ESP8266 RTOS SDK
Programming Guide

Version 1.3.0

Espressif Systems IOT Team

http://bbs.espressif.com/

Copyright © 2016

Disclaimer and Copyright Notice

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

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

The WiFi Alliance Member Logo is a trademark of the WiFi Alliance.

All trade names, trademarks and registered trademarks mentioned in this document are property of their respective owners, and are hereby acknowledged.

Copyright © 2016 Espressif Systems. All rights reserved.

Table of Contents

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.

Notes:

- It is recommended that users set the timer to the periodic mode for periodic checks.

    ▸ In freeRTOS timer or os_timer, do not delay by `while(1)` or in the manner that will block the thread.

    ▸ The timer callback should not occupy CPU more than 15ms.

    ▸ `os_timer_t` should not define a local variable, it has to be global variable or static variable or memory got by `os_malloc`.

- Since ESP8266_RTOS_SDK_v1.2.0, functions are stored in CACHE area 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.

- The highest priority of the RTOS SDK is 14. `xTaskCreate` is an interface of freeRTOS to create tasks. For details of the freeRTOS APIs, 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 `os_malloc` and `os_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 timer (ms) thread is 12; priority of the TCP/IP task is 10; priority of the freeRTOS timer is 2; priority of the idle task is 0.

‣ Users can use tasks with priorities from 1 to 9.

‣ Please do not revise `FreeRTOSConfig.h`. Revision of the head file can only be fulfilled by libraries in SDK.

3.   Sample Codes

3.1.   Catalog Structure of SDK

The catalog structure of ESP8266_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:** sample codes of applications provided by Espressif Systems

Subfolder	Description
driver_lib	• sample codes of driver provided by Espressif Systems
smart config	• sample codes of smart config function provided by Espressif Systems

- **document catalog:** ESP8266_RTOS_SDK files.

- **include catalog:** header files of ESP8266_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 ESP8266_RTOS_SDK.

- **tool catalog:** tools, 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
- Events that will be triggered when WiFi connection state changes
- 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
- How to read and write information from sectors on a flash memory
- Examples of RTC
- How to port APP from non-OS SDK to RTOS SDK

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) ESP8266_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: \ESP8266_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 ESP8266_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)
>
> 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 the working mode of ESP8266 as the 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 the working mode of ESP8266 as 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)　Get the station info when ESP8266 functions as soft-AP

```
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
```

(4)　When functions as soft-AP, the default IP address is 192.168.4.1. The IP address is subject to modification by developers, however, before modifying, DHCP server must be closed first. For example, the IP address can be set as 192.168.5.1

```
wifi_softap_dhcps_stop();  // disable soft-AP DHCP server

struct ip_info info;
IP4_ADDR(&info.ip, 192, 168, 5, 1);          // set IP
IP4_ADDR(&info.gw, 192, 168, 5, 1);          // set gateway
IP4_ADDR(&info.netmask, 255, 255, 255, 0); // set netmask
wifi_set_ip_info(SOFTAP_IF, &info);
```

(5)　Range of IP address allocated by ESP8266 soft-AP can be set by developers. For example, IP address can range from 192.168.5.100 to 192.168.5.105. Please enable DHCP server when the configuration is completed.

```
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);

wifi_softap_dhcps_start();    // enable soft-AP DHCP server
```

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

(7) 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.

无线网络连接   ⌃

DEMO_AP   已连接 ▂▄▆█

Result:

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

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

5. Basic example: events that will be triggered when WiFi connection state changes

(1) The event monitor wifi_set_event_handler_cb watches ESP8266's WiFi connection state, either when it is working as station or soft-AP, and executes a user callback when the connection state changes.

(2) Sample code:

```c
void wifi_handle_event_cb(System_Event_t *evt)
{
    printf("event %x\n", evt->event_id);
    switch (evt->event_id) {
        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 user's own code here....
        wifi_set_event_handler_cb(wifi_handle_event_cb);
    }
```

(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.

Results:

For example, when ESP8266 functions as a station, the process of how it is connected to a router is shown below:

```
    wifi_handle_event_cb : event 1

    connect to ssid Demo_AP, channel 1

    wifi_handle_event_cb : event 4

    IP:192.168.1.126,mask:255.255.255.0,gw:192.168.1.1

    wifi_handle_event_cb : event 2

    disconnect from ssid Demo_AP, reason 8
```

6. 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.

Notes:

- MAC addresses are different when ESP8266 functions as soft-AP and station, so do not set them as the same.

- bit 0 of the first bit of the MAC address should not be 1. For example, the MAC address can be set as "1a:fe:36:97:d5:7b" instead of "15:fe:36:97:d5:7b".

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

7.   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;

    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:

```
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)
```

8.   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!
>
> scan done
>
> (3,"DEMO_AP",-49,"aa:5b:78:30:46:0a",11)

9.   Basic example: read and write information from sectors on a flash memory

(1)   Read information from the sectors on a flash memory. It is essential that four bytes must be aligned. Below is an example of how to read information from a flash.

```
#define SPI_FLASH_SEC_SIZE        4096
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]);
```

(2)   Similarly, when write information into sectors on a flash memory, the four bytes should also be aligned. Use spi_flash_erase_sector to erase the sector first, then call spi_flash_write to write data in. For example,

```
uint32 data[M];
// TODO: fit in the data
spi_flash_erase_sector(N);
spi_flash_write(N*4*1024, data, M*4);
```

(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:

> read data from 0x3E000 : 05 00 04 02

10. Basic example: examples of how to use RTC

(1)   When software restart (system_restart) is executed, the system time will return to zero, while the RTC timer will continue. However, if the chipset is waken up (including periodically waken

up the device from deep-sleep mode) via external hardware including EXT_RST pin or CHIP_EN pin, RTC timer will be restarted. In detail,

```
•   external reset (EXT_RST): RTC memory does not change, register of
        RTC timer counts from zero.

•   watchdog reset: RTC memory does not change, register of RTC timer
        does not change.

•   system_restart: RTC memory does not change, register of RTC timer
        does not change.

•   power on: value of RTC memory is at random, register of RTC timer
        counts from zero.

•   CHIP_EN reset: value of RTC memory is at random, register of RTC
        timer counts from zero.
```

For example, the returned value of system_get_rtc_time is 10 (indicating 10 RTC time cycles), the returned value of system_rtc_clock_cali_proc is 5.75 (indicating time period of one RTC cycle is 5.75 microsecond), then the real time will be 10x5.75 = 57.5 microsecond.

```
rtc_t = system_get_rtc_time();
cal = system_rtc_clock_cali_proc();
os_printf("cal: %d.%d \r\n", ((cal*1000)>>12)/1000, ((cal*1000)>>12)%1000 );
```

Read and write RTC memory. Please be noted that RTC memory can only four-byte lump-sum deposit and withdraw.

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

system_rtc_mem_read(64, &rtc_time, sizeof(rtc_time));
```

(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:

```
rtc_time: 1613921

cal: 6.406
```

11. Basic example: how to port APP from non-OS SDK to RTOS SDK

(1) Codes on how to set the timer do not need to be revised.

(2) Codes on how to execute callback function do not need to be revised.

(3) Method on how to create a new task should be revised. When creating a new task, RTOS SDK uses xTaskCreate, a self-contained interface owned by freeRTOS.

Non-OS SDK: creating a new task

```
#define Q_NUM  (10)
ETSEvent test_q[Q_NUM];

void test_task(ETSEvent *e)
{
 switch(e->sig)
 {
    case 1:
            func1(e->par);
            break;
    case 2:
            func2();
            break;
    case 3:
            func3();
            break;
    default:
            break;
 }
}

void func_send_Sig(void)
{
    ETSSignal sig = 2;
    system_os_post(2,sig,0);
}

void task_ini(void)
{
    system_os_task(test_task, 2, test_q,Q_NUM);
    // test_q is the corresponding array of test_task.
    // (2) is the priority of test_task.
```

```
        // Q_NUM is the queue length of test_task.
    }
```

RTOS SDK: creating a new task

```
        #define Q_NUM  (10)
        xQueueHandle test_q;
        xTaskHandle test_task_hdl;
        void test_task(void *pvParameters)
        {
           int *sig;
           for(;;){
            if(pdTRUE == xQueueReceive(test_q, &sig, (portTickType)portMAX_DELAY) ){
                   vTaskSuspendAll();
                   switch(*sig)
                   {
                       case 1:
                           func1();
                           break;
                       case 2:
                           func2();
                           break;
                       default:
                           break;
                   }
                   free(sig);
                   xTaskResumeAll();
            }
          }
        }


        void func_send_Sig(void)
        {
           int *evt = (int *)malloc(sizeif(int));
           *evt = 2;
           if(xQueueSend(test_q,&evt,10/portTick_RATE_MS)!=pdTRUE){
             os_printf("test_q is full\n");
           }
        // It is the address of parameter that stored in test_q, so int *evt and int
            *sig can be other types.
        }
```

```
void task_ini(void)
{
  test_q = xQueueCreate(Q_NUM,sizeof(void *));
  xTaskCreate(test_task,(signed portCHAR *)"test_task", 512, NULL, (1),
    &test_task_hdl );
// 512 means the heap size of this task, 512 * 4 byte.
// NULL is a pointer of parameter to test_task.
// (1) is the priority of test_task.
// test_task_hdl is the pointer of the task of test_task.
}
```

3.3. Networking Protocol Example

The networking protocol of ESP8266_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

    ESP8266 UDP task > socket ok!
```

> ESP8266 UDP task > bind ok!
>
> ESP8266 UDP task > recv data 16 Bytes from 192.168.1.112, Port 57233

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.
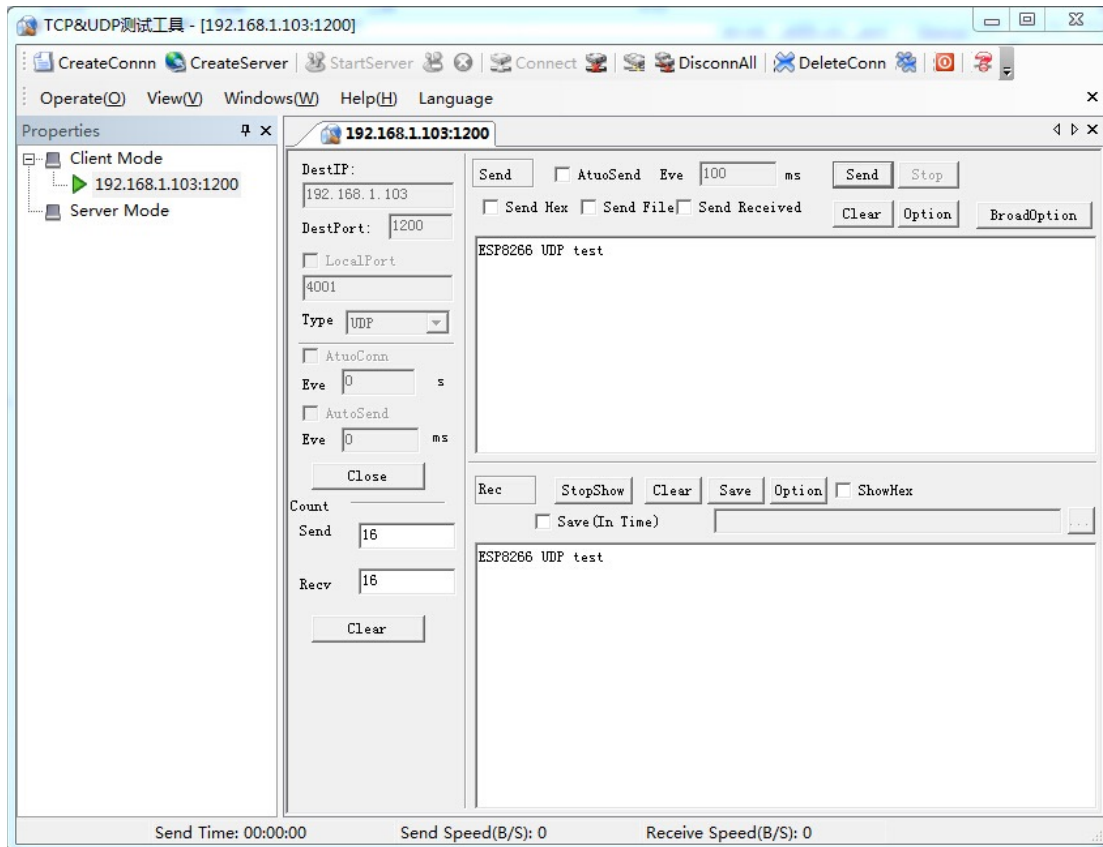
2. Networking protocol example: TCP client

(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.
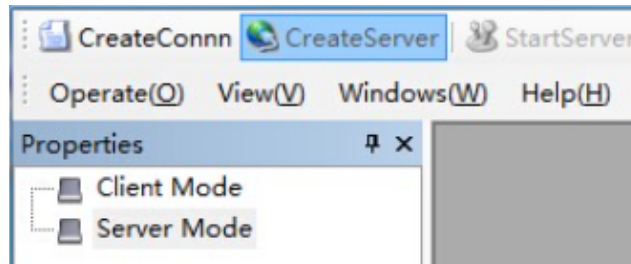

```
#define SERVER_IP          "192.168.1.124"
#define SERVER_PORT        1001
```

(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.
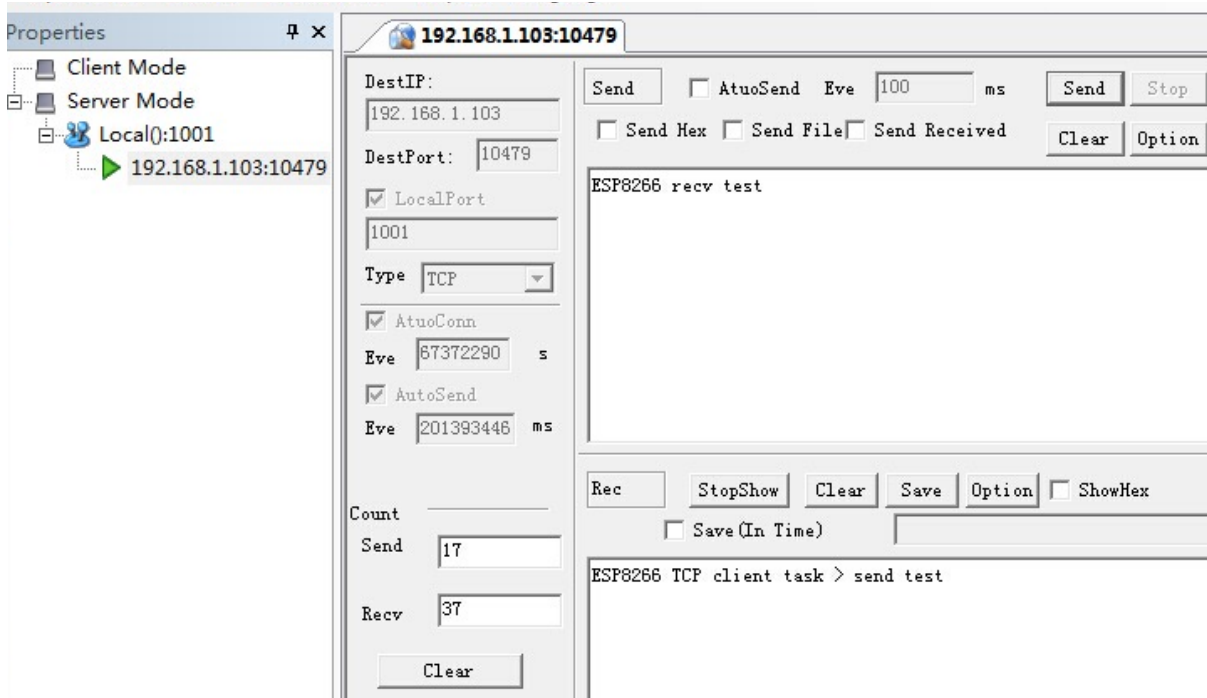
3. Networking protocol example: TCP server

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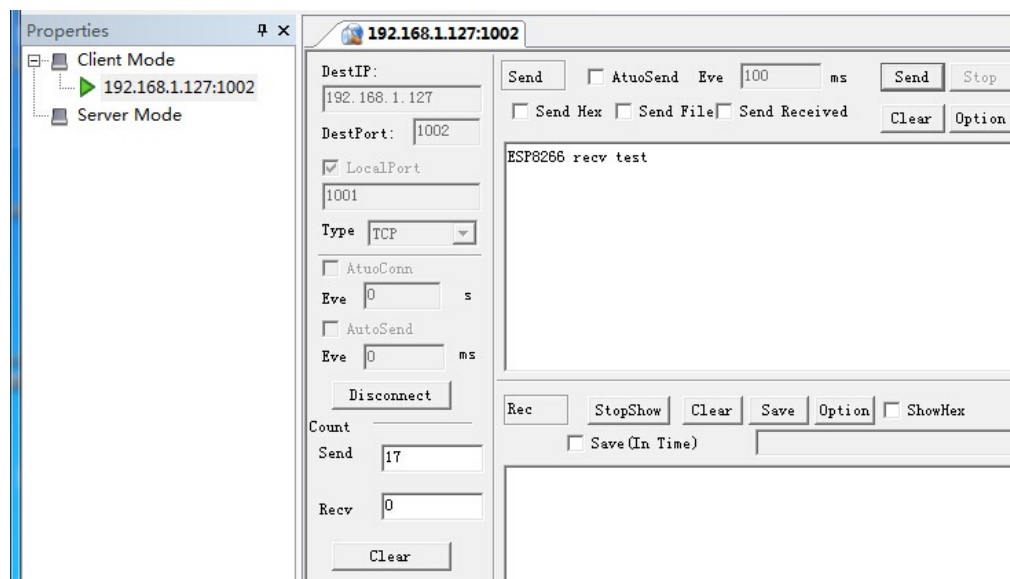
(2)　Compile application program, generate firmware and burn it into ESP8266 module.

(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 Examples

Advanced examples included in ESP8266_RTOS_SDK are listed below:

- Firmware upgrade Over-the-air (OTA)

- Example of force sleep

- spiffs file system

- Examples on how to implement SSL

1. Advanced 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.

Note:

Erasing the flash sector is a slow process. Thus it may take longer time to erase a flash sector while write information into sectors of the flash at the same time. Besides, the stability of the network might also be affected. Consequently, please call function `spi_flash_erase_sector` to erase sectors waiting to be upgraded first, then connect to the network, and download the latest firmware from OTA server, then call function `spi_flash_write` to write information into sectors of the flash.

---

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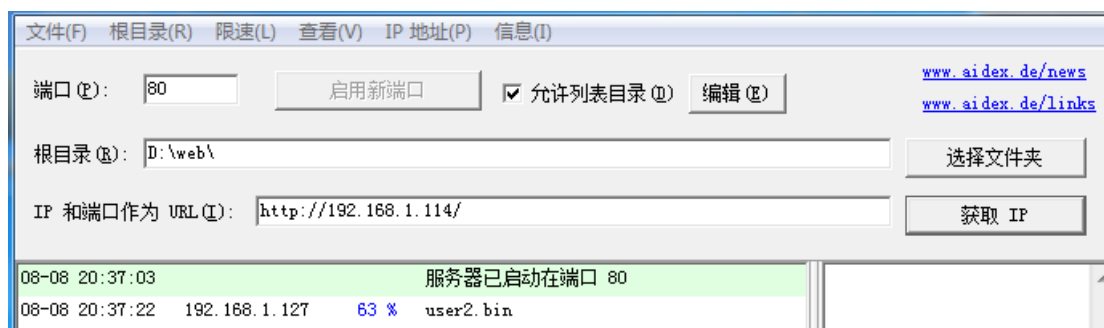
The module will reboot when user2.bin has been downloaded, and start running user2.bin. Then user1.bin will be downloaded from the server. This cycle revolves.

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

connected with Demo_AP, channel 6

ip:192.168.1.127,mask:255.255.255.0,gw:192.168.1.1

socket 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/30.0.1599.101 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!

… …

2. Advanced example: example of force sleep

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

Note:

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

Example one: Modem-sleep mode (Disable RF)

```c
#define FPM_SLEEP_MAX_TIME    0xFFFFFFF

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


void user_func(...)
{
   ...
   wifi_station_disconnect();
   wifi_set_opmode(NULL_MODE);              // set WiFi mode to null mode.
   wifi_fpm_set_sleep_type(MODEM_SLEEP_T);  // modem sleep
   wifi_fpm_open();                         // enable force sleep
#ifdef SLEEP_MAX
   /* For modem sleep, FPM_SLEEP_MAX_TIME can only be wakened by calling
wifi_fpm_do_wakeup. */
   wifi_fpm_do_sleep(FPM_SLEEP_MAX_TIME);
#else
   // wakeup automatically when timeout.
   wifi_fpm_set_wakeup_cb(fpm_wakup_cb_func1);   // Set wakeup callback
   wifi_fpm_do_sleep(10*1000);
#endif
   ...
}
```

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

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

Users need to set a callback by wifi_fpm_set_wakeup_cb, so the program can go on after wake up.

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

void user_func(...)
{
   wifi_station_disconnect();
   wifi_set_opmode(NULL_MODE);          // set WiFi mode to null mode.
   wifi_fpm_set_sleep_type(LIGHT_SLEEP_T); // light sleep
   wifi_fpm_open();                     // enable force sleep
   wifi_fpm_set_wakeup_cb(fpm_wakup_cb_func1);   // Set wakeup callback
   wifi_fpm_do_sleep(10*1000);
   ...
}
```

3.   Advanced example: spiffs file system

(1)   Initialize spiffs file system by call esp_spiffs_init.

```
void spiffs_fs1_init(void)
{
   struct esp_spiffs_config config;

   config.phys_size = FS1_FLASH_SIZE;
   config.phys_addr = FS1_FLASH_ADDR;
```

```
        config.phys_erase_block = SECTOR_SIZE;
        config.log_block_size = LOG_BLOCK;
        config.log_page_size = LOG_PAGE;
        config.fd_buf_size = FD_BUF_SIZE * 2;
        config.cache_buf_size = CACHE_BUF_SIZE;


        esp_spiffs_init(&config);
    }
```

(2)   Open and create a new file, write in the data.

```
    char *buf="hello world";
    char out[20] = {0};


    int pfd = open("myfile", O_TRUNC | O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
    if(pfd <= 3) {
        printf("open file error \n");
    }
    int write_byte = write(pfd, buf, strlen(buf));
    if (write_byte <= 0)
    {
        printf("write file error \n");
    }
    close(pfd);
```

(3)   Read date via the file system.

```
    open("myfile",O_RDWR);
    if (read(pfd, out, 20) < 0)
        printf("read errno \n");
    close(pfd);
    printf("--> %s <--\n", out);
```

4.   Advanced example: how to implement SSL

(1)   Define the IP address and port that SSL server will be connected to.

```
    #define SSL_SERVER_IP     "115.29.202.58"
    #define SSL_SERVER_PORT   443


    esp_test *pTestParamer = (esp_test *)zalloc(sizeof(esp_test));


    pTestParamer->ip.addr = ipaddr_addr(SSL_SERVER_IP);
```

```
    pTestParamer->port = server_port;
```

(2) Create a new task when the device functions as SSL client.

```
    xTaskCreate(esp_client, "esp_client", 1024, (void*)pTestParamer, 4, NULL);
```

(3) When ESP8266 functions as a station, connect it to a router. Then check if it has already get
     the IP address, then start SSL connection.

```
    struct ip_info ipconfig;
    wifi_get_ip_info(STATION_IF, &ipconfig);


    while (ipconfig.ip.addr == 0) {
        vTaskDelay(1000 / portTICK_RATE_MS);
        wifi_get_ip_info(STATION_IF, &ipconfig);
    }
```

(4) Create socket connection.

```
    client_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (client_fd < 0){
        printf("create with the socket err\n");
    }
    memset(&client_addr, 0, sizeof(client_addr));
    client_addr.sin_family = AF_INET;
    client_addr.sin_port = htons(port);
    client_addr.sin_addr.s_addr = sin_addr;


    if(connect(client_fd, (struct sockaddr *)&client_addr, sizeof(client_addr))< 0)
            printf("connect with the host err\n");
```

(5) Create the context of SSL. Please call system_get_free_heap_size to check the memory space
     available since SSL requires a relative large amount of space.

```
    uint32 options = SSL_SERVER_VERIFY_LATER|SSL_DISPLAY_CERTS|SSL_NO_DEFAULT_KEY;
    if ((ssl_ctx = ssl_ctx_new(options, SSL_DEFAULT_CLNT_SESS)) == NULL){
           printf("Error: Client context is invalid\n");
       }
    printf("heap_size %d\n",system_get_free_heap_size());
```

(6) If SSL authentication function is required:

If not use spiffs file system, please run python script esp_iot_sdk_freertos\tools\make_cert.py, generate esp_ca_cert.bin, and write it into the flash.

Below is an example showing how to read information about SSL encryption key and certificate from the flash.

```
    uint8 flash_offset = 0x78;    // Example : Flash address 0x78000


    if (ssl_obj_option_load(ssl_ctx, SSL_OBJ_RSA_KEY, "XX.key", password,
        flash_offset)){
          printf("Error: the Private key is undefined.\n");
    }


    if (ssl_obj_option_load(ssl_ctx, SSL_OBJ_X509_CERT, "XX.cer", NULL,
        flash_offset)){
          printf("Error: the Certificate is undefined.\n");
    }
```

If use spiffs file system, please run tool spiffy (https://github.com/xlfe/spiffy, and please notice that the spiffs_config.h of this tool has to be the same as the one in RTOS SDK), generate spiff_rom.bin, and write it into the flash.

Below is an example showing how to read information about SSL encryption key and certificate from the flash through spiffs.

```
    if (ssl_obj_load(ssl_ctx, SSL_OBJ_RSA_KEY, "XX.key", password)){
          printf("Error: the Private key is undefined.\n");
      }


    if (ssl_obj_load(ssl_ctx, SSL_OBJ_X509_CERT, "XX.cer", NULL)){
          printf("Error: the Certificate is undefined.\n");
    }
```

(7)   Start handshake with SSL client.

```
    ssl = ssl_client_new(ssl_ctx, client_fd, NULL, 0);
    if (ssl != NULL){
        printf("client handshake start\n");
    }
```

(8)   Check the status of SSL connection.

```
    if ((res = ssl_handshake_status(ssl)) == SSL_OK){
        … …
    }
```

(9) If the handshake succeed, then the certificate can be released and more memory space will be available.

```
const char *common_name = ssl_get_cert_dn(ssl,SSL_X509_CERT_COMMON_NAME);
if (common_name){
    printf("Common Name:\t\t\t%s\n", common_name);
}
display_session_id(ssl);
display_cipher(ssl);
quiet = true;
os_printf("client handshake ok! heapsize %d\n",system_get_free_heap_size());
x509_free(ssl->x509_ctx);
ssl->x509_ctx=NULL;
os_printf("certificate free ok! heapsize %d\n",system_get_free_heap_size());
```

(10) Transmit SSL data.

```
uint8 buf[512];
bzero(buf, sizeof(buf));
sprintf(buf,httphead,"/", "iot.espressif.cn",port);
os_printf("%s\n", buf);
if(ssl_write(ssl, buf, strlen(buf)+1) < 0) {
    ssl_free(ssl);
    ssl_ctx_free(ssl_ctx);
    close(client_fd);
    vTaskDelay(1000 / portTICK_RATE_MS);
    os_printf("send fail\n");
    continue;
}
```

(11) Receive SSL data.

```
while((recbytes = ssl_read(ssl, &read_buf)) >= 0) {
    if(recbytes == 0){
        vTaskDelay(500 / portTICK_RATE_MS);
        continue;
    }
    os_printf("%s\n", read_buf);
}

free(read_buf);
if(recbytes < 0) {
```

```
        os_printf("ERROR:read data fail! recbytes %d\r\n",recbytes);
        ssl_free(ssl);
        ssl_ctx_free(ssl_ctx);
        close(client_fd);
        vTaskDelay(1000 / portTICK_RATE_MS);
    }
```

Result:

```
    ip:192.168.1.127,mask:255.255.255.0,gw:192.168.1.1

    -----BEGIN SSL SESSION PARAMETERS-----

    4ae116a6a0445b369f010e0ea5420971497e92179a6602c8b5968c1f35b60483

    -----END SSL SESSION PARAMETERS-----

    CIPHER is AES128-SHA

    client handshake ok! heapsize 38144

    certificate free ok! heapsize 38144

    GET / HTTP/1.1

    Host: iot.espressif.cn:443

    Connection: keep-alive

    ......
```

4. Appendix

4.1. 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
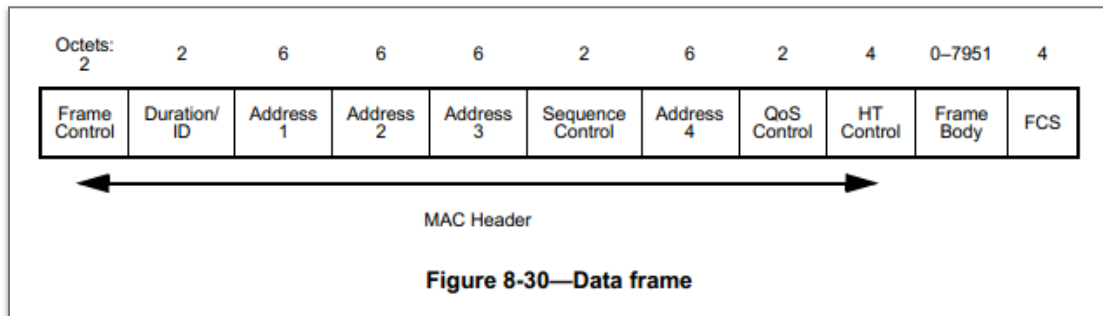
- The first 24 bytes of MAC header of the data packet are needed:

  ‣ `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.

4.2. 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.

Case 3:
  (1) Other stations are connected to the ESP8266 softAP .
  (2) If the ESP8266's station interface has been scanning or trying to connect to a target router, the ESP8266 softAP-end connection may break.

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

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

4.3. ESP8266 boot messages

ESP8266 outputs boot messages through UART0 with baud rate 74880:

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



load 0x4010f000, len 1264, room 16

tail 0

chksum 0x42

csum 0x42
```

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