST_AP_RECV_NEG,    // AP received a request of negotiation from station
    SP_ST_STA_AP_REFUSE_NEG,     // station received the refusal to negotiate from AP
    /* definitions below are error codes */
    SP_ST_WAIT_TIMEOUT,    // Error: time out
    SP_ST_SEND_ERROR,     // Error: error occur when sending data
    SP_ST_KEY_INSTALL_ERR,          // Error: error occur during key installation
    SP_ST_KEY_OVERLAP_ERR,          // Error: one MAC address uses multiple keys
    SP_ST_OP_ERROR,                 // Error: operational error
    SP_ST_UNKNOWN_ERROR,            // Error: unknown error
    SP_ST_MAX,
} SP_ST_t;
```

2. register_simple_pair_status_cb

Function:

Register a callback of status for status of simple pair.

Prototype:

init register_simple_pair_status_cb(simple_pair_status_cb_t cb)

Callback Definition:

typedef void (*simple_pair_status_cb_t)(u8 *sa, u8 status);

Parameter:

u8 *sa : the MAC address of the remote device

u8 status : status of simple pair, refer to SP_ST_t

Parameter:

simple_pair_status_cb_t cb : callback

Return:

0: Success

otherwise: Failure

3. unregister_simple_pair_status_cb

Function:

Unregister the callback of status of simple pair.

Prototype:

void unregister_simple_pair_status_cb(void)

Parameter:

none

Return:

none

4. simple_pair_init

Function:

Simple pair initialization.

Prototype:

int simple_pair_init(void)

Parameter:

none

Return:

0: Success

otherwise: Failure

5. simple_pair_deinit

Function:

Deinitialize simple pair.

Prototype:

int simple_pair_deinit(void)

Parameter:

none

Return:

none

6. simple_pair_state_reset

Function:

Reset the state of simple pair. When simple pair needs to be restarted again, this API can be called to reset the state.

Prototype:

int simple_pair_state_reset(void)

Parameter:

none

Return:

0: Success

otherwise: Failure

7. simple_pair_ap_enter_announce_mode

Function:

The AP peer of simple pair enters announce mode.

Prototype:

int simple_pair_ap_enter_announce_mode(void)

Parameter:

none

Return:

0: Success

otherwise: Failure

8. simple_pair_sta_enter_scan_mode

Function:

The station peer of simple pair enters scan mode.

Prototype:

int simple_pair_sta_enter_scan_mode(void)

Parameter:

none

Return:

0: Success

otherwise: Failure

9. simple_pair_sta_start_negotiate

Function:

The station peer of simple pair starts to negotiate.

Prototype:

int simple_pair_sta_start_negotiate(void)

Parameter:

none

Return:

0: Success

otherwise: Failure

10. simple_pair_ap_start_negotiate

Function:

The AP peer of simple pair agrees to negotiate.

Prototype:

int simple_pair_ap_start_negotiate(void)

Parameter:

none

Return:

0: Success

otherwise: Failure

11. simple_pair_ap_refuse_negotiate

Function:

The AP peer of simple pair refuses to negotiate.

Prototype:

int simple_pair_ap_refuse_negotiate(void)

Parameter:

none

Return:

0: Success

otherwise: Failure

12. simple_pair_set_peer_ref

Function:

Set configuration of the peer which needs to negotiate. Note that this only sets the configuration of the peer, and does not install keys or perform any other relevant operations.

If the device runs as the station peer, this API needs to be called before simple_pair_sta_start_negotiate.

If the device runs as the AP peer, this API needs to be called before simple_pair_ap_start_negotiate or simple_pair_ap_refuse_negotiate.

Prototype:

int simple_pair_set_peer_ref(u8 *peer_mac, u8 *tmp_key, u8 *ex_key)

Parameter:

u8 *peer_mac : MAC address of the target peer of negotiation, length: 6 bytes, can NOT be null.

u8 *tmp_key : a temporary key to encrypt the negotiation, length: 16 bytes, can NOT be null.

u8 *ex_key : a key for exchange, length: 16 bytes. If it is null, the 0x0000...0000 will be used as the ex_key by default.

Return:

0: Success

otherwise: Failure

13. simple_pair_get_peer_ref

Function:

Get the configuration of the negotiation. If passing a null pointer, the corresponding parameter will not be got.

Prototype:

int simple_pair_get_peer_ref(u8 *peer_mac, u8 *tmp_key, u8 *ex_key)

Parameter:

u8 *peer_mac : MAC address of the target peer of negotiation, length: 6 bytes.

u8 *tmp_key : the temporary key to encrypt the negotiation, length: 16 bytes.

u8 *ex_key : the key for exchange, length: 16 bytes. The 0x0000...0000 is used as the ex_key by default.

Return:

0: Success

otherwise: Failure

3.10. Upgrade (FOTA) APIs

FOTA APIs can be found in: **/ESP8266_NONOS_SDK/include/user_interface.h & upgrade.h**.

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:

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:

none

3. system_upgrade_flag_check

Function:

Gets upgrade status flag.

Prototype:

uint8 system_upgrade_flag_check()

Parameter:

none

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 cannot be started.

5. system_upgrade_reboot

Function: reboot system and use new version

Prototype:

void system_upgrade_reboot (void)

Parameters:

none

Return:

none

3.11. Sniffer Related APIs

Sniffer APIs can be found in: **/ESP8266_NONOS_SDK/include/user_interface.h**.

1. wifi_promiscuous_enable

Function:

Enable promiscuous mode for sniffer

Note:

- promiscuous mode can only be enabled in station mode.
- During promiscuous mode(sniffer), ESP8266 station and soft-AP are disabled.
- Before enable promiscuous mode, please call wifi_station_disconnect first
- 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:

none

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:

none

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:

none

4. wifi_get_channel

Function:

Get Wi-Fi channel

Prototype:

uint8 wifi_get_channel(void)

Parameters:

none

Return:

Channel number

5. wifi_set_channel

Function:

Set Wi-Fi channel, for sniffer mode

Prototype:

bool wifi_set_channel (uint8 channel)

Parameters:

uint8 channel : channel number

Return:

true: Success

false: Failure

3.12. Smart Config APIs

Smart-Config APIs can be found in: **/ESP8266_NONOS_SDK/include/smartconfig.h**.

AirKiss APIs can be found in: **/ESP8266_NONOS_SDK/include/airkiss.h**.

Please make sure the target AP is enabled before enable Smart-Config.

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:

- This API can only be called in station mode.
- During smart-config,ESP8266 station and soft-AP are disabled.
- Can not call smartconfig_start twice before it finish, please call smartconfig_stop first.
- Don't call any other APIs during smart-config, please call smartconfig_stop first.

Structure:

```
typedef enum  {
        SC_STATUS_WAIT = 0,       // Please don't start connection in this phase
        SC_STATUS_FIND_CHANNEL,   // Start connection by APP in this phase
        SC_STATUS_GETTING_SSID_PSWD,
        SC_STATUS_LINK,
        SC_STATUS_LINK_OVER,      // Got IP, connect to AP successfully
} sc_status;
typedef enum  {
        SC_TYPE_ESPTOUCH = 0,
        SC_TYPE_AIRKISS,
} sc_type;
```

Prototype:

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

Parameter:

sc_callback_t cb : smart config callback; executed when smart-config status changed;

parameter status of this callback shows the status of smart-config:

- if status == SC_STATUS_GETTING_SSID_PSWD, parameter void *pdata is a pointer of sc_type, means smart-config type: AirKiss or ESP-TOUCH.
- if status == SC_STATUS_LINK, parameter void *pdata is a pointer of struct station_config;
- if status == SC_STATUS_LINK_OVER, parameter void *pdata is a pointer of mobile phone's IP address, 4 bytes. This is only available in ESPTOUCH, otherwise, it is NULL.
- otherwise, parameter void *pdata is NULL.

uint8 log : 1: UART outputs logs; otherwise: UART only outputs the result. It is suggest that this log is only used for debugging. Users should not set it to be 1 while SmartConfig is working properly.

Return:

true: Success

false: Failure

Example:

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

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

2. smartconfig_stop

Function:

Stop smart config, free the buffer taken by smartconfig_start.

Note:

Irrespective of whether connection to AP succeeded or not, this API should be called to free memory taken by smartconfig_start.

Prototype:

bool smartconfig_stop(void)

Parameter:

none

Return:

true: Success

false: Failure

3. smartconfig_set_type

Function:

Set the protocol type of SmartConfig

Note:

This API can only be called before calling smartconfig_start.

Prototype:

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

Parameter:

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

Return:

true: Success;

false: Failure

4. airkiss_version

Function:

Get version information of the AirKiss lib.

Notice:

The length of the version information is unknown.

Prototype:

```
const char* airkiss_version(void)
```

Parameter:

none

Return:

Version information of the AirKiss lib.

5. airkiss_lan_recv

Function:

For the function that AirKiss can detect the ESP8266 devices in LAN, more details of this function refer to WeChat: http://iot.weixin.qq.com.

Workflow : Create a UDP transmission. When UDP data is received in espconn_recv_callback, call API airkiss_lan_recv and input the UDP data, if airkiss_lan_recv returns AIRKISS_LAN_SSDP_REQ, airkiss_lan_pack can be called to make a response packet.

This API is to parse the UDP packet sent by WeChat.

Prototype:

```
int airkiss_lan_recv(
const void* body,
unsigned short length,
const airkiss_config_t* config)
```

Parameter:

const void* body : the received UDP packet

unsigned short length : the length of UDP packet

airkiss_config_t* config : AirKiss structure

Return:

Refer to airkiss_lan_ret_t

>= 0: Success,

< 0: Failure

6. airkiss_lan_pack

Function:

User packet assembly for the function that AirKiss can detect the ESP8266 devices in LAN.

Prototype:

```
int airkiss_lan_pack(
        airkiss_lan_cmdid_t ak_lan_cmdid,
        void* appid,
        void* deviceid,
        void* _datain,
        unsigned short inlength,
        void* _dataout,
        unsigned short* outlength,
        const airkiss_config_t* config)
```

Parameter:

airkiss_lan_cmdid_t ak_lan_cmdid : packet type

void* appid : WeChat public number, got from WeChat

void* deviceid : device ID, got from WeChat

void* _datain : user data waiting for packet assembly

unsigned short inlength : length of the user data

void* _dataout : the packet got by _datain packet assembly

unsigned short* outlength : length of the packet

const airkiss_config_t* config : AirKiss structure

Return:

Refer to airkiss_lan_ret_t

>= 0: Success,

< 0: Failure

3.13. SNTP APIs

SNTP APIs can be found in: **/ESP8266_NONOS_SDK/include/sntp.h**.

1. sntp_setserver

Function:

Set SNTP server by IP address, supports 3 SNTP server at most

Prototype:

void sntp_setserver(unsigned char idx, ip_addr_t *addr)

Parameter:

unsigned char idx ： SNTP server index, support 3 SNTP server at most (0 ~ 2); index 0 is the main server,index 1 and 2 are as backup.

ip_addr_t *addr ： IP address; users need to ensure that it is an SNTP server

Return:

none

2. sntp_getserver

Function:

Get IP address of SNTP server as set by sntp_setserver

Prototype:

ip_addr_t sntp_getserver(unsigned char idx)

Parameter:

unsigned char idx : SNTP server index, supports 3 SNTP servers at most (0 ~ 2)

Return:

IP address

3. sntp_setservername

Function:

Set SNTP server by domain name, support 3 SNTP server at most

Prototype:

void sntp_setservername(unsigned char idx, char *server)

Parameter:

unsigned char idx ： SNTP server index, supports 3 SNTP servers at most (0 ~ 2); index 0 is the main server, index 1 and 2 are as backup.

char *server ： domain name; users need to ensure that it is an SNTP server

Return:

none

4. sntp_getservername

Function:

Get domain name of SNTP server which set by sntp_setservername

Prototype:

char * sntp_getservername(unsigned char idx)

Parameter:

unsigned char idx : SNTP server index, supports 3 SNTP servers at most (0 ~ 2)

Return:

domain name

5. sntp_init

Function:

SNTP initialize

Prototype:

void sntp_init(void)

Parameter:

none

Return:

none

6. sntp_stop

Function:

Stop SNTP

Prototype:

void sntp_stop(void)

Parameter:

none

Return:

none

7. sntp_get_current_timestamp

Function:

Get current timestamp from basic time (1970.01.01 00:00:00 GMT + 8), uint: second

Prototype:

uint32 sntp_get_current_timestamp()

Parameter:

none

Return:

time stamp

8. sntp_get_real_time

Function:

Get real time(GMT + 8)

Prototype:

char* sntp_get_real_time(long t)

Parameter:

long t : time stamp

Return:

real time

9. sntp_set_timezone

Function:

Set time zone

Prototype:

bool sntp_set_timezone (sint8 timezone)

Note:

Before calling sntp_set_timezone, please call sntp_stop first

Parameter:

sint8 timezone – time zone,range:-11 ~ 13

Return:

true: Success

false: Failure

Example:

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

10. sntp_get_timezone

Function:

Get time zone

Prototype:

sint8 sntp_get_timezone (void)

Parameter:

none

Return:

time zone, range: -11 ~ 13

11. SNTP Example

Step 1. enable SNTP

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

Step 2. set a timer to check SNTP timestamp

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

Step 3. timer callback

```
void ICACHE_FLASH_ATTR user_check_sntp_stamp(void *arg){
    uint32 current_stamp;
    current_stamp = sntp_get_current_timestamp();
    if(current_stamp == 0){
        os_timer_arm(&sntp_timer, 100, 0);
    } else{
        os_timer_disarm(&sntp_timer);
        os_printf("sntp: %d, %s \n",current_stamp, sntp_get_real_time(current_stamp));
    }
}
```

3.14. WPA2_Enterprise APIs

ESP8266 station can connect to WPA2_Enterprise APs.

WPA2_Enterprise APIs can be found in: **/ESP8266_NONOS_SDK/include/wpa2_enterprise.h.**

1. wifi_station_set_wpa2_enterprise_auth

Function:

Set authentication of WPA2_Enterprise.

To connect to WPA2_Enterprise AP, wifi_station_set_wpa2_enterprise_auth(1); should be called first. For connecting to a regular AP at a later stage, wifi_station_set_wpa2_enterprise_auth(0); should be called to clear the WPA2_Enterprise status.

Prototype:

int wifi_station_set_wpa2_enterprise_auth(int enable)

Parameter:

int enable : 0, disable authentication of WPA2_Enterprise, clear the status

non-0, enable authentication of WPA2_Enterprise

Return:

0 : Success

non-0 : Failure

2. wifi_station_set_enterprise_cert_key

Function:

Set user certificate and private key for connecting to WPA2_Enterprise AP. It is used for EAP-TLS authentication.

Note:

- Connecting to WPA2-ENTERPRISE AP needs more than 26 KB memory, please ensure enough space (system_get_free_heap_size).
- So far, WPA2-ENTERPRISE can only support unencrypted certificate and private key, and only in PEM format.
  - Header of certificate: - - - - - BEGIN CERTIFICATE - - - - -
  - Header of private key: - - - - - BEGIN RSA PRIVATE KEY - - - - - or - - - - - BEGIN PRIVATE KEY - - - - -
- Please call this API to set certificate and private key before connecting to WPA2_ Enterprise AP and the application needs to hold the certificate and private key. Call wifi_station_clear_enterprise_cert_key to release resources and clear status after being connected to the target AP, and then the application can release the certificate and private key.
- If the private key is encrypted, please use openssl pkey command to change it to unencrypted file to use, or use openssl rsa related commands to change it (or change the start TAG).

Prototype:

```
int wifi_station_set_enterprise_cert_key (
        u8 *client_cert, int client_cert_len,
        u8 *private_key, int private_key_len,
        u8 *private_key_passwd, int private_key_passwd_len,)
```

Parameter:

u8 *client_cert : user certificate, HEX array

int client_cert_len : length of certificate

u8 *private_key : private key, HEX array, can NOT be longer than 2048 bits

int private_key_len : length of private key, less than 2048

u8 *private_key_passwd : password for private key, to be supported, can only be NULL now.

int private_key_passwd_len : length of password, to be supported, can only be 0 now.

Return:

0 : Success

non-0 : Failure

Example:

For example, the private key is - - - - - BEGIN PRIVATE KEY - - - - - … … … …

Then then array should be uint8 key[]={0x2d, 0x2d, 0x2d, 0x2d, 0x2d, 0x42, 0x45, 0x47, … … 0x00 };

It is the ASCII code for the characters, and the array needs to terminate with 0x00.

3. wifi_station_clear_enterprise_cert_key

Function:

Release user certificate and private key resources and clear related status after being connected to the WPA2_Enterprise AP.

Prototype:

void wifi_station_clear_enterprise_cert_key (void)

Parameter:

none

Return:

none

4. wifi_station_set_enterprise_ca_cert

Function:

Set root certificate for connecting to WPA2_Enterprise AP. It is an option in EAP-TTLS/PEAP authentication.

Prototype:

int wifi_station_set_enterprise_ca_cert(u8 *ca_cert, int ca_cert_len)

Parameter:

u8 *ca_cert : root certificate, HEX array

int ca_cert_len : length of root certificate

Return:

0 : Success

non-0 : Failure

5. wifi_station_clear_enterprise_ca_cert

Function:

Release root certificate resources and clear related status after being connected to the WPA2_Enterprise AP.

Prototype:

void wifi_station_clear_enterprise_ca_cert (void)

Parameter:

none

Return:

none

6. wifi_station_set_enterprise_username

Function:

Set ESP8266 station's user name for connecting to WPA2_Enterprise AP.

Note:

- For EAP-TTLS and EAP-PEAP authentication, the user name has to be set. It is used in phase 2 of the authentication, only the user name that the server supported can pass the authentication.
- For EAP-TTLS and EAP-PEAP authentication, the user name is only an option. Without setting user name, the authentication can still be done anonymously.

Prototype:

int wifi_station_set_enterprise_username (u8 *username, int len)

Parameter:

u8 *username : the user name

int len : length of user name

Return:

0 : Success

non-0 : Failure

7. wifi_station_clear_enterprise_username

Function:

Release the user name resources and clear related status after being connected to the WPA2_Enterprise AP.

Prototype:

void wifi_station_clear_enterprise_username (void)

Parameter:

none

Return:

none

8. wifi_station_set_enterprise_password

Function:

Set the password for connecting to WPA2_Enterprise AP. It is used for EAP-TTLS / EAP-PEAP authentication.

Prototype:

int wifi_station_set_enterprise_password (u8 *password, int len)

Parameter:

u8 *password : the user password

int len : length of the password

Return:

0 : Success

non-0 : Failure

9. wifi_station_clear_enterprise_password

Function:

Clear the password resources and clear related status data after being connected to the WPA2_Enterprise AP.

Prototype:

void wifi_station_clear_enterprise_password (void)

Parameter:

none

Return:

none

10. wifi_station_set_enterprise_new_password

Function:

Set the new password for connecting to WPA2_Enterprise AP. It is used for MSCHAPV2.

Prototype:

int wifi_station_set_enterprise_new_password (u8 *new_password, int len)

Parameter:

u8 *new_password : the new password

int len : length of the password

Return:

0 : Success

non-0 : Failure

11. wifi_station_clear_enterprise_new_password

Function:

Release the new password resources and clear related status after being connected to the WPA2_Enterprise AP.

Prototype:

void wifi_station_clear_enterprise_new_password (void)

Parameter:

none

Return:

none

12. wifi_station_set_enterprise_disable_time_check

Function:

Determines whether expiration time is checked in authentication. The expiration time will not be checked by default.

Prototype:

void wifi_station_set_enterprise_disable_time_check (bool disable)

Parameter:

bool disable :

true, will NOT check the expiration time;

false, check the expiration time, wpa2_enterprise_set_user_get_time has to be called.

Return:

none

13. wifi_station_get_enterprise_disable_time_check

Function:

Check whether the expiration time will be observed in authentication.

Prototype:

bool wifi_station_get_enterprise_disable_time_check (void)

Parameter:

none

Return:

True : will NOT check the expiration time

False : check the expiration time

14. wpa2_enterprise_set_user_get_time

Function:

Set time callback to get current time from user.

wifi_station_set_enterprise_disable_time_check(false); should be called as the example below.

Prototype:

void wpa2_enterprise_set_user_get_time(get_time_func_t cb)

Parameter:

get_time_func_t cb : callback

Return:

none

Example:

```
static int sys_get_current_time(struct os_time *t) {
        t->sec = CURRENT_TIME;    // User set current time.
         return 0;
}

//Set Callback
wpa2_enterprise_set_user_get_time(sys_get_current_time);

//Enable Time check
wifi_station_set_enterprise_disable_time_check(false);
```

15. WPA2_Enterprise work flow

Here is the work flow that prepares ESP266 station to connect to WPA2_Enterprise AP.

1. Call wifi_station_set_config to set the configuration of target AP.
2. Call wifi_station_set_wpa2_enterprise_auth(1); to enable WPA2_Enterprise authentication.
    2.1. For EAP-TLS authentication, call wifi_station_set_enterprise_cert_key to set certificate and private key. wifi_station_set_enterprise_username is an optional choice, it can be called to set user name.
    2.2. For EAP-TTLS or EAP-PEAP authentication, call wifi_station_set_enterprise_username and wifi_station_set_enterprise_password to set user name and password.
    wifi_station_set_enterprise_ca_cert is an optional choice, it can be called to set root certificate.
3. Call wifi_station_connect to connect to target AP.
4. After being connected to an AP, or failing to connect to AP and on stopped retries, please call the corresponding wifi_station_clear_enterprise_XXX APIs to release the resources.

4. TCP/UDP APIs

Found in **ESP8266_NONOS_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.
- mDNS APIs: APIs that related to mDNS.

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 : Success

Non-0 : error, return error code

ESPCONN_ARG - illegal argument, cannot find network transmission according to structure espconn

ESPCONN_INPROGRESS - the connection is still in progress, please call espconn_disconnect to disconnect before deleting it.

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 - Success

ESPCONN_INPROGRESS - Error code : already connected

ESPCONN_ARG - Error code : illegal argument, cannot find network transmission according to structure espconn

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;

                        if (ipaddr != NULL)
    os_printf(user_esp_platform_dns_found %d.%d.%d.%d/n,
      *((uint8 *)&ipaddr->addr), *((uint8 *)&ipaddr->addr + 1),
      *((uint8 *)&ipaddr->addr + 2), *((uint8 *)&ipaddr->addr + 3));
}
void dns_test(void) {
   espconn_gethostbyname(pespconn,"iot.espressif.cn", &esp_server_ip,
         user_esp_platform_dns_found);
}
```

3. espconn_port

Function: get an available port

Prototype:

```
uint32 espconn_port(void)
```

Parameter:

none

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 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find network transmission according to structure espconn

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
)
```

Parameters:

struct espconn *espconn : corresponding connected control block structure

espconn_connect_callback connect_cb : registered callback function

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find network transmission according to structure espconn

6. espconn_sent_callback

Function:

Callback after the data are sent

Prototype:

void espconn_sent_callback (void *arg)

Parameters:

void *arg : pointer corresponding structure espconn. This pointer may be different in different callbacks, please don't use this pointer directly to distinguish one from another in multiple connections, use remote_ip and remote_port in espconn instead.

Return:

null

7. espconn_recv_callback

Function:

callback after data are received

Prototype:

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

Parameters:

void *arg : pointer corresponding structure espconn. This pointer may be different in different callbacks, please don't use this pointer directly to distinguish one from another in multiple connections, use remote_ip and remote_port in espconn instead.

char *pdata : received data entry parameters

unsigned short len : received data length

Return:

none

8. espconn_get_connection_info

Function:

Get the information about a TCP connection or UDP transmission. Usually used in the espconn_recv_callback.

Prototype:

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

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 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

Example:

```
 void user_udp_recv_cb(void *arg, char *pusrdata, unsigned short length)
 {
    struct espconn *pesp_conn = arg;
    remot_info *premot = NULL;
    if (espconn_get_connection_info(pesp_conn,&premot,0) == ESPCONN_OK){
          pesp_conn->proto.tcp->remote_port = premot->remote_port;
          pesp_conn->proto.tcp->remote_ip[0] = premot->remote_ip[0];
          pesp_conn->proto.tcp->remote_ip[1] = premot->remote_ip[1];
          pesp_conn->proto.tcp->remote_ip[2] = premot->remote_ip[2];
          pesp_conn->proto.tcp->remote_ip[3] = premot->remote_ip[3];
          espconn_sent(pesp_conn, pusrdata, os_strlen(pusrdata));
    }
 }
```

9. espconn_send

Function:

Send data through network

Note:

- Please call espconn_send after espconn_sent_callback of the pre-packet.
- If it is a UDP transmission, please set espconn->proto.udp->remote_ip and remote_port before every calling of espconn_send.

Prototype:

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

Parameters:

struct espconn *espconn : corresponding connected control block structure

uint8 *psent : pointer of data

uint16 length : data length

Return:

0 : Success

Non-0 : Error code

ESPCONN_MEM - Out of memory

ESPCONN_ARG - illegal argument,cannot find network transmission according to structure espconn

ESPCONN_MAXNUM - buffer (or 8 packets at most) of sending data is full

ESPCONN_IF - send UDP data fail

10. espconn_sent

[@deprecated] This API is deprecated, please use espconn_send instead.

Function:

Send data through network

Note:

- Please call espconn_sent after espconn_sent_callback of the pre-packet.
- If it is a UDP transmission, please set espconn->proto.udp->remote_ip and remote_port before every calling of espconn_sent.

Prototype:

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

Parameters:

struct espconn *espconn : corresponding connected control block structure

uint8 *psent : sent data pointer

uint16 length : sent data length

Return:

0 : Success

Non-0 : Error code

ESPCONN_MEM - Out of memory

ESPCONN_ARG - illegal argument, cannot find network transmission according to structure espconn

ESPCONN_MAXNUM - buffer of sending data is full

ESPCONN_IF - send UDP data fail

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)

Parameter:

struct espconn *espconn : corresponding connected control block structure

Return:

0 : Success

Non-0 : Error code

ESPCONN_MEM - Out of memory

ESPCONN_ISCONN - Already connected

ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

2. espconn_regist_time

Function:

Register timeout interval of ESP8266 TCP server.

Note:

Call this API after espconn_accept, before listened to a TCP connection. This API can not be used for SSL connection.

This timeout interval is not very precise, only as reference.

If timeout is set to 0, timeout will be disabled and ESP8266 TCP server will not disconnect if a TCP client has stopped communication. This usage of timeout=0, is deprecated.

Prototype:

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

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 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

3. espconn_connect

Function:

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

Note:

- If espconn_connect fail, returns non-0 value, there is no connection, so it won't enter any espconn callback.
- It is suggested to use espconn_port to get an available local port.

Prototype:

```
sint8 espconn_connect(struct espconn *espconn)
```

Parameters:

struct espconn *espconn : corresponding connected control block structure

Return:

0 : Success

Non-0 : Error code

ESPCONN_RTE - Routing Problem

ESPCONN_MEM - Out of memory

ESPCONN_ISCONN - Already connected

ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

4. espconn_connect_callback

Function: Callback for successful connection (ESP8266 as TCP server or ESP8266 as TCP client).

Callback can be registered by espconn_regist_connectcb

Prototype:

void espconn_connect_callback (void *arg)

Parameter:

void *arg : pointer to corresponding structure espconn. This pointer may be different in different callbacks, please don't use this pointer directly to distinguish one from another in multiple connections, use remote_ip and remote_port in espconn instead.

Return:

none

5. espconn_regist_connectcb

Function:

Register a connected callback which will be called on successful TCP connection

Prototype:

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

Parameters:

struct espconn *espconn : corresponding connected control block structure

espconn_connect_callback connect_cb : registered callback function

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

6. espconn_set_opt

Function: Set configuration options for TCP connection

Note:

This API can NOT be used for SSL connection.

In general, this API need not be called.

However, if espconn_set_opt is called, please call it from espconn_connect_callback.

Prototype:

sint8 espconn_set_opt(struct espconn *espconn, uint8 opt)

Structure:

```
enum espconn_option{
        ESPCONN_START = 0x00,
        ESPCONN_REUSEADDR = 0x01,
        ESPCONN_NODELAY = 0x02,
        ESPCONN_COPY = 0x04,
        ESPCONN_KEEPALIVE = 0x08,
        ESPCONN_END
}
```

Parameter:

struct espconn *espconn : corresponding connected control structure

uint8 opt : options for TCP connection, refer to espconn_option

bit 0: 1: free memory after TCP disconnection - need not wait 2 minutes;

bit 1: 1: disable nagle algorithm during TCP data transmission, quiken the data transmission.

bit 2: 1: enable espconn_regist_write_finish. Enter write finish callback once the data has been sent using espconn_send (data was written to 2920 bytes write-buffer for sending or has already been sent).

bit 3: 1: enable TCP keep alive

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

7. espconn_clear_opt

Function:

Clear option of TCP connection.

Prototype:

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

Structure:

```
enum espconn_option{
        ESPCONN_START = 0x00,
        ESPCONN_REUSEADDR = 0x01,
        ESPCONN_NODELAY = 0x02,
        ESPCONN_COPY = 0x04,
        ESPCONN_KEEPALIVE = 0x08,
        ESPCONN_END
}
```

Parameters:

struct espconn *espconn : corresponding connected control block structure

uint8 opt : option of TCP connection, refer to espconn_option

Return:

0 : Success

Non-0 : error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

8. espconn_set_keepalive

Function:

Set configuration of TCP keep alive.

Prototype**:**

```
sint8 espconn_set_keepalive(struct espconn *espconn, uint8 level, void* optarg)
```

Structure:

```
enum espconn_level{
        ESPCONN_KEEPIDLE,
        ESPCONN_KEEPINTVL,
        ESPCONN_KEEPCNT
}
```

Parameters:

struct espconn *espconn : corresponding connected control block structure

uint8 level : Default to do TCP keep-alive detection every ESPCONN_KEEPIDLE, if there in no response, retry ESPCONN_KEEPCNT times every ESPCONN_KEEPINTVL. If there is still no response, it is considered as a broken TCP connection and program calls espconn_reconnect_callback.

Notice, keep alive interval is not precise, only for reference, it depends on priority.

Description:

ESPCONN_KEEPIDLE - TCP keep-alive interval, unit:second

ESPCONN_KEEPINTVL - packet interval during TCP keep-alive, unit: second

ESPCONN_KEEPCNT - maximum packet count of TCP keep-alive

void* optarg : value of parameter

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

Note:

In general, this API need not be called.

If needed, please call it in espconn_connect_callback and call espconn_set_opt to enable keep alive first.

9. espconn_get_keepalive

Function:

Get value of TCP keep-alive parameter

Prototype:

sint8 espconn_set_keepalive(struct espconn *espconn, uint8 level, void* optarg)

Structure:

```
enum espconn_level{
    ESPCONN_KEEPIDLE,
    ESPCONN_KEEPINTVL,
    ESPCONN_KEEPCNT
}
```

Parameter:

struct espconn *espconn : corresponding connected control block structure

uint8 level :

ESPCONN_KEEPIDLE - TCP keep-alive interval, unit:second

ESPCONN_KEEPINTVL - packet interval during TCP keep-alive, unit: second

ESPCONN_KEEPCNT - maximum packet count of TCP keep-alive

void* optarg : value of parameter

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

10. espconn_reconnect_callback

Function:

This callback is entered when an error occurs, TCP connection broken. This callback is registered by espconn_regist_reconcb

Prototype:

void espconn_reconnect_callback (void *arg, sint8 err)

Parameter:

void *arg : pointer corresponding structure espconn. This pointer may be different in different callbacks, please do not use this pointer directly to distinguish one from another in multiple connections, use remote_ip and remote_port in espconn instead.

sint8 err : error code

ESCONN_TIMEOUT - Timeout

ESPCONN_ABRT - TCP connection aborted

ESPCONN_RST - TCP connection reset

ESPCONN_CLSD - TCP connection closed

ESPCONN_CONN - TCP connection

ESPCONN_HANDSHAKE - TCP SSL handshake fail

ESPCONN_PROTO_MSG - SSL application invalid

Return:

none

11. espconn_regist_reconcb

Function:

Register reconnect callback

Note:

espconn_reconnect_callback is more like a network-broken error handler; it handles errors that occurs in any phase of the connection. For instance, if espconn_send fails, espconn_reconnect_callback will be called because the network is broken.

Prototype:

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

Parameters:

struct espconn *espconn : corresponding connected control block structure

espconn_reconnect_callback recon_cb : registered callback function

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

12. espconn_disconnect

Function:

Disconnect a TCP connection

Note:

Do not call this API in any espconn callback. If needed, please use system_os_task and system_os_post to trigger espconn_disconnect

Prototype:

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

Parameters:

struct espconn *espconn : corresponding connected control structure

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

13. 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
)
```

Parameters:

struct espconn *espconn : corresponding connected control block structure

espconn_connect_callback connect_cb : registered callback function

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

14. espconn_abort

Function:

Forcefully abort a TCP connection

Note:

Do not call this API in any espconn callback. If needed, please use system_os_task and system_os_post to trigger espconn_abort.

Prototype:

```
sint8 espconn_abort(struct espconn *espconn)
```

Parameters:

struct espconn *espconn : corresponding network connection

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

15. espconn_regist_write_finish

Function:

Register a callback which will be called when all sending data is completely written into write buffer or sent. Need to call espconn_set_opt to enable write-buffer first.

Note:

- This API can not be used for SSL connection.
- write-buffer is used to keep TCP data that is waiting to be sent, queue number of the write-buffer is 8 which means that it can keep 8 packets at most. The size of write-buffer is 2920 bytes.
- Users can enable it by using espconn_set_opt.
- Users can call espconn_send to send the next packet in write_finish_callback instead of using espconn_sent_callback.

Prototype:

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

Parameters:

struct espconn *espconn ： corresponding network connection

espconn_connect_callback write_finish_fn : registered callback function

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

16. espconn_tcp_get_max_con

Function:

Get the number of maximum TCP connections allowed.

Prototype:

```
uint8 espconn_tcp_get_max_con(void)
```

Parameter:

none

Return:

Maximum number of TCP connections allowed.

17. espconn_tcp_set_max_con

Function:

Set the maximum number of TCP connections allowed.

Prototype:

sint8 espconn_tcp_set_max_con(uint8 num)

Parameter:

uint8 num : Maximum number of TCP connections allowed.

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

18. espconn_tcp_get_max_con_allow

Function:

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

Prototype:

sint8 espconn_tcp_get_max_con_allow(struct espconn *espconn)

Parameter:

struct espconn *espconn : corresponding network connection

Return:

> 0 : Maximum number of TCP clients allowed.

< 0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

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 network connection

uint8 num : Maximum number of TCP clients allowed.

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

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 network connection

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure 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 network connection

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

22. espconn_secure_accept

Function:

Creates an SSL TCP server.

Note:

- This API can be called only once, only one SSL server is allowed to be created, and only one SSL client can be connected.
- If SSL encrypted packet size is larger than ESP8266 SSL buffer size (default 2KB, set by espconn_secure_set_size), SSL connection will fail, will enter espconn_reconnect_callback
- SSL related APIs named as espconn_secure_XXX are different from normal TCP APIs and must not be used interchangeably. In SSL connection, only espconn_secure_XXX APIs, espconn_regist_XXXcb APIs and espconn_port can be used.
- Users should call API espconn_secure_set_default_certificate and espconn_secure_set_default_private_key to set SSL certificate and secure key first.

Prototype:

sint8 espconn_secure_accept(struct espconn *espconn)

Parameter:

struct espconn *espconn : corresponding network connection

Return:

0 : Success

Non-0 : Error code

ESPCONN_MEM - Out of memory

ESPCONN_ISCONN - Already connected

ESPCONN_ARG - illegal argument,cannot find TCP connection according to structure espconn

23. espconn_secure_delete

Function:

Delete the SSL connection when ESP8266 runs as SSL server.

Prototype:

sint8 espconn_secure_delete(struct espconn *espconn)

Parameter:

struct espconn *espconn : corresponding SSL connection

Return:

0 : Success

Non-0 : Error, return error code

ESPCONN_ARG - illegal argument, cannot find network transmission according to structure espconn

ESPCONN_INPROGRESS - the SSL connection is still in progress, please call espconn_secure_disconnect to disconnect before deleting it.

24. espconn_secure_set_size

Function:

Set buffer size of encrypted data (SSL)

Note:

Buffer size default to be 2Kbytes. If need to change, please call this API before espconn_secure_accept (ESP8266 as TCP SSL server) or espconn_secure_connect (ESP8266 as TCP SSL client)

Prototype:

bool espconn_secure_set_size (uint8 level, uint16 size)

Parameters:

uint8 level : set buffer for ESP8266 SSL server/client:

0x01	SSL client;
0x02	SSL server;
0x03	both SSL client and SSL server

uint16 size : buffer size, range: 1 ~ 8192, unit: byte, default to be 2048

Return:

true : Success

false : Failure

25. espconn_secure_get_size

Function:

Get buffer size of encrypted data (SSL)

Prototype:

sint16 espconn_secure_get_size (uint8 level)

Parameters:

uint8 level : buffer for ESP8266 SSL server/client:

0x01	SSL client;
0x02	SSL server;
0x03	both SSL client and SSL server

Return:

buffer size

26. espconn_secure_connect

Function:

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

Note:

- If espconn_connect fails, returns non-0 value, it is not connected and therefore will not enter any espconn callback.
- Only one connection is allowed when the ESP8266 acts as a SSL client, this API can be called only once, or call espconn_secure_disconnect to disconnect first, then call this API to create another SSL connection.
- If SSL encrypted packet size is larger than the ESP8266 SSL buffer size (default 2KB, set by espconn_secure_set_size), the SSL connection will fail, will enter espconn_reconnect_callback
- SSL related APIs named as espconn_secure_XXX are different from normal TCP APIs and must not be used interchangeably. In SSL connection, only espconn_secure_XXX APIs, espconn_regist_XXXcb APIs and espconn_port can be used.

Prototype:

sint8 espconn_secure_connect (struct espconn *espconn)

Parameters:

struct espconn *espconn : corresponding network connection

Return:

0 : Success

Non-0 : Error code

ESPCONN_MEM - Out of memory

ESPCONN_ISCONN - Already connected

ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

27. espconn_secure_send

Function: send encrypted data (SSL)

Note:

Please call espconn_secure_send after espconn_sent_callback of the pre-packet.

Prototype:

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

Parameters:

struct espconn *espconn : corresponding network connection

uint8 *psent : sent data pointer

uint16 length : sent data length

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

28. espconn_secure_sent

[@deprecated] This API is deprecated, please use espconn_secure_send instead.

Function: send encrypted data (SSL)

Note:

Please call espconn_secure_sent after espconn_sent_callback of the pre-packet.

Prototype:

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

Parameters:

struct espconn *espconn : corresponding network connection

uint8 *psent : sent data pointer

uint16 length : sent data length

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

29. espconn_secure_disconnect

Function: secure TCP disconnection(SSL)

Note:

Do not call this API in any espconn callback. If needed, please use system_os_task and system_os_post to trigger espconn_secure_disconnect

Prototype:

sint8 espconn_secure_disconnect(struct espconn *espconn)

Parameters:

struct espconn *espconn : corresponding network connection

Return:

0 : Success

Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

30. espconn_secure_ca_disable

Function:

Disable SSL CA (certificate authenticate) function

Note:

- CA function is disabled by default, more details in document "5A-ESP8266__SDK__SSL_User_Manual"

Prototype:

bool espconn_secure_ca_disable (uint8 level)

Parameter:

uint8 level : set configuration for ESP8266 SSL server/client:

0x01	SSL client;
0x02	SSL server;
0x03	both SSL client and SSL server

Return:

true: Success

false: Failure

31. espconn_secure_ca_enable

Function:

Enable SSL CA (certificate authenticate) function

Note:

- CA function is disabled by default, more details in document "5A-ESP8266__SDK__ SSL_User_Manual"
- This API must be called before espconn_secure_accept (ESP8266 as TCP SSL server) or espconn_secure_connect (ESP8266 as TCP SSL client)

Prototype:

bool espconn_secure_ca_enable (uint8 level, uint32 flash_sector)

Parameter:

uint8 level : set configuration for ESP8266 SSL server/client:

0x01	SSL client;
0x02	SSL server;
0x03	both SSL client and SSL server

uint32 flash_sector : flash sector in which CA (esp_ca_cert.bin) is downloaded. For example, if the flash_sector is 0x3B, then esp_ca_cert.bin must be downloaded to flash at 0x3B000

Return:

true : Success

false: Failure

32. espconn_secure_cert_req_enable

Function:

Enable certification verification function when ESP8266 runs as SSL client

Note:

- Certification verification function is disabled by defaults. If the SSL server does not require certification verification, this API need not be called.
- This API must be called before espconn_secure_connect is called

Prototype:

bool espconn_secure_cert_req_enable (uint8 level, uint32 flash_sector)

Parameter:

uint8 level : can only be set as 0x01 when ESP8266 runs as SSL client;

uint32 flash_sector : set the address where secure key (esp_cert_private_key.bin) will be written into the flash. For example, parameters 0x3A should be written into address 0x3A000 in the flash. Please note that the secure key written into flash must not overlap with code BINs or system parameter BINs in the flash memory.

Return:

true: Success

false: Failure

33. espconn_secure_cert_req_disable

Function:

Disable certification verification function when ESP8266 runs as SSL client

Note:

- Certification verification function is disabled by default

Prototype:

bool espconn_secure_ca_disable (uint8 level)

Parameter:

uint8 level : can only be set as 0x01, when ESP8266 runs as SSL client.

Return:

true : Success

false: Failure

34. espconn_secure_set_default_certificate

Function:

Set the certificate when ESP8266 runs as SSL server

Note:

- Demos can be found in ESP8266_NONOS_SDK\examples\IoT_Demo
- This API has to be called before espconn_secure_accept.

Prototype:

bool espconn_secure_set_default_certificate (const uint8_t* certificate, uint16_t length)

Parameter:

const uint8_t* certificate : pointer to the certificate

uint16_t length : length of the certificate

Return:

true: Success

false: Failure

35. espconn_secure_set_default_private_key

Function:

Set the secure key when ESP8266 runs as SSL server

Note:

- Demos can be found in ESP8266_NONOS_SDK\examples\IoT_Demo
- This API has to be called before espconn_secure_accept.

Prototype:

bool espconn_secure_set_default_private_key (const uint8_t* key, uint16_t length)

Parameter:

const uint8_t* key : pointer to the secure key

uint16_t length : length of the secure key

Return:

true : Success

false: Failure

4.3. UDP APIs

1. espconn_create

Function:

Create UDP transmission.

Note:

Parameter remote_ip and remote_port need to be set, do not set to be 0.

Prototype:

sin8 espconn_create(struct espconn *espconn)

Parameter:

struct espconn *espconn : corresponding network transmission

Return:

0 : Success

Non-0 : Error code

ESPCONN_ISCONN - Already connected

ESPCONN_MEM - Out of memory

ESPCONN_ARG - illegal argument, cannot find UDP transmission according to structure espconn

2. espconn_sendto

Function:

Send UDP data

Prototype:

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

Parameter:

struct espconn *espconn : corresponding network transmission

uint8 *psent : pointer of data

uint16 length : data length

Return:

0 : Success

Non-0 : Error code

ESPCONN_ISCONN - Already connected

ESPCONN_MEM - Out of memory

ESPCONN_IF - send UDP data fail

3. espconn_igmp_join

Function:

Join a multicast group

Note:

This API can only be called after the ESP8266 station connects to a router.

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 : Success

Non-0 : Error code ESPCONN_MEM - Out of memory

4. 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 : Success

Non-0 : Error code ESPCONN_MEM - Out of memory

5. espconn_dns_setserver

Function:

Set default DNS server. Two DNS servers are allowed to be set.

Note:

Only if ESP8266 DHCP client is disabled (wifi_station_dhcpc_stop), this API can be used.

Prototype:

void espconn_dns_setserver(char numdns, ip_addr_t *dnsserver)

Parameter:

char numdns : DNS server ID, 0 or 1

ip_addr_t *dnsserver : DNS server IP

Return:

none

4.4. mDNS APIs

1. espconn_mdns_init

Function:

mDNS initialization

Note:

- In soft-AP+station mode, call wifi_set_broadcast_if(STATIONAP_MODE); first to enable broadcast for both soft-AP and station interface.
- Using station interface, please obtain IP address of the ESP8266 station first before calling the API to initialize mDNS;
- txt_data has to be set as " key = value ", as **Example**;

Structure:

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

Prototype:

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

Parameter:

struct mdns_info *info ： mDNS information

Return:

none

2. espconn_mdns_close

Function:

Close mDNS, corresponding creation API : espconn_mdns_init

Prototype:

```
void espconn_mdns_close(void)
```

Parameter:

none

Return:

none

3. espconn_mdns_server_register

Function:

Register mDNS server

Prototype:

```
void espconn_mdns_server_register(void)
```

Parameter:

none

Return:

none

4. espconn_mdns_server_unregister

Function:

Unregister mDNS server

Prototype:

void espconn_mdns_server_unregister(void)

Parameter:

none

Return:

none

5. espconn_mdns_get_servername

Function:

Get mDNS server name

Prototype:

char* espconn_mdns_get_servername(void)

Parameter:

none

Return:

server name

6. espconn_mdns_set_servername

Function:

Set mDNS server name

Prototype:

void espconn_mdns_set_servername(const char *name)

Parameter:

const char *name : server name

Return:

none

7. espconn_mdns_set_hostname

Function:

Set mDNS host name

Prototype:

void espconn_mdns_set_hostname(char *name)

Parameter:

char *name : host name

Return:

none

8. espconn_mdns_get_hostname

Function:

Get mDNS host name

Prototype:

char* espconn_mdns_get_hostname(void)

Parameter:

none

Return:

host name

9. espconn_mdns_disable

Function:

Disable mDNS, corresponding creation API : espconn_mdns_enable

Prototype:

void espconn_mdns_disable(void)

Parameter:

none

Return:

none

10. espconn_mdns_enable

Function:

Enable mDNS

Prototype:

void espconn_mdns_enable(void)

Parameter:

none

Return:

none

11. Example of mDNS

Please do not use special characters (for example, "." character), or use a protocol name (for example, "http"), when defining "host_name" and "server_name" for mDNS.

```
struct mdns_info info;

void user_mdns_config()

{

        struct ip_info ipconfig;

        wifi_get_ip_info(STATION_IF, &ipconfig);

        info->host_name = "espressif";
        info->ipAddr = ipconfig.ip.addr; //ESP8266 station IP
        info->server_name = "iot";
        info->server_port = 8080;
        info->txt_data[0] = "version = now";
        info->txt_data[1] = "user1 = data1";
        info->txt_data[2] = "user2 = data2";
        espconn_mdns_init(&info);

}
```

5. Mesh APIs

For more information on Mesh, please refer to documentation "30A_ESP8266__ Mesh_User Guide".

Download: http://bbs.espressif.com/viewtopic.php?f=51&t=1977

6. Application Related

6.1. AT APIs

AT APIs can be found in **/ESP8266_NONOS_SDK/include/at_custom.h**.

For AT APIs examples, refer to ESP8266_NONOS_SDK/examples/at.

1. at_response_ok

Function:

Output OK to AT Port (UART0)

Prototype:

void at_response_ok(void)

Parameter:

none

Return:

none

2. at_response_error

Function:

output ERROR to AT Port (UART0)

Prototype:

void at_response_error(void)

Parameter:

none

Return:

none

3. at_cmd_array_regist

Function:

register user-defined AT commands.

Can be called only once to register all user-defined 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-defined AT commands

uint32 cmd_num : Number counts of user-defined AT commands

Return:

none

Example:

refer to ESP8266_NONOS_SDK/examples/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 ESP8266_NONOS_SDK/examples/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 needs to be parsed

int32 max_len : max string length allowed

Return:

length of string:

>=0: Success, and returns the length of the string

<0 : Failure, and returns -1

Example:

refer to ESP8266_NONOS_SDK/examples/at/user/user_main.c

6. at_init

Function:

AT initialize

Prototype:

void at_init (void)

Parameter:

none

Return:

none

Example:

refer to ESP8266_NONOS_SDK/examples/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:

none

Example:

refer to ESP8266_NONOS_SDK/examples/at/user/user_main.c

8. at_set_custom_info

Function:

User-defined 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:

none

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:

none

Return:

none

10. at_leave_special_state

Function:

Exit from AT processing state.

Prototype:

void at_leave_special_state (void)

Parameter:

none

Return:

none

11. at_get_version

Function:

Get Espressif AT lib version.

Prototype:

uint32 at_get_version (void)

Parameter:

none

Return:

Espressif AT lib version

12. at_register_uart_rx_intr

Function:

Set UART0 to be used by user or AT commands.

Note:

This API can be called multiple times.

Running AT,UART0 default to be used by AT commands.

Prototype:

void at_register_uart_rx_intr(at_custom_uart_rx_intr rx_func)

Parameter:

at_custom_uart_rx_intr : register a UART0 RX interrupt handler so that UART0 can be used by the customer, but if it is NULL, UART0 is assigned to AT commands.

Return:

none

Example:

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

        // change UART0 for AT
        at_register_uart_rx_intr(NULL);
}
void user_init(void){ at_register_uart_rx_intr(user_uart_rx_intr); }
```

13. at_response

Function:

Set AT response

Note:

at_response outputs from UART0 TX by default which is same as at_port_print. But on calling at_register_response_func, the string of at_response will be the parameter of response_func, users can define their own behavior.

Prototype:

void at_response (const char *str)

Parameter:

const char *str : string

Return:

none

14. at_register_response_func

Function:

Register callback of at_response for user-definedd responses. After calling at_register_response_func, the string of at_response will be the parameter of response_func, users can define their own behavior.

Prototype:

void at_register_response_func (at_custom_response_func_type response_func)

Parameter:

at_custom_response_func_type : callback of at_response

Return:

none

15. at_fake_uart_enable

Function:

Enable UART simulation, can be used to develop AT commands through SDIO or network.

Prototype:

bool at_fake_uart_enable(bool enable, at_fake_uart_tx_func_type func)

Parameter:

bool enable : enable UART simulation

at_fake_uart_tx_func_type func : callback for UART TX simulation

Return:

true: Success

false: Failure

16. at_fake_uart_rx

Function:

UART RX simulation, can be used to develop AT commands through SDIO or network.

Prototype:

uint32 at_fake_uart_rx(uint8* data, uint32 length)

Parameter:

uint8* data : data for UART(simulation) RX

uint32 length : length of data

Return:

If successful, the return value will be equal to length, otherwise, failure

17. at_set_escape_character

Function:

Set an escape character for AT commands. Default escape character is "\".

Prototype:

bool at_set_escape_character(uint8 ch)

Parameter:

uint8 ch : escape character, can be character !, or #, or $, or @, or &, or \.

Return:

true: Success

false: Failure

6.2. Related JSON APIs

Found in : **ESP8266_NONOS_SDK/include/json/jsonparse.h & jsontree.h**

1. jsonparse_setup

Function:

Initialize JSON parser

Prototype:

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

Parameters:

struct jsonparse_state *state : json parsing pointer

const char *json : json parsing character string

int len : character string length

Return:

none

2. jsonparse_next

Function:

Returns jsonparse next object

Prototype:

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

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
)
```

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)

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)

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)

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)

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)

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)
)
```

Parameters:

struct jsontree_context *js_ctx : json tree element pointer

struct jsontree_value *root : root element pointer

int (* putchar)(int) : input function

Return:

none

10. jsontree_reset

Function:

Resets json tree

Prototype:

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

Parameters:

`struct jsontree_context *js_ctx` : json data tree pointer

Return:

none

11. jsontree_path_name

Function:

get json tree parameters

Prototype:

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

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
)
```

Parameters:

struct jsontree_context *js_ctx : json tree pointer

int value : integer value

Return:

none

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
)
```

Parameters:

struct jsontree_context *js_ctx : json tree pointer

int *text : array entry address

uint32 length : array length

Return:

none

14. jsontree_write_string

Function:

Writes string to json tree

Prototype:

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

Parameters:

struct jsontree_context *js_ctx : json tree pointer

const char* text : character string pointer

Return:

none

15. jsontree_print_next

Function:

json tree depth

Prototype:

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

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
)
```

Parameters:

struct jsontree_context *js_ctx : json tree pointer

int : type

Return:

struct jsontree_value * : json tree element pointer

7. Definitions & Structures

7.1. 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;
```

7.2. 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, SSID is checked till a termination character is found; 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;

    uint8 bssid[6];
    uint8 ssid[32];
    uint8 ssid_len;
    uint8 channel;
    sint8 rssi;
    AUTH_MODE authmode;
    uint8 is_hidden;            // SSID of current AP is hidden or not.
    sint16 freq_offset;         // AP’s frequency offset
    sint16 freqcal_val;
    uint8 *esp_mesh_ie;
    uint8 simple_pair;
};
typedef void (* scan_done_cb_t)(void *arg, STATUS status);
```

4. Wi-Fi event related structures

```
enum {
    EVENT_STAMODE_CONNECTED = 0,
    EVENT_STAMODE_DISCONNECTED,
    EVENT_STAMODE_AUTHMODE_CHANGE,
    EVENT_STAMODE_GOT_IP,
    EVENT_STAMODE_DHCP_TIMEOUT,
    EVENT_SOFTAPMODE_STACONNECTED,
    EVENT_SOFTAPMODE_STADISCONNECTED,
    EVENT_SOFTAPMODE_PROBEREQRECVED,
    EVENT_MAX
};

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

```
        REASON_BEACON_TIMEOUT       = 200,
        REASON_NO_AP_FOUND          = 201,
        REASON_AUTH_FAIL            = 202,
        REASON_ASSOC_FAIL           = 203,
        REASON_HANDSHAKE_TIMEOUT    = 204,
};

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

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

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

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

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

typedef struct {
```

```
        uint8 mac[6];
        uint8 aid;
} Event_SoftAPMode_StaDisconnected_t;

typedef struct {
        int rssi;
        uint8 mac[6];
} Event_SoftAPMode_ProbeReqRecved_t;

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

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

5. smart config structures

```
typedef enum  {
   SC_STATUS_WAIT = 0,      // Please don't start connection in this phase
   SC_STATUS_FIND_CHANNEL,  // Start connection by APP in this phase
   SC_STATUS_GETTING_SSID_PSWD,
   SC_STATUS_LINK,
   SC_STATUS_LINK_OVER,     // Got IP, connect to AP successfully
} sc_status;
typedef enum  {
   SC_TYPE_ESPTOUCH = 0,
   SC_TYPE_AIRKISS,
   SC_TYPE_ESPTOUCH_AIRKISS,
} sc_type;
```

7.3. JSON Related Structure

1. json structures

```
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 definitions

```
#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 }
```

7.4. espconn parameters

1. callback functions

```
/** 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 structures

```
typedef void* espconn_handle;
typedef struct _esp_tcp {
    int remote_port;
    int local_port;
    uint8 local_ip[4];
    uint8 remote_ip[4];
            espconn_connect_callback connect_callback;
            espconn_reconnect_callback reconnect_callback;
            espconn_connect_callback disconnect_callback;
            espconn_connect_callback write_finish_fn;
} esp_tcp;

typedef struct _esp_udp {
    int remote_port;
    int local_port;
    uint8 local_ip[4];
    uint8 remote_ip[4];
} esp_udp;

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

/** 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
};

enum espconn_option{
        ESPCONN_START = 0x00,
        ESPCONN_REUSEADDR = 0x01,
        ESPCONN_NODELAY = 0x02,
        ESPCONN_COPY = 0x04,
        ESPCONN_KEEPALIVE = 0x08,
        ESPCONN_END
}

enum espconn_level{
        ESPCONN_KEEPIDLE,
        ESPCONN_KEEPINTVL,
        ESPCONN_KEEPCNT
}
/** A espconn descriptor */
struct espconn {
    /** type of the espconn (TCP, UDP) */
    enum espconn_type type;
    /** current state of the espconn */
    enum espconn_state state;
    union {
        esp_tcp *tcp;
        esp_udp *udp;
    } proto;
    /** A callback function that is informed about events for this espconn */
    espconn_recv_callback recv_callback;
    espconn_sent_callback sent_callback;
    uint8 link_cnt;
    void *reverse; // reversed for customer use
};
```

7.5. interrupt related definitions

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

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

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

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

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

/* enable a interrupt */
```

```
#define ETS_INTR_ENABLE(inum)   ets_isr_unmask((1<<inum))

/* disable a interrupt */

#define ETS_INTR_DISABLE(inum)  ets_isr_mask((1<<inum))


/* enable SPI interrupt */

#define ETS_SPI_INTR_ENABLE()   ETS_INTR_ENABLE(ETS_SPI_INUM)

/* enable UART interrupt */

#define ETS_UART_INTR_ENABLE()          ETS_INTR_ENABLE(ETS_UART_INUM)

/* disable UART interrupt */

#define ETS_UART_INTR_DISABLE()         ETS_INTR_DISABLE(ETS_UART_INUM)


/* enable frc1 timer interrupt */

#define ETS_FRC1_INTR_ENABLE()          ETS_INTR_ENABLE(ETS_FRC_TIMER1_INUM)

/* disable frc1 timer interrupt */

#define ETS_FRC1_INTR_DISABLE()         ETS_INTR_DISABLE(ETS_FRC_TIMER1_INUM)


/* enable GPIO interrupt */

#define ETS_GPIO_INTR_ENABLE()ETS_INTR_ENABLE(ETS_GPIO_INUM)

/* disable GPIO interrupt */

#define ETS_GPIO_INTR_DISABLE()         ETS_INTR_DISABLE(ETS_GPIO_INUM)
```

8. Peripheral Related Drivers

Peripheral drivers can refer to /ESP8266_NONOS_SDK/driver_lib.

8.1. GPIO Related APIs

GPIO APIs can be found in **/ESP8266_NONOS_SDK/include/eagle_soc.h & gpio.h**.

Please refer to /ESP8266_NONOS_SDK/examples/IoT_Demo/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_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 output bit

uint32 disable_mask : enable input bit

Return:

none

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 macros

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_ANYEDGE = 3,
    GPIO_PIN_INTR_LOLEVEL = 4,
    GPIO_PIN_INTR_HILEVEL = 5
} GPIO_INT_TYPE;
```

Return:

none

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);
```

8.2. UART Related APIs

By default, UART0 is a debug output interface. In the case of a dual UART, UART0 works as data receive and transmit interface, while UART1 acts as the debug output interface. Please make sure all hardware is correctly connected.

1. uart_init

Function:

Initialize 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:

none

2. uart0_tx_buffer

Function:

Send user-defined data through UART0

Prototype:

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

Parameter:

uint8 *buf : data to be sent

uint16 len : the length of data to be sent

Return:

none

3. uart0_rx_intr_handler

Function:

UART0 interrupt processing function. Users can process the received data in this function.

Prototype:

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

Parameter:

void *para : the pointer pointing to RcvMsgBuff structure

Return:

none

8.3. I2C Master Related APIs

1. i2c_master_gpio_init

Function:

Set GPIO in I2C master mode

Prototype:

void i2c_master_gpio_init (void)

Parameters:

none

Return:

none

2. i2c_master_init

Function:

Initialize I2C

Prototype:

void i2c_master_init(void)

Parameters:

none

Return:

none

3. i2c_master_start

Function: configures I2C to start sending data

Prototype:

void i2c_master_start(void)

Parameters:

none

Return:

none

4. i2c_master_stop

Function:

configures I2C to stop sending data

Prototype:

void i2c_master_stop(void)

Parameters:

none

Return:

none

5. i2c_master_send_ack

Function:

Sends I2C ACK

Prototype:

void i2c_master_send_ack (void)

Parameters:

none

Return:

none

6. i2c_master_send_nack

Function:

Sends I2C NACK

Prototype:

void i2c_master_send_nack (void)

Parameters:

none

Return:

none

7. i2c_master_checkAck

Function:

Checks ACK from slave

Prototype:

bool i2c_master_checkAck (void)

Parameters:

none

Return:

true: ACK received from I2C slave

false: NACK received from I2C slave

8. i2c_master_readByte

Function:

Read one byte from I2C slave

Prototype:

uint8 i2c_master_readByte (void)

Input Parameters:

none

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:

none

8.4. PWM Related

PWM APIs can not be called when APIs in hw_timer.c are in use, because they use the same hardware timer.

Do not set the system to be Light Sleep mode (wifi_set_sleep_type(LIGT_SLEEP);), because that Light Sleep will stop the CPU, it can not be interrupted by NMI during light sleep.

To enter Deep Sleep mode, PWM needs to be stopped first.

1. pwm_init

Function:

Initialize PWM function, including GPIO selection, period and duty cycle.

Note:

This API can be called only once.

Prototype:

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

Parameter:

uint32 period : PWM period

uint8 *duty : duty cycle of each output

uint32 pwm_channel_num: PWM channel number

uint32 (*pin_info_list)[3]: GPIO parameter of PWM channel, it is a pointer of n * 3 array which defines GPIO register, IO reuse of corresponding PIN and GPIO number.

Return:

none

Example:

```
uint32 io_info[][3] =
        {{PWM_0_OUT_IO_MUX,PWM_0_OUT_IO_FUNC,PWM_0_OUT_IO_NUM},
        {PWM_1_OUT_IO_MUX,PWM_1_OUT_IO_FUNC,PWM_1_OUT_IO_NUM},
        {PWM_2_OUT_IO_MUX,PWM_2_OUT_IO_FUNC,PWM_2_OUT_IO_NUM}};

pwm_init(light_param.pwm_period, light_param.pwm_duty, 3, io_info);
```

2. pwm_start

Function:

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

Prototype:

void pwm_start (void)

Parameter:

none

Return:

none

3. pwm_set_duty

Function:

Sets duty cycle of a PWM output. Set the time that high-level signal will last, duty depends on period, the maximum value can be Period * 1000 /45. For example, 1KHz PWM, duty range is 0 ~ 22222

Note:

After updating the configuration, pwm_start must be called for the changes to take effect.

Prototype:

void pwm_set_duty(uint32 duty, uint8 channel)

Input Parameters:

uint32 duty : the time that high-level single will last, duty cycle will be (duty*45)/ (period*1000)

uint8 channel : PWM channel, depends on how many PWM channels is used, in IOT_Demo it depends on #define PWM_CHANNEL

Return:

none

4. pwm_get_duty

Function:

Gets duty cycle of PWM output, duty cycle will be (duty*45)/ (period*1000)

Prototype:

uint8 pwm_get_duty(uint8 channel)

Input Parameters:

uint8 channel : PWM channel, depends on how many PWM channels is used, in IOT_Demo it depends on #define PWM_CHANNEL

Return:

uint8 : duty cycle of PWM output

5. pwm_set_period

Function:

Sets PWM period, unit: us. For example, for 1KHz PWM, period is 1000 us

Note:

After updating the configuration, pwm_start must be called for the changes to take effect.

Prototype:

void pwm_set_period(uint32 period)

Input Parameters:

uint32 period : PWM period, unit: us

Return:

none

6. pwm_get_period

Function:

Gets PWM period.

Prototype:

uint32 pwm_get_period(void)

Parameter:

none

Return:

PWM period, unit: us.

7. get_pwm_version

Function:

Get version information of PWM.

Prototype:

uint32 get_pwm_version(void)

Parameter:

none

Return:

PWM version

8.5. SDIO APIs

ESP8266 can only work as SDIO slave.

1. sdio_slave_init

Function:

SDIO slave initialization.

Prototype:

void sdio_slave_init(void)

Parameter:

none

Return:

none

2. sdio_load_data

Function:

Load data into SDIO buffer, and inform SDIO host to read it.

Prototype:

int32 sdio_load_data(const uint8* data, uint32 len)

Parameter:

const uint8* data : data that will be transmitted

uint32 len : the length of data

Return:

The length of data that be loaded successfully.

If the data length is too long to fit in SDIO buffer, this API will return 0 which means it failed to load data.

3. sdio_register_recv_cb

Function:

Register a callback which will be called when ESP8266 receives data from the host through SDIO.

Callback Function:

typedef void(*sdio_recv_data_callback)(uint8* data, uint32 len)

- The sdio_recv_data_callback can not be stored in cache, so please do not define ICACHE_FLASH_ATTR before it.

Prototype:

bool sdio_register_recv_cb(sdio_recv_data_callback cb)

Parameter:

sdio_recv_data_callback cb : callback

Return:

true: Success

false: Failure

9. Appendix

9.1. ESPCONN Programming

1. TCP Client Mode

> Notes:
>
> - ESP8266, working in Station mode, will start client connections when given an IP address.
> - ESP8266, working in soft-AP mode, will start client connections when the devices connected to the ESP8266 are given IP addresses.

Steps

1. Initialize espconn parameters according to protocols.
2. Register connect callback function, and register reconnect callback function.
   - (Call espconn_regist_connectcb and espconn_regist_reconcb)
3. Call espconn_connect function and set up the connection with TCP Server.
4. Registered connected callback functions will be called after successful connection, which will register corresponding callback function. We recommend registering a disconnect callback function.
   - (Call espconn_regist_recvcb , espconn_regist_sentcb and espconn_regist_disconcb in connected callback)
5. 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 firmware functions are completed.

2. TCP Server Mode

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

Steps

1. Initialize espconn parameters according to protocols.
2. Register connect callback and reconnect callback function.
   - (Call espconn_regist_connectcb and espconn_regist_reconcb)
3. Call espconn_accept function to listen to the connection with host.
4. Registered connect function will be called after a successful connection, which will register a corresponding callback function.

- (Call espconn_regist_recvcb , espconn_regist_sentcb and espconn_regist_disconcb in connected callback)

3. espconn callback

Register Function	Callback	Description
espconn_regist_connectcb	espconn_connect_callback	TCP connected successfully
espconn_regist_reconcb	espconn_reconnect_callback	Error occur,TCP disconnected
espconn_regist_sentcb	espconn_sent_callback	Sent TCP or UDP data
espconn_regist_recvcb	espconn_recv_callback	Received TCP or UDP data
espconn_regist_write_finish	espconn_write_finish_callback	Write data into TCP-send-buffer
espconn_regist_disconcb	espconn_disconnect_callback	TCP disconnected successfully

⚠ Notice:

- Parameter arg of callback is the pointer corresponding structure espconn. This pointer may be different in different callbacks, please do not use this pointer directly to distinguish one from another in multiple connections, use remote_ip and remote_port in espconn instead.
- If espconn_connect (or espconn_secure_connect) fails and returns non-0 value, there is no connection, so it won't enter any espconn callback.
- Don't call espconn_disconnect (or espconn_secure_disconnect) to break the TCP connection in any espconn callback.
  - If it is needed, please use system_os_task and system_os_post to trigger the disconnection (espconn_disconnect or espconn_secure_disconnect).

9.2. 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){
        os_printf("rtc time init...\r\n");
        rtc_time.magic = RTC_MAGIC;
        rtc_time.time_acc= 0;
        rtc_time.time_base = system_get_rtc_time();
        os_printf("time base : %d \r\n",rtc_time.time_base);
    }

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

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

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

    st2 = system_get_time();
    rtc_t2 = system_get_rtc_time();
    cal2 = system_rtc_clock_cali_proc();
    os_printf(" rtc_t2-t1 : %d \r\n",rtc_t2-rtc_t1);
    os_printf(" st2-t2 :  %d  \r\n",st2-st1);
    os_printf("cal 1  : %d.%d  \r\n", ((cal1*1000)>>12)/1000, ((cal1*1000)>>12)%1000 );
    os_printf("cal 2  : %d.%d \r\n",((cal2*1000)>>12)/1000,((cal2*1000)>>12)%1000 );
    os_printf("==================\r\n\r\n");
    rtc_time.time_acc += (  ((uint64)(rtc_t2 - rtc_time.time_base))  *  ( (uint64)((cal2*1000)>>12))  ) ;
```

```
    os_printf("rtc time acc  : %lld \r\n",rtc_time.time_acc);
    os_printf("power on time :  %lld  us\r\n", rtc_time.time_acc/1000);
    os_printf("power on time :  %lld.%02lld  S\r\n", (rtc_time.time_acc/10000000)/100, (rtc_time.time_acc/
10000000)%100);

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

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

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

9.3. Sniffer Structure Introduction

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

The following HT20 packet types are supported:

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

The following packet types are not supported:

- HT40
- LDPC

Although the ESP8266 can not decipher some IEEE80211 packets completely, it can Get the length of these packets.

Therefore, when in the sniffer mode, the ESP8266 can either (1) completely capture the packets or (2) Get the length of the packets.

- For packets that ESP8266 can decipher completely, the ESP8266 returns with the
  - MAC addresses of both communication sides and the encryption type
  - the length of the entire packet.
- For packets that ESP8266 cannot completely decipher, the ESP8266 returns with
  - the length of the entire 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 not 11n packet; non-0: is 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
};
```

The callback function wifi_promiscuous_rx contains two parameters (buf and len). len shows the length of buf, it can be: len = 128, len = X * 10, len = 12.

LEN == 128

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

LEN == X * 10

- buf contains structure sniffer_buf: this structure is reliable, data packets represented by it have been verified by CRC.
- sniffer_buf.cnt shows the number of packets in buf. The value of len is decided by 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 shows the length of a packet. lenseq[0] shows the length of the first packet. If there are two packets where (sniffer_buf.cnt == 2), lenseq[1] shows the length of the second packet.

- If sniffer_buf.cnt > 1, it is a AMPDU packet. Because headers of each MPDU packets are similar, we only provide the length of each packet (from the header of MAC packet to FCS)
- This structure contains: length of packet, MAC address of both communication sides, length of the packet header.

LEN == 12

- buf contains structure RxControl; but this structure is not reliable. It cannot show the MAC addresses of both communication sides, or the length of the packet header.
- It does not show the number or the length of the sub-packets of AMPDU packets.
- This structure contains: length of the packet, rssi and FEC_CODING.
- RSSI and FEC_CODING are used to judge whether the packets are from the same device.

Summary

It is recommended that users speed up the processing of individual packets, otherwise, some follow-up packets may be lost.

Format of an entire IEEE802.11 packet is shown as below.

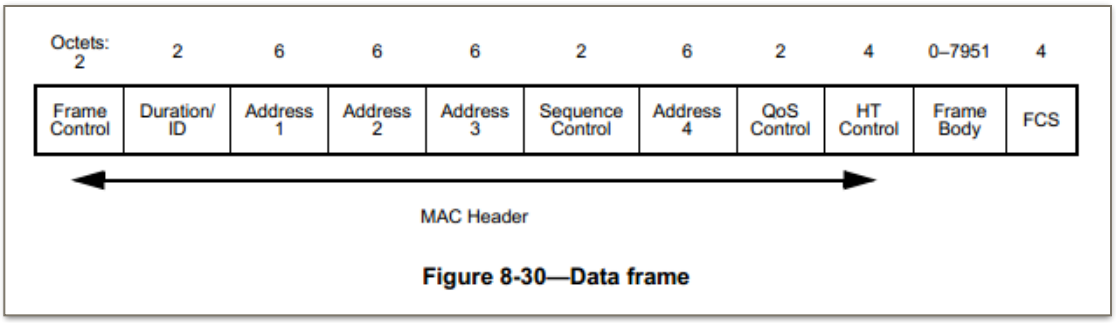

Figure 8-30—Data frame

- The first 24 bytes of MAC header of the data packet are needed:
  - Address 4 field is decided by FromDS and ToDS in Frame Control;
  - QoS Control field is decided by Subtype in Frame Control;
  - HT Control field is decided by Order Field in Frame Control;
  - For more details, refer to *IEEE Std 80211-2012*.
- For WEP encrypted packets, the MAC header is followed by an 4-byte IV, and there is a 4-byte ICV before the FCS.
- For TKIP encrypted packets, the MAC header is followed by a 4-byte IV and a 4-byte EIV, and there are an 8-byte MIC and a 4-byte ICV before the FCS.
- For CCMP encrypted packets, the MAC header is followed by an 8-byte CCMP header, and there is an 8-byte MIC before the FCS.

9.4. ESP8266 soft-AP and station channel configuration

Even though ESP8266 supports the softAP+station mode, it is limited to only one hardware channel.

In the softAP+station mode, the ESP8266 soft-AP will adjust its channel configuration to be same as the ESP8266 station.

This limitation may cause some inconveniences in the softAP+station mode that users need to pay special attention to, for example:

Case 1:

(1) When the user connects the ESP8266 to a router (for example, channel 6),

(2) and sets the ESP8266 soft-AP through wifi_softap_set_config,

(3) If the value is effective, the API will return to true. However, the channel will be automatically adjusted to channel 6 in order to be in line with the ESP8266 station interface. This is because there is only one hardware channel in this mode.

Case 2:

(1) If the user sets the channel of the ESP8266 soft-AP through wifi_softap_set_config (for example, channel 5),

(2) other stations will connect to the ESP8266 soft-AP,

(3) then the user connects the ESP8266 station to a router (for example, channel 6),

(4) the ESP8266 softAP will adjust its channel to be as same as the ESP8266 station (which is channel 6 in this case).

(5) As a result of the change of channel, the station Wi-Fi connected to the ESP8266 soft-AP in step two will be disconnected.

Case 3:

(1) Other stations are connected to the ESP8266 softAP .

(2) If the ESP8266‘s station interface has been scanning or trying to connect to a target router, the ESP8266 softAP-end connection may break.

This is because the ESP8266 station will try to find its target router in different channels, which means it will keep changing channels, and as a result, the ESP8266 channel is changing, too. Therefore, the ESP8266 softAP-end connection may break.

In cases like this, users can set a timer to call wifi_station_disconnect to stop the ESP8266 station from continuously trying to connect to the router. Or use wifi_station_set_reconnect_policy or wifi_station_set_auto_connect to disable the ESP8266 station from reconnecting to the router.

9.5. ESP8266 boot messages

ESP8266 outputs boot messages through UART0 with baud rate 74880:

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


load 0x4010f000, len 1264, room 16

tail 0

chksum 0x42

csum 0x42
```

Messages	Description
rst cause	1: power on
	2: external reset
	4: hardware watchdog-reset
boot mode (first parameter)	1 :ESP8266 is in UART-down mode (download firmware into Flash)
	3 :ESP8266 is in Flash-boot mode (boot up from Flash)
chksum	If chksum == csum, it means that read Flash correctly during booting.

Espressif System

IOT Team

http://bbs.espressif.com