ESP8266 RTOS SDK

Programming Guide

Version 1.0.5

Espressif Systems IOT Team

http://bbs.espressif.com/

Table of Contents

1. **Preambles** 8

2. **Overview**..9

3. **Sample Codes**..10
    - 3.1. Catalog Structure of SDK 10
    - 3.2. Basic Examples 11
        1. Basic example: initialisation..11
        2. Basic example: how to read the ID of the chipset..13
        3. Basic example: connect to AP when ESP8266 functions as station 13
        4. Basic example: ESP8266 functions as soft-AP 14
        5. Basic example: read and set the MAC address of ESP8266 15
        6. Basic example: scan AP nearby..16
        7. Basic example: get RSSI (Received Signal Strength Indicator) of AP 17
    - 3.3. Networking Protocol Example 18
        1. Networking protocol example: UDP transmission..18
        2. Networking protocol example: TCP client 21
        3. Networking protocol example: TCP server 23
    - 3.4. Advanced Application Examples..26
        1. Advanced application example: Firmware upgrade Over-the-air 26

4. **Software APIs** 30
    - 4.1. Software Timer 30
        1. os_timer_arm..30
        2. os_timer_disarm 31
        3. os_timer_setfn 31
        4. os_timer_arm_us 31
    - 4.2. System APIs 32
        1. system_get_sdk_version..32
        2. system_restore 32
        3. system_restart 33
        4. system_get_rst_info 33

5. system_get_chip_id 34
6. system_get_vdd33 34
7. system_adc_read 35
8. system_deep_sleep 35
9. system_deep_sleep_set_option 36
10. system_phy_set_rfoption 36
11. system_phy_set_max_tpw 37
12. system_phy_set_tpw_via_vdd33 37
13. system_print_meminfo 38
14. system_get_free_heap_size 38
15. system_get_time 38
16. system_get_rtc_time 39
17. system_rtc_clock_cali_proc 39
18. system_rtc_mem_write 40
19. system_rtc_mem_read 41
20. system_uart_swap 41
21. system_uart_de_swap 42
22. system_get_boot_version 42
23. system_get_userbin_addr 42
24. system_get_boot_mode 43
25. system_restart_enhance 43
26. system_get_flash_size_map 44
27. os_delay_us 44
28. os_install_putc1 44
29. os_putc 45

4.3. SPI Flash Related APIs 45
1. spi_flash_get_id 45
2. spi_flash_erase_sector 46
3. spi_flash_write 46
4. spi_flash_read 46
5. system_param_save_with_protect 47
6. system_param_load 48

4.4. Wi-Fi Related APIs 49

1. wifi_get_opmode 49
2. wifi_get_opmode_default 49
3. wifi_set_opmode....50
4. wifi_set_opmode_current 50
5. wifi_station_get_config 51
6. wifi_station_get_config_default....51
7. wifi_station_set_config 51
8. wifi_station_set_config_current 53
9. wifi_station_connect 53
10. wifi_station_disconnect 54
11. wifi_station_get_connect_status 54
12. wifi_station_scan....54
13. scan_done_cb_t 55
14. wifi_station_ap_number_set 56
15. wifi_station_get_ap_info 56
16. wifi_station_ap_change 57
17. wifi_station_get_current_ap_id 57
18. wifi_station_get_auto_connect 57
19. wifi_station_set_auto_connect 58
20. wifi_station_dhcpc_start 58
21. wifi_station_dhcpc_stop 59
22. wifi_station_dhcpc_status 59
23. wifi_station_set_reconnect_policy 59
24. wifi_station_get_reconnect_policy 60
25. wifi_softap_get_config 60
26. wifi_softap_get_config_default 61
27. wifi_softap_set_config....61
28. wifi_softap_set_config_current 62
29. wifi_softap_get_station_num 62
30. wifi_softap_get_station_info 62
31. wifi_softap_free_station_info 63
32. wifi_softap_dhcps_start 64
33. wifi_softap_dhcps_stop 64

34. wifi_softap_set_dhcps_lease 64
35. wifi_softap_dhcps_status 66
36. wifi_softap_set_dhcps_offer_option 66
37. wifi_set_phy_mode 67
38. wifi_get_phy_mode 67
39. wifi_get_ip_info 68
40. wifi_set_ip_info 68
41. wifi_set_macaddr 69
42. wifi_get_macaddr 70
43. wifi_status_led_install 70
44. wifi_status_led_uninstall 71
45. wifi_set_event_handler_cb 71
4.5. Upgrade (FOTA) APIs 73
1. system_upgrade_userbin_check 73
2. system_upgrade_flag_set 73
3. system_upgrade_flag_check 73
4. system_upgrade_reboot 74
4.6. Sniffer Related APIs 75
1. wifi_promiscuous_enable 75
2. wifi_promiscuous_set_mac 75
3. wifi_set_promiscuous_rx_cb 76
4. wifi_get_channel 76
5. wifi_set_channel 76
4.7. smart config APIs 77
1. smartconfig_start 77
2. smartconfig_stop 79
4.8. cJSON APIs 80
1. cJSON_Parse 80
2. cJSON_Print 80
3. cJSON_Delete 80
4. cJSON_GetArraySize 80
5. cJSON_GetArrayItem 81
6. cJSON_GetObjectItem 81

7. cJSON_CreateXXX 81
8. cJSON_CreateXXXArray 82
9. cJSON_InitHooks 82
10. cJSON_AddItemToArray 83
11. cJSON_AddItemReferenceToArray 83
12. cJSON_AddItemToObject 83
13. cJSON_AddItemReferenceToObject 84
14. cJSON_DetachItemFromArray 84
15. cJSON_DeleteItemFromArray 85
16. cJSON_DetachItemFromObject 85
17. cJSON_DeleteItemFromObject 85
18. cJSON_InsertItemInArray 86
19. cJSON_ReplaceItemInArray 86
20. cJSON_ReplaceItemInObject 86
21. cJSON_Duplicate 87
22. cJSON_ParseWithOpts 87

5. Definitions & Structures 88

5.1. Timer 88
5.2. WiFi Related Structures 88
1. Station Related 88
2. soft-AP related 88
3. scan related 89
4. WiFi event related structure 89

6. Appendix 93

6.1. RTC APIs Example 93
6.2. Sniffer Structure Introduction 95
6.3. ESP8266 soft-AP and station channel configuration 98

1. Preambles

The ESP8266EX offers a complete and self-contained Wi-Fi network solution. It can be used to host the applications or to offload Wi-Fi network functions from other application processors. When the ESP8266 hosts an application as the only processor in the device, it boots up directly from an external flash. It has an in-built, high-speed cache to improve the performance of the system and reduce the memory occupation. Alternately, when the ESP8266 is used as a Wi-Fi adapter, wireless internet access can be added to any micro controller-based device through the UART interface or the CPU AHB bridge interface, and thus provide the users with a simple Wi-Fi solution.

ESP8266EX enjoys high level of on-chip integration. It integrates the antenna switch, RF balun, power amplifier, low noise receive amplifier, filters, and power management modules. It requires minimal external circuitry, and the entire solution, including the front-end module, is designed to occupy minimal PCB space

The ESP8266EX also integrates an enhanced version of the 32-bit processor of Tensilica's L106 Diamond series, with on-chip SRAM. The ESP8266EX is often integrated with external sensors and other application specific devices through its GPIOs. The SDK files provide examples of the softwares of the related applications.

The ESP8266EX system has many cutting-edge advantages, including energy-efficient VoIP that can switch rapidly between sleep and wake modes, adaptive radio bias for low-power operations, front-end signal processing capacity, problem-shooting capacity, and the radio system co-existence feature to remove cellular, bluetooth, DDR, LVDS and LCD interference.

The SDK based on the ESP8266 IoT platform offers users a simple, high-speed and efficient software platform for IoT device development. This programming guide provides an overview of the SDK as well as details of the APIs. The target readers are embedded software developers who use the ESP8266 IoT platform for software development.

2. Overview

The SDK provides its users with a set of interfaces for data reception and transmission. Users do not need to worry about the set-up of the network, including Wi-Fi and TCP/IP stack. Instead, they can focus on the IoT application development. They can do so by receiving and transmitting data through the interfaces.

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

`void user_init(void)` is the entrance function of the application. It provides users with an initialization interface, and users can add more functions to the interface, including hardware initialization, network parameters setting, and timer initialization.

Notices:

- It is recommended that users set the timer to the periodic mode for periodic checks. In freeRTOS timer or os_timer, do not delay in the manner of `while(1)`.
- Since esp_iot_rtos_sdk_v1.0.4, functions are stored in CACHE by default, need not be added `ICACHE_FLASH_ATTR` any more. The interrupt functions can also be stored in CACHE. If users want to store some frequently called functions in RAM, please add `IRAM_ATTR` before functions' name.
- Network programming use socket, please do not bind to the same port.
- Priority of the RTOS SDK is 15. `xTaskCreate` is an interface of freeRTOS. For details of the freeRTOS and APIs of the system, please visit http://www.freertos.org
    - When using `xTaskCreate` to create a task, the task stack range is [176, 512].
    - If an array whose length is over 60 bytes is used in a task, it is suggested that users use `malloc` and `free` rather than local variable to allocate array. Large local variables could lead to task stack overflow.
    - The RTOS SDK takes some priorities. Priority of the pp task is 13; priority of precise gets timer thread is 12; priority of the lwip task is 10; priority of the ferrets timer is 2; priority of the idle is 0.
    - Users can use tasks with priorities from 1 to 9. Do not revise `FreeRTOSConfig.h`. task priorities are decided by source code inside the RTOS SDK, and therefore, users can't change `FreeRTOSConfig.h`.

3. Sample Codes

3.1. Catalog Structure of SDK

The catalog structure of `esp_iot_rtos_sdk` is illustrated below:

- **app catalog:** programming path of application programs. Users can add codes to this path and start compiling, or they can create and self-define a new subfolder as the programming path, the level of the new folder should be the same with app catalog.
- **bin catalog:** path that firmwares are stored. Firmwares generated by codes in programming app will be stored under this path, too.

Subfolder	Description
bin root directory	• boot and firmware initialization parameters provide by Espressif Systems • if users choose **none boot mode** to compile firmwares generated by application programs, eagle.flash.bin and eagle.irom0text.bin, then firmware upgrade over the air is not supported. stored in this path.
upgrade	• if users choose **with boot mode** to compile firmwares generated by application programs, user1.bin and user2.bin, then firmware upgrade over the air is supported. stored in this path.

- **example catalog:** users can copy the application sample codes provided by Espressif Systems to the app catalog, or to subfolder that falls to the same level of app catalog, and then start compiling. When firmwares are generated, burn them into ESP8266 modules.

Subfolder	Description
IoT_Demo	• sample codes of smart devices provided by Espressif Systems
smart config	• sample codes of smart config function provided by Espressif Systems

- **document catalog:** esp_iot_rtos_sdk files.
- **include catalog:** header files of esp_iot_rtos_sdk, including software interfaces and macro functions for users to use.
- **ld catalog:** link files used when compiling, users don't need to modify them.
- **lib catalog:** library file of esp_iot_rtos_sdk.
- **tool catalog:** tools used when compiling, users don't need to modify them.

3.2. Basic Examples

Some basic examples are listed below:

- initialization
- how to read the ID of the chipset
- how to set the WiFi work mode
  - when ESP8266 works under station mode, it can be connected to the AP (router)
  - when ESP8266 works under soft-AP mode, it can be connected to other stations
- how to read and set the MAC address of the chipset
- how to scan AP nearby
- how to get RSSI (Received Signal Strength Indicator) of AP

1. Basic example: initialisation

(1) Initialisation of application programs can be implemented in `user_main.c`. `void user_init(void)`, which is the EntryPoint Function, can be used by users to implement initialisation process. It is suggested that version information of SDK should be printed, and WiFi work mode should be set.

```
void user_init(void)
{
    printf("SDK version:%s\n", system_get_sdk_version());
    /* station + soft-AP mode */
    wifi_set_opmode(STATIONAP_MODE);
    ……
}
```

(2) esp_iot_rtos_sdk adopts UART0 to print debugging information by default, and the baud rate is 74880 by default. UART initialization can be self-defined by users in `user_init`. Please refer to `uart_init_new` on how to implement this.

Sample of UART driver: \esp_iot_rtos_sdk\examples\driver_lib\driver\uart.c

Take the initialization of UART0 for example. Config parameters of UART:

```
UART_ConfigTypeDef uart_config;
uart_config.baud_rate    = BIT_RATE_74880;
uart_config.data_bits    = UART_WordLength_8b;
uart_config.parity       = USART_Parity_None;
uart_config.stop_bits    = USART_StopBits_1;
uart_config.flow_ctrl    = USART_HardwareFlowControl_None;
uart_config.UART_RxFlowThresh = 120;
```

```
        uart_config.UART_InverseMask  = UART_None_Inverse;
        UART_ParamConfig(UART0, &uart_config);
```

Register UART interrupt function and enable UART interrupt:

```
        UART_IntrConfTypeDef uart_intr;
        uart_intr.UART_IntrEnMask = UART_RXFIFO_TOUT_INT_ENA | UART_FRM_ERR_INT_ENA
        | UART_RXFIFO_FULL_INT_ENA | UART_TXFIFO_EMPTY_INT_ENA;
        uart_intr.UART_RX_FifoFullIntrThresh = 10;
        uart_intr.UART_RX_TimeOutIntrThresh = 2;
        uart_intr.UART_TX_FifoEmptyIntrThresh = 20;
        UART_IntrConfig(UART0, &uart_intr);

        UART_SetPrintPort(UART0);
        UART_intr_handler_register(uart0_rx_intr_handler);
        ETS_UART_INTR_ENABLE();
```

(3) Multi-thread is supported by esp_iot_rtos_sdk, therefore, multi tasks can be created. The interface `xTaskCreate` used to create tasks is self-contained by freeRTOS. When using `xTaskCreate` to create a new task, the range of task stack should be [176, 512].

```
        xTaskCreate(task2, "tsk2", 256, NULL, 2, NULL);
        xTaskCreate(task3, "tsk3", 256, NULL, 2, NULL);
```

Register the task and execute the function. Take the execution of task 2 as an example:

```
    void task2(void *pvParameters)
    {
        printf("Hello, welcome to task2!\r\n");
        while (1) {
            ……
        }
        vTaskDelete(NULL);
    }
```

(4) Compile application program, generate firmware and burn it into ESP8266 module.

(5) Power off the module, and change to operation mode, then power on the module and run the program.

Result:

```
SDK version:1.0.3(601f5cd)
mode : sta(18:fe:34:97:f7:40) + softAP(1a:fe:34:97:f7:40)
add if0
dhcp server start:(ip:192.168.4.1,mask:255.255.255.0,gw:192.168.4.1)
add if1
bcn 100
Hello, welcome to task2!
Hello, welcome to task3!
```

2. Basic example: how to read the ID of the chipset

(1) Introduction of software interface:

`system_get_chip_id` returned value is chip ID of the module. Every chip has one exclusive ID.

```
printf("ESP8266 chip ID:0x%x\n", system_get_chip_id());
```

(2) Compile application program, generate firmware and burn it into ESP8266 module.

(3) Power off the module, and change to operation mode, then power on the module and run the program.

Result:

```
ESP8266 chip ID:0x97f740
```

3. Basic example: connect to AP when ESP8266 functions as station

(1) Set ESP8266 to work under station mode, or coexistence of station+soft-AP mode.

```
wifi_set_opmode(STATION_MODE);
```

(2) Set the SSID and password of the AP.

```
#define DEMO_AP_SSID      "DEMO_AP"
#define DEMO_AP_PASSWORD  "12345678"
```

`wifi_station_set_config` is used to set the AP information when ESP8266 functions as station. Please be noted that the initialised value of bssid_set in station_config should be 0, unless the MAC of AP must be specified.

`wifi_station_connect` set the connection of AP.

```
struct station_config * config = (struct station_config *)zalloc(sizeof(struct 
    station_config));
    sprintf(config->ssid, DEMO_AP_SSID);
    sprintf(config->password, DEMO_AP_PASSWORD);

    wifi_station_set_config(config);
    free(config);
wifi_station_connect();
```

(3) Compile application program, generate firmware and burn it into ESP8266 module.

(4) Power off the module, and change to operation mode, then power on the module and run the program.

Result:

```
connected with DEMO_AP, channel 11
dhcp client start...
ip:192.168.1.103,mask:255.255.255.0,gw:192.168.1.1
```

4. Basic example: ESP8266 functions as soft-AP

(1) Set ESP8266 to work under soft-AP mode, or coexistence of station+soft-AP mode.

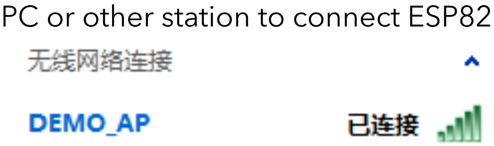
```
wifi_set_opmode(SOFTAP_MODE);
```

(2) Config when ESP8266 functions as soft-AP

```
#define DEMO_AP_SSID      "DEMO_AP"
#define DEMO_AP_PASSWORD  "12345678"
struct softap_config *config = (struct softap_config *)zalloc(sizeof(struct softap_config));

wifi_softap_get_config(config); // Get soft-AP config first.

sprintf(config->ssid, DEMO_AP_SSID);
sprintf(config->password, DEMO_AP_PASSWORD);

config->authmode = AUTH_WPA_WPA2_PSK;
config->ssid_len = 0;        // or its actual SSID length
config->max_connection = 4;

wifi_softap_set_config(config); // Set ESP8266 soft-AP config
free(config);
```

(3) Compile application program, generate firmware and burn it into ESP8266 module.

(4) Power off the module, and change to operation mode, then power on the module and run the program. Please use PC or other station to connect ESP8266 soft-AP.



Result:

When ESP8266 functions as soft-AP, the following information will be printed when station is connected to:

```
station: c8:3a:35:cc:14:94 join, AID = 1
```

5. Basic example: read and set the MAC address of ESP8266

(1) ESP8266 can work under station+soft-AP coexistence mode. The MAC addresses of station and soft-AP interfaces are different. It is guaranteed that the MAC address of every chipset is unique and exclusive. If users want to reset the MAC address, the uniqueness of the MAC should be assured.

(2) Set ESP8266 as station+soft-AP coexistence mode.

```
wifi_set_opmode(STATIONAP_MODE);
```

(3) Read the MAC addresses of station and soft-AP interfaces respectively.

```
wifi_get_macaddr(SOFTAP_IF, sofap_mac);
wifi_get_macaddr(STATION_IF, sta_mac);
```

(4) Set the MAC addresses of station and soft-AP interfaces respectively. The setting of MAC address is not stored in the flash, and the setting can be operated only when the interface is enabled first.

```
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);
```

(5) Compile application program, generate firmware and burn it into ESP8266 module.

(6) Power off the module, and change to operation mode, then power on the module and run the program.

Result:

```
ESP8266 station MAC :18:fe:34:97:f7:40
ESP8266 soft-AP MAC :1a:fe:34:97:f7:40
ESP8266 station new MAC :12:34:56:78:90:ab
ESP8266 soft-AP new MAC :16:34:56:78:90:ab
```

6. Basic example: scan AP nearby

(1) Set ESP8266 to work under soft-AP mode, or coexistence of station+soft-AP mode.

```
wifi_set_opmode(STATIONAP_MODE);
```

(2) Scan AP nearby

If the first parameter of `wifi_station_scan` is NULL, then all AP around will be scanned; if certain information including SSID and channel is defined in the first parameter, then that specific AP will be returned.

```
wifi_station_scan(NULL,scan_done);
```

Callback function when AP scanning is completed

```
void scan_done(void *arg, STATUS status)
{
  uint8 ssid[33];
  char temp[128];

  if (status == OK){
    struct bss_info *bss_link = (struct bss_info *)arg;
    bss_link = bss_link->next.stqe_next;//ignore the first one , it's invalid.

    while (bss_link != NULL)
    {
      memset(ssid, 0, 33);
      if (strlen(bss_link->ssid) <= 32)
        memcpy(ssid, bss_link->ssid, strlen(bss_link->ssid));
      else
        memcpy(ssid, bss_link->ssid, 32);

      printf("(%d,\"%s\",%d,\""MACSTR"\",%d)\r\n",
                 bss_link->authmode, ssid, bss_link->rssi,
                 MAC2STR(bss_link->bssid),bss_link->channel);
      bss_link = bss_link->next.stqe_next;
    }
  }
  else{
```

```
            printf("scan fail !!!\r\n");
        }
    }
```

(3) Compile application program, generate firmware and burn it into ESP8266 module.

(4) Power off the module, and change to operation mode, then power on the module and run the program.

Result:

```
mode : sta(18:fe:34:97:f7:40) + softAP(1a:fe:34:97:f7:40)
add if0
dhcp server start:(ip:192.168.4.1,mask:255.255.255.0,gw:192.168.4.1)
add if1
bcn 100
Hello, welcome to scan-task!
scandone
(0,"ESP_A13319",-41,"1a:fe:34:a1:33:19",1)
(4,"sscgov217",-75,"80:89:17:79:63:cc",1)
(0,"ESP_97F0B1",-46,"1a:fe:34:97:f0:b1",1)
(0,"ESP_A1327E",-36,"1a:fe:34:a1:32:7e",1)
```

7. Basic example: get RSSI (Received Signal Strength Indicator) of AP

(1) If ESP8266 (functions as station) is not connected to AP, users can obtain RSSI (Received Signal Strength Indicator) of AP by scanning AP with a specified SSID.

Specified SSID of target AP:

```
#define DEMO_AP_SSID     “DEMO_AP"
```

Scan AP with a specified SSID, after the scan is completed, `scan_done` will be called back.

```
    struct scan_config config;

    memset(&config, 0, sizeof(config));
    config.ssid = DEMO_AP_SSID;

    wifi_station_scan(&config,scan_done);
```

(2) Compile application program, generate firmware and burn it into ESP8266 module.

(3) Power off the module, and change to operation mode, then power on the module and run the program.

Result:

```
Hello, welcome to scan-task!
scandone
(3,"DEMO_AP",-49,"aa:5b:78:30:46:0a",11)
```

3.3. Networking Protocol Example

The networking protocol of `esp_iot_rtos_sdk` is programming of socket, including the following examples:

- Example of UDP transmission
- Example of TCP connection
  - ESP8266 functions as TCP client
  - ESP8266 functions as TCP server

1. Networking protocol example: UDP transmission

(1) Set the local port number of UDP. Below is an example when the port number is 1200.

```
#define UDP_LOCAL_PORT  1200
```

(2) Create socket.

```
LOCAL int32 sock_fd;
struct sockaddr_in server_addr;

memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = INADDR_ANY;
server_addr.sin_port = htons(UDP_LOCAL_PORT);
server_addr.sin_len = sizeof(server_addr);

do{
    sock_fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock_fd == -1) {
        printf("ESP8266 UDP task > failed to create sock!\n");
        vTaskDelay(1000/portTICK_RATE_MS);
```

```
        }
    }while(sock_fd == -1);

    printf("ESP8266 UDP task > socket OK!\n");
```

(3) Bind a local port

```
    do{
         ret = bind(sock_fd, (struct sockaddr *)&server_addr, sizeof(server_addr));
         if (ret != 0) {
               printf("ESP8266 UDP task > captdns_task failed to bind sock!\n");
               vTaskDelay(1000/portTICK_RATE_MS);
          }
        }while(ret != 0);

        printf("ESP8266 UDP task > bind OK!\n");
```

(4) Receiving and transmission of UDP data

```
    while(1){
         memset(udp_msg, 0, UDP_DATA_LEN);
         memset(&from, 0, sizeof(from));

         setsockopt(sock_fd, SOL_SOCKET, SO_RCVTIMEO, (char *)&nNetTimeout,
       sizeof(int));
         fromlen = sizeof(struct sockaddr_in);
         ret = recvfrom(sock_fd, (uint8 *)udp_msg, UDP_DATA_LEN, 0,(struct sockaddr
       *)&from,(socklen_t *)&fromlen);
         if (ret > 0) {
             printf("ESP8266 UDP task > recv %d Bytes from Port %d %s\n",ret,
       ntohs(from.sin_port), inet_ntoa(from.sin_addr));

             sendto(sock_fd,(uint8*)udp_msg, ret, 0, (struct sockaddr *)&from,
       fromlen);
            }
    }

    if(udp_msg){
         free(udp_msg);
         udp_msg = NULL;
    }
    close(sock_fd);
```

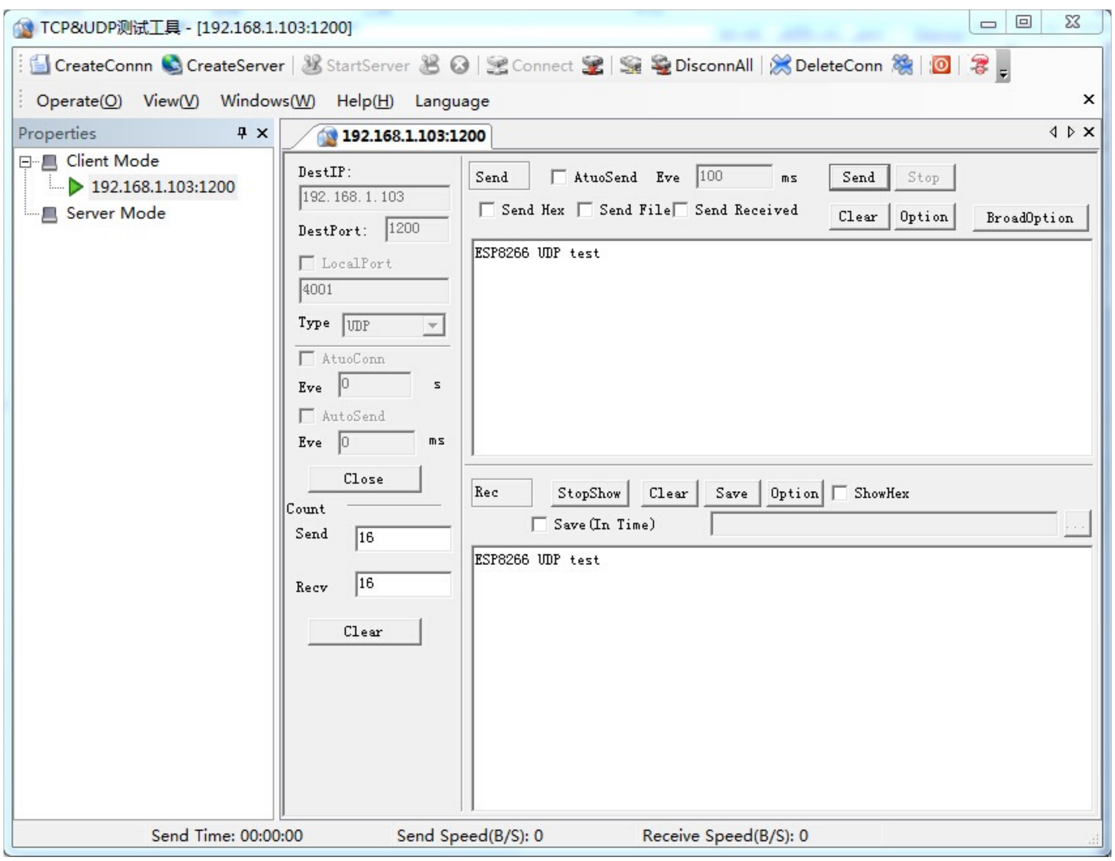
(5) Compile application program, generate firmware and burn it into ESP8266 module.

(6) Power off the module, and change to operation mode, then power on the module and run the program.

Result:

```
ip:192.168.1.103,mask:255.255.255.0,gw:192.168.1.1
got ip !!!
ESP8266 UDP task > socket OK!
ESP8266 UDP task > bind OK!
ESP8266 UDP task > recv 16 Bytes from Port 57233 192.168.1.112
```

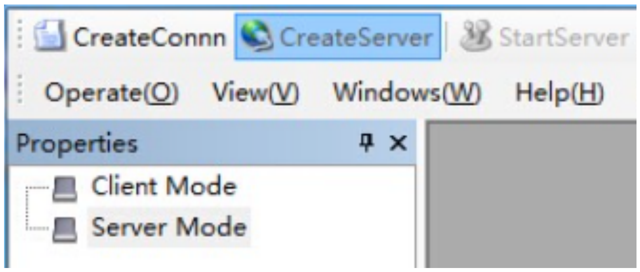
UDP communication can be set up at the PC terminal by using network debugging tools, then "ESP8266 UDP test" will be sent to ESP8266 UDP port. When the UDP data is received by ESP8266, the same message will be sent to the PC terminal, too.



2. Networking protocol example: TCP client

(1) Connect ESP8266 (when it is functions as station) to AP. Users can refer to previous examples.

(2) Establish a TCP server using network debugging tools.




```
#define SERVER_IP          "192.168.1.124"
#define SERVER_PORT        1001
```

(3) Implement TCP communication via programming of the socket.

Create socket:

```
sta_socket = socket(PF_INET, SOCK_STREAM, 0);
if (-1 == sta_socket) {
     close(sta_socket);
     vTaskDelay(1000 / portTICK_RATE_MS);
     printf("ESP8266 TCP client task > socket fail!\n");
     continue;
}
printf("ESP8266 TCP client task > socket ok!\n");
```

Create TCP connection:

```
bzero(&remote_ip, sizeof(struct sockaddr_in));
remote_ip.sin_family = AF_INET;
remote_ip.sin_addr.s_addr = inet_addr(SERVER_IP);
remote_ip.sin_port = htons(SERVER_PORT);

if (0 != connect(sta_socket, (struct sockaddr *)(&remote_ip), sizeof(struct
sockaddr))) {
    close(sta_socket);
    vTaskDelay(1000 / portTICK_RATE_MS);
    printf("ESP8266 TCP client task > connect fail!\n");
    continue;
}
```

```
        printf("ESP8266 TCP client task > connect ok!\n");
```

TCP communication, sending data packets:

```
        if (write(sta_socket, pbuf, strlen(pbuf) + 1) < 0){
            close(sta_socket);
            vTaskDelay(1000 / portTICK_RATE_MS);
            printf("ESP8266 TCP client task > send fail\n");
            continue;
        }
        printf("ESP8266 TCP client task > send success\n");
        free(pbuf);
```

TCP communication, receiving packets:

```
        char *recv_buf = (char *)zalloc(128);
        while ((recbytes = read(sta_socket , recv_buf, 128)) > 0) {
             recv_buf[recbytes] = 0;
             printf("ESP8266 TCP client task > recv data %d bytes!\nESP8266 TCP 
        client task > %s\n", recbytes, recv_buf);
        }
        free(recv_buf);

        if (recbytes <= 0) {
              close(sta_socket);
              printf("ESP8266 TCP client task > read data fail!\n");
        }
```

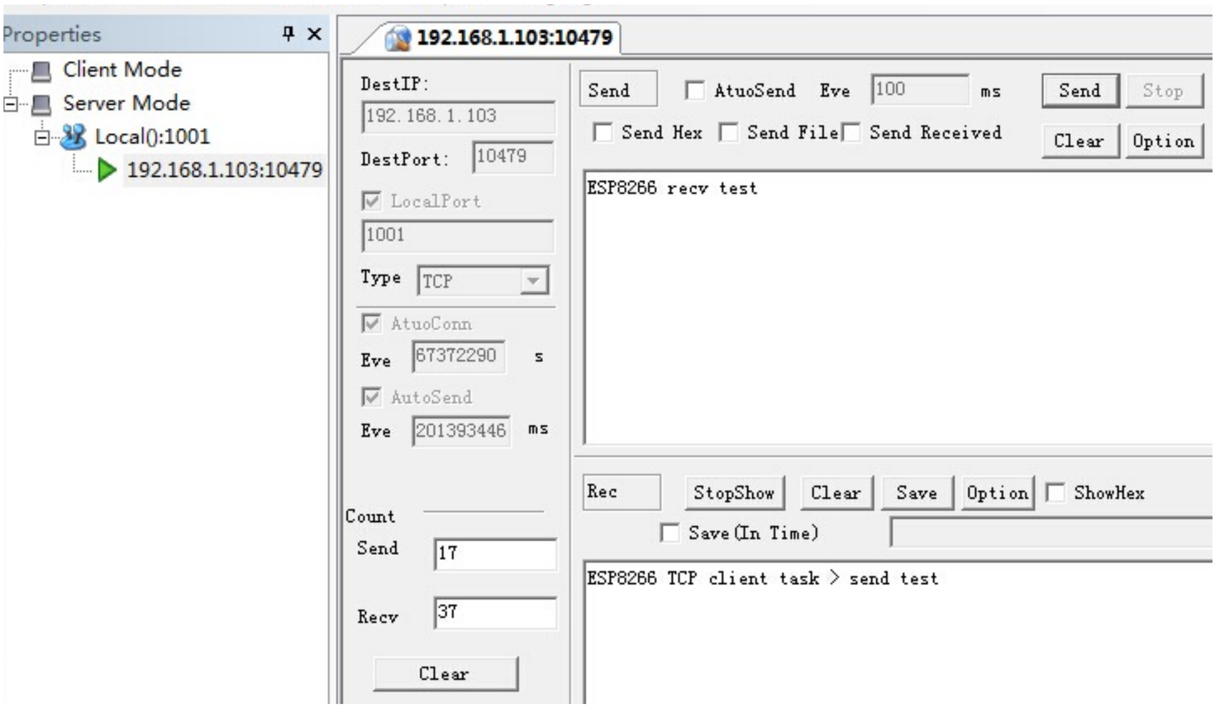
(4) Compile application program, generate firmware and burn it into ESP8266 module.

(5) Power off the module, and change to operation mode, then power on the module and run the program.

Result:

```
ESP8266 TCP client task > socket ok!
ESP8266 TCP client task > connect ok!
ESP8266 TCP client task > send success
ESP8266 TCP client task > recv data 17 bytes!
ESP8266 TCP client task > ESP8266 recv test
```

The picture below shows when TCP server established at the terminal of network debugging tool communicates with ESP8266 successfully.



3. Networking protocol example: TCP server

(1) Establish TCP server, bind local port.

```
#define SERVER_PORT 1002
int32 listenfd;
int32 ret;
struct sockaddr_in server_addr,remote_addr;
int stack_counter=0;

/* Construct local address structure */
memset(&server_addr, 0, sizeof(server_addr)); /* Zero out structure */
server_addr.sin_family = AF_INET;            /* Internet address family */
server_addr.sin_addr.s_addr = INADDR_ANY;   /* Any incoming interface */
server_addr.sin_len = sizeof(server_addr);
server_addr.sin_port = htons(httpd_server_port); /* Local port */

/* Create socket for incoming connections */
do{
    listenfd = socket(AF_INET, SOCK_STREAM, 0);
```

```
        if (listenfd == -1) {
            printf("ESP8266 TCP server task > socket error\n”);
            vTaskDelay(1000/portTICK_RATE_MS);
        }
    }while(listenfd == -1);

    printf("ESP8266 TCP server task > create socket: %d\n", server_sock);

    /* Bind to the local port */
    do{
        ret = bind(listenfd, (struct sockaddr *)&server_addr,
    sizeof(server_addr));
        if (ret != 0) {
             printf("ESP8266 TCP server task > bind fail\n”);
             vTaskDelay(1000/portTICK_RATE_MS);
        }
    }while(ret != 0);

    printf("ESP8266 TCP server task > port:%d\n”,ntohs(server_addr.sin_port));
```

Establish TCP server interception:

```
    do{
        /* Listen to the local connection */
        ret = listen(listenfd, MAX_CONN);
        if (ret != 0) {
            printf("ESP8266 TCP server task > failed to set listen queue!\n");
            vTaskDelay(1000/portTICK_RATE_MS);
        }
    }while(ret != 0);

    printf("ESP8266 TCP server task > listen ok\n”);
```

Wait until TCP client is connected with the server, then start receiving data packets when TCP communication is established:

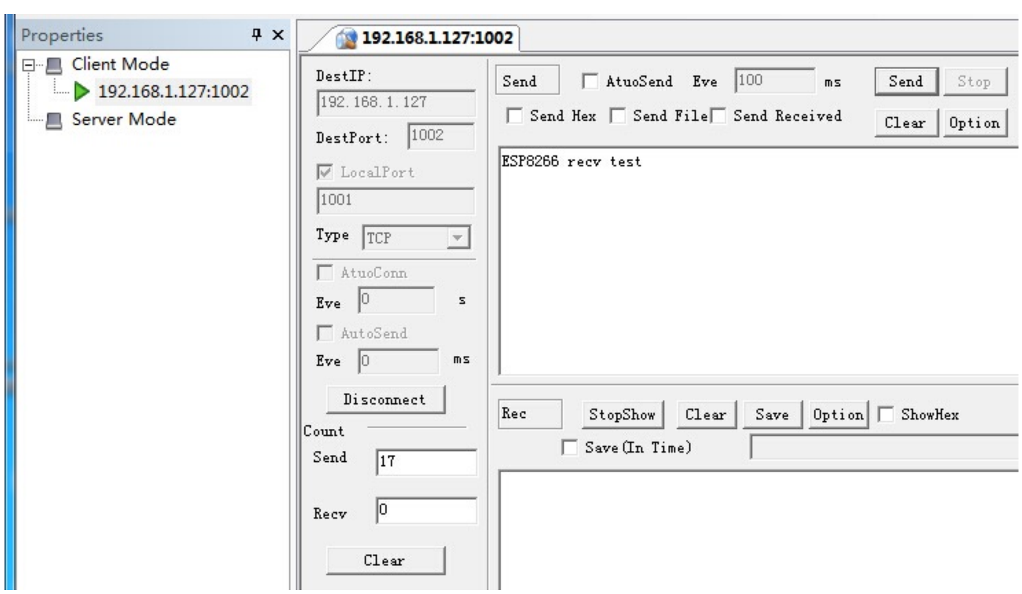
```
    int32 client_sock;
    int32 len = sizeof(struct sockaddr_in);

    for (;;) {
       printf("ESP8266 TCP server task > wait client\n”);
       /*block here waiting remote connect request*/
```

```
    if ((client_sock = accept(listenfd, (struct sockaddr *)&remote_addr,
(socklen_t *)&len)) < 0) {
        printf("ESP8266 TCP server task > accept fail\n");
        continue;
    }
    printf("ESP8266 TCP server task > Client from %s %d\n",
inet_ntoa(remote_addr.sin_addr), htons(remote_addr.sin_port));

    char *recv_buf = (char *)zalloc(128);
    while ((recbytes = read(client_sock , recv_buf, 128)) > 0) {
      recv_buf[recbytes] = 0;
      printf("ESP8266 TCP server task > read data success %d!\nESP8266 TCP
server task > %s\n", recbytes, recv_buf);
    }
    free(recv_buf);

    if (recbytes <= 0) {
      printf("ESP8266 TCP server task > read data fail!\n");
      close(client_sock);
    }
}
```

(2) Compile application program, generate firmware and burn it into ESP8266 module.

(3) Power off the module, and change to operation mode, then power on the module and run the program.

(4) Establish a TCP client via network debugging tool, then connect the TCP client with ESP8266 TCP server, and start sending data.



Result:

```
ip:192.168.1.127,mask:255.255.255.0,gw:192.168.1.1
got ip !!!
Hello, welcome to ESP8266 TCP server task!
ESP8266 TCP server task > create socket: 0
ESP8266 TCP server task > bind port: 1002
ESP8266 TCP server task > listen ok
ESP8266 TCP server task > wait client
ESP8266 TCP server task > Client from 192.168.1.108 1001
ESP8266 TCP server task > read data success 17!
ESP8266 TCP server task > ESP8266 recv test
```

3.4. Advanced Application Examples

Advanced examples included in `esp_iot_rtos_sdk` are listed below:

- Firmware upgrade Over-the-air (OTA)

1. Advanced application example: Firmware upgrade Over-the-air

Firmware upgrade OTA refers to downloading new software upgrade from the server via WiFi networking and realise firmware upgrade.

(1) Users can establish their own cloud server, or they can adopt cloud server provided by Espressif.

(2) Upload the new firmware to the cloud server.

(3) Descriptions of the codes are explained below:

Connect ESP8266 module to AP (users can refer to previous examples), then check if ESP8266 station can get the IP address through function `upgrade_task`.

```
wifi_get_ip_info(STATION_IF, &ipconfig);

/* check the IP address or net connection state*/
while (ipconfig.ip.addr == 0) {
      vTaskDelay(1000 / portTICK_RATE_MS);
      wifi_get_ip_info(STATION_IF, &ipconfig);
}
```

When IP address is obtained by ESP8266, the module will be connected with cloud server. (Users can refer to previous socket programming).

`system_upgrade_flag_set` set a flag to indicate the upgrade status:

- `UPGRADE_FLAG_IDLE` : idle.
- `UPGRADE_FLAG_START` : start upgrade.

- `UPGRADE_FLAG_FINISH` : finish downloading new firmware from the cloud server.

`system_upgrade_userbin_check` : check the user bin file that the system is running. If the system is running user1.bin, then user2.bin will be downloaded; if the systems is running user2.bin, then user1.bin will be downloaded.

```
system_upgrade_init();
system_upgrade_flag_set(UPGRADE_FLAG_START);
```

Send downloading request to the server. After the upgraded firmware data is received successfully, burn it into the flash.

```
if(write(sta_socket,server->url,strlen(server->url)+1) < 0) {
        ……
}

while((recbytes = read(sta_socket ,precv_buf,UPGRADE_DATA_SEG_LEN)) > 0) {
      // write the new firmware into flash by spi_flash_write
}
```

Set a software timer to check the upgrade status of the firmware periodically.

If the timer indicates time-out, and the firmware has not been updated from the cloud server, then upgrade is failed. The status of firmware upgrade will turn back to idle and quit.

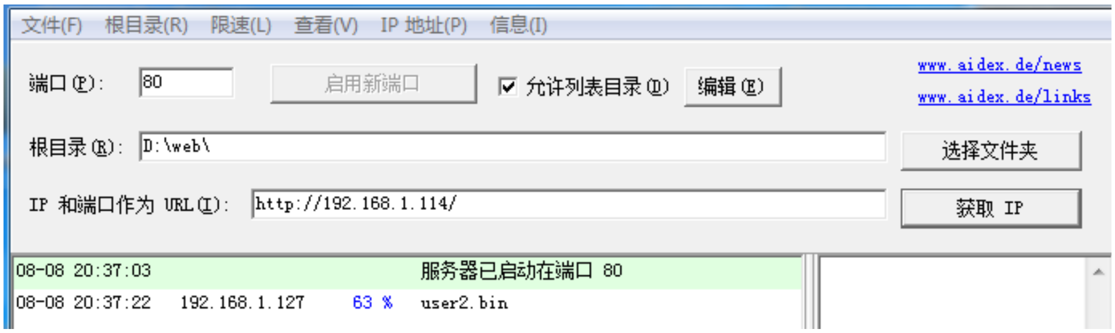
If firmware has been successfully downloaded from the server, upgrade status will be shown as `UPGRADE_FLAG_FINISH`. Call function `system_upgrade_reboot`, reboot ESP8266, and start running the newly updated firmware.

(4) Compile application program, generate firmware and burn it into ESP8266 module.

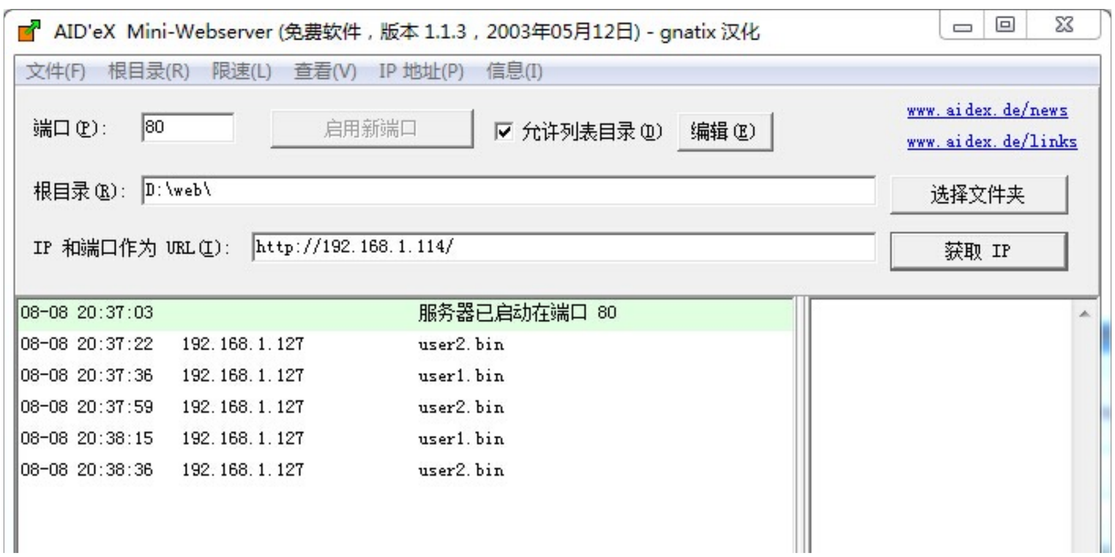
(5) Power off the module, and change to operation mode, then power on the module and run the program.

Result:

Establish a server at the PC terminal via webserver, then upload user1.bin and user2.bin to the server. After the firmware has been burnt into ESP8266, user1.bin will start running first by default, then user2.bin will be downloaded from the server.



The module will reboot when user2.bin has been downloaded, and start running user2.bin. Then user1.bin will be downloaded from the server. This cycle revolves.



Below is a picture showing the print information during ESP8266 upgrading process:

```
ip:192.168.1.127,mask:255.255.255.0,gw:192.168.1.1
Hello, welcome to client!
socket ok!
connect ok!
GET /user2.bin HTTP/1.0
Host: "192.168.1.114":80
Connection: keep-alive
Cache-Control: no-cache
User-Agent: Mozilla/5.0 (Windows NT 5.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/3
0.0.1599.101 Safari/537.36
Accept: */*
Accept-Encoding: gzip,deflate,sdch
Accept-Language: zh-CN,zh;q=0.8

send success
read data success!
upgrade file download start.
read data success!
totallen = 1460
read data success!
totallen = 2920
read data success!
```

4. Software APIs

4.1. Software Timer

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

Notice: Timers of the following interfaces are software timers. Functions of the timers are executed during the tasks. Since a task can be stopped, or be delayed because there are other tasks with higher priorities, the following os_timer interfaces cannot guarantee the precise execution of the timers.

- 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 the millisecond timer.

Functional definition:
    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, the maximum value allowed
    is 0x41893

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

Return:
    null
```

2. os_timer_disarm

Function:

Disarm the timer

Functional definition:

```
void os_timer_disarm (os_timer_t *ptimer)
```

Parameters:

`os_timer_t *ptimer` : Timer structure

Return:

null

3. os_timer_setfn

Function:

Set the timer callback function.

Notices:

- The callback function must be set in order to enable the timer.
- Operating system scheduling is disabled in timer callback.

Functional definition:

```
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
`void *parg` : callback function parameter

Return:

null

4. os_timer_arm_us

Function:

Enable the microsecond timer.

Functional definition:

```
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 allowed is 0x64, the maximum value allowed is 0xFFFFFFF

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

Return:

null

4.2. System APIs

1. system_get_sdk_version

Function:

Get information of the SDK version.

Functional definition:

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

Parameter:

none

Return:

Information of the SDK version

Example:

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

2. system_restore

Function:

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

Functional definition:

```
void system_restore(void)
```

Parameters:

null

Return:

null

3. system_restart

Function:

Restart the system.

Functional definition:

```
void system_restart(void)
```

Parameters:

null

Return:

null

4. system_get_rst_info

Function:

Get the reason of restart.

Structure:

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

struct rst_info {
    uint32 reason;    // enum rst_reason
    uint32 exccause;
    uint32 epc1;
    uint32 epc2;
    uint32 epc3;
```

```
        uint32 excvaddr;
        uint32 depc;
    };
```

Prototype:

`struct rst_info* system_get_rst_info(void)`

Parameter:

none

Return:

Reason of restart.

5. system_get_chip_id

Function:

Get the chip ID

Functional definition:

`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

Notices:

- `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

Functional definition:

`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

Notices:

- system_adc_read can only be called when the TOUT pin is connected to the external circuitry, and the TOUT pin input voltage should be limited to 0 ~1.0V.
- When the TOUT pin is connected to the external circuitry, the 107th byte (vdd33_const) of esp_init_data_default.bin(0~127byte) should be set as the real power voltage of VDD3P3 pin 3 and 4.
- The unit of vdd33_const is 0.1V, the effective value range is [18, 36]; if vdd33_const is in [0, 18) or (36, 255), 3.3V is used to optimize RF by default.

Functional definition:

```
uint16 system_adc_read(void)
```

Parameter:

none

Return:

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

8. system_deep_sleep

Function:

Set the chip to deep-sleep mode. The device will automatically wake up after the deep-sleep time set by the users. Upon waking up, the device boots up from user_init.

Notices:

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

Functional definition:

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

Parameters:

uint32 time_in_us : deep-sleep time, unit: microsecond

Return:

null

9. system_deep_sleep_set_option

Function:

Call this API before system_deep_sleep to set the activity after the next deep-sleep wakeup. If this API is not called, default to be system_deep_sleep_set_option(1).

Functional definition:

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

Parameter:

uint8 option :

0 : Radio calibration after the deep-sleep wakeup is decided by byte 108 of esp_init_data_default.bin (0~127byte).

1 : Radio calibration will be done after the deep-sleep wakeup. This will lead to stronger current.

2 : Radio calibration will not be done after the deep-sleep wakeup. This will lead to weaker current.

4 : Disable radio calibration after the deep-sleep wakeup (the same as modem-sleep). This will lead to the weakest current, but the device can't receive or transmit data after waking up.

Return:

true : succeed

false : fail

10. system_phy_set_rfoption

Function:

Enable RF or not when wakeup from deep-sleep.

Notices:

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

Functional definition:

`void system_phy_set_rfoption(uint8 option)`

Parameter:

`uint8 option` :

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

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

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

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

Return:

null

11. system_phy_set_max_tpw

Function:

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

Functional definition:

`void system_phy_set_max_tpw(uint8 max_tpw)`

Parameter:

`uint8 max_tpw` : the 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:

null

12. system_phy_set_tpw_via_vdd33

Function:

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

Notices:

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

When TOUT pin is connected to the external circuitry, system_get_vdd33 can not be used to measure VDD33.

Functional definition:

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

Parameter:

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

Return:

null

13. system_print_meminfo

Function:

Print the system memory distribution, including data/rodata/bss/heap.

Functional definition:

```
void system_print_meminfo (void)
```

Parameters:

null

Return:

null

14. system_get_free_heap_size

Function:

Get the size of available heap.

Functional definition:

```
uint32 system_get_free_heap_size(void)
```

Parameters:

null

Return:

uint32 : available heap size

15. system_get_time

Function:

Get system time, unit: microsecond.

Functional definition:

```
uint32 system_get_time(void)
```

Parameter:

null

Return:

System time, unit: microsecond.

16. system_get_rtc_time

Function:

Get RTC time, unit: RTC clock cycle.

Example:

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

Notices:

System time will return to zero because of system_restart, but the RTC time still goes on. If the chip is reset by pin EXT_RST or pin CHIP_EN (including the deep-sleep wakeup), situations are shown as below:

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

Functional definition:

```
uint32 system_get_rtc_time(void)
```

Parameter:

null

Return:

RTC time

17. system_rtc_clock_cali_proc

Function:

Get the RTC clock cycle.

Notices:

The RTC clock cycle has decimal part.

The RTC clock cycle will change according to the temperature, so RTC timer is not very precise.

Functional definition:

```
uint32 system_rtc_clock_cali_proc(void)
```

Parameter:

null

Return:

RTC clock period (unit: microsecond), bit11~ bit0 are decimal. ((RTC_CAL * 100)>> 12)

Notice:

see RTC demo in Appendix.

18. system_rtc_mem_write

Function:

During deep-sleep, only RTC is working. So users can store their data in RTC memory if it is needed. The user data segment below (512 bytes) is used to store the user data.

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

Notices:

Read and write unit for data stored in the RTC memory is 4 bytes. des_addr is the block number (4 bytes per block). So when storing data at the beginning of the user data segment, des_addr will be 256/4 = 64, save_size will be data length.

Functional definition:

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

Parameters:

uint32 des_addr : destination address (block number) in RTC memory, des_addr >=64

void * src_addr : data pointer.

uint32 save_size : data length (unit: byte)

Return:

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

19. system_rtc_mem_read

Function:

Read user data from the RTC memory.The user data segment (512 bytes, as shown below) is used to store user data.

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

Notice:

RRead and write unit for data stored in the RTC memory is 4 bytes. src_addr is the block number (4 bytes per block). So when storing data at the beginning of the user data segment, src_addr will be 256/4 = 64, save_size will be data length.

Functional definition:

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

Parameters:

```
uint32 src_addr  :  source address of rtc memory, src_addr >= 64
void * des_addr  :  data pointer
uint32 save_size :  data length, unit: byte
```

Return:

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

20. system_uart_swap

Function:

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

Functional definition:

```
void system_uart_swap (void)
```

Parameter:

null

Return:

null

21. system_uart_de_swap

Function:

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

Functional definition:

```
void system_uart_de_swap (void)
```

Parameter:

null

Return:

null

22. system_get_boot_version

Function:

Get information of the boot version.

Functional definition:

```
uint8 system_get_boot_version (void)
```

Parameter:

null

Return:

Information of the boot version.

Notice:

If boot version >= 3 , users can enable the enhanced boot mode (refer to system_restart_enhance)

23. system_get_userbin_addr

Function:

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

Functional definition:

```
uint32 system_get_userbin_addr (void)
```

Parameter:

null

Return:

The address of the current running user bin.

24. system_get_boot_mode

Function:

Get the boot mode.

Functional definition:

```
uint8 system_get_boot_mode (void)
```

Parameter:

null

Return:

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

Notices:

Enhanced boot mode: It can load and run FW at any address;
Regular boot mode: It can only load and run at some addresses of user1.bin (or user2.bin).

25. system_restart_enhance

Function:

Restarts the system, and enters the enhanced boot mode.

Functional definition:

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

Parameters:

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 : starting address of the bin file

Return:

true: succeed
false: Fail

Notice:

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

26. system_get_flash_size_map

Function:

Get the current flash size and flash map.

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

Structure:

```
enum flash_size_map {
    FLASH_SIZE_4M_MAP_256_256 = 0,
    FLASH_SIZE_2M,
    FLASH_SIZE_8M_MAP_512_512,
    FLASH_SIZE_16M_MAP_512_512,
    FLASH_SIZE_32M_MAP_512_512,
    FLASH_SIZE_16M_MAP_1024_1024,
    FLASH_SIZE_32M_MAP_1024_1024
};
```

Functional definition:

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

Parameter:

null

Return:

flash map

27. os_delay_us

Function:

delay function, maximum value: 65535 us

Functional definition:

```
void os_delay_us(uint16 us)
```

Parameter:

uint16 us - delay time

Return:

null

28. os_install_putc1

Function:

Register the print output function.

Functional definition:

```
void os_install_putc1(void(*p)(char c))
```

Parameter:

`void(*p)(char c)` - pointer of print function

Return:

null

Example:

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

29. os_putc

Function:

Print a character. Start from from UART0 by default.

Functional definition:

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

Parameter:

`char c` - character to be printed.

Return:

null

4.3. SPI Flash Related APIs

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

1. spi_flash_get_id

Function:

Get ID info of spi flash

Functional definition:

```
uint32 spi_flash_get_id (void)
```

Parameter:

null

Return:

SPI flash ID

2. spi_flash_erase_sector

Function:

Erase the flash sector.

Functional definition:

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

Parameter:

`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.

Functional definition:

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

Parameters:

`uint32 des_addr` : destination address in flash.

`uint32 *src_addr` : source address of the data.

`uint32 size` :length of data

Return:

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

4. spi_flash_read

Function:

Read data from flash.

Functional definition:

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

Parameters:

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

Return:

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

Example:

```
uint32 value;

uint8 *addr = (uint8 *)&value;

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

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

5. system_param_save_with_protect

Function:

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

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

Notice:

For more details of protection of flash read/write, refer to *99A-SDK-Espressif IOT Flash RW Operation* at http://bbs.espressif.com/viewtopic.php?f=21&t=413

Functional definition:

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

Parameters:

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

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

void *param : pointer of the data to be written

uint16 len : data length, should be less than a sector, which is 4 * 1024

Return:

true, succeed;

false, fail

6. system_param_load

Function:

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

Read/write protection of flash: use 3 sectors (4KB 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.

Notice:

For more details of the read/write protection of flash, refer to *99A-SDK-Espressif IOT Flash RW Operation* at http://bbs.espressif.com/viewtopic.php?f=21&t=413

Functional definition:

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

```
Parameters:
    uint16 start_sec  : start sector (sector 0) of the 3 sectors used for flash
    read/write protection. It cannot be sector 1 or sector 2.

    For example, in IOT_Demo, the 3 sectors (3 * 4KB) starting from flash
    0x3D000 can be used for flash read/write protection. The parameter start_sec
    is 0x3D, and it cannot be 0x3E or 0x3F.

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

Return:
    true, succeed;

    false, fail
```

4.4. Wi-Fi Related APIs

wifi_station APIs and other APIs which sets/gets configuration of the ESP8266 station can only be called if the ESP8266 station is enabled.

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

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

1. wifi_get_opmode

```
Function:
    get the current operating mode of the Wifi

Functional definition:
    uint8 wifi_get_opmode (void)

Parameter:
    null

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

2. wifi_get_opmode_default

```
Function:
    Get the operating mode of the WiFi saved in the flash.
```

Functional definition:

```
uint8 wifi_get_opmode_default (void)
```

Parameter:

null

Return:

WiFi operating modes:

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

3. wifi_set_opmode

Function:

Set the WiFi operating mode as station, soft-AP or station+soft-AP, and save it to flash. The default mode is soft-AP mode.

Notices:

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

Functional definition:

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

Parameters:

uint8 opmode: WiFi operating modes:

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

Return:

true: succeed
false: fail

4. wifi_set_opmode_current

Function:

Set the WiFi operating mode as station, soft-AP or station+soft-AP, and the mode won't be saved to the flash.

Functional definition:

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

```
Parameters:
    uint8 opmode: WiFi operating modes:
        0x01: station mode
        0x02: soft-AP mode
        0x03: station+soft-AP mode
Return:
    true:  succeed
    false: fail
```

5. wifi_station_get_config

```
Function:
    Get the configuration parameters of the current WiFi station.
Functional definition:
    bool wifi_station_get_config (struct station_config *config)
Parameter:
    struct station_config *config : WiFi station configuration pointer
Return:
    true:  succeed
    false: fail
```

6. wifi_station_get_config_default

```
Function:
    Get the configuration parameters saved in the flash of the current WiFi
    station.
Functional definition:
    bool wifi_station_get_config_default (struct station_config *config)
Parameters:
    struct station_config *config : WiFi station configuration pointer
Return:
    true:  succeed
    false: fail
```

7. wifi_station_set_config

```
Function:
    Set the configuration parameters of the WiFi station and save them to the
    flash.
```

Notices:

- This API can be called only when the ESP8266 station is enabled.
- If wifi_station_set_config is called in user_init , there is no need to call wifi_station_connect. The ESP8266 station will automatically connect to the AP (router) after the system initialization. Otherwise, wifi_station_connect should be called.
- Generally, station_config.bssid_set needs to be 0; and it needs to be 1 only when users need to check the MAC address of the AP.
- This configuration will be saved in the flash system parameter area if changed.

Functional definition:

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

Parameters:

struct station_config *config: WiFi station configuration pointer

Return:

true: succeed
false: fail

Example:

```
void ICACHE_FLASH_ATTR
user_set_station_config(void)
{
    char ssid[32] = SSID;
    char password[64] = PASSWORD;
    struct station_config stationConf;

    stationConf.bssid_set = 0;  //need not check MAC address of AP

    os_memcpy(&stationConf.ssid, ssid, 32);
    os_memcpy(&stationConf.password, password, 64);
    wifi_station_set_config(&stationConf);
}
void user_init(void)
{
    wifi_set_opmode(STATIONAP_MODE); //Set softAP + station mode
    user_set_station_config();
}
```

8. wifi_station_set_config_current

Function:

Set parameters of the WiFi station. They won't be saved to the flash.

Notices:

- This API can be called only when the ESP8266 station is enabled.
- If wifi_station_set_config_current is called in user_init , there is no need to call wifi_station_connect The ESP8266 station will automatically connect to the AP (router) after the system initialization. Otherwise, wifi_station_connect should be called.
- Generally, station_config.bssid_set needs to be 0, and it needs to be 1 only when users need to check the MAC address of the AP.

Functional definition:

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

Parameters:

struct station_config *config: WiFi station configuration pointer

Return:

true: succeed
false: fail

9. wifi_station_connect

Function:

connect THE ESP8266 WiFi station to the AP.

Notices:

- Do not call this API in user_init. This API should be called when the ESP8266 station is enabled, and the system initialization is completed.
- If the ESP8266 is connected to an AP, call wifi_station_disconnect to disconnect.

Functional definition:

```
bool wifi_station_connect (void)
```

Parameter:

null

Return:

true: succeed
false: fail

10. wifi_station_disconnect

Function:

Disconnects WiFi station from AP.

Notices:

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

Functional definition:

```
bool wifi_station_disconnect (void)
```

Parameters:

null

Return:

true: succeed
false: fail

11. wifi_station_get_connect_status

Function:

Get the connection status of the ESP8266 WiFi station.

Functional definition:

```
uint8 wifi_station_get_connect_status (void)
```

Parameter:

null

Return:

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

12. wifi_station_scan

Function:

Scan all available APs.

Notice:

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

Functional definition:

```
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: scan the AP configuration parameters

if config==null: scan all available APs

if config.ssid==null && config.bssid==null && config.channel!=null: ESP8266 will scan the APs in specific channels

scan_done_cb_t cb: callback function after scanning

Return:

true: succeed

false: fail

13. scan_done_cb_t

Function:

Callback function for wifi_station_scan

Functional definition:

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

Parameters:

void *arg: information of APs that are found; save them as linked list; refer to struct bss_info

STATUS status: get status

Return:

null

Example:

```
wifi_station_scan(&config, scan_done);
static void ICACHE_FLASH_ATTR scan_done(void *arg, STATUS status) {
    if (status == OK) {
        struct bss_info *bss_link = (struct bss_info *)arg;
        bss_link = bss_link->next.stqe_next; //ignore first
        ...
    }
}
```

14. wifi_station_ap_number_set

Function:

Set the number of APs that can be recorded in the ESP8266 station. When the ESP8266 station is connected to an AP, the SSID and password of the AP will be recorded.

Notice:

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

Functional definition:

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

Parameter:

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

Return:

true: succeed
false: fail

15. wifi_station_get_ap_info

Function:

Get the information of APs (5 at most) recorded by ESP8266 station.

Functional definition:

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

Parameter:

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

Return:

The number of APs recorded.

Example:

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

16. wifi_station_ap_change

Function:

Switch the ESP8266 station connection to a recorded AP.

Functional definition:

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

Parameter:

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

Return:

true: succeed

false: fail

17. wifi_station_get_current_ap_id

Function:

Get the current record id of the AP. The ESP8266 can record the AP of every configuration; start counting from 0.

Functional definition:

```
uint8 wifi_station_get_current_ap_id ();
```

Parameter:

null

Return:

The recorded id of the current AP.

18. wifi_station_get_auto_connect

Function:

Check if the ESP8266 station will connect to the recorded AP automatically when the power is on.

Functional definition:

```
uint8 wifi_station_get_auto_connect(void)
```

Parameter:

null

Return:

0: not connect to the AP automatically;

Non-0: connect to the AP automatically.

19. wifi_station_set_auto_connect

Function:

Set whether the ESP8266 station will connect to the recorded AP automatically when the power is on. It will do so by default.

Notices:

If this APIP is called in user_init, it is effective immediately after the power is on. If it is called in other places, it will be effective the next time when the power is on.

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

Functional definition:

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

Parameters:

uint8 set: If it will automatically connect to the AP when the power is on

0: it will not connect automatically

1: it will connect automatically

Return:

true: succeed

false: fail

20. wifi_station_dhcpc_start

Function:

Enable the ESP8266 station DHCP client.

Notices:

(1) The DHCP is enabled by default.

(2) the DHCP and the static IP API ((wifi_set_ip_info)) influence each other, and if the DHCP is enabled, the static IP will be disabled; if the static IP is enabled, the DHCP will be disabled. It depends on the latest configuration.

Functional definition:

```
bool wifi_station_dhcpc_start(void)
```

Parameter:

null

Return:

true: succeed

false: fail

21. wifi_station_dhcpc_stop

Function:

Disable the ESP8266 station DHCP client.

Notices:

(1) The DHCP is enabled by default.

(2) the DHCP and the static IP API ((wifi_set_ip_info)) influence each other, and if the DHCP is enabled, the static IP will be disabled; if the static IP is enabled, the DHCP will be disabled.

Functional definition:

```
bool wifi_station_dhcpc_stop(void)
```

Parameter:

null

Return:

true: succeed

false: fail

22. wifi_station_dhcpc_status

Function:

Get the ESP8266 station DHCP client status.

Functional definition:

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

Parameter:

null

Return:

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

23. wifi_station_set_reconnect_policy

Function:

Set whether the ESP8266 station will reconnect to the AP after disconnection. It will do so by default.

Notice:

It is suggested that users call this API in user_init.

Functional definition:

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

Parameter:

bool set - if it's true, it will enable reconnection; if it's false, it will disable reconnection

Return:

true: succeed

false: fail

24. wifi_station_get_reconnect_policy

Function:

Check whether the ESP8266 station will reconnect to the AP after disconnection. It will do so by default.

Functional definition:

```
bool wifi_station_get_reconnect_policy(void)
```

Parameter:

null

Return:

true: enable the reconnection

false: disable the reconnection

25. wifi_softap_get_config

Function:

Get the current configuration of the ESP8266 WiFi soft-AP.

Functional definition:

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

Parameter:

struct softap_config *config : ESP8266 soft-AP configuration

Return:

true: succeed

false: fail

26. wifi_softap_get_config_default

Function:

Get the configuration of the ESP8266 WiFi soft-AP saved in the flash.

Functional definition:

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

Parameter:

struct softap_config *config : ESP8266 soft-AP configuration

Return:

true: succeed

false: fail

27. wifi_softap_set_config

Function:

Set the configuration of the WiFi soft-AP and save it to the flash.

Notices:

- Call this API when the ESP8266 soft-AP is enabled.
- This configuration will be saved in flash system parameter area if changed.
- The ESP8266 is limited to only one channel, so when in the soft-AP + station mode, the soft-AP will adjust its channel automatically to be the same as the channel of the ESP8266 station. For more details, refer to *5. appendix*.

Functional definition:

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

Parameter:

struct softap_config *config : WiFi soft-AP configuration

Return:

true: succeed

false: fail

28. wifi_softap_set_config_current

Function:

Set the configuration of the WiFi soft-AP; the configuration won't be saved to the flash.

Notices:

- Call this API when the ESP8266 soft-AP is enabled.
- The ESP8266 is limited to only one channel, so when in the soft-AP + station mode, the soft-AP will adjust its channel automatically to be the same as the channel of the ESP8266 station. For more details, refer to *5. appendix*.

Functional definition:

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

Parameter:

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

Return:

true: succeed
false: fail

29. wifi_softap_get_station_num

Function:

Get the number of stations connected to the ESP8266 soft-AP.

Functional definition:

```
uint8 wifi_softap_get_station_num(void)
```

Parameter:

null

Return:

the number of stations connected to the ESP8266 soft-AP

30. wifi_softap_get_station_info

Function:

Get the number of stations connected to the ESP8266 soft-AP, including MAC and IP.

Notice:

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

Functional definition:

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

Input Parameters:

null

Return:

`struct station_info*` : station information structure

31. wifi_softap_free_station_info

Function:

Free the space occupied by `station_info` when `wifi_softap_get_station_info` is called.

Functional definition:

```
void wifi_softap_free_station_info(void)
```

Parameter:

null

Return:

null

Examples 1 (Getting MAC and IP information):

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

Examples 2 (Getting MAC and IP information):

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

32. wifi_softap_dhcps_start

Function:

Enable the ESP8266 soft-AP DHCP server.

Notices:

(1) The DHCP is enabled by default.

(2) the DHCP and the static IP API ((wifi_set_ip_info)) influence each other, and if the DHCP is enabled, the static IP will be disabled; if the static IP is enabled, the DHCP will be disabled. It depends on the latest configuration.

Functional definition:

```
bool wifi_softap_dhcps_start(void)
```

Parameter:

null

Return:

true: succeed

false: fail

33. wifi_softap_dhcps_stop

Function:

Disable the ESP8266 soft-AP DHCP server. the DHCP is enabled by default.

Functional definition:

```
bool wifi_softap_dhcps_stop(void)
```

Parameter:

null

Return:

true: succeed

false: fail

34. wifi_softap_set_dhcps_lease

Function:

Set the IP range of the ESP8266 soft-AP DHCP server.

Notices:

- The IP range should be in the same sub-net with the ESP8266 soft-AP IP address
- This API should only be called when the DHCP server is disabled (wifi_softap_dhcps_stop).

• This configuration will only take effect the next time when the DHCP server is enabled (wifi_softap_dhcps_start). If the DHCP server is disabled again, this API should be called to set the IP range; otherwise, when the DHCP server is enabled later, the default IP range will be used.

Functional definition:

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

Parameter:

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

Return:

true: succeed
false: fail

Example:

```
void dhcps_lease_test(void)
{
   struct dhcps_lease dhcp_lease;
   const char* start_ip = “192.168.5.100”;
   const char* end_ip = “192.168.5.105”;

   dhcp_lease.start_ip.addr = ipaddr_addr(start_ip);
   dhcp_lease.end_ip.addr = ipaddr_addr(end_ip);
   wifi_softap_set_dhcps_lease(&dhcp_lease);
}
```

or

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

```
        wifi_softap_dhcps_stop();

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

        dhcps_lease_test();

        wifi_softap_dhcps_start();
    }
```

35. wifi_softap_dhcps_status

Function:

Get the ESP8266 soft-AP DHCP server status.

Functional definition:

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

Parameter:

null

Return:

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

36. wifi_softap_set_dhcps_offer_option

Function:

Set the ESP8266 soft-AP DHCP server option.

Structure:

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

Functional definition:

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

Parameters:

```
uint8 level -   OFFER_ROUTER set the router option
void* optarg -
   bit0, 0 disable the router information;
   bit0, 1 enable the router information
```

Return:

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

Example:

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

37. wifi_set_phy_mode

Fuction:

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

Notice:

The ESP8266 soft-AP only supports bg.

Functional definition:

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

Parameters:

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

Return:

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

38. wifi_get_phy_mode

Function:

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

Functional definition:

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

Parameter:

null

Return:

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

39. wifi_get_ip_info

Function:

Get the IP address of the WiFi station or the soft-AP interface.

Functional definition:

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

Parameters:

uint8 if_index : get the IP address of the station or the soft-AP interface

0x00 for STATION_IF, 0x01 for SOFTAP_IF.

struct ip_info *info : the IP information obtained

Return:

true: succeed

false: fail

40. wifi_set_ip_info

Function:

Set the IP address of the station or the soft-AP interface.

Notice:

This API should be called in user_init.

Functional definition:

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

Parameters:

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

Example:

```
struct ip_info info;

wifi_station_dhcpc_stop();
wifi_softap_dhcps_stop();

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

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

wifi_softap_dhcps_start();
```

Return:

true: succeed

false: fail

41. wifi_set_macaddr

Function:

Set the MAC address

Notices:

- This API should be called in user_init.

Functional definition:

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

Parameters:

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

Example:

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

wifi_set_opmode(STATIONAP_MODE);
wifi_set_macaddr(SOFTAP_IF, sofap_mac);
wifi_set_macaddr(STATION_IF, sta_mac);
```

Return:

true: succeed

false: fail

42. wifi_get_macaddr

Function: get MAC address

Functional definition:

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

Parameter:

uint8 if_index : s set the IP address of the station or the soft-AP interface

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

uint8 *macaddr : the MAC address

Return:

true: succeed

false: fail

43. wifi_status_led_install

Function:

Install the WiFi status LED.

Functional definition:

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

```
Parameters:
    uint8 gpio_id   : GPIO id
    uint8 gpio_name : GPIO mux name
    uint8 gpio_func : GPIO function
Return:
    null
```

44. wifi_status_led_uninstall

```
Function:
    Uninstall the WiFi status LED.
Functional definition:
    void wifi_status_led_uninstall ()
Parameter:
    null
Return:
    null
```

45. wifi_set_event_handler_cb

```
Function:
    Register the Wi-Fi event handler.
Functional definition:
    void wifi_set_event_handler_cb(wifi_event_handler_cb_t cb)
Parameter:
    wifi_event_handler_cb_t cb - callback function
Return:
    none
Example:
void wifi_handle_event_cb(System_Event_t *evt)
{
    printf("event %x\n", evt->event);
    switch (evt->event) {
        case EVENT_STAMODE_CONNECTED:
                printf("connect to ssid %s, channel %d\n",
                            evt->event_info.connected.ssid,
                            evt->event_info.connected.channel);
                break;
```

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

4.5. Upgrade (FOTA) APIs

1. system_upgrade_userbin_check

Function:

Check the user bin.

Functional definition:

```
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:

Set the upgrade status flag.

Notice:

After downloading new softwares, set the flag to UPGRADE_FLAG_FINISH and call system_upgrade_reboot to reboot the system in order to run the new software.

Functional definition:

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

Parameter:

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

Return:

null

3. system_upgrade_flag_check

Function:

Check the upgrade status flag.

Functional definition:

```
uint8 system_upgrade_flag_check()
```

Parameter:

null

Return:

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

4. system_upgrade_reboot

Function: reboot system to use the new software.

Functional definition:

```
void system_upgrade_reboot (void)
```

Parameters:

null

Return:

null

4.6. Sniffer Related APIs

1. wifi_promiscuous_enable

Function:

Enable the promiscuous mode.

Notices:

(1) The promiscuous mode can only be enabled in the ESP8266 station mode.

(2) When in the promiscuous mode, the ESP8266 station and soft-AP are disabled.

(3)Call wifi_station_disconnect to disconnect before enabling the promiscuous mode.

(4) Don't call any other APIs when in the promiscuous mode. Call wifi_promiscuous_enable(0) to quit sniffer before calling other APIs.

Functional definition:

```
void wifi_promiscuous_enable(uint8 promiscuous)
```

Parameter:

uint8 promiscuous :

0: to disable the promiscuous mode

1: to enable the promiscuous mode

Return:

null

2. wifi_promiscuous_set_mac

Function:

Set the MAC address filter for the sniffer mode.

Notices:

This filter works only for the current sniffer mode.

If users disable and then enable the sniffer mode, and then enable sniffer, they need to set the MAC address filter again.

Functional definition:

```
void wifi_promiscuous_set_mac(const uint8_t *address)
```

Parameter:

const uint8_t *address : MAC address

Return:

null

Example:

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

3. wifi_set_promiscuous_rx_cb

Function:

Register the RX callback function in the promiscuous mode. Each time a packet is received, the callback function will be registered.

Functional definition:

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

Parameter:

`wifi_promiscuous_cb_t cb` : callback function

Return:

null

4. wifi_get_channel

Function:

Get the channel number for sniffer functions

Functional definition:

```
uint8 wifi_get_channel(void)
```

Parameters:

null

Return:

channel number

5. wifi_set_channel

Function:

Set the channel number for sniffer functions

Functional definition:

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

Parameters:

`uint8 channel` : channel number

Return:

true: succeed
false: fail

4.7. smart config APIs

Herein we only introduce smart-config APIs，users can inquire Espressif Systems for smart-config documentation which will contain more details. 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:

(1)This api can only be called in station mode.

(2)During smart-config，ESP8266 station and soft-AP are disabled.

(3)Can not call smartconfig_start twice before it finish，please call smartconfig_stop first.

(4)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 output logs; otherwise: UART only outputs the result.

Return:

true: succeed
false: fail

Example:

```
void smartconfig_done(sc_status status, void *pdata)
  {
      switch(status) {
          case SC_STATUS_WAIT:
              printf("SC_STATUS_WAIT\n");
              break;
          case SC_STATUS_FIND_CHANNEL:
              printf("SC_STATUS_FIND_CHANNEL\n");
              break;
          case SC_STATUS_GETTING_SSID_PSWD:
              printf("SC_STATUS_GETTING_SSID_PSWD\n");
              sc_type *type = pdata;
              if (*type == SC_TYPE_ESPTOUCH) {
                  printf("SC_TYPE:SC_TYPE_ESPTOUCH\n");
              } else {
                  printf("SC_TYPE:SC_TYPE_AIRKISS\n");
              }
              break;
          case SC_STATUS_LINK:
              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:
                    printf("SC_STATUS_LINK_OVER\n");
                        if (pdata != NULL) {
                        uint8 phone_ip[4] = {0};
                        memcpy(phone_ip, (uint8*)pdata, 4);
                        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:

Whether connect to AP succeed or not, this API should be called to free memory taken by smartconfig_start.

Prototype:

```
bool smartconfig_stop(void)
```

Parameter:

null

Return:

true: succeed
false: fail

4.8. cJSON APIs

1. cJSON_Parse

Function:
Parse the cJSON character string.

Functional definition:
`cJSON *cJSON_Parse (const char *value)`

Parameters:
`const char *value` : the incoming character string

Return:

Return to the cJSON structure.

2. cJSON_Print

Function:
Change the cJSON structure to character string.

Functional definition:
`char *cJSON_Print (cJSON *item)`

Parameters:
`cJSON *item` : transmit the cJSON structure

Return:
character string

3. cJSON_Delete

Function:
Delete the cJSON structure to free the cJSON space.

Functional definition:
`void cJSON_Delete (cJSON *c)`

Parameters:
`cJSON *c` : transmit the cJSON

Return:
null

4. cJSON_GetArraySize

Function:
Get the cJSON array size or object size

Functional definition:

`int cJSON_GetArraySize (cJSON *array)`

Parameters:

`cJSON *array` : transmit the cJSON array

Return:

the cJSON array size or object size

5. cJSON_GetArrayItem

Function:

Get the array or object items by index.

Functional definition:

`cJSON *cJSON_GetArrayItem(cJSON *array,int item)`

Parameters:

`cJSON *array` : transmit cJSON

`int item` : array or object items

Return:

Return to items of the array or the object by index. If items cannot be found, then return to `NULL`.

6. cJSON_GetObjectItem

Function:

Get the array or object items by name.

Functional definition:

`cJSON *cJSON_GetObjectItem (cJSON *object, const char *string)`

Parameters:

`cJSON *object` : transmit cJSON

`const char *string` : array or object names

Return:

Return to items of the array or the object by name. If items cannot be found, then return to `NULL`.

7. cJSON_CreateXXX

Function:

Create cJSON items of different types.

Functional definition:

```
cJSON *cJSON_CreateNull (void)   // create Null cJSON structure
cJSON *cJSON_CreateTrue (void)   // create True cJSON structure
cJSON *cJSON_CreateFalse (void)  // create False cJSON structure
cJSON *cJSON_CreateBool (int b)  // create bool cJSON structure
cJSON *cJSON_CreateNumber (double num)   // create double num cJSON
structure
cJSON *cJSON_CreateString (const char *string) // create character string
cJSON structure
cJSON *cJSON_CreateArray (void)   // create array cJSON structure
cJSON *cJSON_CreateObject (void)  // create JSON tree cJSON structure
```

Return:

cJSON

8. cJSON_CreateXXXArray

Function:

Create arrays of multiple types.

Functional definition:

```
cJSON *cJSON_CreateIntArray(const int *numbers,int count)
cJSON *cJSON_CreateFloatArray(const float *numbers,int count)
cJSON *cJSON_CreateDoubleArray(const double *numbers,int count)
cJSON *cJSON_CreateStringArray(const char **strings,int count)
```

Parameters:

numbers : data values

int count : number of arrays created

Return:

cJSON

9. cJSON_InitHooks

Function:

Redefine malloc, realloc, free, etc.

Functional definition:

```
void cJSON_InitHooks (cJSON_Hooks* hooks)
```

Parameters:

cJSON_Hooks* hooks : cJSON callback function

Return:

null

10. cJSON_AddItemToArray

Function:

Add an item to a specific array, and destroy the current JSON.

Functional definition:

```
void cJSON_AddItemToArray (cJSON *array, cJSON *item)
```

Parameters:

cJSON *array : the specific array

cJSON *item : the item to be added

Return:

null

11. cJSON_AddItemReferenceToArray

Function:

Add an item to a specific array, and not destroy the current JSON.

Functional definition:

```
void cJSON_AddItemReferenceToArray (cJSON *array, cJSON *item)
```

Parameters:

cJSON *array : the specific array

cJSON *item : the item to be added

Return:

null

12. cJSON_AddItemToObject

Function:

Add an item to a specific object, and destroy the current JSON.

Functional definition:

```
void cJSON_AddItemToObject (cJSON *object, const char *string, cJSON *item)

void cJSON_AddItemToObjectCS (cJSON *object, const char *string, cJSON
*item)
```

Parameters:

cJSON *object : the current object

const char *string : the item to be added

cJSON *item : the specific object

Return:

null

13. cJSON_AddItemReferenceToObject

Function:

Add an item to a specific object, and not destroy the current JSON.

Functional definition:

```
void cJSON_AddItemReferenceToObject (
    cJSON *object,
    const char *string,
    cJSON *item)
```

Parameters:

cJSON *object : the current object

const char *string : the item to be added

cJSON *item : the specific object

Return:

null

14. cJSON_DetachItemFromArray

Function:

Detach an item from a specific array.

Functional definition:

```
cJSON *cJSON_DetachItemFromArray (cJSON *array, int which)
```

Parameters:

cJSON *array : the specific array

int which : address of the item to be detached

Return:

cJSON

15. cJSON_DeleteItemFromArray

Function:

Delete an item from a specific array.

Functional definition:

`void cJSON_DeleteItemFromArray (cJSON *array, int which)`

Parameters:

`cJSON *array` : the specific array

`int which` : address of the item to be deleted

Return:

null

16. cJSON_DetachItemFromObject

Function:

Detach an item from a specific object.

Functional definition:

`cJSON *cJSON_DetachItemFromObject (cJSON *object, const char *string)`

Parameters:

`cJSON *object` : the specific object

`const char *string` : the item to be detached

Return:

cJSON structure

17. cJSON_DeleteItemFromObject

Function:

Delete an item from a specific object

Functional definition:

`void cJSON_DetachItemFromObject (cJSON *object, const char *string)`

Parameters:

`cJSON *object` : the specific object

`const char *string` : the item to be deleted

Return:

null

18. cJSON_InsertItemInArray

Function:

insert an item in an array.

Functional definition:

```
void cJSON_InsertItemInArray (cJSON *array, int which, cJSON *newitem)
```

Parameters:

`cJSON *array` : the specific array

`int which` : the array position of the new item.

`cJSON *newitem` : the item to be inserted

Return:

null

19. cJSON_ReplaceItemInArray

Function:

Replace an item in an array.

Functional definition:

```
void cJSON_ReplaceItemInArray (cJSON *array, int which, cJSON *newitem)
```

Parameters:

`cJSON *array` : the specific array

`int which` : the array position of the item to be replaced

`cJSON *newitem` : the new item

Return:

null

20. cJSON_ReplaceItemInObject

Function:

Replace an item in an object.

Functional definition:

```
void cJSON_ReplaceItemInObject (
    cJSON *object,
    const char *string,
    cJSON *newitem)
```

Parameters:

`cJSON *object` : the specific object

const char *string : the item to be replaced

cJSON *newitem : the new item

Return:

null

21. cJSON_Duplicate

Function:

duplicate a cJSON and create an identical one.

Functional definition:

```
cJSON *cJSON_Duplicate (cJSON *item, int recurse)
```

Parameters:

cJSON *item : the cJSON to be duplicated

int recurse : depth of recursion

Return:

cJSON

22. cJSON_ParseWithOpts

Function:

Parse the cJSON.

Functional definition:

```
cJSON *cJSON_ParseWithOpts (
    const char *value,
    const char **return_parse_end,
    int require_null_terminated)
```

Parameters:

const char *value : transmit the character string

const char **return_parse_end : Parse the address at the end of the character string

int require_null_terminated : the location where the parsing terminates

Return:

cJSON

5. Definitions & Structures

5.1. Timer

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

5.2. WiFi Related Structures

1. Station Related

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

Notices:

BSSID is the MAC address of the AP. It will be used when several APs have the same SSID.

If `station_config.bssid_set==1` , `station_config.bssid` must be set, otherwise, the connection will fail.

Generally, `station_config.bssid_set` should be set as 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
};

Notices:
    If softap_config.ssid_len==0, check the SSID until there is a termination
    character; otherwise, set the SSID length according to
    softap_config.ssid_len.
```

3. scan related

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

4. WiFi event related structure

```
enum {
```

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

        REASON_BEACON_TIMEOUT           = 200,
        REASON_NO_AP_FOUND              = 201,
};

typedef struct {
        uint8 ssid[32];
```

```
        uint8 ssid_len;
        uint8 bssid[6];
        uint8 channel;
} Event_StaMode_Connected_t;

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

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

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

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

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

typedef union {
        Event_StaMode_Connected_t           connected;
        Event_StaMode_Disconnected_t        disconnected;
        Event_StaMode_AuthMode_Change_t     auth_change;
        Event_StaMode_Got_IP_t                      got_ip;
        Event_SoftAPMode_StaConnected_t     sta_connected;
```

```
            Event_SoftAPMode_StaDisconnected_t      sta_disconnected;
    } Event_Info_u;

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

6. Appendix

6.1. RTC APIs Example

The example below shows how to check the RTC time, the system time, and changes during system_restart, as well as how to read and write the RTC memory.

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

os_timer_t rtc_test_t;
#define  RTC_MAGIC  0x55aaaa55

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

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

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

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

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

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

    cal1 = system_rtc_clock_cali_proc();
    os_delay_us(300);

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

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

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

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

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

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

6.2. 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 11n packet; 1:is not 11n packet;
    unsigned legacy_length:12; // if not 11n packet, shows length of packet.
    unsigned damatch0:1;
    unsigned damatch1:1;
    unsigned bssidmatch0:1;
    unsigned bssidmatch1:1;
    unsigned MCS:7;             // if is 11n packet, shows the modulation
                                // and code used (range from 0 to 76)
    unsigned CWB:1; // if is 11n packet, shows if is HT40 packet or not
    unsigned HT_length:16;// if is 11n packet, shows length of packet.
    unsigned Smoothing:1;
    unsigned Not_Sounding:1;
    unsigned:1;
    unsigned Aggregation:1;
    unsigned STBC:2;
    unsigned FEC_CODING:1; // if is 11n packet, shows if is LDPC packet or not.
    unsigned SGI:1;
    unsigned rxend_state:8;
    unsigned ampdu_cnt:8;
    unsigned channel:4; //which channel this packet in.
    unsigned:12;
};

struct LenSeq{
    u16 len; // length of packet
    u16 seq; // serial number of packet, the high 12bits are serial number,
         //    low 14 bits are Fragment number (usually be 0)
    u8 addr3[6]; // the third address in packet
};

struct sniffer_buf{
    struct RxControl rx_ctrl;
    u8 buf[36 ]; // head of ieee80211 packet
    u16 cnt;     // number count of packet
```

```
        struct LenSeq lenseq[1];  //length of packet
    };

    struct sniffer_buf2{
        struct RxControl rx_ctrl;
        u8 buf[112];
        u16 cnt;
        u16 len;  //length of packet
    };
```

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.

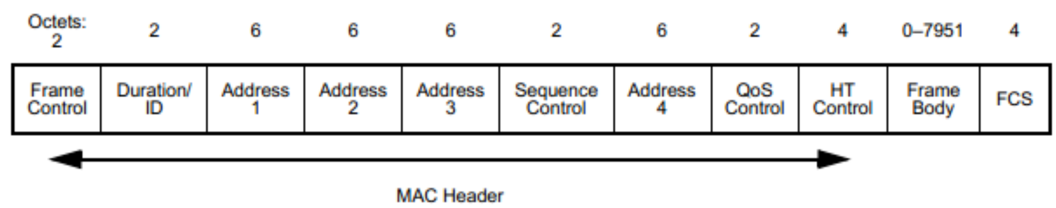


Figure 8-30—Data frame

- 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.

6.3. ESP8266 soft-AP and station channel configuration

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

In the soft-AP + 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-F connected to the ESP8266 soft-AP in step two will be disconnected.