Espressif IoT SDK 编程手册

Status	Released
Current version	V0.9.1
Author	Fei Yu
Completion Date	2014.9.23
Reviewer	Jiangang Wu
Completion Date	2014.9.23

[ ] CONFIDENTIAL

[ ] INTERNAL

[√] PUBLIC

版本信息

日期	版本	撰写人	审核人	修改说明
2013.12.25	0.1	Jiangang Wu		初稿
2013.12.25	0.1.1	Han Liu		修订 json 处理 API
2014.1.15	0.2	Jiangang Wu		增加、修改部分接口配置函数
2014.1.29	0.3	Han Liu		增加 client/server 接口函数
2014.3.20	0.4	Jiangang Wu / Han Liu		1.增双 uart 接口； 2.增加 i2c master 接口； 3.修改 client/server 接口函数； 4.增加加密接口； 5.增加 upgrade 接口
2014.4.17	0.5	Jiangang Wu / Han Liu		1.修改 espconn 接口； 2.增加 gpio 接口 api 说明； 3.增加其他说明；
2014.5.14	0.6	Jiangang Wu		增加若干 API
2014.6.18	0.7	Jiangang Wu		1.增加 dns 接口； 2.修改、添加用户参数保存接口； 3.添加任务接口； 4.添加 softap 模式下获取已连设备信息接口； 5.添加打印系统内存空间及获取可用 heap 区大小接口； 6.其他
2014.7.10	0.8	Fei Yu		1.修改 upgrade 接口； 2.增加开关打印接口； 3.增加 station 连接状态查询接口

				4.增加 SSL 加密 Server 接口；修改加密 client 接口名称
2014.8.13	0.9	Fei Yu		1.修改 espconn 接口； 2.增加 sniffer 接口； 3.增加 chip 查询接口； 4.增加获取、修改 mac&ip 接口；
2014.9.23	0.9.1	Fei Yu		1、增加休眠接口 2、修改 flash 读写接口 3、记录连接过 AP 4、修改 UDP 接口

免责申明和版权公告

本文中的信息，包括供参考的 URL 地址，如有变更，恕不另行通知。

文档“按现状”提供，不负任何担保责任，包括对适销性、适用于特定用途或非侵权性的任何担保，和任何提案、规格或样品在他处提到的任何担保。本文档不负任何责任，包括使用本文档内信息产生的侵犯任何专利权行为的责任。本文档在此未以禁止反言或其他方式授予任何知识产权使用许可，不管是明示许可还是暗示许可。

Wi-Fi 联盟成员标志归 Wi-Fi 联盟所有。

文中提到的所有商标名称、商标和注册商标均属其各自所有者的财产，特此声明。

目录

版本信息.....2
目录.....5
1. 前言.....9
2. 软件框架.....10
3. SDK 提供的 API 接口.....11
3.1. 定时器接口.....11
3.1.1. os_timer_arm.....11
3.1.2. os_timer_disarm.....11
3.1.3. os_timer_setfn.....11
3.2. 底层用户接口.....12
3.2.1. system_restore.....12
3.2.2. system_restart.....12
3.2.3. system_get_chip_id.....13
3.2.4. system_deep_sleep.....13
3.2.5. system_upgrade_userbin_check.....13
3.2.6. system_upgrade_start.....14
3.2.7. system_upgrade_reboot.....14
3.2.8. system_set_os_print.....14
3.2.9. system_timer_reinit.....15
3.2.10. system_print_meminfo.....15
3.2.11. system_get_free_heap_size.....16
3.2.12. system_os_task.....16
3.2.13. system_os_post.....17
3.2.14. spi_flash_erase_sector.....18
3.2.15. spi_flash_write.....18
3.2.16. spi_flash_read.....19
3.2.17. wifi_get_opmode.....19
3.2.18. wifi_set_opmode.....20
3.2.19. wifi_station_get_config.....20
3.2.20. wifi_station_set_config.....20
3.2.21. wifi_station_connect.....21
3.2.22. wifi_station_disconnect.....21
3.2.23. wifi_station_get_connect_status.....21
3.2.24. wifi_station_scan.....22
3.2.25. scan_done_cb_t.....23
3.2.26. wifi_station_ap_number_set.....23
3.2.27. wifi_station_ap_change.....23
3.2.28. wifi_station_get_current_ap_id.....24
3.2.29. wifi_softap_get_config.....24
3.2.30. wifi_softap_set_config.....24

3.2.31. wifi_softap_get_station_info 25
3.2.32. wifi_softap_free_station_info 25
3.2.33. wifi_get_ip_info 26
3.2.34. wifi_set_ip_info 27
3.2.35. wifi_set_macaddr 27
3.2.36. wifi_get_macaddr 28
3.2.37. wifi_status_led_install 28
3.2.38. wifi_promiscuous_enable 29
3.2.39. wifi_set_promiscuous_rx_cb 29
3.2.40. wifi_get_channel 30
3.2.41. wifi_set_channel 30
3.3. espconn 接口 30
3.3.1. 通用接口 31
3.3.1.1. espconn_gethostbyname 31
3.3.1.2. espconn_port 32
3.3.1.3. espconn_regist_sentcb 32
3.3.1.4. espconn_regist_recvcb 33
3.3.1.5. espconn_sent_callback 33
3.3.1.6. espconn_recv_callback 33
3.3.1.7. espconn_sent 34
3.3.2. TCP 连接接口 34
3.3.2.1. espconn_accept 34
3.3.2.2. espconn_secure_accept 35
3.3.2.3. espconn_regist_time 35
3.3.2.4. espconn_get_connection_info 36
3.3.2.5. espconn_connect 36
3.3.2.6. espconn_connect_callback 37
3.3.2.7. espconn_disconnect 37
3.3.2.8. espconn_regist_connectcb 37
3.3.2.9. espconn_regist_reconcb 38
3.3.2.10. espconn_regist_disconcb 38
3.3.2.11. espconn_secure_connect 39
3.3.2.12. espconn_secure_sent 39
3.3.2.13. espconn_secure_disconnect 40
3.3.3. UDP 接口 40
3.3.3.1. espconn_create 40
3.3.3.2. espconn_delete 40
3.4. json API 接口 41
3.4.1. jsonparse_setup 41
3.4.2. jsonparse_next 41
3.4.3. jsonparse_copy_value 42
3.4.4. jsonparse_get_value_as_int 42
3.4.5. jsonparse_get_value_as_long 42
3.4.6. jsonparse_get_len 43

3.4.7. jsonparse_get_value_as_type 43
3.4.8. jsonparse_strcmp_value 43
3.4.9. jsontree_set_up 44
3.4.10. jsontree_reset 44
3.4.11. jsontree_path_name 44
3.4.12. jsontree_write_int 45
3.4.13. jsontree_write_int_array 45
3.4.14. jsontree_write_string 45
3.4.15. jsontree_print_next 46
3.4.16. jsontree_find_next 46
4. 数据结构定义 47
4.1. 定时器结构 47
4.2. wifi 参数 47
4.2.1. station 配置参数 47
4.2.2. softap 配置参数 47
4.2.3. scan 参数 48
4.3. json 相关结构 48
4.3.1. json 结构 48
4.3.2. json 宏定义 50
4.4. espconn 参数 51
4.4.1 回调 function 51
4.4.2 espconn 51
5. 驱动接口 54
5.1. GPIO 接口 API 54
5.1.1. PIN 脚功能设置宏 54
5.1.2. gpio_output_set 54
5.1.3. GPIO 输入输出相关宏 55
5.1.4. GPIO 中断控制相关宏 55
5.1.5. gpio_pin_intr_state_set 56
5.1.6. GPIO 中断处理函数 56
5.2. 双 UART 接口 API 56
5.2.1. uart_init 57
5.2.2. uart0_tx_buffer 57
5.2.3. uart0_rx_intr_handler 58
5.3. i2c master 接口 58
5.3.1. i2c_master_gpio_init 58
5.3.2. i2c_master_init 58
5.3.3. i2c_master_start 59
5.3.4. i2c_master_stop 59
5.3.5. i2c_master_setAck 59
5.3.6. i2c_master_getAck 60
5.3.7. i2c_master_readByte 60
5.3.8. i2c_master_writeByte 60
5.4. pwm 61

5.4.1. pwm_init 61
5.4.2. pwm_start 61
5.4.3. pwm_set_duty 61
5.4.4. pwm_set_freq 62
5.4.5. pwm_get_duty 62
5.4.6. pwm_get_freq 62
6. 无附录 63
A. ESPCONN 编程 63
A.1. client 模式 63
A.1.1. 说明 63
A.1.2. 步骤 63
A.2. server 模式 63
A.2.1. 说明 63
A.2.2. 步骤 64

1. 前言

基于 ESP8266 物联网平台的 SDK 为用户提供了一个简单、快速、高效开发物联网产品的软件平台。

本文旨在介绍该 SDK 的基本框架，以及相关的 API 接口。主要的阅读对象为需要在 ESP8266 物联网平台进行软件开发的嵌入式软件开发人员。

2. 软件框架

为了让用户不用关心底层网络，如 WIFI、TCP/IP 等的具体实现，仅专注于物联网应用的开发，SDK 为用户提供了一套数据接收、发送函数接口，用户只需利用相应接口即可完成网络数据的收发。

ESP8266 物联网平台的所有网络功能均在库中实现，对用户不透明，用户初始化功能在 user_main.c 文件中实现。

函数 void usre_init(void)的作用是给用户提供一个初始化接口，用户可在该函数内增加硬件初始化、网络参数设置、定时器初始化等功能。

SDK 中提供了对 json 包的处理 API，用户也可以采用自定义数据包格式，自行对数据进行处理。

3. SDK 提供的 API 接口

3.1. 定时器接口

说明：定时器接口接口函数或宏以及所使用的参数结构体定义在（工程目录\include\osapi.h）。

3.1.1. os_timer_arm

功能：初始化定时器

函数定义：

Void os_timer_arm(ETSTimer *ptimer,uint32_t milliseconds, bool repeat_flag)

输入参数：

ETSTimer*ptimer——定时器结构（该结构体参见 4.1 说明）

uint32_t milliseconds——定时时间，单位毫秒

bool repeat_flag——该定时是否重复

返回：

无

3.1.2. os_timer_disarm

功能：取消定时器定时

函数定义：

Void os_timer_disarm(ETSTimer *ptimer)

输入参数：

ETSTimer *ptimer——定时器结构（该结构体参见 4.1 说明）

返回：

无

3.1.3. os_timer_setfn

功能：设置定时器回调函数

函数定义：

Void os_timer_setfn(ETSTimer *ptimer, ETSTimerFunc *pfunction, void *parg)

输入参数：

ETSTimer *ptimer——定时器结构（该结构体参见 4.1 说明）

TESTimerFunc *pfunction——定时器回调函数

void*parg——回调函数参数

返回：

无

3.2. 底层用户接口

说明：以下所有接口函数或宏以及所使用的参数结构体定义在（工程目录\include\user_interface.h）。

3.2.1. system_restore

功能：恢复出厂设置

函数定义：

void system_restore(void)

输入参数：

无

返回：

无

3.2.2. system_restart

功能：重启

函数定义：

void system_restart(void)

输入参数：

无

返回：

无

3.2.3. system_get_chip_id

功能：获取芯片 id

函数定义：

uint32 system_get_chip_id(void)

输入参数：

无

返回：

芯片 id 值

3.2.4. system_deep_sleep

功能：设置进入 deep sleep 模式，每休眠多长时间（us）唤醒一次。唤醒以后整个系统重新跑，程序从 user_init 开始。

函数定义：

void system_deep_sleep(uint32 time_in_us)

输入参数：

uint32 time_in_us – 设置休眠时间，单位：us

返回：

无

3.2.5. system_upgrade_userbin_check

功能：检查当前正在使用的 firmware 是 user1 还是 user2。

函数定义：

uint8 system_upgrade_userbin_check()

输入参数：

无

返回：

0x00 : UPGRADE_FW_BIN1 ， 即 user1.bin

0x01 : UPGRADE_FW_BIN2 ， 即 user2.bin

3.2.6. system_upgrade_start

功能：配置参数，开始升级。

函数定义：

bool system_upgrade_start (struct upgrade_server_info *server)

参数：

struct upgrade_server_info *server - server 相关的参数。

返回

True ：开始进行升级。

False ：已正在升级过程中，无法执行 upgrade start。

3.2.7. system_upgrade_reboot

功能：重启系统，运行新版本软件。

函数定义：

void system_upgrade_reboot (void)

输入参数：

无

返回：

无

3.2.8. system_set_os_print

功能：开关打印 log 功能。

函数定义：

Void system_set_os_print(uint8 onoff)

输入参数：

Uint8 onoff —— 打开/关闭打印功能；

0x00 代表关闭打印功能

0x01 代表打开打印功能

默认为打开打印功能。

返回：

无

3.2.9. system_timer_reinit

功能：当需要使用 us 级 timer 时，需要重新初始化 timer。

注意：1、同时定义 USE_US_TIMER；

2、system_timer_reinit 需放在最开始，user_init 第一句。

函数定义：

Void system_timer_reinit (void)

输入参数：

无

返回：

无

3.2.10. system_print_meminfo

功能：打印系统内存空间分配，打印信息包括 data/rodata/bss/heap

函数定义：

Void system_print_meminfo (void)

输入参数：

无

返回：

无

3.2.11. system_get_free_heap_size

功能：获取系统可用 heap 区空间大小

函数定义：

Uint32 system_get_free_heap_size(void)

输入参数：

无

返回：

Uint32 ——可用 heap 区大小

3.2.12. system_os_task

功能：建立系统任务

函数定义：

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

输入参数：

Os_task_t task——任务函数

Uint8 prio——任务优先级，当前支持 3 个邮件级的任务 0/1/2，0 最低

Os_event_t *queue——消息队列指针

Uint8 qlen——消息队列深度

返回：

无

示例：

```
#define SIG_RX   0
#define TEST_QUEUE_LEN     4
os_event_t *testQueue;
void test_task (os_event_t *e)
{
    switch (e->sig) {
        case SIG_RX:
```

```
            os_printf("sig_rx %c\n", (char)e->par);
            break;
        default:
            break;
    }
}
void task_init(void)
{
lwipIf0EvtQueue = (os_event_t *)os_malloc(sizeof(os_event_t)* TEST_QUEUE_LEN);
system_os_task(test_task, USER_TASK_PRIO_0, testQueue, TEST_QUEUE_LEN);
}
```

3.2.13. system_os_post

功能：向任务发送消息

函数定义：

void system_os_post (uint8 prio, os_signal_t sig, os_param_t par)

输入参数：

Uint8 prio——任务优先级，与建立时的优先级对应

Os_signal_t sig——消息类型

Os_param_t par——消息参数

返回：

无

结合上一节的示例：

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

打印输出：sig_rx a

3.2.14. spi_flash_erase_sector

功能：擦除 Flash 的某个扇区。

说明：flash 读写操作的介绍，详见文档“Espressif IOT Flash 读写说明”。

函数定义：

SpiFlashOpResult spi_flash_erase_sector (uint16 sec)

参数：

uint16 sec - 扇区号，从扇区 0 开始计数，每扇区 4KB

返回：

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

3.2.15. spi_flash_write

功能：将数据存到 Flash。

说明：flash 读写操作的介绍，详见文档“Espressif IOT Flash 读写说明”。

函数定义：

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

参数：

uint32 des_addr - 写入 Flash 的地址，起始位置。

uint32 *src_addr - 写入 Flash 的数据指针。

Uint32 size - 写入数据长度

返回：

```
Typedef enum{
    SPI_FLASH_RESULT_OK,
    SPI_FLASH_RESULT_ERR,
```

```
    SPI_FLASH_RESULT_TIMEOUT
}SpiFlashOpResult;
```

3.2.16. spi_flash_read

功能： 从 flash 读取数据。

说明：flash 读写操作的介绍，详见文档“Espressif IOT Flash 读写说明”。

函数定义：

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

参数：

uint32 des_addr - 读取 Flash 的地址，起始位置。

uint32 *src_addr – 存储读取到数据的指针。

Uint32 size - 读取数据长度

返回：

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

3.2.17. wifi_get_opmode

功能：获取 wifi 工作模式

函数定义：

```
uint8 wifi_get_opmode (void)
```

输入参数：

无

返回：

wifi 工作模式，其中 0x01 时为 STATION_MODE，0x02 时为 SOFTAP_MODE，0x03 时为 STATIONAP_MODE。

3.2.18. wifi_set_opmode

功能：设置 wifi 工作模式为 STATION、SOFTAP、STATION+SOFTAP

说明：在 esp_iot_sdk_v0.9.2 之前版本，本接口调用后，需要重启生效；v0.9.2（含）之后版本，无需重启即可生效。

函数定义：

bool wifi_set_opmode (uint8 opmode)

输入参数：

uint8 opmode——wifi 工作模式

其中 STATION_MODE 为 0x01，SOFTAP_MODE 为 0x02，STATIONAP_MODE 为 0x03。

返回：

True ，成功；False ，失败

3.2.19. wifi_station_get_config

功能：获取 wifi 的 station 接口参数

函数定义：

bool wifi_station_get_config (struct station_config *config)

输入参数：

struct station_config *config——wifi 的 station 接口参数指针

返回：

True ，成功；False ，失败

3.2.20. wifi_station_set_config

功能：设置 wifi 的 station 接口参数。

注意：如果在 user_init 中调用 wifi_station_set_config，底层会自动连接对应路由，不需要调用 wifi_station_connect 来进行连接。

函数定义：

bool wifi_station_set_config (struct station_config *config)

输入参数：

struct station_config *config——wifi 的 station 接口参数指针（结构体定义参见 4.2.1）

返回：

True ， 成功；False ， 失败

3.2.21. wifi_station_connect

功能：wifi 的 station 接口连接所配置的路由

注意：如果连接过路由，请先 wifi_station_disconnect，再调用 wifi_station_connect。

函数定义：

bool wifi_station_connect(void)

输入参数：

无

返回：

True ， 成功；False ， 失败

3.2.22. wifi_station_disconnect

功能：wifi 的 station 接口断开所连接的路由

函数定义：

bool wifi_station_disconnect(void)

输入参数：

无

返回：

True ， 成功；False ， 失败

3.2.23. wifi_station_get_connect_status

功能：获取 wifi station 接口连接 AP 的状态

函数定义：

uint8 wifi_station_get_connect_status (void)

输入参数：

无

返回：

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

3.2.24. wifi_station_scan

功能：获取 AP 热点信息

函数定义：

bool wifi_station_scan (struct scan_config *config, scan_done_cb_t cb);

结构体：

```
struct scan_config{
    uint8 *ssid;          // AP 的 ssid
    uint8 *bssid;         // AP 的 bssid
    uint8 channel;        //扫描某特定信道
};
```

输入参数：

struct scan_config *config – 扫描 AP 的相关参数，传 NULL 表示此项不设限。

例如，若 config 传 NULL，则扫描获取所有 AP 的信息；

若 config 中 ssid、bssid 传 NULL，仅设置 channel，则扫描该特定 channel 上的 AP 信息。

scan_done_cb_t cb - 获取 AP 热点信息回调 function

返回：

True ， 成功；False ， 失败

3.2.25. scan_done_cb_t

功能：scan 回调 function

函数定义：

void scan_done_cb_t (void *arg, STATUS status);

输入参数：

void *arg——获取的 AP 热点信息入口参数，arg 指针需要转换为 struct bss_info 结构体指针来解析所扫描的 AP 信息，AP 热点信息以链表形式储存（参见 4.2.3 中 struct bss_info 结构体定义）

STATUS status——获取结果

返回：

无

3.2.26. wifi_station_ap_number_set

功能：设置 ESP8266 station 最多可记录几个 AP 的信息

函数定义：

bool wifi_station_ap_number_set (uint8 ap_number);

输入参数：

uint8 ap_number—— 最多可记录 AP 的数目（MAX: 5）。

返回：

True ， 成功；False ， 失败

3.2.27. wifi_station_ap_change

功能：ESP8266 station 切换到第几号 AP 配置进行连接

函数定义：

bool wifi_station_ap_change (uint8 current_ap_id);

输入参数：

uint8 current_ap_id——第几号 AP 配置。从 0 开始计数。

返回：

True ，成功；False ，失败

3.2.28. wifi_station_get_current_ap_id

功能：获取 AP 热点信息

函数定义：

Uint8 wifi_station_get_current_ap_id ();

输入参数：

无

返回：

当前使用的 AP id，即当前 AP 为 ESP8266 记录的第几号配置信息。

3.2.29. wifi_softap_get_config

功能：设置 wifi 的 softap 接口参数

函数定义：

bool wifi_softap_get_config(struct softap_config *config)

输入参数：

struct softap_config *config——wifi 的 softap 接口参数指针（结构体定义参见 4.2.2）

返回：

True ，成功；False ，失败

3.2.30. wifi_softap_set_config

功能：设置 wifi 的 softap 接口参数

函数定义：

bool wifi_softap_set_config (struct softap_config *config)

输入参数：

struct softap_config *config——wifi 的 softap 接口参数指针（结构体定义参见 4.2.2）

返回：

True ，成功；False ，失败

3.2.31. wifi_softap_get_station_info

功能：获取 softap 模式下连接的 station 设备信息，包括 mac 和 ip

函数定义：

struct station_info * wifi_softap_get_station_info(void)

输入参数：

无

返回：

struct station_info *——所连接的 station 信息链表

3.2.32. wifi_softap_free_station_info

功能：释放由于调用 wifi_softap_get_station_info 函数生成的 struct station_info 空间

函数定义：

void wifi_softap_free_station_info (void)

输入参数：

无

返回：

无

获取 mac、ip 信息示例，注意释放资源：

方法一：

```
struct station_info * station = wifi_softap_get_station_info();
struct station_info * next_station;
while(station){
        os_printf("bssid : "MACSTR", ip : "IPSTR"\n", MAC2STR(station->bssid),
```

```
IP2STR(&station->ip));
            next_station = STAILQ_NEXT(station, next);
            os_free(station);     //直接释放
            station = next_station;
}
```

方法二：

```
struct station_info * station = wifi_softap_get_station_info();
while(station){
   os_printf("bssid : "MACSTR", ip : "IPSTR"\n", MAC2STR(station->bssid),
IP2STR(&station->ip));
   station = STAILQ_NEXT(station, next);
}
wifi_softap_free_station_info();  //调用函数释放
```

3.2.33. wifi_get_ip_info

功能：获取 wifi 的 station 或 softap 接口 ip 信息

注意：默认未连接情况下 station 模式的 ip 地址都为 0，因此可以使用此函数来判断本地是否成功连接 DHCP 模式的 Ap 或路由，另外 softap 模式下默认的 ip 地址为：192.168.4.1

函数定义：

bool wifi_get_ip_info(uint8 if_index, struct ip_info *info)

输入参数：

uint8 if_index——获取 ip 信息的接口，其中 STATION_IF 为 0x00，SOFTAP_IF 为 0x01。

struct ip_info *info——获取的指定接口的 ip 信息指针

返回：

True ，成功；False ，失败

3.2.34. wifi_set_ip_info

功能：修改 ip 地址

注意：仅在 user_init 中可以修改生效，不支持实时修改。

函数定义：

bool wifi_set_ip_info(uint8 if_index, struct ip_info *info)

输入参数：

uint8 if_index – 设置 station ip 还是 softAP ip

#define STATION_IF 0x00

#define SOFTAP_IF 0x01

struct ip_info *info – ip 信息

示例：

```
struct ip_info info;

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

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

返回：

True ， 成功；False ， 失败

3.2.35. wifi_set_macaddr

功能：设置 mac 地址

注意：仅在 user_init 中可以修改生效，不支持实时修改。

函数定义：

bool wifi_set_macaddr(uint8 if_index, uint8 *macaddr)

输入参数：

uint8 if_index – 设置 station ip 还是 softAP ip

#define STATION_IF 0x00

#define SOFTAP_IF 0x01

uint8 *macaddr – mac 地址

示例：

```
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);
```

返回：

True ，成功；False ，失败

3.2.36. wifi_get_macaddr

功能：获取 wifi 的 station 或 softap 接口 ip 信息

函数定义：

Bool wifi_get_macaddr(uint8 if_index , uint8 *macaddr)

输入参数：

uint8 if_index——获取 mac 信息的接口，其中 STATION_IF 为 0x00，SOFTAP_IF 为 0x01。

uint8 *macaddr——获取的指定接口的 mac 信息指针

返回：

True ，成功；False ，失败

3.2.37. wifi_status_led_install

功能：注册 wifi led 状态 led

函数定义：

Void wifi_status_led_install (uint8 gpio_id, uint32 gpio_name, uint8 gpio_func)

输入参数：

uint8 gpio_id——gpio 号

uint8 gpio_name——gpio mux 名

uint8 gpio_func——gpio 功能

返回：

无

示例：

使用 GPIO0 作为 wifi 状态 LED

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

3.2.38. wifi_promiscuous_enable

功能：使能混杂模式，用于 sniffer

函数定义：

Void wifi_promiscuous_enable(uint8 promiscuous)

输入参数：

uint8 promiscuous —— 0, disable promiscuous

1, enable promiscuous

返回：

无

示例：可向 Espressif 申请实现 sniffer 的 demo

3.2.39. wifi_set_promiscuous_rx_cb

功能：注册 wifi 混杂模式下的 call back 函数，每收到一包数据，都会进入注册的

回调函数。

函数定义：

Void wifi_set_promiscuous_rx_cb(wifi_promiscuous_cb_t cb)

输入参数：

wifi_promiscuous_cb_t cb——回调函数

返回：

无

3.2.40. wifi_get_channel

功能：用于 sniffer 功能，获取信道号

函数定义：

Uint8 wifi_get_channel (void)

输入参数：

无

返回：

信道号

3.2.41. wifi_set_channel

功能：用于 sniffer 功能，设置信道号

函数定义：

bool wifi_set_channel (uint8 channel)

输入参数：

uint8 channel——信道号

返回：

True ，成功；False ，失败

3.3. espconn 接口

说明：以下所有接口函数定义在（工程目录\include\espconn.h）

通用接口：TCP 和 UDP 均可以调用的接口。

TCP 连接接口：仅建立 TCP 连接时，使用的接口。

UDP 接口：仅收发 UDP 包时，使用的接口。

3.3.1. 通用接口

3.3.1.1. espconn_gethostbyname

功能：域名解析

函数定义：

Err_t espconn_gethostbyname(struct espconn *pespconn, const char *hostname, ip_addr_t *addr, dns_found_callback found)

输入参数：

struct espconn *espconn——相应连接的控制块结构

const char *hostname——域名 string 指针

ip_addr_t *addr——ip 地址

dns_found_callback found——回调

返回：

Err_t——ESPCONN_OK

ESPCONN_INPROGRESS

ESPCONN_ARG

示例如下，具体可参考 IoT Demo 代码：

```
ip_addr_t esp_server_ip;
LOCAL void ICACHE_FLASH_ATTR
user_esp_platform_dns_found(const char *name, ip_addr_t *ipaddr, void *arg)
{
    struct espconn *pespconn = (struct espconn *)arg;

    os_printf("user_esp_platform_dns_found %d.%d.%d.%d\n",
            *((uint8 *)&ipaddr->addr), *((uint8 *)&ipaddr->addr + 1),
            *((uint8 *)&ipaddr->addr + 2), *((uint8 *)&ipaddr->addr + 3));
```

```
}
Void dns_test(void)
{
espconn_gethostbyname(pespconn,”iot.espressif.cn”,&esp_server_ip,user_esp_platf
orm_dns_found);
}
```

3.3.1.2. espconn_port

功能：获取未使用的端口

函数定义：

uint32 espconn_port(void);

输入参数：

无

返回：

uint32——获取的端口号

3.3.1.3. espconn_regist_sentcb

功能：注册数据发送完成的回调函数，数据发送成功后回调

函数定义：

Sint8 espconn_regist_sentcb(struct espconn *espconn, espconn_sent_callback sent_cb)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_sent_callback sent_cb——注册的回调函数

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.3.1.4. espconn_regist_recvcb

功能：注册数据接收函数，收到数据时回调

函数定义：

Sint8 espconn_regist_recvcb(struct espconn *espconn, espconn_recv_callback recv_cb)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_connect_callback connect_cb——注册的回调函数

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.3.1.5. espconn_sent_callback

功能：数据发送结束回调

函数定义：

void espconn_sent_callback (void *arg)

输入参数：

void *arg——回调函数参数

返回：

无

3.3.1.6. espconn_recv_callback

功能：接收数据回调函数

函数定义：

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

输入参数：

void *arg——回调函数参数

char *pdata——接收数据入口参数

unsigned short len——接收数据长度

返回：

无

3.3.1.7. espconn_sent

功能：发送数据

函数定义：

Sint8 espconn_sent(struct espconn *espconn, uint8 *psent, uint16 length)

输入参数：

struct espconn *espconn——相应连接的控制块结构

uint8 *psent——sent 数据指针

uint16 length——sent 数据长度

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.3.2. TCP 连接接口

3.3.2.1. espconn_accept

功能：TCP 连接，侦听连接

函数定义：

Sin8 espconn_accept(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.3.2.2. espconn_secure_accept

功能：TCP 连接，加密侦听连接

函数定义：

Sint8 espconn_secure_accept(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.3.2.3. espconn_regist_time

功能：ESP8266 作为 TCP Server，设置超时断开连接的超时时长。

函数定义：

Sin8 espconn_regist_time(struct espconn *espconn, uint32 interval, uint8 type_flag)

输入参数：

struct espconn *espconn——相应连接的控制块结构

uint32 interval —— 超时时间，单位为秒，最大值为 7200 秒

uint8 type_flag —— 0，设置全部连接；1，设置单个连接

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.3.2.4. espconn_get_connection_info

功能：TCP 连接，针对多连接的情况，获得某个连接的信息

函数定义：

Sin8 espconn_get_connection_info (struct espconn *espconn, remot_info **pcon_info, uint8 typeflags)

输入参数：

struct espconn *espconn——相应连接的控制块结构

remot_info **pcon_info—— 连接 client 的信息

uint8 typeflags —— 0，普通 server；1，ssl server

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.3.2.5. espconn_connect

功能：建立 TCP 连接

函数定义：

Sint8 espconn_connect(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.3.2.6. espconn_connect_callback

功能：TCP 侦听或连接成功回调

函数定义：

Void espconn_connect_callback (void *arg)

输入参数：

void *arg——回调函数参数

返回：

无

3.3.2.7. espconn_disconnect

功能：断开 TCP 连接

函数定义：

Sin8 espconn_disconnect(struct espconn *espconn);

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.3.2.8. espconn_regist_connectcb

功能：注册 TCP 连接函数，成功连接时回调

函数定义：

Sint8 espconn_regist_connectcb(struct espconn *espconn, espconn_connect_callback connect_cb)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_connect_callback connect_cb——注册的回调函数

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.3.2.9. espconn_regist_reconcb

功能：注册 TCP 重连函数，出错重连时回调

函数定义：

Sint8 espconn_regist_reconcb(struct espconn *espconn, espconn_connect_callback recon_cb)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_connect_callback connect_cb——注册的回调函数

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.3.2.10. espconn_regist_disconcb

功能：注册断开 TCP 连接函数，断开连接成功时回调

函数定义：

Sint8 espconn_regist_disconcb(struct espconn *espconn, espconn_connect_callback discon_cb)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_connect_callback connect_cb——注册的回调函数

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.3.2.11. espconn_secure_connect

功能：建立加密 TCP 连接

函数定义：

Sint8 espconn_secure_connect (struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.3.2.12. espconn_secure_sent

功能：TCP 加密发送数据

函数定义：

Sint8 espconn_secure_sent (struct espconn *espconn, uint8 *psent, uint16 length)

输入参数：

struct espconn *espconn——相应连接的控制块结构

uint8 *psent——sent 数据指针

uint16 length——sent 数据长度

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.3.2.13. espconn_secure_disconnect

功能：加密断开 TCP 连接

函数定义：

Sint8 espconn_secure_disconnect(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.3.3. UDP 接口

3.3.3.1. espconn_create

功能：建立 UDP 传输

函数定义：

Sin8 espconn_create(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.3.3.2. espconn_delete

功能：删除 UDP 传输

函数定义：

Sin8 espconn_delete(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

0 - succeed，#define ESPCONN_OK 0

非 0 - Erro，详见 espconn.h

3.4. json API 接口

说明：jsonparse 相关接口函数或宏定义在（工程目录\include\json\jsonparse.h），另外 jsontree 相关接口函数或宏定义在（工程目录\include\json\jsontree.h）。

3.4.1. jsonparse_setup

功能：json 解析初始化

函数定义：

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

输入参数：

struct jsonparse_state *state——json 解析指针

const char *json——json 解析字符串

int len——字符串长度

返回：

无

3.4.2. jsonparse_next

功能：解析 json 格式下一个元素

函数定义：

Int jsonparse_next(struct jsonparse_state *state)

输入参数：

struct jsonparse_state *state——json 解析指针

返回：

int——解析结果

3.4.3. jsonparse_copy_value

功能：复制当前解析字符串到指定缓存

函数定义：

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

输入参数：

struct jsonparse_state *state——json 解析指针

char *str——缓存指针

int size——缓存大小

返回：

int——复制结果

3.4.4. jsonparse_get_value_as_int

功能：解析 json 格式为整形数据

函数定义：

Int jsonparse_get_value_as_int(struct jsonparse_state *state)

输入参数：

struct jsonparse_state *state——json 解析指针

返回：

int——解析数据

3.4.5. jsonparse_get_value_as_long

功能：解析 json 格式为长整形数据

函数定义：

Long jsonparse_get_value_as_long(struct jsonparse_state *state)

输入参数：

struct jsonparse_state *state——json 解析指针

返回：

long——解析数据

3.4.6. jsonparse_get_len

功能：解析 json 格式数据长度

函数定义：

Int jsonparse_get_value_len(struct jsonparse_state *state)

输入参数：

struct jsonparse_state *state——json 解析指针

返回：

int——解析的 json 格式数据长度

3.4.7. jsonparse_get_value_as_type

功能：解析 json 格式数据类型

函数定义：

Int jsonparse_get_value_as_type(struct jsonparse_state *state)

输入参数：

struct jsonparse_state *state——json 解析指针

返回：

int——json 格式数据类型

3.4.8. jsonparse_strcmp_value

功能：比较解析的 json 数据与特定字符串

函数定义：

Int jsonparse_strcmp_value(struct jsonparse_state *state, const char *str)

输入参数：

struct jsonparse_state *state——json 解析指针

const char *str——字符缓存

返回：

int——比较结果

3.4.9. jsontree_set_up

功能：生成 json 格式数据树

函数定义：

void jsontree_setup(struct jsontree_context *js_ctx,

struct jsontree_value *root, int (* putchar)(int))

输入参数：

struct jsontree_context *js_ctx——json 格式树元素指针

struct jsontree_value *root——根树元素指针

int (* putchar)(int)——输入函数

返回：

无

3.4.10. jsontree_reset

功能：设置 json 树

函数定义：

void jsontree_reset(struct jsontree_context *js_ctx)

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

返回：

无

3.4.11. jsontree_path_name

功能：json 树参数获取

函数定义：

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

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

int depth——json 格式树深度

返回：

char*——参数指针

3.4.12. jsontree_write_int

功能：整形数写入 json 树

函数定义：

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

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

int value——整形值

返回：

无

3.4.13. jsontree_write_int_array

功能：整形数数组写入 json 树

函数定义：

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

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

int *text——数组入口地址

uint32 length——数组长度

返回：

无

3.4.14. jsontree_write_string

功能：字符串写入 json 树

函数定义：

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

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

const char* text——字符串指针

返回：

无

3.4.15. jsontree_print_next

功能：json 树深度

函数定义：

int jsontree_print_next(struct jsontree_context *js_ctx)

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

返回：

int——json 树深度

3.4.16. jsontree_find_next

功能：查找 json 树元素

函数定义：

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

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

int——类型

返回：

struct jsontree_value *——json 格式树元素指针

4. 数据结构定义

4.1. 定时器结构

```
typedef void ETSTimerFunc(void *timer_arg);


typedef struct _ETSTIMER_ {
struct _ETSTIMER_       *timer_next;
      uint32_t                    timer_expire;
      uint32_t                    timer_period;
     ETSTimerFunc              *timer_func;
     void                        *timer_arg;
} ETSTimer;
```

4.2. wifi 参数

4.2.1. station 配置参数

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

4.2.2. softap 配置参数

```
typedef enum _auth_mode {
      AUTH_OPEN                = 0,
      AUTH_WEP,
      AUTH_WPA_PSK,
      AUTH_WPA2_PSK,
      AUTH_WPA_WPA2_PSK
```

```
} AUTH_MODE;
struct softap_config {
    uint8 ssid[32];
    uint8 password[64];
    uint8 channel;
    uint8 authmode;
    uint8 ssid_hidden;
    uint8 max_connection;
};
```

4.2.3. scan 参数

```
struct bss_info {
    STAILQ_ENTRY(bss_info)    next;
    u8 bssid[6];
    u8 ssid[32];
    u8 channel;
    s8 rssi;
    u8 authmode;
};
typedef void (* scan_done_cb_t)(void *arg, STATUS status);
```


4.3. json 相关结构

4.3.1. json 结构

```
struct jsontree_value {
uint8_t type;
};
```

```
struct jsontree_pair {
const char *name;
struct jsontree_value *value;
};

struct jsontree_context {
struct jsontree_value *values[JSONTREE_MAX_DEPTH];
uint16_t index[JSONTREE_MAX_DEPTH];
int (* putchar)(int);
uint8_t depth;
uint8_t path;
int callback_state;
};

struct jsontree_callback {
uint8_t type;
int (* output)(struct jsontree_context *js_ctx);
int (* set)(struct jsontree_context *js_ctx, struct jsonparse_state *parser);
};

struct jsontree_object {
uint8_t type;
uint8_t count;
struct jsontree_pair *pairs;
};

struct jsontree_array {
uint8_t type;
uint8_t count;
```

```
struct jsontree_value **values;
};

struct jsonparse_state {
const char *json;
int pos;
int len;
int depth;
int vstart;
int vlen;
char vtype;
char error;
char stack[JSONPARSE_MAX_DEPTH];
};
```

4.3.2. json 宏定义

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

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

```
sizeof(jsontree_value_##name)/sizeof(struct jsontree_value*),          \
    jsontree_value_##name }
```

4.4. espconn 参数

4.4.1 回调 function

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

4.4.2 espconn

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

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

/** Protocol family and type of the espconn */
enum espconn_type {
```

```
        ESPCONN_INVALID      = 0,
        /* ESPCONN_TCP Group */
        ESPCONN_TCP          = 0x10,
        /* ESPCONN_UDP Group */
        ESPCONN_UDP          = 0x20,
    };

    /** Current state of the espconn. Non-TCP espconn are always in state ESPCONN_NONE! */
    enum espconn_state {
        ESPCONN_NONE,
        ESPCONN_WAIT,
        ESPCONN_LISTEN,
        ESPCONN_CONNECT,
        ESPCONN_WRITE,
        ESPCONN_READ,
        ESPCONN_CLOSE
    };

    /** A espconn descriptor */
    struct espconn {
        /** type of the espconn (TCP, UDP) */
        enum espconn_type type;
        /** current state of the espconn */
        enum espconn_state state;
        union {
            esp_tcp *tcp;
            esp_udp *udp;
        } proto;
```

```
    /** A callback function that is informed about events for this espconn */
    espconn_recv_callback recv_callback;
    espconn_sent_callback sent_callback;
    espconn_handle esp_pcb;
    uint8 *ptrbuf;
    uint16 cntr;
};
```

5. 驱动接口

5.1. GPIO 接口 API

关于 gpio 接口 API 操作的具体应用，可参看\user\ user_plug.c。

5.1.1. PIN 脚功能设置宏

- ✓ PIN_PULLUP_DIS(PIN_NAME)

  管脚上拉屏蔽

- ✓ PIN_PULLUP_EN(PIN_NAME)

  管脚上拉使能

- ✓ PIN_PULLDWN_DIS(PIN_NAME)

  管脚下拉屏蔽

- ✓ PIN_PULLDWN_EN(PIN_NAME)

  管脚下拉使能

- ✓ PIN_FUNC_SELECT(PIN_NAME, FUNC)

  管脚功能选择

  例如：PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U, FUNC_GPIO12);

  选择 MTDI 管脚为复用 GPIO12。

5.1.2. gpio_output_set

功能：设置 gpio 口属性

函数定义：

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

输入参数：

uint32 set_mask——设置输出为高的位，对应位为 1，输出高，对应位为 0，不改变状态

uint32 clear_mask——设置输出为低的位，对应位为 1，输出低，对应位为 0，

不改变状态

uint32 enable_mask——设置使能输出的位

uint32 disable_mask——设置使能输入的位

返回：

无

例子：

- ✓ 设置 GPIO12 输出高电平，则：gpio_output_set(BIT12, 0, BIT12, 0);
- ✓ 设置 GPIO12 输出低电平，则：gpio_output_set(0, BIT12, BIT12, 0);
- ✓ 设置 GPIO12 输出高电平，GPIO13 输出低电平，则：gpio_output_set(BIT12, BIT13, BIT12|BIT13, 0);
- ✓ 设置 GPIO12 为输入，则 gpio_output_set(0, 0, 0, BIT12);

5.1.3. GPIO 输入输出相关宏

- ✓ GPIO_OUTPUT_SET(gpio_no, bit_value)

  设置 gpio_no 管脚输出 bit_value，同 5.1.2 例子中输出高低电平的功能。
- ✓ GPIO_DIS_OUTPUT(gpio_no)

  设置 gpio_no 管脚为输入，同 5.1.2 例子中输入。
- ✓ GPIO_INPUT_GET(gpio_no)

  获取 gpio_no 管脚的电平状态。

5.1.4. GPIO 中断控制相关宏

- ✓ ETS_GPIO_INTR_ATTACH(func, arg)

  注册 GPIO 中断处理函数。
- ✓ ETS_GPIO_INTR_DISABLE()

  关 GPIO 中断。
- ✓ ETS_GPIO_INTR_ENABLE()

  开 GPIO 中断。

5.1.5. gpio_pin_intr_state_set

功能：设置 gpio 脚中断触发状态

函数定义：

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

输入参数：

uint32 i——GPIO 管脚 ID，如需设置 GPIO14，则为 GPIO_ID_PIN(14)；

GPIO_INT_TYPE intr_state——中断触发状态

其中：

```
typedef enum{
    GPIO_PIN_INTR_DISABLE = 0,
    GPIO_PIN_INTR_POSEDGE= 1,
    GPIO_PIN_INTR_NEGEDGE= 2,
    GPIO_PIN_INTR_ANYEGDE=3,
    GPIO_PIN_INTR_LOLEVEL= 4,
    GPIO_PIN_INTR_HILEVEL = 5
}GPIO_INT_TYPE;
```

返回：

无

5.1.6. GPIO 中断处理函数

在 GPIO 中断处理函数内，需要做如下操作来清除响应位的中断状态：

```
uint32 gpio_status;
gpio_status = GPIO_REG_READ(GPIO_STATUS_ADDRESS);
//clear interrupt status
GPIO_REG_WRITE(GPIO_STATUS_W1TC_ADDRESS, gpio_status);
```

5.2. 双 UART 接口 API

默认情况下，UART0 作为系统的 debug 输出接口，当配置为双 UART 时，UART0 作为数据收发接口，UART1 作为 debug 输出接口。

使用时，请确保硬件连接正确。

5.2.1. uart_init

功能：双 uart 模式，两个 uart 波特率初始化

函数定义：

void uart_init(UartBautRate uart0_br, UartBautRate uart1_br)

输入参数：

UartBautRate uart0_br——uart0 波特率

UartBautRate uart1_br——uart1 波特率

其中：

```
typedef enum {
    BIT_RATE_9600    = 9600,
    BIT_RATE_19200   = 19200,
    BIT_RATE_38400   = 38400,
    BIT_RATE_57600   = 57600,
    BIT_RATE_74880   = 74880,
    BIT_RATE_115200 = 115200,
    BIT_RATE_230400 = 230400,
    BIT_RATE_460800 = 460800,
    BIT_RATE_921600 = 921600
} UartBautRate;
```

返回：

无

5.2.2. uart0_tx_buffer

功能：通过 UART0 发送用户自定义数据

函数定义：

Void uart0_tx_buffer(uint8 *buf, uint16 len)

输入参数：

Uint8 *buf——待发送数据

Uint16 len——待发送数据长度

返回：

无

5.2.3. uart0_rx_intr_handler

功能：UART0 中断处理函数，用户可在该函数内添加对接收到数据包的处理。（接收缓冲区大小为 0x100，如果接受数据大于 0x100，请自行处理）

函数定义：

Void uart0_rx_intr_handler(void *para)

输入参数：

Void*para——指向 RcvMsgBuff 结构的指针

返回：

无

5.3. i2c master 接口

5.3.1. i2c_master_gpio_init

功能：i2c master 模式时，对相应 GPIO 口进行设置

函数定义：

Void i2c_master_gpio_init (void)

输入参数：

无

返回：

无

5.3.2. i2c_master_init

功能：初始化 i2c 操作

函数定义：

Void i2c_master_init(void)

输入参数：

无

返回：

无

5.3.3. i2c_master_start

功能：设置 i2c 进入发送状态

函数定义：

Void i2c_master_start(void)

输入参数：

无

返回：

无

5.3.4. i2c_master_stop

功能：设置 i2c 进入停止发送状态

函数定义：

Void i2c_master_stop(void)

输入参数：

无

返回：

无

5.3.5. i2c_master_setAck

功能：设置 i2c ACK

函数定义：

Void i2c_master_setAck (uint8 level)

输入参数：

Uint8 level——ack 0 or 1

返回：

无

5.3.6. i2c_master_getAck

功能：获取 slave 的 ACK

函数定义：

Uint8 i2c_master_getAck (void)

输入参数：

无

返回：

uint8——0 or 1

5.3.7. i2c_master_readByte

功能：从 slave 读取一字节

函数定义：

Uint8 i2c_master_readByte (void)

输入参数：

无

返回：

uint8——读取到的值

5.3.8. i2c_master_writeByte

功能：向 slave 写一字节

函数定义：

Void i2c_master_writeByte (uint8 wrdata)

输入参数：

uint8 wrdata——待写数据

返回：

无

5.4. pwm

当前支持 4 路 PWM，可在 pwm.h 中对采用的 GPIO 口进行配置选择。

5.4.1. pwm_init

功能：pwm 功能初始化，包括 gpio，频率以及占空比

函数定义：

Void pwm_init(uint16 freq, uint8 *duty)

输入参数：

Uint16 freq——pwm 的频率；

uint8 *duty——各路的占空比

返回：

无

5.4.2. pwm_start

功能：pwm 开始，每次更新 pwm 数据后，都需要重新调用本接口进行计算。

函数定义：

Void pwm_start (void)

输入参数：

无

返回：

无

5.4.3. pwm_set_duty

功能：对某一路设置占空比

函数定义：

Void pwm_set_duty(uint8 duty, uint8 channel)

输入参数：

uint8 duty——占空比

uint8 channel——某路

返回：

无

5.4.4. pwm_set_freq

功能：设置 pwm 频率

函数定义：

Void pwm_set_freq(uint16 freq)

输入参数：

Uint16 freq——pwm 频率

返回：

无

5.4.5. pwm_get_duty

功能：获取某路的占空比

函数定义：

uint8 pwm_get_duty(uint8 channel)

输入参数：

uint8 channel——待获取占空比的 channel

返回：

uint8——占空比

5.4.6. pwm_get_freq

功能：获取 pwm 频率

函数定义：

Uint16 pwm_get_freq(void)

输入参数：

无

返回：

Uint16——频率

6. 无附录

A. ESPCONN 编程

A.1. client 模式

A.1.1. 说明

ESP8266 工作在 Station 模式下，确认已经分配到 IP 地址时，启用 client 连接。

ESP8266 工作在 softap 模式下，确认连接 ESP8266 的设备分配到 ip 地址，启用 client 连接。

A.1.2. 步骤

1) 依据工作协议初始化 espconn 参数。
2) 注册 connect 回调函数。udp 协议可以省略该步骤，注册 recv 回调函数。
3) 调用 espconn_connect 函数建立与 host 的连接。
4) 连接成功后将调用注册的 connect 函数，该函数中根据应用注册相应的回调函数，建议 disconnect 回调函数必须注册。udp 协议省略该步骤。
5) 在 recv 回调函数或 sent 回调函数执行 disconnect 操作时，建议适当延时一定时间，确保底层函数执行结束。

A.2. server 模式

A.2.1. 说明

ESP8266 工作在 Station 模式下，确认已经分配到 IP 地址，启用 server 侦听。

ESP8266 工作在 softap 模式下，启用 server 侦听。

A.2.2. 步骤

1) 依据工作协议初始化 espconn 参数。

2) 注册 connect 回调函数。udp 协议可以省略该步骤，注册 recv 回调函数。

3) 调用 espconn_accept 函数侦听 host 的连接。

连接成功后将调用注册的 connect 函数，该函数中根据应用注册相应的回调函数。udp 协议省略该步骤。