ESP8266 RTOS SDK

Programming Guide

Version 1.5

Copyright © 2017

About This Guide

This document provides sample codes based on ESP8266_RTOS_SDK.

The document is structured as follows:

Chapter	Title	Subject
Chapter 1	Introduction	Provides preliminary information on ESP8266EX.
Chapter 2	Overview	Provides an overview of ESP8266_RTOS_SDK.
Chapter 3	Sample Codes	Provides sample codes based on ESP8266_RTOS_SDK.
Chapter 4	Appendix	Provides relevant extra information.

Release Notes

Date	Version	Release notes
2016.04	V1.4	First Release.
2017.02	V1.5	Important updates based on ESP8266_RTOS_SDK v1.5.

Content

1. Introduction

The ESP8266EX offers a complete and self-contained Wi-Fi SoC solution. With low power consumption, a compact design and high stability, it caters to the needs of its users.

It can be used to host 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 built-in, high-speed cache which improves the performance of the system and optimizes the memory. Alternatively, when the ESP8266EX is used as a Wi-Fi adapter, wireless internet access can be added to any microcontroller-based device through the SPI/SDIO interface or the I2C/UART interface, and thus provide the users with a simple Wi-Fi solution.

Furthermore, the ESP8266EX offers a 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 software of the related applications.

The ESCP–Espressif Systems' Smart Connectivity Platform 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 ESP8266EX 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 present document should be particularly helpful to embedded software developers who use the ESP8266EX IoT platform for software development.

2. Overview

2.1. RTOS SDK Introduction

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. Users can initialize the interface in **user_main.c**.

`void user_init(void)` is the entry-point 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.

2.2. Notes on Programming

- It is recommended that users set the timer to the periodic mode for regular checks.
  - In freeRTOS timer or os_timer, do not delay by using while(1) or in any other manner that will block the thread. For example, a socket send operation can not be done in the timer callback for the send function will block threads.
  - The timer callback should not occupy the CPU for more than 15 ms.
  - `os_timer_t` should not define a local variable; it should define a global variable, or a static variable, or a pointer allocated by `os_malloc`.
- Since the release of ESP8266_RTOS_SDK_v1.2.0, functions are stored in CACHE area by default, and there is no need to add the macro `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.
- In network programming using general-purpose sockets, please do not bind the sockets to the same port in network communication.
- For details on the freeRTOS and its APIs, please refer to *http://www.freertos.org*.
- The highest priority of the RTOS SDK is 14. `xTaskCreate` is an interface of freeRTOS for creating tasks.When using `xTaskCreate` to create a task, the task stack range is [176, 512].
  - If an array with a length of over 60 bytes is used in a task, it is suggested that users use `os_malloc` and `os_free` rather than the local variables to allocate the array. Large local variables could lead to task stack overflow.
  - The RTOS SDK takes up certain priorities. The priority of the watchdog task is 14; pp task is 13; the priority of the precise timer (ms) thread is 12; the priority of the TCP/IP task is 10; the priority of the freeRTOS timer is 2; and the priority of the idle task is 0.

- Users can create tasks with priorities from 1 to 9.
- Please do not revise *FreeRTOSConfig.h*. Revision of the head file can only be done by libraries in the SDK.

3.   Sample Codes

3.1.   Directory Structure of RTOS SDK

ESP8266 RTOS SDK can be downloaded via the following link: *ESP8266 RTOS SDK*.

Below is the directory structure of the ESP8266 RTOS SDK.

- *bin*: boot and initialization firmware.

- *documents*: ESP8266_RTOS_SDK files.

- *driver_lib*: sample codes for drivers.

- *examples*: sample codes for Espressif's application programs.

  - *openssl_demo*: OpenSSL-API-related sample codes.

  - *project_template*: sample codes for a project template.

  - *smart_config*: SmartConfig-related sample codes.

  - *spiffs_test*: SPIFFS-related sample codes.

  - *websocket_demo*: WebSocket-related sample codes.

- *include*: header files of ESP8266_RTOS_SDK, including software interfaces and macro functions for users to use.

- *ld*: linker scripts used for compiling; users do not need to modify them.

- *lib*: library files provided in ESP8266_RTOS_SDK.

- *third_party*: third-party library of Espressif's open-source codes, currently including freeRTOS, JSON, lwIP, mbedTLS, noPoll, OpenSSL, SPIFFS, and SSL.

- *tools*: tools; users do not need to modify them.

3.2.   Basic Examples

Some basic examples are listed below:

- Initialization

- How to read the ID of the chip

- How to set the Wi-Fi working mode

  ‣ when ESP8266 works in Station mode, it can be connected to the AP (router)

  ‣ when ESP8266 works in SoftAP mode, it can be connected to other Stations

- Events that will be triggered when the Wi-Fi connection state changes

- How to read and set the MAC address of the chip

- How to scan APs nearby

- How to get the RSSI (Received Signal Strength Indicator) of an AP

- How to read and write information from sectors on a flash memory

- Examples of RTC

- How to port apps from non-OS SDK to RTOS SDK

3.2.1. Initialization

1. The initialization of application programs can be implemented in `user_main.c`. The function `void user_init(void)`, which is the entry-point function, can be used by users to implement the initialization process. It is suggested that the SDK's version information be printed, and the Wi-Fi working mode be set.

```
void user_init(void)

{

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

    /* station + soft-AP mode */

    Wi-Fi_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 defined by users themselves in `user_init`. Please refer to `uart_init_new` on how to implement this.

   Sample of UART driver: ***ESP8266_RTOS_SDK\\driver_lib\\driver\\uart.c*

   Take the initialization of UART0 as an example. Configure the 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 the UART interrupt function and enable the 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. ESP8266_RTOS_SDK supports multi-threading, therefore, multiple tasks can be created. The interface `xTaskCreate` used to create tasks is a self-contained interface owned by freeRTOS. When using `xTaskCreate` to create a new task, the range of the 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 the application program, generate firmware and download it into the ESP8266 module.

5. Power off the module, and change it 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!
```

3.2.2. How to Read the ID of the Chip

1. Introduction of the Software Interface:

`system_get_chip_id` returns the value signifying the chip ID of the module. Every chip has one exclusive ID.

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

2. Compile the application program, generate firmware and download it into the ESP8266 module.

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

Result:

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

3.2.3. Connect to AP When ESP8266 Functions as Station

1. Set the working mode of ESP8266 to Station mode, or Station+SoftAP mode.

```
Wi-Fi_set_opmode(STATION_MODE);
```

2. Set the SSID and password of the AP.

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

`Wi-Fi_station_set_config` is used to set the AP information when ESP8266 functions as Station. Please note that the initialized value of `bssid_set` in `station_config` should be 0. It is only when the MAC address of AP need be specified that the initialized value is set to be 1.

`Wi-Fi_station_connect` sets 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);


Wi-Fi_station_set_config(config);

free(config);

Wi-Fi_station_connect();
```

3. Compile the application program, generate firmware and program it into ESP8266 module.

4. Power off the module, and change it 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
```

3.2.4. ESP8266 Functions as SoftAP

1. Set the working mode of ESP8266 as the Station mode, or Station+SoftAP mode.

```
Wi-Fi_set_opmode(SOFTAP_MODE);
```

2. Configure ESP8266 as SoftAP.

```
#define DEMO_AP_SSID     "DEMO_AP"

#define DEMO_AP_PASSWORD  "12345678"

struct softap_config *config = (struct softap_config *)zalloc(sizeof(struct
softap_config));                    // initialization
```

```
        Wi-Fi_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;


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

3. Get the Station info when ESP8266 functions as SoftAP.

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

4. When ESP8266 functions as SoftAP, its default IP address is 192.168.4.1. The IP address is subject to modification by developers; however, the DHCP server must be closed first before modifying the address. Below is an example of setting the IP address as 192.168.5.1.

```
        Wi-Fi_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
        Wi-Fi_set_ip_info(SOFTAP_IF, &info);
```

5. The range of the IP address allocated by ESP8266 SoftAP can be set by developers. For example, the IP address can range from 192.168.5.100 to 192.168.5.105. Please enable the 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);
        Wi-Fi_softap_set_dhcps_lease(&dhcp_lease);


        Wi-Fi_softap_dhcps_start();     // enable soft-AP DHCP server
```

6. Compile the application program, generate firmware and download it into the ESP8266 module.

7. Power off the module, and change it to operation mode; then power on the module and run the program. Connect a PC or other Stations to the ESP8266 SoftAP.

Result:

ESP8266 works as SoftAP, and the following information will be printed when a Station is connected to it:

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

3.2.5. Events Triggered When Wi-Fi Connection State Changes

1. The event monitor `Wi-Fi_set_event_handler_cb` monitors ESP8266's Wi-Fi connection state, either when it is working as Station or SoftAP, and executes a user callback when the connection state changes.

2. Sample code:

```
void Wi-Fi_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....
    Wi-Fi_set_event_handler_cb(Wi-Fi_handle_event_cb);
}
```

3. Compile the application program, generate firmware and download it into the ESP8266 module.

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

Result:

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

```
Wi-Fi_handle_event_cb : event 1

connect to ssid Demo_AP, channel 1

Wi-Fi_handle_event_cb : event 4

IP:192.168.1.126,mask:255.255.255.0,gw:192.168.1.1

Wi-Fi_handle_event_cb : event 2

disconnect from ssid Demo_AP, reason 8
```

3.2.6. Read and Set the MAC Address of ESP8266

1. ESP8266 can work in Station+SoftAP mode. The MAC addresses of the Station and SoftAP interfaces are different. It is guaranteed that the MAC address of every chip is

unique and exclusive. If users want to reset the MAC address, the uniqueness of the MAC address should be assured.

2. Set ESP8266 to Station+SoftAP mode.

```
Wi-Fi_set_opmode(STATIONAP_MODE);
```

3. Read the MAC addresses of the Station and SoftAP interfaces.

```
Wi-Fi_get_macaddr(SOFTAP_IF, sofap_mac);

Wi-Fi_get_macaddr(STATION_IF, sta_mac);
```

4. Set the MAC addresses of the Station and SoftAP interfaces. The setting of MAC addresses is not stored in the flash, and the setting can only be done when the corresponding interface is enabled first.

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

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


Wi-Fi_set_macaddr(SOFTAP_IF, sofap_mac);

Wi-Fi_set_macaddr(STATION_IF, sta_mac);
```

5. Compile the application program, generate firmware and download it into the ESP8266 module.

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

> ⚠️ *Notice:*
>
> - *Mac addresses are different for the two distinct modes of ESP8266 (SoftAP and Station). Please do not set the same MAC address for ESP8266 SoftAP and ESP8266 Station.*
>
> - *Bit 0 of the first byte of the MAC address should not be 1. For example, the MAC address can be set as "1a:fe:36:97:d5:7b", but not as "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
```

3.2.7. Scan Nearby APs

1. Set ESP8266 to work in Station mode, or Station+SoftAP mode.

```
Wi-Fi_set_opmode(STATIONAP_MODE);
```

2. Scan nearby APs.

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

```
Wi-Fi_station_scan(NULL,scan_done);
```

Callback function when the AP scanning is completed:

```c
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 the application program, generate firmware and download it into the ESP8266 module.

4. Power off the module, and change it 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)
```

3.2.8. Get RSSI (Received Signal Strength Indicator) of AP

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

   Specify the SSID of the target AP:

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

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

```
struct scan_config config;


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


Wi-Fi_station_scan(&config,scan_done);
```

2. Compile the application program, generate firmware and download it into the ESP8266 module.

3. Power off the module, and change it 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)
```

3.2.9. Read and Write of Flash Memory

1. Read data from a flash memory. It is essential that the data address be four-byte 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 writing data into sectors on a flash memory, the data address should also be four-byte aligned. Use `spi_flash_erase_sector` to erase the sector first, then call `spi_flash_write` to write data into it. 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 the application program, generate firmware and download it into the ESP8266 module.

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

Result:

```
read data from 0x3E000 : 05 00 04 02
```

3.2.10. 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 chip is woken up (including being periodically woken up from Deep-sleep mode) via external hardware (including EXT_RST pin or CHIP_EN pin), the RTC timer will be restarted.

   - external reset (EXT_RST): the RTC memory does not change; the register of the RTC timer counts from zero.

   - watchdog reset: the RTC memory does not change; the register of the RTC timer does not change.

   - `system_restart`: the RTC memory does not change; the register of the RTC timer does not change.

   - Power on: the value of the RTC memory is random; the register of the RTC timer counts from zero.

   - CHIP_EN reset: the value of the RTC memory is random; the register of the RTC timer counts from zero.

   For example, if the returned value of `system_get_rtc_time` is 10 (indicating 10 RTC cycles), and the returned value of `system_rtc_clock_cali_proc` is 5.75 µs (indicating that the duration of one RTC cycle is 5.75 microseconds), then the real time will be 10x5.75 = 57.5 microseconds.

```
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 note that RTC memory access must be four-byte aligned.

```
 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 the application program, generate firmware and download it into the ESP8266 module.

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

Result:

```
rtc_time: 1613921
cal: 6.406
```

3.2.11. How to Port Apps from Non-OS SDK to RTOS SDK

1. Codes for setting the timer do not need to be revised.

2. Codes for executing callback functions do not need to be revised.

3. The method of creating 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 the 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

3.3.1. UDP Transmission

1. Set the local port number of UDP. Below is an example of setting the port number as 1200.

```
#define UDP_LOCAL_PORT  1200
```

2. Create a 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. Receive and transmit the 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 %s, Port %d\n" ,ret,
inet_ntoa(from.sin_addr), ntohs(from.sin_port));


          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 the application program, generate firmware and download it into the ESP8266 module.

6. Power off the module, and change it to operation mode; then power on the module and run the program.
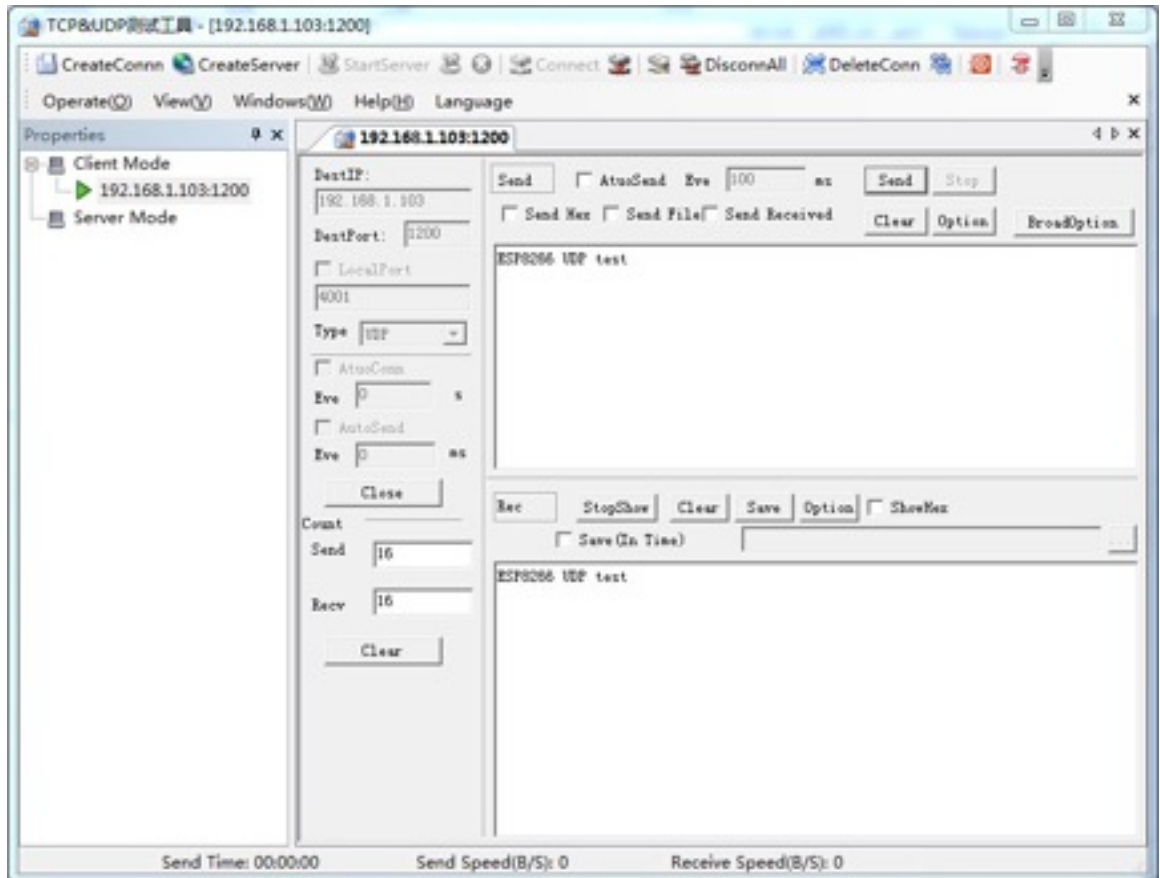
Result:

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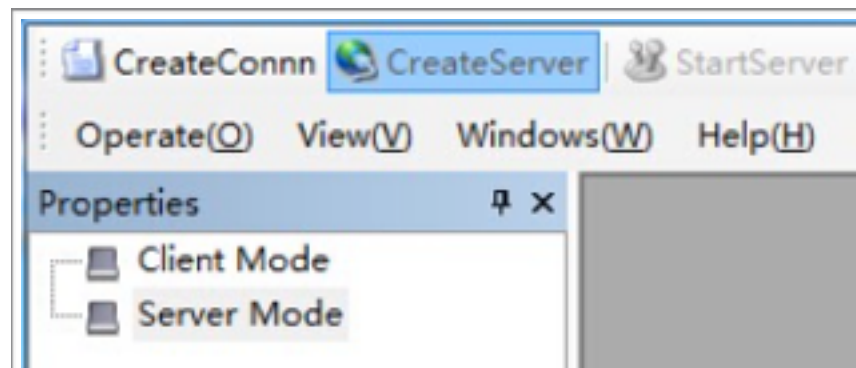
```
ESP8266 UDP task > bind ok!
ESP8266 UDP task > recv data 16 Bytes from 192.168.1.112, Port 57233
```

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

3.3.2. TCP Client

1. When ESP8266 functions as Station, connect it to an AP. Users can refer to previous examples for details on how to do this.

2. Establish a TCP server using network debugging tools.

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

3. Implement the TCP communication via socket programming.

Create a 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 a 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");
```

Send data packets via TCP communication:

```
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);
```

Receive data packets via TCP communication:

```
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");

    }
```
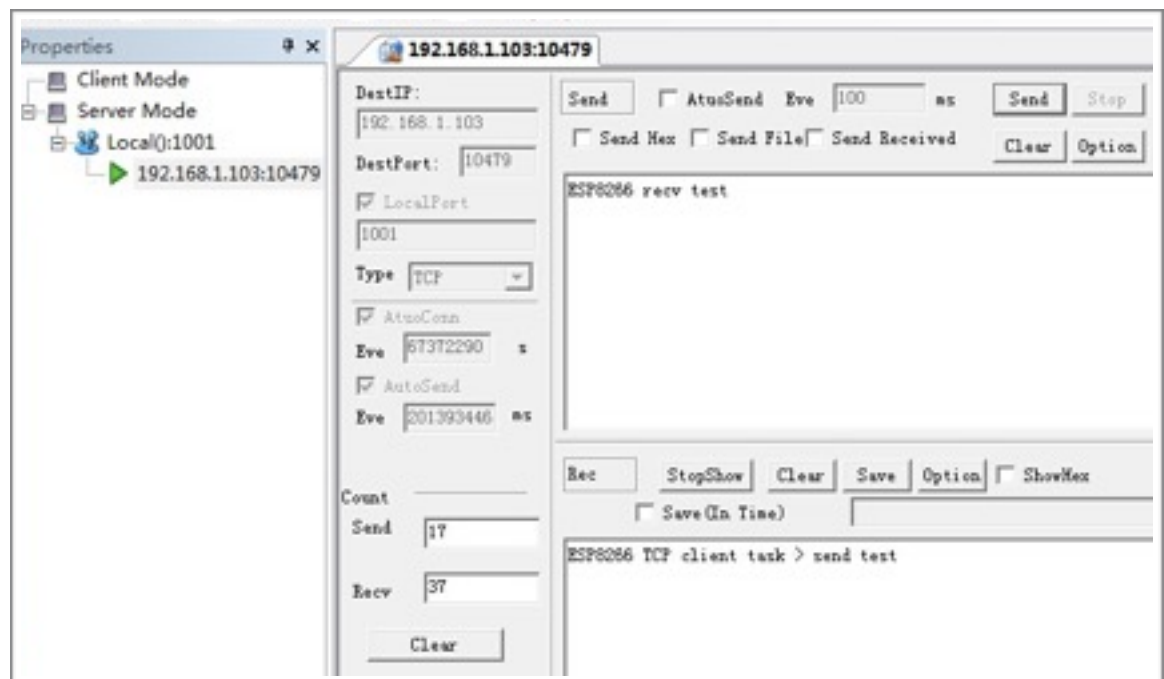
4. Compile the application program, generate firmware and download it into the ESP8266 module.

5. Power off the module, and change it 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
```

Below is an example showing that the TCP server established at the terminal of network debugging tool communicates with the ESP8266 successfully.

3.3.3. TCP Server

1. Establish a TCP server, and bind it with the 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);
```
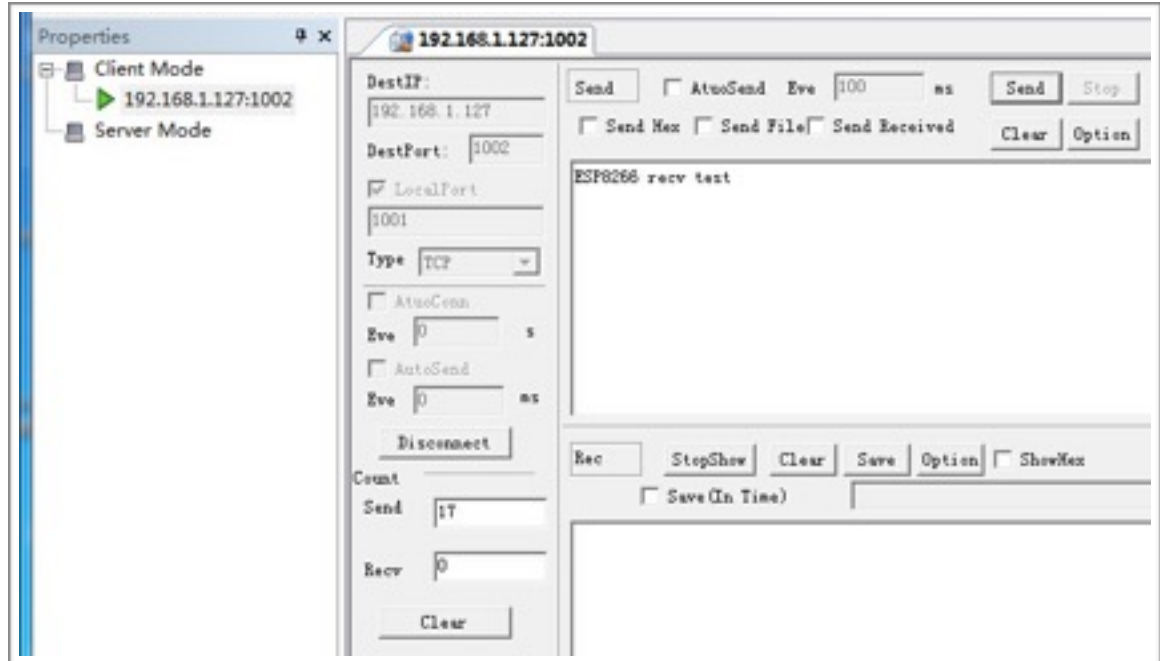
```
            }
        }while(ret != 0);


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

Wait until the TCP client is connected to the server; then start receiving data packets when the 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 the application program, generate firmware and download it into the ESP8266 module.

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

4. Establish a TCP client using the network debugging tool, then connect the TCP client with the 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 Example of OTA Firmware Upgrade

Advanced examples included in ESP8266_RTOS_SDK are listed below:

- Firmware over-the-air (OTA) upgrade
- Example of forced sleep
- SPIFFS file system
- Examples of how to implement SSL

3.4.1. Firmware Over-the-Air (OTA) Upgrade

OTA Firmware upgrade refers to downloading new firmware from the server over a Wi-Fi network and realizing firmware upgrade.

> ⚠️ **Notice:**
>
> *Erasing the flash memory is a slow process. Thus it may take longer time to erase a flash memory sector while writing information into other sectors of the flash at the same time. Besides, the stability of the network might also be affected. Consequently, it is suggested that users call function* `spi_flash_erase_sector` *to erase sectors to be upgraded first; connect to the network, and download the latest firmware from OTA server; then, call function* `spi_flash_write` *to write information into the flash.*

1. Users can establish their own cloud server, or use the cloud server provided by Espressif.

2. Upload the new firmware to the cloud server.

3. Descriptions of the codes are listed below:

   Connect the ESP8266 module to the AP (for details of this process users can refer to previous examples). Then check if the ESP8266 Station can get the IP address through function `upgrade_task`.

```
Wi-Fi_get_ip_info(STATION_IF, &ipconfig);


/* check the IP address or net connection state*/

while (ipconfig.ip.addr == 0) {

    vTaskDelay(1000 / portTICK_RATE_MS);

    Wi-Fi_get_ip_info(STATION_IF, &ipconfig);

}
```

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

- `system_upgrade_flag_set`: sets a flag to indicate the upgrade status.

  - `UPGRADE_FLAG_IDLE`: idle.

  - `UPGRADE_FLAG_START`: starts the upgrade.

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

- `system_upgrade_userbin_check`: checks the user binary 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 the downloading request to the server. After the upgraded firmware data is received successfully, program 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 a time-out, and the firmware has not been updated from the cloud server, it means the upgrade failed. The status of the firmware upgrade will become idle and the upgrade will stop.

- If the firmware has been successfully downloaded from the server, the 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 the application program, generate firmware and download it into the ESP8266 module.

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

Result:

- Establish a server at the PC terminal via the webserver, then upload *user1.bin* and *user2.bin* to the same server. After the firmware has been programmed into ESP8266, *user1.bin* will is run first by default, then *user2.bin* will be downloaded from the server.

- The module will reboot automatically when *user2.bin* has been downloaded, and will start running *user2.bin*. Until the next FOTA upgrade, ESP8266 will not run *user1.bin*. When there is a new firmware on the server and a new FOTA upgrade is requested by the users, *user1.bin* will be downloaded from the server. When the module reboots automatically, *user1.bin* is run in this case. This process repeats.

- The print information during ESP8266 upgrading process:

```
connected with Demo_AP, channel 6

dhcp client start...

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/535.36 (KHTML, like Gecko) Chrome/
30.0.1599.101 Safari/535.36

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!

... ...
```

3.4.2. Examples of Forced Sleep

The forced-sleep interface can be called, and the RF circuit can be closed mandatorily so as to lower the power.

> ⚠️ *Notice:*
>
> - *When forced-sleep interface is called, the chip will not enter sleep mode instantly, it will enter sleep mode when the system is executing the idle task.*
>
> - *Below are the sample codes for sleep modes.*

Example One: Modem-sleep Mode (RF Disabled)

```
      #define FPM_SLEEP_MAX_TIME          0xFFFFFFF
void fpm_wakup_cb_func1(void)
{
    Wi-Fi_fpm_close();                   // disable force sleep function
    Wi-Fi_set_opmode(STATION_MODE);      // set station mode
    Wi-Fi_station_connect();                     // connect to AP
}
void user_func(…)
{
    …
    Wi-Fi_station_disconnect();
    Wi-Fi_set_opmode(NULL_MODE);                 // set Wi-Fi mode to null mode.
    Wi-Fi_fpm_set_sleep_type(MODEM_SLEEP_T);     // modem sleep
    Wi-Fi_fpm_open();                            // enable force sleep
#ifdef SLEEP_MAX
/* For modem sleep, FPM_SLEEP_MAX_TIME can only be wakened by calling Wi-Fi_fpm_do_wakeup.
*/
    Wi-Fi_fpm_do_sleep(FPM_SLEEP_MAX_TIME);
#else
// wakeup automatically when timeout.
    Wi-Fi_fpm_set_wakeup_cb(fpm_wakup_cb_func1) ;    // Set wakeup callback
     Wi-Fi_fpm_do_sleep(50*1000);
#endif
    …
}


#ifdef SLEEP_MAX
void func1(void)
{
    Wi-Fi_fpm_do_wakeup();
    Wi-Fi_fpm_close();                           // disable force sleep function
```

```
        Wi-Fi_set_opmode(STATION_MODE);              // set station mode
        Wi-Fi_station_connect();                     // connect to AP
    }
    #endif
```

Example Two: Light-sleep Mode (RF and CPU Disabled)

ESP8266 is forced into Light-sleep mode, and both RF and CPU are disabled. Users need to set a callback so that the program can continue after wakeup.

```
        void fpm_wakup_cb_func1(void)
        {
            Wi-Fi_fpm_close();              // disable force sleep function
            Wi-Fi_set_opmode(STATION_MODE); // set station mode
            Wi-Fi_station_connect();        // connect to AP
        }


        #ifndef SLEEP_MAX
        // Wakeup till time out.
        void user_func(···)
        {
            Wi-Fi_station_disconnect();

            Wi-Fi_set_opmode(NULL_MODE);                    // set Wi-Fi mode to null mode.
            Wi-Fi_fpm_set_sleep_type(LIGHT_SLEEP_T);        // light sleep
            Wi-Fi_fpm_open();                               // enable force sleep
            Wi-Fi_fpm_set_wakeup_cb(fpm_wakup_cb_func1);    // Set wakeup callback
            Wi-Fi_fpm_do_sleep(50*1000);
        }
        #else
        // Or wakeup by GPIO
        void user_func(···)
        {
             Wi-Fi_station_disconnect();
            Wi-Fi_set_opmode(NULL_MODE);                    // set Wi-Fi mode to null mode.
            Wi-Fi_fpm_set_sleep_type(LIGHT_SLEEP_T);        // light sleep
            Wi-Fi_fpm_open();                               // enable force sleep
            PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTCK_U,3);
            gpio_pin_wakeup_enable(13, GPIO_PIN_INTR_LOLEVEL);

            Wi-Fi_fpm_set_wakeup_cb(fpm_wakup_cb_func1);    // Set wakeup callback
            Wi-Fi_fpm_do_sleep(0xFFFFFFFF);
        }
```

```
#endif
```

3.4.3. How to Use the SPIFFS File System

1. Initialize the SPIFFS file system by calling `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, and write the data in it.

```
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);
```

3.4.4. How to Implement SSL

1. Define the IP address and port that the 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 an 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 obtained the IP address before setting up the SSL connection.

```
struct ip_info ipconfig;
Wi-Fi_get_ip_info(STATION_IF, &ipconfig);


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

4. Create a 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 the SSL requires a relatively 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. When the SSL authentication function is required:

- If not using 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 the 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 using SPIFFS file system, please run the tool spiffy (*https://github.com/xlfe/spiffy*. Please note that the **spiffs_config.h** of this tool has to be the same as the one in the RTOS SDK). Then generate **spiffs_rom.bin**, and write it into the flash; for details on this process please refer to the example of `esp_spiffs_init`.

  Below is an example showing how to read information about the SSL encryption key and certificate from the flash using 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 a handshake with the 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 the SSL connection.

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

9. If the handshake is successful, 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

......
```

A. Appendix

A.1. Sniffer Introduction

For more details on sniffer, please refer to *ESP8266 Technical Reference*.

A.2. ESP8266 SoftAP and Station Channel Configuration

Even though ESP8266 supports the SoftAP+Station mode, only one hardware channel can be used for communication.

In the SoftAP+Station mode, the ESP8266 SoftAP will adjust its channel configuration to be same as that of 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 One:

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

2. and sets the ESP8266 SoftAP through `Wi-Fi_softap_set_config`,

3. if the value is valid, the API will return `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 Two

1. If the user sets the channel of the ESP8266 SoftAP through `Wi-Fi_softap_set_config` (for example, channel 5),

2. other Stations will connect to the ESP8266 SoftAP.

3. Then if 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 that of the ESP8266 Station (which is channel 6 in this case).

5. As a result of the change of channel, the Wi-Fi connection of the Stations connected to the ESP8266 SoftAP in Step Two will be disconnected.

Case Three

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.

3. 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 SoftAP channel is changing, too. Therefore, the ESP8266 SoftAP-end connection may break.

4. In cases like this, users can set a timer to call `Wi-Fi_station_disconnect` in order to stop the ESP8266 Station from continuously trying to connect to the router. Users can also call `Wi-Fi_station_set_reconnect_policy` or `Wi-Fi_station_set_auto_connect` to disable the ESP8266 Station from reconnecting to the router.

A.3. ESP8266 Boot Messages

ESP8266 outputs boot messages through UART0 with a baud rate of 74880:

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



load 0x4010f000, len 1264, room 16

tail 0

chksum 0x42

csum 0x42
```

Below is the description about the boot messages:

Boot messages	Description
rst cause	1: power on
	2: external reset
	4: hardware watchdog reset
boot mode (the first parameter)	1: ESP8266 is in UART-down mode; so firmware is downloaded into the flash via UART.
	3: ESP8266 is in Flash-boot mode and boots up from the flash.
chksum	If chksum == csum, it means that flash is read correctly during booting.

Espressif IOT Team

www.espressif.com