ESP8266 Non-OS SDK

API 参考

版本 3.0
乐鑫信息科技

关于本手册

本文档提供 ESP8266_NONOS_SDK 的 API 说明。

发布说明

日期	版本	发布说明
2016.03	V1.5.2	更新章节 3.2、A.5 和 3.3.37
2016.04	V1.5.3	新增章节 3.5.11 和 3.5.12 更新章节 3.5.67 和 3.7.9
2016.05	V1.5.4	新增章节 3.3.8、3.3.46、3.3.47、3.3.48 和 3.7.8 更新章节 3.7
2016.07	V2.0.0	更新章节 3.8.6 和 3.5.65 新增章节 3.9、3.14、3.3.48、3.5.72 和 3.5.73
2016.11	V2.0.1	将章节 3.5.30 中的函数定义 `wifi_station_get_hostname` 改为 `wifi_station_set_hostname`
2017.01	V2.0.2	更新第 2 章
2017.05	V2.1.0	新增章节 3.3.49，3.3.50，4.3.6 和 8.2.4
2017.05	V2.1.1	更新第 2 章
2017.06	V2.1.2	更新章节 3.3.9
2018.02	V2.2	更新章节 6.2.1，6.2.3，6.2.4，3.3.49 新增章节 3.4.8，3.4.9，3.5.74，3.5.75
2018.05	V2.2.1	更新章节 2.4，3.5.54，3.7
2018.08	V3.0	新增章节 3.3.50，3.3.51，3.3.52，3.5.76，3.5.77，3.5.78，3.5.79，附录 A.6 更新章节 2.5 删除 system_phy_freq_trace_enable

文档变更通知

用户可通过乐鑫官网订阅页面 *https://www.espressif.com/zh-hans/subscribe* 订阅技术文档变更的电子邮件通知。

证书下载

用户可通过乐鑫官网证书下载页面 *https://www.espressif.com/zh-hans/certificates* 下载产品证书。

目录

1. 前言

ESP8266EX 由乐鑫公司开发，提供了一套高度集成的 Wi-Fi SoC 解决方案，其低功耗、紧凑设计和高稳定性可以满足用户的需求。

ESP8266EX 拥有完整的且自成体系的 Wi-Fi 网络功能，既能够独立应用，也可以作为从机搭载于其他主机 MCU 运行。当 ESP8266EX 独立应用时，能够直接从外接 Flash 中启动。内置的高速缓冲存储器有利于提高系统性能，并且优化存储系统。此外 ESP8266EX 只需通过 SPI/SDIO 接口或 I2C/UART 口即可作为 Wi-Fi 适配器，应用到基于任何微控制器的设计中。

ESP8266EX 集成了天线开关、射频 balun、功耗放大器、低噪放大器、过滤器和电源管理模块。这样紧凑的设计仅需极少的外部电路并且将 PCB 的尺寸降到最小。

ESP8266EX 还集成了增强版的 Tensilica's L106 钻石系列 32-bit 内核处理器，带片上 SRAM。ESP8266EX 可以通过 GPIO 外接传感器和其他设备。软件开发包 (SDK) 提供了一些应用的示例代码。

乐鑫智能互联平台 (ESCP-Espressif Systems' Smart Connectivity Platform) 表现出来的领先特征有：睡眠/唤醒模式之间的快速切换以实现节能、配合低功耗操作的自适应射频偏置、前端信号的处理功能、故障排除和射频共存机制可消除蜂窝/蓝牙/DDR/LVDS/LCD 干扰。

基于 ESP8266EX 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. Non-OS SDK

2.1. Non-OS SDK 简介

Non-OS SDK 为用户提供了一套应用程序编程接口 (API)，能够实现 ESP8266 的核心功能改，例如数据接收/发送、TCP/IP 功能、硬件接口功能，以及基本的系统管理功能等。用户不必关心底层网络，如 Wi-Fi、TCP/IP 等的具体实现，只需要专注于物联网上层应用的开发，利用相应接口实现各种功能即可。

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。用户应用的初始化功能可以在 ***user_main.c*** 中实现。

`void user_init(void)` 是上层程序的入口函数，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。

- 对于 ***ESP8266_NONOS_SDK_v3.0.0*** 及之后版本，请在 ***user_main.c*** 增加函数 `void ICACHE_FLASH_ATTR user_pre_init(void)`，并且在 `user_pre_init` 中注册自己的 partition table。
- 对于 ***ESP8266_NONOS_SDK_v1.5.2*** 至 ***ESP8266_NONOS_SDK_v2.2.1*** 之间的版本，请在 ***user_main.c*** 增加函数 `void user_rf_pre_init(void)` 和 `uint32 user_rf_cal_sector_set(void)`，可参考 ***IOT_Demo*** 的 ***user_main.c***。用户可在 `user_rf_pre_init` 中配置 RF 初始化，RF 设置接口为 `system_phy_set_rfoption`，或者在 Deep-sleep 前调用 `system_deep_sleep_set_option`。如果设置为 RF 不打开，则 ESP8266 Station 及 SoftAP 均无法使用，请勿调用 Wi-Fi 相关接口及网络功能。RF 关闭时，Wi-Fi 射频功能和网络堆栈管理 API 均无法使用。

对于 ***ESP8266_NONOS_SDK_v2.1.0*** 及之后版本，用户如果并未使用 DIO-To-QIO flash，可以在 ***user_main.c*** 中增加空函数 `void user_spi_flash_dio_to_qio_pre_init(void)` 来优化 iRAM 空间。

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。

2.2. 代码结构

Non-OS SDK 适用于用户需要完全控制代码执行顺序的应用程序。由于没有操作系统，non-OS SDK 不支持任务调度，也不支持基于优先级的抢占。

Non-OS SDK 最适合用于事件驱动的应用程序。由于没有操作系统，non-OS SDK 没有单个任务堆栈大小的限制或者执行时隙要求。

而 RTOS SDK 可用于基于任务的模块化编程。要了解有关 RTOS SDK 的更多信息，请参阅《ESP8266 SDK 入门指南》。

Non-OS SDK 中的代码结构具有以下特征：

- Non-OS SDK 不像基于 RTOS 的应用程序支持任务调度。Non-OS SDK 使用四种类型的函数：
  - 应用函数
  - 回调函数
  - 用户任务
  - 中断服务程序 (Interrupt Service Routines, ISR)

  应用函数类似于嵌入式 C 编程中的常用 C 函数。这些函数必须由另一个函数调用。应用函数在定义时建议添加 `ICACHE_FLASH_ATTR` 宏，相应程序将存放在 flash 中，被调用时才加载到 cache 运行。而如果添加了 `IRAM_ATTR` 宏的函数，则会在上电启动时就加载到 iRAM 中。

  回调函数是指不直接从用户程序调用的函数，而是当某系统事件发生时，相应的回调函数由 non-OS SDK 内核调用执行。这使得开发者能够在不使用 RTOS 或者轮询事件的情况下响应实时事件。

  要编写回调函数，用户首先需要使用相应的 `register_cb` API 注册回调函数。回调函数的示例包括定时器回调函数和网络事件回调函数。

  中断服务程序 (ISR) 是一种特殊类型的回调函数。发生硬件中断时会调用这些函数。当使能中断时，必须注册相应的中断处理函数。请注意，ISR 必须添加 `IRAM_ATTR`。

  用户任务可以分为三个优先级：0、1、2。任务优先级为 2 > 1 > 0。即 Non-OS SDK 最多只支持 3 个用户任务，优先级分别为 0、1、2。

  用户任务一般用于函数不能直接被调用的情况下。要创建用户任务，请参阅本文档中的 `system_os_task()` 的 API 描述。例如，`espconn_disconnect()` API 不能直接在 `espconn` 的回调函数中调用，因此建议开发者可以在 `espconn` 回调中创建用户任务来执行 `espconn_disconnect`。

- 如前所述，non-OS SDK 不支持抢占任务或进程切换。因此开发者需要自行保证程序的正确执行，用户代码不能长期占用 CPU。否则会导致看门狗复位，ESP8266 重启。

  如果某些特殊情况下，用户线程必须执行较长时间（比如大于 500 ms），建议经常调用 `system_soft_wdt_feed()` API 来喂软件看门狗，而不建议禁用软件看门狗。

- 请注意，***esp_init_data.bin*** 和 ***blank.bin*** 文件至少需要烧录一次，以用于正确的初始化系统。对于 ***ESP8266_NONOS_SDK_v2.2.1*** 及之前的版本，应用程序必须在 `user_rf_cal_sector_set` 中设置 RF 校准扇区。

2.3. 定时器 (timer) 和中断

- 对于需要进行轮询的应用，建议使用系统定时器定期检查事件。
  - 如果使用循环（`while` 或 `for`），不仅效率低下，而且阻塞 CPU，不建议使用。
  - 如果需要在定时器回调中执行 `os_delay_us` 或 `while` 或 `for`，请勿占用 CPU 超过 15 ms。
- 请勿频繁调用定时器，建议频率不高于每 5 ms 一次（微秒计时器则为 100 µs）。有关定时器使用的详细信息，请参阅 `os_timer_arm()` 和相关的 API 说明。
- 微秒定时器不是很精确，请在回调中考虑 500 µs 的抖动。如需实现高精度的定时，可以参考驱动程序 (***driver_lib***) 使用硬件定时器。请注意，PWM API 不能与硬件定时器同时使用。
- 请勿长时间关闭中断。ISR 执行时间也应当尽可能短（即微秒级）。

2.4. 系统性能

- ESP8266 通常的运行速率为 80 MHz，在高性能应用中也可以配置为160 MHz。请注意，外设不受 CPU 频率设置的影响，因为它们使用了不同的时钟源。
- 设置更高的时钟频率或者禁用睡眠模式，会导致更大的功耗，但能获得更好的性能。应用程序应考虑两者之间的平衡。
- 添加了 `ICACHE_FLASH_ATTR` 的代码通常比使用 `IRAM_ATTR` 标记的代码执行得慢。然而，像大多数嵌入式平台一样，ESP8266 的 iRAM 空间有限，因此建议一般代码添加 `ICACHE_FLASH_ATTR`，仅对执行效率要求高的代码添加 `IRAM_ATTR` 宏。
- Flash 模式和频率直接影响代码执行速度。将 flash 设置为更高的频率和 QIO 模式会产生更好的性能，但会导致更大的功耗。

2.5. 系统存储

- ESP8266 支持高达 128 Mbits 的外部 QSPI flash，用于存储代码和数据。也可以使用辅助存储芯片来存储用户数据。
- ESP8266 没有存储用户代码或数据的非易失性存储。ESP8285 是一款在 ESP8266 的基础上集成了 flash 的芯片。更多详细信息请参考 ESP8285 技术规格书。

- ESP8266 带有 160 KB 的 RAM，其中 64 KB 为 iRAM，96 KB 为 dRAM。iRAM 进一步分成两块：32 KB iRAM 块运行标有 `IRAM_ATTR` 的代码，另一个 32 KB 块用作 cache，运行标有 `ICACHE_FLASH_ATTR` 的代码。
- 从 ESP8266_NonOS_SDK_V3.0 开始，增加了支持使用 iRAM 作为内存的功能，能够多提供约 17 KB 的内存， 对性能可能有一定的影响，请根据实际应用需求设置，并建议做详细测试进行确认。使用方法如下：
  - 在应用中定义 `user_iram_memory_is_enabled` 函数并设置返回值为 1。示例：

```
#define CONFIG_ENABLE_IRAM_MEMORY   1


#ifdef CONFIG_ENABLE_IRAM_MEMORY
uint32 user_iram_memory_is_enabled(void)
{
       return CONFIG_ENABLE_IRAM_MEMORY;
}
#endif
```

  - 如上设置后，默认使用 IRAM 作为内存，`os_malloc`、`os_zalloc` 和 `os_calloc` 优先从 iRAM 分配，iRAM 用尽后会继续使用 dRAM 分配；
  - 或者直接调用 `os_malloc_iram`、`os_zalloc_iram`、`os_calloc_iram` 指定从 iRAM 分配内存，iRAM 用尽后会继续使用 dRAM 分配；直接调用 `os_malloc_dram`、`os_zalloc_dram`、`os_calloc_dram` 指定从 dRAM 分配内存；
  - 如需与旧版本兼容，可使能宏 `MEM_DEFAULT_USE_DRAM`，`os_malloc`、`os_zalloc` 和 `os_calloc` 将从 dRAM 分配，而 `os_malloc_iram`、`os_zalloc_iram`、`os_calloc_iram` 可以指定从 iRAM 分配，iRAM 用尽后会继续使用 dRAM 分配。例如，在 makefile 中添加：

```
CONFIGURATION_DEFINES += -DMEM_DEFAULT_USE_DRAM
```

在 include/mem.h 中的具体定义如下：

```
#ifdef MEM_DEFAULT_USE_DRAM
#define os_malloc os_malloc_dram
#define os_zalloc os_zalloc_dram
#define os_calloc os_calloc_dram
#else
#define os_malloc os_malloc_iram
#define os_zalloc os_zalloc_iram
#define os_calloc os_calloc_iram
#endif
```

- RAM 和 flash 访问必须是 4 字对齐的，请勿直接进行指针转换。请使用 `os_memcpy` 或其他 API 进行内存操作。

3. 应用程序接口 (API)

3.1. 软件定时器

以下软件定时器接口位于 */ESP8266_NONOS_SDK/include/osapi.h*。请注意，以下接口使用的定时器由软件实现，定时器的函数在任务中被执行。因为任务可能被中断，或者被其他高优先级的任务延迟，因此以下 os_timer 系列的接口并不能保证定时器精确执行。

如果需要精确的定时，例如，周期性操作某 GPIO，请使用硬件中断定时器，具体可参考 *hw_timer.c*，硬件定时器的执行函数在中断里被执行。

注意：

- 对于同一个 timer，os_timer_arm 或 os_timer_arm_us 不能重复调用，必须先 os_timer_disarm。
- os_timer_setfn 必须在 timer 未使能的情况下调用，在 os_timer_arm 或 os_timer_arm_us 之前或者 os_timer_disarm 之后。

3.1.1. os_timer_arm

功能	使能毫秒级定时器
函数定义	`void os_timer_arm (` `  os_timer_t *ptimer,` `  uint32_t milliseconds,` `  bool repeat_flag` `)`
参数	• os_timer_t *ptimer：定时器结构 • uint32_t milliseconds：定时时间，单位：ms - 如未调用 system_timer_reinit，可支持范围 5 ~ 0x68D7A3 - 如调用了 system_timer_reinit，可支持范围 100 ~ 0x689D0 • bool repeat_flag：定时器是否重复
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 (os_timer_t *ptimer)`
参数	os_timer_t *ptimer：定时器结构
返回	无

3.1.3. os_timer_setfn

功能	设置定时器回调函数。使用定时器，必须设置回调函数。
函数定义	`void os_timer_setfn(` `  os_timer_t *ptimer,` `  os_timer_func_t *pfunction,` `  void *parg` `)`
参数	• `os_timer_t *ptimer`：定时器结构 • `os_timer_func_t *pfunction`：定时器回调函数 • `void *parg`：回调函数的参数
返回	无

3.1.4. system_timer_reinit

功能	重新初始化定时器，当需要使用微秒级定时器时调用
注意	• 同时定义 `USE_US_TIMER` • `system_timer_reinit` 在程序最开始调用，`user_init` 的第一句。
函数定义	`void system_timer_reinit (void)`
参数	无
返回	无

3.1.5. os_timer_arm_us

功能	使能微秒级定时器。
注意	• 请定义 `USE_US_TIMER`，并在 `user_init` 起始第一句，先调用 `system_timer_reinit`。 • 最高精度为 500 μs。
函数定义	`void os_timer_arm_us (` `  os_timer_t *ptimer,` `  uint32_t microseconds,` `  bool repeat_flag` `)`
参数	• `os_timer_t *ptimer`：定时器结构 • `uint32_t microseconds`：定时时间，单位：μs，最小定时 0x64 ，最大可输入 0xFFFFFFF • `bool repeat_flag`：定时器是否重复
返回	无

3.2. 硬件中断定时器

以下硬件中断定时器接口位于 */ESP8266_NONOS_SDK/examples/driver_lib/hw_timer.c*。用户可根据 *driver_lib* 文件夹下的 *readme.txt* 文件使用。

注意：

- 如果使用 NMI 中断源，且为自动填装的定时器，调用 `hw_timer_arm` 时参数 `val` 必须大于 100。
- 如果使用 NMI 中断源，那么该定时器将为最高优先级，可打断其他 ISR。
- 如果使用 FRC1 中断源，那么该定时器无法打断其他 ISR。
- `hw_timer.c` 的接口不能跟 PWM 驱动接口函数同时使用，因为二者共用了同一个硬件定时器。
- 硬件中断定时器的回调函数定义，请勿添加 `ICACHE_FLASH_ATTR` 宏。
- 使用 `hw_timer.c` 的接口，请勿调用 `wifi_set_sleep_type(LIGHT_SLEEP);` 将自动睡眠模式设置为 Light-sleep。因为 Light-sleep 在睡眠期间会停 CPU，停 CPU 期间不能响应 NMI 中断。

3.2.1. hw_timer_init

功能	初始化硬件 ISR 定时器
函数定义	`void hw_timer_init (` `  FRC1_TIMER_SOURCE_TYPE source_type,` `  u8 req` `)`
参数	• `FRC1_TIMER_SOURCE_TYPE source_type`：定时器的 ISR 源 - `FRC1_SOURCE`：使用 FRC1 中断源 - `NMI_SOURCE`：使用 NMI 中断源 • `u8 req` - 0：不自动填装； - 1：自动填装
返回	无

3.2.2. hw_timer_arm

功能	使能硬件中断定时器
函数定义	`void hw_timer_arm (uint32 val)`
参数	`uint32 val`：定时时间 自动填装模式： - 使用 FRC1 中断源 `FRC1_SOURCE`，取值范围：50 ~ 0x199999 μs； - 使用 NMI 中断源 `NMI_SOURCE`，取值范围：100 ~ 0x199999 μs； 非自动填装模式，取值范围：10 ~ 0x199999 μs
返回	无

3.2.3. hw_timer_set_func

功能	设置定时器回调函数。使用定时器，必须设置回调函数。
注意	回调函数前不能添加 ICACHE_FLASH_ATTR 宏定义，中断响应不能存放在 Flash 中。
函数定义	`void hw_timer_set_func (void (* user_hw_timer_cb_set)(void) )`
参数	`void (* user_hw_timer_cb_set)(void)`：定时器回调函数，函数定义时请勿添加 ICACHE_FLASH_ATTR 宏。
返回	无

3.2.4. 硬件定时器示例

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

    if( (WDEV_NOW() - tick_now2) >= 1000000 )
    {
    static u32 idx = 1;
        tick_now2 = WDEV_NOW();
        os_printf("b%u:%d\n",idx++,j);
        j = 0;
    }
}

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

3.3. 系统接口

系统接口位于 */ESP8266_NONOS_SDK/include/user_interface.h*。

os_XXX 系列接口位于 */ESP8266_NONOS_SDK/include/osapi.h*。

3.3.1. system_get_sdk_version

功能	查询 SDK 版本信息
函数定义	`const char* system_get_sdk_version(void)`
参数	无

返回	SDK 版本信息
示例	printf("SDK version: %s \n", system_get_sdk_version());

3.3.2. system_restore

功能	恢复出厂设置。本接口将清除以下接口的设置，恢复默认值： `wifi_station_set_auto_connect`、`wifi_set_phy_mode`、`wifi_softap_set_config` 相关，`wifi_station_set_config` 相关，`wifi_set_opmode` 以及 `#define AP_CACHE` 记录的 AP 信息。
注意	恢复出厂设置后，请务必重新启动 `system_restart`，再正常使用。
函数定义	`void system_restore(void)`
参数	无
返回	无

3.3.3. system_restart

功能	系统重启
注意	调用本接口后，ESP8266 模块并不会立刻重启，请勿在本接口之后调用其他功能接口。
函数定义	`void system_restart(void)`
参数	无
返回	无

3.3.4. system_init_done_cb

功能	在 `user_init` 中调用，注册系统初始化完成的回调函数。
注意	接口 `wifi_station_scan` 必须在系统初始化完成后，并且 Station 模式使能的情况下调用。
函数定义	`void system_init_done_cb(init_done_cb_t cb)`
参数	`init_done_cb_t cb`：系统初始化完成的回调函数
返回	无
示例	`void to_scan(void)  { wifi_station_scan(NULL,scan_done); }` `void user_init(void)  {` `  wifi_set_opmode(STATION_MODE);` `  system_init_done_cb(to_scan);` `}`

3.3.5. system_get_chip_id

功能	查询芯片 ID
函数定义	`uint32 system_get_chip_id (void)`
参数	无

返回	芯片 ID

3.3.6. system_get_vdd33

功能	测量 VDD3P3 管脚 3 和 4 的电压值，单位：1/1024V
注意	• system_get_vdd33 必须在 TOUT 管脚悬空的情况下使用。 • TOUT 管脚悬空的情况下，esp_init_data_default.bin (0 ~ 127 byte) 中的第 107 byte 为 vdd33_const，必须设为 0xFF，即 255。 • 不同 Wi-Fi 模式下，例如，Modem-sleep 模式或者普通 Wi-Fi 工作模式时，VDD33 的测量值会稍有差异。
函数定义	uint16 system_get_vdd33(void)
参数	无
返回	VDD33 电压值。单位：1/1024V

3.3.7. system_adc_read

功能	测量 TOUT 管脚 6 的输入电压，单位：1/1024V
注意	• system_adc_read 必须在 TOUT 管脚接外部电路情况下使用，TOUT 管脚输入电压范围限定为 0 ~ 1.0V。 • TOUT 管脚接外部电路的情况下，esp_init_data_default.bin (0 ~ 127 byte) 中的第 107 byte vdd33_const，必须设为 VDD3P3 管脚 3 和 4 上真实的电源电压，且必须小于 0xFF。 • 第 107 byte vdd33_const 的单位是 0.1V，有效取值范围是 [18, 36]；当 vdd33_const 处于无效范围 [0, 18) 或者 (36, 255) 时，使用默认值 3.3V 来优化 RF 电路工作状态。 • 不同 Wi-Fi 模式下，例如，Modem-sleep 模式或者普通 Wi-Fi 工作模式时，ADC 的测量值会稍有差异。 • 若需要高精度的 ADC，请使用 system_adc_read_fast 接口。
函数定义	uint16 system_adc_read(void)
参数	无
返回	TOUT 管脚 6 的输入电压，单位：1/1024V

3.3.8. system_adc_read_fast

功能	快速高精度的 ADC 采样。
注意	• 本接口必须在关闭 Wi-Fi 的状态下使用。如需进行连续测量 ADC，则还需要在关闭所有中断的状态下使用。因此，调用 `system_adc_read_fast` 时，不能使用 PWM 或者 NMI 类型的硬件定时器。 • 本接口必须在 TOUT 管脚接外部电路情况下使用，TOUT 管脚输入电压范围限定为 0 ~ 1.0V。 • TOUT 管脚接外部电路作为 ADC 输入时，`esp_init_data_default.bin` (0 ~ 127 byte) 中的 [107] byte `vdd33_const` 必须小于 0xFF。 • [107] byte `vdd33_const` 的具体用法如下： - [107] byte = 0XFF 时，内部测量 VDD33，TOUT 管脚不能作为外部 ADC 输入； - [107] byte 有效取值范围是 [18, 36] 时，单位是 0.1V，设置为实际的 VDD33 电源电压，优化 RF 电路工作状态，TOUT 管脚可以作为外部 ADC 输入； - [107] byte 有效取值范围是 [0, 18) 或者 (36, 255) 时，使用默认值 3.3V 作为电源电压来优化 RF 电路工作状态，TOUT 管脚可以作为外部 ADC 输入。
函数定义	`void system_adc_read_fast(uint16 *adc_addr, uint16 adc_num, uint8 adc_clk_div)`
参数	• `uint16 *adc_addr`：ADC 连续采样输出的地址指针。 • `uint16 adc_num`：ADC 连续采样的点数，输入范围 [1, 65535]。 • `uint8 adc_clk_div`：ADC 工作时钟 = 80M/adc_clk_div，输入范围 [8, 32]，推荐值为 8。
返回	无

示例	`extern void system_adc_read_fast(uint16 *adc_addr, uint16 adc_num, uint8 adc_clk_div);` `os_timer_t timer;` `void ICACHE_FLASH_ATTR ADC_TEST(void *p)` `{` `wifi_set_opmode(NULL_MODE);` `ets_intr_lock();  //close interrupt` `uint16 adc_addr[10];` `uint16 adc_num = 10;` `uint8 adc_clk_div = 8;` `uint32 i;` `system_adc_read_fast(adc_addr, adc_num, adc_clk_div);` `for(i=0; i<adc_num; i++)` `os_printf("i=%d, adc_v=%d\n", i, adc_addr[i]);` `ets_intr_unlock();  //open interrupt` `os_timer_disarm(&timer);` `os_timer_setfn(&timer, ADC_TEST, NULL);` `os_timer_arm(&timer,1000,1);` `}`

3.3.9. system_deep_sleep

功能	设置芯片进入 Deep-sleep 模式，休眠设定时间后自动唤醒，唤醒后程序从 `user_init` 重新运行。
注意	• 硬件需要将 XPD_DCDC 通过 0Ω 电阻连接到 EXT_RSTB，用作 Deep-sleep 唤醒。 • `system_deep_sleep(0)` 未设置唤醒定时器，可通过外部 GPIO 拉低 RST 脚唤醒。 • 本接口设置后，芯片并不会立刻进入 Deep-sleep，而是等待 Wi-Fi 底层功能安全关闭后，才进入 Deep-sleep 休眠。
函数定义	`bool system_deep_sleep(uint64 time_in_us)`
参数	`uint64 time_in_us`：休眠时间，单位：μs • 参数 `time_in_us` 的理论最大值可由公式 `(time_in_us/cali)<<12 = 2^31 -1` 计算。 - 其中 `cali = system_rtc_clock_cali_proc()`，表示 RTC 的时钟周期，bit11 ~ bit0 为小数部分，受温度或电源电压变化而偏移，并不精确，详细可参考 `system_rtc_clock_cali_proc` 函数说明。 • 由于计算并不精确，设置时传入的 `time_in_us` 值需小于理论最大值。
返回	True，设置成功 False，设置失败

3.3.10. system_deep_sleep_set_option

功能	设置下一次 Deep-sleep 唤醒后的行为，如需调用此 API，必须在 system_deep_sleep 之前调用。默认 option 为 1。
函数定义	bool system_deep_sleep_set_option(uint8 option)
参数	uint8 option：设置下一次 Deep-sleep 唤醒后的行为。 • 0：由 esp_init_data_default.bin (0~127 byte) 的 byte 108 和 Deep-sleep 的次数（deep_sleep_number，上电时初始化为 0）共同控制 Deep-sleep 唤醒后的行为，以每 (byte 108 + 1) 次 Deep-sleep 唤醒为周期循环。 - 若 deep_sleep_number <= byte 108，则 Deep-sleep 唤醒后不进行任何 RF_CAL，初始电流较小； - 若 deep_sleep_number = byte 108 + 1，则 Deep-sleep 唤醒后的行为与上电的行为一致，且将 deep_sleep_number 归零； • 1：Deep-sleep 唤醒后的行为与上电的行为一致； • 2：Deep-sleep 唤醒后不进行 RF_CAL，初始电流较小； • 4：Deep-sleep 唤醒后不打开 RF，与 Modem-sleep 行为一致，这样电流最小，但是设备唤醒后无法发送和接收数据。
返回	true：成功 false：失败

3.3.11. system_phy_set_rfoption

功能	设置此次 ESP8266 Deep-sleep 醒来，是否打开 RF。
注意	• 本接口只允许在 user_rf_pre_init 中调用。 • 本接口与 system_deep_sleep_set_option 功能相似，system_deep_sleep_set_option 在 Deep-sleep 前调用，本接口在 Deep-sleep 醒来初始化时调用，以本接口设置为准。 • 调用本接口前，要求至少调用过一次 system_deep_sleep_set_option。
函数定义	void system_phy_set_rfoption(uint8 option)

参数	`uint8 option`：设置下一次 Deep-sleep 唤醒后的行为。 • 0：由 `esp_init_data_default.bin` (0~127 byte) 的 byte 108 和 Deep-sleep 的次数（`deep_sleep_number`，上电时初始化为 0）共同控制 Deep-sleep 唤醒后的行为，以每 (byte 108 + 1) 次 Deep-sleep 唤醒为周期循环。 - 若 `deep_sleep_number` <= byte 108，则 Deep-sleep 唤醒后不进行任何 `RF_CAL`，初始电流较小； - 若 `deep_sleep_number` = byte 108 + 1，则 Deep-sleep 唤醒后的行为与上电的行为一致，且将 `deep_sleep_number` 归零； • 1：Deep-sleep 唤醒后的行为与上电的行为一致； • 2：Deep-sleep 唤醒后不进行 `RF_CAL`，初始电流较小； • 4：Deep-sleep 唤醒后不打开 RF，与 Modem-sleep 行为一致，这样电流最小，但是设备唤醒后无法发送和接收数据。
返回	无

3.3.12. system_phy_set_powerup_option

功能	设置上电时 RF 初始化的行为，默认为 option 0。
函数定义	`void system_phy_set_powerup_option(uint8 option)`
参数	`uint8 option`：power up 时，RF 初始化的行为 • 0：由 `esp_init_data_default.bin` (0 ~ 127 byte) byte 114 控制 RF 初始化行为，详细可参考 ESP8266 SDK 入门指南。 • 1：RF 初始化仅做 VDD33 和 TX power CAL，耗时约 18 ms，初始电流较小。 • 2：RF 初始化仅做 VDD33 校准，耗时约 2 ms，初始电流最小。 • 3：RF 初始化进行全部 RF CAL，耗时约 200 ms，初始电流较大。
返回	无

3.3.13. system_phy_set_max_tpw

功能	设置 RF TX Power 最大值，单位：0.25 dBm
函数定义	`void system_phy_set_max_tpw(uint8 max_tpw)`
参数	`uint8 max_tpw`：RF Tx Power 的最大值，可参考 `esp_init_data_default.bin` (0 ~ 127 byte) 的第 34 byte `target_power_qdb_0` 设置，单位：0.25 dBm，参数范围 [0, 82]
返回	无

3.3.14. system_phy_set_tpw_via_vdd33

功能	根据改变的 VDD33 电压值，重新调整 RF TX Power，单位：1/1024V
注意	• 在 TOUT 管脚悬空的情况下，VDD33 电压值可通过 `system_get_vdd33` 测量获得。 • 在 TOUT 管脚接外部电路情况下，不可使用 `system_get_vdd33` 测量 VDD33 电压值。

函数定义	`void system_phy_set_tpw_via_vdd33(uint16 vdd33)`
参数	`uint16 vdd33`：重新测量的 VDD33 值，单位：1/1024V，有效值范围：[1900, 3300]
返回	无

3.3.15. system_set_os_print

功能	开关打印 log 功能
函数定义	`void system_set_os_print (uint8 onoff)`
参数	`uint8 onoff` • 0：打印功能关； • 1：打印功能开
默认值	打印功能开
返回	无

3.3.16. 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)`
参数	无
返回	无

3.3.17. system_get_free_heap_size

功能	查询系统剩余可用 heap 区空间大小
函数定义	`uint32 system_get_free_heap_size(void)`
参数	无
返回	`uint32`：可用 heap 空间大小

3.3.18. system_os_task

功能	创建系统任务，最多支持创建 3 个任务，优先级分别为 0/1/2
函数定义	`bool system_os_task(` `  os_task_t  task,` `  uint8  prio,` `  os_event_t  *queue,` `  uint8  qlen` `)`
参数	• `os_task_t task`：任务函数 • `uint8 prio`：任务优先级，可为 0/1/2；0 为最低优先级。这表示最多只支持建立 3 个任务 • `os_event_t *queue`：消息队列指针 • `uint8 qlen`：消息队列深度

返回	true：成功 false：失败
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) {` `  testQueue=(os_event_t *)os_malloc(sizeof(os_event_t)*TEST_QUEUE_LEN);` `  system_os_task(test_task,USER_TASK_PRIO_0,testQueue,TEST_QUEUE_LEN);` `}`

3.3.19. system_os_post

功能	向任务发送消息
函数定义	`bool system_os_post (` `  uint8 prio,` `  os_signal_t sig,` `  os_param_t par` `)`
参数	• `uint8 prio`：任务优先级，与建立时的任务优先级对应。 • `os_signal_t sig`：消息类型 • `os_param_t par`：消息参数
返回	true：成功 false：失败
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`

3.3.20. system_get_time

功能	查询系统时间，单位：μs
函数定义	`uint32 system_get_time(void)`
参数	无
返回	系统时间，单位：μs。

3.3.21. system_get_rtc_time

功能	查询 RTC 时间，单位：RTC 时钟周期
示例	例如 `system_get_rtc_time` 返回 10（表示 10 个 RTC 周期）， `system_rtc_clock_cali_proc` 返回 5.75（表示 1 个 RTC 周期为 5.75 μs），则实际时间为 10 x 5.75 = 57.5 μs。
注意	`system_restart` 时，系统时间归零，但是 RTC 时间仍然继续。但是如果外部硬件通过 EXT_RST 脚或者 CHIP_EN 脚，将芯片复位后（包括 Deep-sleep 定时唤醒的情况），RTC 时钟会复位。具体如下： • 外部复位 EXT_RST：RTC memory 不变，RTC timer 寄存器从零计数 • 看门狗复位：RTC memory 不变，RTC timer 寄存器不变 • `system_restart`：RTC memory 不变，RTC timer 寄存器不变 • 电源上电：RTC memory 随机值，RTC timer 寄存器从零计数 • CHIP_EN 复位：RTC memory 随机值，RTC timer 寄存器从零计数
函数定义	`uint32 system_get_rtc_time(void)`
参数	无
返回	RTC 时间

3.3.22. system_rtc_clock_cali_proc

功能	查询 RTC 时钟周期
注意	• RTC 时钟周期含有小数部分。 • RTC 时钟周期会随温度或电源电压变化发生偏移，因此 RTC 时钟适用于在精度可接受的范围内进行计时，建议最多每分钟调用一次即可。
函数定义	`uint32 system_rtc_clock_cali_proc(void)`
参数	无
返回	RTC 时钟周期，单位：μs，bit11 ~ bit0 为小数部分
示例	`os_printf( “clk cal : %d \r\n” ,system_rtc_clock_cali_proc()>>12);` 详细 RTC 示例请见附录。

3.3.23. system_rtc_mem_write

功能	由于 Deep-sleep 时，仅 RTC 仍在工作，用户如有需要，可将数据存入 RTC memory 中。提供如下图中的 user data 段共 512 bytes 供用户存储数据。 `|<--------system data--------->|<-----------------user data--------------->|` `|  256 bytes  |  512 bytes  |`
注意	RTC memory 只能 4 字节整存整取，函数中参数 `des_addr` 为 block number，每 block 4 字节，因此若写入上图 user data 区起始位置，`des_addr` 为 256/4 = 64，`save_size` 为存入数据的字节数。

函数定义	`bool system_rtc_mem_write (` `  uint32 des_addr,` `  void * src_addr,` `  uint32 save_size` `)`
参数	• `uint32 des_addr`：写入 rtc memory 的位置，des_addr >=64 • `void * src_addr`：数据指针 • `uint32 save_size`：数据长度，单位：字节
返回	`true`：成功 `false`：失败

3.3.24. system_rtc_mem_read

功能	读取 RTC memory 中的数据，提供如下图中 user data 段共 512 bytes 给用户存储数据。 `|<--------system data--------->|<-----------------user data--------------->|` `|  256 bytes  |  512 bytes  |`
注意	RTC memory 只能 4 字节整存整取，函数中参数 `des_addr` 为 block number，每 block 4 字节，因此若写入上图 user data 区起始位置，`des_addr` 为 256/4 = 64，`save_size` 为存入数据的字节数。
函数定义	`bool system_rtc_mem_read (` `  uint32 src_addr,` `  void * des_addr,` `  uint32 save_size` `)`
参数	• `uint32 des_addr`：写入 rtc memory 的位置，des_addr >=64 • `void * src_addr`：数据指针 • `uint32 save_size`：数据长度，单位：字节
返回	`true`：成功 `false`：失败

3.3.25. system_uart_swap

功能	UART0 转换。将 MTCK 作为 UART0 RX，MTDO 作为 UART0 TX。硬件上也从 MTDO (U0RTS) 和 MTCK (U0CTS) 连出 UART0，从而避免上电时从 UART0 打印出 ROM log。
函数定义	`void system_uart_swap (void)`
参数	无
返回	无

3.3.26. system_uart_de_swap

功能	取消 UART0 转换，仍然使用原有 UART0，而不是将 MTCK、MTDO 作为 UART0。
函数定义	`void system_uart_de_swap (void)`

参数	无
返回	无

3.3.27. system_get_boot_version

功能	读取 boot 版本信息
函数定义	`uint8 system_get_boot_version (void)`
参数	无
返回	boot 版本信息
注意	如果 boot 版本号 >= 3 时，支持 boot 增强模式，详见 `system_restart_enhance`。

3.3.28. system_get_userbin_addr

功能	读取当前正在运行的 user bin（***user1.bin*** 或者 ***user2.bin***）的存放地址。
函数定义	`uint32 system_get_userbin_addr (void)`
参数	无
返回	正在运行的 user bin 的存放地址

3.3.29. system_get_boot_mode

功能	查询 boot 模式
函数定义	`uint8 system_get_boot_mode (void)`
参数	无
返回	`#define SYS_BOOT_ENHANCE_MODE 0` `#define SYS_BOOT_NORMAL_MODE  1`
注意	boot 增强模式：支持跳转到任意位置运行程序； boot 普通模式：仅能跳转到固定的 ***user1.bin***（或 ***user2.bin***）位置运行。

3.3.30. system_restart_enhance

功能	重启系统，进入 boot 增强模式。
函数定义	`bool system_restart_enhance(` `  uint8 bin_type,` `  uint32 bin_addr` `)`
参数	• `uint8 bin_type`：bin 类型 - `#define SYS_BOOT_NORMAL_BIN  0  // user1.bin 或者 user2.bin` - `#define SYS_BOOT_TEST_BIN  1  // 向乐鑫申请的 test bin` • `uint32 bin_addr`：bin 的起始地址
返回	`true`：成功 `false`：失败

注意	***SYS_BOOT_TEST_BIN*** 用于量产测试，用户可以向乐鑫申请获得。

3.3.31. system_update_cpu_req

功能	设置 CPU 频率。默认为 80 MHz
注意	系统总线时钟频率始终为 80 MHz，不受 CPU 频率切换的影响。UART、SPI 等外设频率由系统总线时钟分频而来，因此也不受 CPU 频率切换的影响。
函数定义	`bool system_update_cpu_freq(uint8 freq)`
参数	`uint8 freq`：CPU 频率 `#define SYS_CPU_80MHz  80` `#define SYS_CPU_160MHz 160`
返回	`true`：成功 `false`：失败

3.3.32. system_get_cpu_freq

功能	查询 CPU 频率
函数定义	`uint8 system_get_cpu_freq(void)`
参数	无
返回	CPU 频率，单位：MHz

3.3.33. system_get_flash_size_map

功能	查询当前的 Flash size 和 Flash map Flash map 对应编译时的选项，详细介绍请参考 ESP8266 SDK 入门指南。
结构体	`enum flash_size_map {` `  FLASH_SIZE_4M_MAP_256_256 = 0,` `  FLASH_SIZE_2M,` `  FLASH_SIZE_8M_MAP_512_512,` `  FLASH_SIZE_16M_MAP_512_512,` `  FLASH_SIZE_32M_MAP_512_512,` `  FLASH_SIZE_16M_MAP_1024_1024,` `  FLASH_SIZE_32M_MAP_1024_1024,` `  FLASH_SIZE_64M_MAP_1024_1024,` `  FLASH_SIZE_128M_MAP_1024_1024,` `};`
函数定义	`enum flash_size_map system_get_flash_size_map(void)`
参数	无
返回	flash map

3.3.34. system_get_rst_info

功能	查询当前启动的信息
结构体	`enum rst_reason {` `REANSON_DEFAULT_RST = 0, // normal startup by power on` `REANSON_WDT_RST = 1, // hardware watch dog reset` `// exception reset, GPIO status won't change` `REANSON_EXCEPTION_RST = 2,` `// software watch dog reset, GPIO status won't change` `REANSON_SOFT_WDT_RST = 3,` `// software restart ,system_restart , GPIO status won't change` `REANSON_SOFT_RESTART = 4,` `REANSON_DEEP_SLEEP_AWAKE = 5, // wake up from deep-sleep` `REANSON_EXT_SYS_RST = 6, // external system reset` `};` `struct rst_info {` `uint32 reason; // enum rst_reason` `uint32 exccause;` `uint32 epc1; // the address that error occurred` `uint32 epc2;` `uint32 epc3;` `uint32 excvaddr;` `uint32 depc;` `};`
函数定义	`struct rst_info* system_get_rst_info(void)`
参数	无
返回	启动的信息

3.3.35. system_soft_wdt_stop

功能	关闭软件看门狗
注意	请勿将软件看门狗关闭太长时间（小于 5s），否则将触发硬件看门狗复位
函数定义	`void system_soft_wdt_stop(void)`
参数	无
返回	无

3.3.36. system_soft_wdt_restart

功能	重启软件看门狗
注意	仅支持在软件看门狗关闭 `system_soft_wdt_stop` 的情况下，调用本接口

函数定义	`void system_soft_wdt_restart(void)`
参数	无
返回	无

3.3.37. system_soft_wdt_feed

功能	喂软件看门狗
注意	仅支持在软件看门狗开启的情况下，调用本接口
函数定义	`void system_soft_wdt_feed(void)`
参数	无
返回	无

3.3.38. system_show_malloc

功能	打印目前所分配的堆空间所有内存块，包括分配该内存块的文件名、行号和分配大小。在怀疑有内存泄露时，可以调用本接口查看当前内存状态。
注意	• 在 *user_config.h* 定义 `#define MEMLEAK_DEBUG`。参考 *ESP8266_NONOS_SDK\included\mem.h* 开始位置的注释使用。 • 泄露的内存一般在打印结果中，但打印结果中的内存不保证一定是泄露的内存。 • 本接口仅用于调试，无法确保使用本接口后，程序能继续正常执行，因此请勿在正常运行情况下，调用本接口。
函数定义	`void system_show_malloc(void)`
参数	无
返回	无

3.3.39. os_memset

功能	封装 C 语言函数，在一段内存块中填充某个给定值。
函数定义	`os_memset(void *s, int ch, size_t n)`
参数	• `void *s`：内存块指针 • `int ch`：填充值 • `size_t n`：填充大小
返回	无
示例	`uint8 buffer[32];` `os_memset(buffer, 0, sizeof(buffer));`

3.3.40. os_memcpy

功能	封装 C 语言函数，内存拷贝。
函数定义	`os_memcpy(void *des, void *src, size_t n)`
参数	• `void *des`：目标内存块指针 • `void *src`：源内存块指针 • `size_t n`：拷贝内存大小
返回	无
示例	`uint8 buffer[4] = {0};` `os_memcpy(buffer, "abcd", 4);`

3.3.41. os_strlen

功能	封装 C 语言函数，计算字符串长度。
函数定义	`os_strlen(char *s)`
参数	`char *s`：字符串
返回	字符串长度
示例	`char *ssid = "ESP8266";` `os_memcpy(softAP_config.ssid, ssid, os_strlen(ssid));`

3.3.42. os_printf

功能	格式化输出，打印字符串。
注意	• 本接口默认从 UART 0 打印。IOT_Demo 中的 `uart_init` 可以设置波特率，将 `os_printf` 改为从 UART 1 打印：`os_install_putc1((void *)uart1_write_char);` • 请勿调用本接口打印超过 125 字节的数据，或者频繁连续调用本接口打印，否则可能会丢失部分待打印数据。
函数定义	`void os_printf(const char *s)`
参数	const `char *s`：字符串
返回	无
示例	`os_printf("SDK version: %s \n", system_get_sdk_version());`

3.3.43. os_bzero

功能	置字符串 p 的前 n 个字节为零且包含 `\0`
函数定义	`void os_bzero(void *p, size_t n)`
参数	• `void *p`：要置零的数据的起始地址 • `size_t n`：要置零的数据字节数

返回	无

3.3.44. os_delay_us

功能	延时函数。最大值 65535 μs
函数定义	`void os_delay_us(uint16 us)`
参数	`uint16 us`：延时时间
返回	无

3.3.45. os_install_putc1

功能	注册打印接口函数
函数定义	`void os_install_putc1(void(*p)(char c))`
参数	`void(*p)(char c)`：打印接口函数指针
返回	无
示例	参考 ***UART.c***，`uart_init` 中的 `os_install_putc1((void *)uart1_write_char)` 将 `os_printf` 改为从 UART 1 打印。否则，`os_printf` 默认从 UART 0 打印。

3.3.46. os_random

功能	获取随机数
函数定义	`unsigned long os_random(void)`
参数	无
返回	随机数

3.3.47. os_get_random

功能	获取指定长度的随机数
函数定义	`int os_get_random(unsigned char *buf, size_t len)`
参数	• `unsigned char *buf`：获得的随机数 • `size_t len`：随机数的字节长度
返回	`true`：成功 `false`：失败
示例	`int ret = os_get_random((unsigned char *)temp, 7);` `os_printf("ret %d, value 0x%08x%08x\n\r", ret, temp[1], temp[0]);`

3.3.48. user_rf_cal_sector_set

功能	用户自定义 `RF_CAL` 参数存放在 Flash 的扇区号

注意	• 用户必须在程序中实现此函数，否则编译链接时会报错。但用户程序无需调用此函数，SDK 底层会调用它，将 RF_CAL 参数保存在用户指定的 Flash 扇区里，这将占用用户参数区的一个扇区。 • SDK 预留的 4 个扇区的系统参数区已经使用，因此 RF_CAL 参数需要占用到用户参数区的空间，由用户通过此函数设置一个可用扇区供 SDK 底层使用。 • 建议整个系统需要初始化时，或需要重新进行 RF_CAL 时，烧录 ***blank.bin*** 初始化 RF_CAL 参数区，并烧录 ***esp_init_data.bin***。注意，***esp_init_data.bin*** 至少需要烧录一次。
函数定义	`uint32 user_rf_cal_sector_set(void)`
参数	无
返回	存储 RF_CAL 参数的 Flash 扇区号
示例	将 RF 参数设置存放在 Flash 倒数第 5 个扇区

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

    switch (size_map) {
        case FLASH_SIZE_4M_MAP_256_256:
            rf_cal_sec = 128 - 5;
            break;

        case FLASH_SIZE_8M_MAP_512_512:
            rf_cal_sec = 256 - 5;
            break;

        case FLASH_SIZE_16M_MAP_512_512:
        case FLASH_SIZE_16M_MAP_1024_1024:
            rf_cal_sec = 512 - 5;
            break;

        case FLASH_SIZE_32M_MAP_512_512:
        case FLASH_SIZE_32M_MAP_1024_1024:
            rf_cal_sec = 512 - 5;
            break;

        case FLASH_SIZE_64M_MAP_1024_1024:
            rf_cal_sec = 2048 - 5;
            break;

        case FLASH_SIZE_128M_MAP_1024_1024:
            rf_cal_sec = 4096 - 5;
            break;

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

3.3.49. system_deep_sleep_instant

功能	设置芯片立刻进入 Deep-sleep 模式，休眠设定时间后自动唤醒，唤醒后程序从 `user_init` 重新运行。
注意	• 硬件需要将 XPD_DCDC 通过 0Ω 电阻连接到 EXT_RSTB，用作 Deep-sleep 唤醒。 • `system_deep_sleep_instant(0)` 未设置唤醒定时器，可通过外部 GPIO 拉低 RST 脚唤醒。 • 本接口设置后，芯片立刻进入 Deep-sleep 休眠，不会等待 Wi-Fi 底层功能安全关闭。如需等待 Wi-Fi 安全关闭，可使用接口 `system_deep_sleep`。
函数定义	`bool system_deep_sleep_instant(uint64 time_in_us)`
参数	`uint64 time_in_us`：休眠时间，单位：μs • 参数 `time_in_us` 的理论最大值可由公式 `(time_in_us/cali)<<12 = 2^32 -1` 计算。 - 其中 `cali = system_rtc_clock_cali_proc()`，表示 RTC 的时钟周期，bit11 ~ bit0 为小数部分，受温度或电源电压变化而偏移，并不精确，详细可参考 `system_rtc_clock_cali_proc` 函数说明。 • 由于计算并不精确，设置时传入的 `time_in_us` 值需小于理论最大值。
返回	True，设置成功 False，设置失败

3.3.50. system_partition_table_regist

功能	注册 partition table。
注意	• 本接口必须在 `user_pre_init` 中调用注册，如果注册失败，请检查 partition table 的定义。 • 示例可参考 ESP8266_NONOS_SDK/examples/IoT_Demo/user/user_main.c。
函数定义	`bool system_partition_table_regist(` `  const partition_item_t* partition_table,` `  uint32_t partition_num,` `  uint32_t map` `)`
参数	`const partition_item_t* partition_table`：分区表 `uint32_t partition_num`：分区数目 `uint32_t map`：flash map；必须与编译烧录时选择的 flash map 一致，否则将会启动异常；建议直接传入宏 `SPI_FLASH_SIZE_MAP`，它是系统在编译时记录的 flash map 值。
返回	True，partition table 注册成功 False，partition table 注册失败

3.3.51. system_partition_get_ota_partition_size

功能	查询 ota partition 的大小。

注意	ota partition 是用于存放 user1.bin 或者 user2.bin 的 flash 分区。
函数定义	`uint32_t system_partition_get_ota_partition_size(void)`
参数	-
返回	ota partition 的大小

3.3.52. system_partition_get_item

功能	查询指定类型的 partition 信息。
函数定义	`bool system_partition_get_item(partition_type_t type, partition_item_t* partition_item)`
参数	`partition_type_t type`：分区类型 `partition_item_t* partition_item`：查询到的分区信息
返回	True，查询成功 False，查询失败

3.4. SPI Flash 接口

SPI Flash 接口位于 */ESP8266_NONOS_SDK/include/spi_flash.h*。

`system_param_xxx` 接口位于 */ESP8266_NONOS_SDK/include/user_interface.h*。

关于 SPI Flash 读写操作，详见文档 ESP8266 Flash 读写说明。

3.4.1. spi_flash_get_id

功能	查询 SPI Flash 的 ID
函数定义	`uint32 spi_flash_get_id (void)`
参数	无
返回	`spi flash id`

3.4.2. spi_flash_erase_sector

功能	擦除 Flash 扇区
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 4 KB
返回	`typedef enum{` `  SPI_FLASH_RESULT_OK,` `  SPI_FLASH_RESULT_ERR,` `  SPI_FLASH_RESULT_TIMEOUT` `} SpiFlashOpResult;`

3.4.3. spi_flash_write

功能	写入数据到 Flash。Flash 读写必须 4 字节对齐。
函数定义	`SpiFlashOpResult spi_flash_write (` `  uint32 des_addr,` `  uint32 *src_addr,` `  uint32 size` `)`
参数	• `uint32 des_addr`：写入 Flash 目的地址 • `uint32 *src_addr`：写入数据的指针 • `uint32 size`：数据长度，单位 byte，必须 4 字节对齐进行读写
返回	`typedef enum{` `  SPI_FLASH_RESULT_OK,` `  SPI_FLASH_RESULT_ERR,` `  SPI_FLASH_RESULT_TIMEOUT` `} SpiFlashOpResult;`

3.4.4. spi_flash_read

功能	从 Flash 读取数据。Flash 读写必须 4 字节对齐。
函数定义	`SpiFlashOpResult spi_flash_read(` `  uint32 src_addr,` `  uint32 * des_addr,` `  uint32 size` `)`
参数	• `uint32 des_addr`：写入 Flash 目的地址 • `uint32 *des_addr`： 存放读取到数据的指针 • `uint32 size`：数据长度，单位 byte，必须 4 字节对齐进行读写
返回	`typedef enum {` `  SPI_FLASH_RESULT_OK,` `  SPI_FLASH_RESULT_ERR,` `  SPI_FLASH_RESULT_TIMEOUT` `} SpiFlashOpResult;`
示例	`uint32 value;` `  uint8 *addr = (uint8 *)&value;` `  spi_flash_read(0x3E * SPI_FLASH_SEC_SIZE, (uint32 *)addr, 4);` `  os_printf("0x3E sec:%02x%02x%02x%02x\r\n", addr[0], addr[1], addr[2], addr[3]);`

3.4.5. system_param_save_with_protect

功能	使用带读写保护机制的方式，写入数据到 Flash。Flash 读写必须 4 字节对齐。 Flash 读写保护机制：使用 3 个 sector（4 KB 每 sector）保存 1 个 sector 的数据，sector 0 和 sector 1 互相为备份，交替保存数据，sector 2 作为 flag sector，指示最新的数据保存在 sector 0 还是 sector 1。
注意	关于 SPI Flash 读写操作，详见文档 ESP8266 Flash 读写说明。
函数定义	`bool system_param_save_with_protect (` `  uint16 start_sec,` `  void *param,` `  uint16 len` `)`
参数	• `uint16 start_sec`：读写保护机制使用的 3 个 sector 的起始 sector 0 值。 例如，IOT_Demo 中可使用 0x3D000 开始的 3 个 sector（3×4 KB）建立读写保护机制，则参数 start_sec 传 0x3D。 • `void *param`：写入数据的指针 • `uint16 len`：数据长度，不能超过 1 个 sector 大小，即 4×1024
返回	`true`：成功 `false`：失败

3.4.6. system_param_load

功能	使用带读写保护机制的方式，写入数据到 Flash。Flash 读写必须 4 字节对齐。 Flash 读写保护机制：使用 3 个 sector（4 KB 每 sector）保存 1 个 sector 的数据，sector 0 和 sector 1 互相为备份，交替保存数据，sector 2 作为 flag sector，指示最新的数据保存在 sector 0 还是 sector 1。
注意	关于 SPI Flash 读写操作，详见文档 ESP8266 Flash 读写说明。
函数定义	`bool system_param_load (` `  uint16 start_sec,` `  uint16 offset,` `  void *param,` `  uint16 len` `)`
参数	• `uint16 start_sec`：读写保护机制使用的 3 个 sector 的起始 sector 0 值。 例如，IOT_Demo 中可使用 0x3D000 开始的 3 个 sector（3×4 KB）建立读写保护机制，则参数 start_sec 传 0x3D，请勿传入 0x3E 或者 0x3F。 • `uint16 offset`：需读取数据，在 sector 中的偏移地址 • `void *param`：读取数据的指针 • `uint16 len`：数据长度，不能超过 1 个 sector 大小，即 offset+len ≤ 4*1024

返回	`true`：成功 `false`：失败

3.4.7. spi_flash_set_read_func

功能	注册用户自定义的 SPI Flash 读取接口函数
注意	仅支持在 SPI overlap 模式下使用，请用户参考 *ESP8266_NONOS_SDK\driver_lib\driver\spi_overlap.c*
函数定义	`void spi_flash_set_read_func (user_spi_flash_read read)`
参数	`typedef SpiFlashOpResult (*user_spi_flash_read)(` `SpiFlashChip *spi,` `uint32 src_addr,` `uint32 * des_addr,` `uint32 size` `)`
返回	无

3.4.8. spi_flash_erase_protect_enable

功能	使能 flash 擦写保护。使能后，将保护 flash 不会误操作擦写了正在运行的应用程序。
函数定义	`bool spi_flash_erase_protect_enable(void)`
参数	无
返回	True: 设置成功 False: 设置失败

3.4.9. spi_flash_erase_protect_disable

功能	关闭 flash 擦写保护功能。
函数定义	`bool spi_flash_erase_protect_disable(void)`
参数	无
返回	True: 设置成功 False: 设置失败

3.5. Wi-Fi 接口

Wi-Fi 接口位于 ***/ESP8266_NONOS_SDK/include/user_interface.h***。

`wifi_station_xxx` 系列接口以及 ESP8266 Station 相关的设置、查询接口，请在 ESP8266 Station 使能的情况下调用；

`wifi_softap_xxx` 系列接口以及 ESP8266 SoftAP 相关的设置、查询接口，请在 ESP8266 SoftAP 使能的情况下调用。

ESP8266 station 支持的认证类型有：OPEN，WEP，WPAPSK，WPA2PSK；支持的加密方式有：AUTO，TKIP，AES，WEP。

ESP8266 softAP 支持的认证类型有：OPEN，WPAPSK，WPA2PSK；支持的加密方式有：AUTO，TKIP，AES；但 group key 加密方式，只支持 TKIP，不支持 AES。

后文的“Flash 系统参数区”位于 Flash 的最后 16 KB。

3.5.1. wifi_get_opmode

功能	查询 Wi-Fi 当前工作模式
函数定义	`uint8 wifi_get_opmode (void)`
参数	无
返回	Wi-Fi 工作模式： • `0x01`：Station 模式 • `0x02`：SoftAP 模式 • `0x03`：Station+SoftAP 模式

3.5.2. wifi_get_opmode_default

功能	查询保存在 Flash 中的 Wi-Fi 工作模式设置
函数定义	`uint8 wifi_get_opmode_default (void)`
参数	无
返回	Wi-Fi 工作模式： • `0x01`：Station 模式 • `0x02`：SoftAP 模式 • `0x03`：Station+SoftAP 模式

3.5.3. wifi_set_opmode

功能	设置 Wi-Fi 工作模式（Station，SoftAP 或者 Station+SoftAP），并保存到 Flash。 默认为 SoftAP 模式

注意	• ESP8266_NONOS_SDK_V0.9.2 以及之前版本，设置之后需要调用 `system_restart()` 重启生效； • ESP8266_NONOS_SDK_V0.9.2 之后的版本，不需要重启，即时生效。 • 本设置如果与原设置不同，会更新保存到 Flash 系统参数区。
函数定义	`bool wifi_set_opmode (uint8 opmode)`
参数	`uint8 opmode`：Wi-Fi 工作模式 • `0x01`：Station 模式 • `0x02`：SoftAP 模式 • `0x03`：Station+SoftAP 模式
返回	`true`：成功 `false`：失败

3.5.4. wifi_set_opmode_current

功能	设置 Wi-Fi 工作模式（Station，SoftAP 或者 Station + SoftAP），不保存到 Flash。
函数定义	`bool wifi_set_opmode_current (uint8 opmode)`
参数	`uint8 opmode`：Wi-Fi 工作模式 • `0x01`：Station 模式 • `0x02`：SoftAP 模式 • `0x03`：Station+SoftAP 模式
返回	`true`：成功 `false`：失败

3.5.5. wifi_station_get_config

功能	查询 Wi-Fi Station 接口的当前配置参数。
函数定义	`bool wifi_station_get_config (struct station_config *config)`
参数	`struct station_config *config`：Wi-Fi Station 接口参数指针
返回	`true`：成功 `false`：失败

3.5.6. wifi_station_get_config_default

功能	查询 Wi-Fi Station 接口保存在 Flash 中的配置参数。
函数定义	`bool wifi_station_get_config_default (struct station_config *config)`
参数	`struct station_config *config`：Wi-Fi Station 接口参数指针
返回	`true`：成功 `false`：失败

3.5.7. wifi_station_set_config

功能	设置 Wi-Fi Station 接口的配置参数，并保存到 Flash
注意	• 请在 ESP8266 Station 使能的情况下，调用本接口。 • 如果 `wifi_station_set_config` 在 `user_init` 中调用，则 ESP8266 Station 接口会在系统初始化完成后，自动连接 AP（路由），无需再调用 `wifi_station_connect` • 否则，需要调用 `wifi_station_connect` 连接 AP（路由）。 • `station_config.bssid_set` 一般设置为 0，仅当需要检查 AP 的 MAC 地址时（多用于有重名 AP 的情况下）设置为 1。 • 本设置如果与原设置不同，会更新保存到 Flash 系统参数区。
函数定义	`bool wifi_station_set_config (struct station_config *config)`
参数	`struct station_config *config`：Wi-Fi Station 接口配置参数指针
返回	`true`：成功 `false`：失败
示例	`void ICACHE_FLASH_ATTR` `user_set_station_config(void)` `{` `  char ssid[32] = SSID;` `  char password[64] = PASSWORD;` `  struct station_config stationConf;` `  stationConf.bssid_set = 0;  //need not check MAC address of AP` `  os_memcpy(&stationConf.ssid, ssid, 32);` `  os_memcpy(&stationConf.password, password, 64);` `  wifi_station_set_config(&stationConf);` `}` `void user_init(void)` `{` `  wifi_set_opmode(STATIONAP_MODE); //Set softAP + station mode` `  user_set_station_config();` `}`

3.5.8. wifi_station_set_config_current

功能	设置 Wi-Fi Station 接口的配置参数，不保存到 Flash

注意	• 请在 ESP8266 Station 使能的情况下，调用本接口。 • 如果 wifi_station_set_config 在 user_init 中调用，则 ESP8266 Station 接口会在系统初始化完成后，自动连接 AP（路由），无需再调用 wifi_station_connect • 否则，需要调用 wifi_station_connect 连接 AP（路由）。 • station_config.bssid_set 一般设置为 0，仅当需要检查 AP 的 MAC 地址时（多用于有重名 AP 的情况下）设置为 1。 • 本设置如果与原设置不同，会更新保存到 Flash 系统参数区。
函数定义	bool wifi_station_set_config_current (struct station_config *config)
参数	struct station_config *config：Wi-Fi Station 接口配置参数指针
返回	true：成功 false：失败

3.5.9. wifi_station_set_cert_key

功能	不建议使用本接口，请使用 wifi_station_set_enterprise_cert_key 代替。 设置 ESP8266 Wi-Fi Station 接口连接 WPA2-ENTERPRISE AP 使用的证书。
注意	• 支持 WPA2-ENTERPRISE AP 需占用 26 KB 以上的内存，调用本接口时请注意内存是否足够。 • 目前 WPA2-ENTERPRISE 只支持非加密的私钥文件和证书文件，且仅支持 PEM 格式 - 支持的证书文件头信息为：- - - - - BEGIN CERTIFICATE - - - - - - 支持的私钥文件头信息为：- - - - - BEGIN RSA PRIVATE KEY - - - - - 或者 - - - - - BEGIN PRIVATE KEY - - - - - • 请在连接 WPA2-ENTERPRISE AP 之前调用本接口设置私钥文件和证书文件，在成功连接 AP 后先调用 wifi_station_clear_cert_key 清除内部状态，应用层再释放私钥文件和证书文件信息。 • 如果遇到加密的私钥文件，请使用 openssl pkey 命令改为非加密文件使用，或者使用 openssl rsa 等命令，对某些私钥文件进行加密-非加密的转换（或起始 TAG 转化）。
函数定义	bool wifi_station_set_cert_key (uint8 *client_cert, int client_cert_len, uint8 *private_key, int private_key_len, uint8 *private_key_passwd, int private_key_passwd_len,)
参数	uint8 *client_cert：十六进制数组的证书指针 int client_cert_len：证书长度 uint8 *private_key：十六进制数组的私钥指针，暂不支持超过 2048 的私钥 int private_key_len：私钥长度，请勿超过 2048 uint8 *private_key_passwd：私钥的提取密码，目前暂不支持，请传入 NULL int private_key_passwd_len：提取密码的长度，目前暂不支持，请传入 0

返回	0：成功 非 0：失败
示例	假设私钥文件的信息为 - - - - - BEGIN PRIVATE KEY - - - - - … … … … 那么对应的数组为：uint8 key[]={0x2d, 0x2d, 0x2d, 0x2d, 0x2d, 0x42, 0x45, 0x47, … … 0x00 }; 即各字符的 ASCII 码，请注意，数组必须添加 0x00 作为结尾。

3.5.10. wifi_station_clear_cert_key

功能	不建议使用本接口，请使用 wifi_station_clear_enterprise_cert_key 代替。 释放连接 WPA2-ENTERPRISE AP 使用证书占用的资源，并清除相关状态。
函数定义	void wifi_station_clear_cert_key (void)
参数	无
返回	无

3.5.11. wifi_station_set_username

功能	不建议使用本接口，请使用 wifi_station_set_enterprise_username 代替。 设置连接 WPA2-ENTERPRISE AP 时，ESP8266 Station 的用户名。
函数定义	int wifi_station_set_username (uint8 *username, int len)
参数	uint8 *username：用户名称 int len：名称长度
返回	0：成功 其他：失败

3.5.12. wifi_station_clear_username

功能	不建议使用本接口，请使用 wifi_station_clear_enterprise_username 代替。 释放连接 WPA2-ENTERPRISE AP 设置用户名占用的资源，并清除相关状态。
函数定义	void wifi_station_clear_username (void)
参数	无
返回	无

3.5.13. wifi_station_connect

功能	ESP8266 Wi-Fi Station 接口连接 AP
注意	请勿在 user_init 中调用本接口，请在 ESP8266 Station 使能并初始化完成后调用； 如果 ESP8266 已经连接某个 AP，请先调用 wifi_station_disconnect 断开上一次连接。
函数定义	bool wifi_station_connect (void)
参数	无

返回	true：成功 false：失败

3.5.14. wifi_station_disconnect

功能	ESP8266 Wi-Fi Station 接口从 AP 断开连接
注意	请勿在 user_init 中调用本接口，本接口必须在系统初始化完成后，并且 ESP8266 Station 接口使能的情况下调用。
函数定义	bool wifi_station_disconnect (void)
参数	无
返回	true：成功 false：失败

3.5.15. wifi_station_get_connect_status

功能	查询 ESP8266 Wi-Fi Station 接口连接 AP 的状态。
注意	若为特殊应用场景：调用 wifi_station_set_reconnect_policy 关闭重连功能，且未调用 wifi_set_event_handler_cb 注册 Wi-Fi 事件回调，则本接口失效，无法准确获得连接状态。
函数定义	uint8 wifi_station_get_connect_status (void)
参数	无
返回	enum{ STATION_IDLE = 0, STATION_CONNECTING, STATION_WRONG_PASSWORD, STATION_NO_AP_FOUND, STATION_CONNECT_FAIL, STATION_GOT_IP };

3.5.16. wifi_station_scan

功能	获取 AP 的信息
注意	请勿在 user_init 中调用本接口，本接口必须在系统初始化完成后，并且 ESP8266 Station 接口使能的情况下调用。
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' s ssid uint8 *bssid; // AP' s bssid uint8 channel; //scan a specific channel uint8 show_hidden; //scan APs of which ssid is hidden. wifi_scan_type_t scan_type; // scan type, active or passive wifi_scan_time_t scan_time; // scan time per channel };

参数	• `struct scan_config *config`：扫描 AP 的配置参数 - 若 `config==null`：扫描获取所有可用 AP 的信息 - 若 `config.ssid==null && config.bssid==null && config.channel!=null`：ESP8266 Station 接口扫描获取特定信道上的 AP 信息。 - 若 `config.ssid!=null && config.bssid==null && config.channel==null`：ESP8266 Station 接口扫描获取所有信道上的某特定名称 AP 的信息。 • `scan_done_cb_t cb`：扫描完成的 callback
返回	`true`：成功 `false`：失败

3.5.17. scan_done_cb_t

功能	`wifi_station_scan` 的回调函数
注意	请勿在 `user_init` 中调用本接口，本接口必须在系统初始化完成后，并且 ESP8266 Station 接口使能的情况下调用。
函数定义	`void scan_done_cb_t (void *arg, STATUS status)`
参数	• `void *arg`：扫描获取到的 AP 信息指针，以链表形式存储，数据结构 `struct bss_info` • `STATUS status`：扫描结果
返回	无
示例	`wifi_station_scan(&config, scan_done);` `static void ICACHE_FLASH_ATTR scan_done(void *arg, STATUS status) {` `  if (status == OK) {` `  struct bss_info *bss_link = (struct bss_info *)arg;` `  ...` `  }` `}`

3.5.18. wifi_station_ap_number_set

功能	设置 ESP8266 Station 最多可记录几个 AP 的信息。 ESP8266 Station 成功连入一个 AP 时，可以保存 AP 的 SSID 和 password 记录。 本设置如果与原设置不同，会更新保存到 Flash 系统参数区。
函数定义	`bool wifi_station_ap_number_set (uint8 ap_number)`
参数	`uint8 ap_number`：记录 AP 信息的最大数目（最大值为 5）
返回	`true`：成功 `false`：失败

3.5.19. wifi_station_get_ap_info

功能	获取 ESP8266 Station 保存的 AP 信息，最多记录 5 个。
函数定义	`uint8 wifi_station_get_ap_info(struct station_config config[])`
参数	`struct station_config config[]`：AP 的信息，数组大小必须为 5

返回	记录 AP 的数目
示例	`struct station_config config[5];` `int i = wifi_station_get_ap_info(config);`

3.5.20. wifi_station_ap_change

功能	ESP8266 Station 切换到已记录的某号 AP 配置连接
函数定义	`bool wifi_station_ap_change (uint8 new_ap_id)`
参数	`uint8 new_ap_id`：AP 记录的 ID 值，从 0 开始计数
返回	`true`：成功 `false`：失败

3.5.21. wifi_station_get_current_ap_id

功能	获取当前连接的 AP 保存记录 ID 值。ESP8266 可记录每一个配置连接的 AP，从 0 开始计数。
函数定义	`uint8 wifi_station_get_current_ap_id ();`
参数	无
返回	当前连接的 AP 保存记录的 ID 值。

3.5.22. wifi_station_get_auto_connect

功能	查询 ESP8266 Station 上电是否会自动连接已记录的 AP（路由）。
函数定义	`uint8 wifi_station_get_auto_connect(void)`
参数	无
返回	`0`：不自动连接 AP 非 0：自动连接 AP。

3.5.23. wifi_station_set_auto_connect

功能	设置 ESP8266 Station 上电是否自动连接已记录的 AP（路由），默认为自动连接。
注意	• 本接口如果在 `user_init` 中调用，则当前这次上电就生效；如果在其他地方调用，则下一次上电生效。 • 本设置如果与原设置不同，会更新保存到 Flash 系统参数区。
函数定义	`bool wifi_station_set_auto_connect(uint8 set)`
参数	`uint8 set`：上电是否自动连接 AP • `0`：不自动连接 AP • `1`：自动连接 AP
返回	`true`：成功 `false`：失败

3.5.24. wifi_station_dhcpc_start

功能	开启 ESP8266 Station DHCP client
注意	• DHCP 默认开启。 • DHCP 与静态 IP 功能 `wifi_set_ip_info` 互相影响，以最后设置的为准： DHCP 开启，则静态 IP 失效；设置静态 IP，则关闭 DHCP。
函数定义	`bool wifi_station_dhcpc_start(void)`
参数	无
返回	`true`：成功 `false`：失败

3.5.25. wifi_station_dhcpc_stop

功能	关闭 ESP8266 Station DHCP client
注意	• DHCP 默认开启。 • DHCP 与静态 IP 功能 `wifi_set_ip_info` 互相影响： DHCP 开启，则静态 IP 失效；设置静态 IP，则 DHCP 关闭。
函数定义	`bool wifi_station_dhcpc_stop(void)`
参数	无
返回	`true`：成功 `false`：失败

3.5.26. wifi_station_dhcpc_status

功能	查询 ESP8266 Station DHCP client 状态
函数定义	`enum dhcp_status wifi_station_dhcpc_status(void)`
参数	无
返回	`enum dhcp_status { DHCP_STOPPED, DHCP_STARTED };`

3.5.27. wifi_station_dhcpc_set_maxtry

功能	设置 ESP8266 Station DHCP client 最大重连次数。默认会一直重连。
函数定义	`bool wifi_station_dhcpc_set_maxtry(uint8 num)`
参数	`uint8 num`：最大重连次数
返回	`true`：成功 `false`：失败

3.5.28. wifi_station_set_reconnect_policy

功能	设置 ESP8266 Station 连接 AP 失败或断开后是否重连。默认重连。
注意	建议在 `user_init` 中调用本接口
函数定义	`bool wifi_station_set_reconnect_policy(bool set)`
参数	`bool set` • `true`：断开则重连 • `false`：断开不重连
返回	`true`：成功 `false`：失败

3.5.29. wifi_station_get_rssi

功能	关闭 ESP8266 Station 已连接的 AP 信号强度
函数定义	`sint8 wifi_station_get_rssi(void)`
参数	无
返回	<10：查询成功，返回信号强度 31：查询失败，返回错误码

3.5.30. wifi_station_set_hostname

功能	设置 ESP8266 Station DHCP 分配的主机名称。
函数定义	`bool wifi_station_set_hostname(char* hostname)`
参数	`char* hostname`：主机名称，最长 32 个字符。
返回	`true`：成功 `false`：失败

3.5.31. wifi_station_get_hostname

功能	查询 ESP8266 Station DHCP 分配的主机名称
函数定义	`char* wifi_station_get_hostname(void)`
参数	无
返回	主机名称

3.5.32. wifi_softap_get_config

功能	查询 ESP8266 Wi-Fi SoftAP 接口的当前配置
函数定义	`bool wifi_softap_get_config(struct softap_config *config)`

参数	`struct softap_config *config`：ESP8266 SoftAP 配置参数
返回	`true`：成功 `false`：失败

3.5.33. wifi_softap_get_config_default

功能	查询 ESP8266 Wi-Fi SoftAP 接口保存在 Flash 中的配置
函数定义	`bool wifi_softap_get_config_default(struct softap_config *config)`
参数	`struct softap_config *config`：ESP8266 SoftAP 配置参数
返回	`true`：成功 `false`：失败

3.5.34. wifi_softap_set_config

功能	设置 Wi-Fi SoftAP 接口配置，并保存到 Flash
注意	• 请在 ESP8266 SoftAP 使能的情况下，调用本接口。 • 本设置如果与原设置不同，将更新保存到 Flash 系统参数区。 • 因为 ESP8266 只有一个信道，因此 SoftAP+Station 共存模式时，ESP8266 SoftAP 接口会自动调节信道与 ESP8266 Station 一致，详细说明请参考附录。
函数定义	`bool wifi_softap_set_config (struct softap_config *config)`
参数	`struct softap_config *config`：ESP8266 Wi-Fi SoftAP 配置参数
返回	`true`：成功 `false`：失败

3.5.35. wifi_softap_set_config_current

功能	设置 Wi-Fi SoftAP 接口配置，不保存到 Flash
注意	• 请在 ESP8266 SoftAP 使能的情况下，调用本接口。 • 因为 ESP8266 只有一个信道，因此 SoftAP+Station 共存模式时，ESP8266 SoftAP 接口会自动调节信道与 ESP8266 Station 一致，详细说明请参考附录。
函数定义	`bool wifi_softap_set_config_current (struct softap_config *config)`
参数	`struct softap_config *config`：ESP8266 Wi-Fi SoftAP 配置参数
返回	`true`：成功 `false`：失败

3.5.36. wifi_softap_get_station_num

功能	获取 ESP8266 SoftAP 下连接的 Station 个数

函数定义	`uint8 wifi_softap_get_station_num(void)`
参数	无
返回	ESP8266 SoftAP 下连接的 Station 个数

3.5.37. wifi_softap_get_station_info

功能	获取 ESP8266 SoftAP 接口下连入的 Station 的信息，包括 MAC 和 IP
注意	本接口基于 DHCP 实现，因此不支持静态 IP 或者其他没有重新 DHCP 的情况。
函数定义	`struct station_info * wifi_softap_get_station_info(void)`
参数	无
返回	`struct station_info*`：Station 信息的结构体

3.5.38. wifi_softap_free_station_info

功能	释放调用 `wifi_softap_get_station_info` 时结构体 `station_info` 占用的空间
函数定义	`void wifi_softap_free_station_info(void)`
参数	无
返回	无
示例	获取 MAC 和 IP 信息示例，注意释放资源：
示例 1	`struct station_info * station = wifi_softap_get_station_info();` `struct station_info * next_station;` `while(station) {` `  os_printf(bssid : MACSTR, ip : IPSTR/n,` `  MAC2STR(station->bssid), IP2STR(&station->ip));` `  next_station = STAILQ_NEXT(station, next);` `  os_free(station);  // Free it directly` `  station = next_station;` `}`
示例 2	`struct station_info * station = wifi_softap_get_station_info();` `while(station){` `  os_printf(bssid : MACSTR, ip : IPSTR/n,` `  MAC2STR(station->bssid), IP2STR(&station->ip));` `  station = STAILQ_NEXT(station, next);` `}` `wifi_softap_free_station_info();  // Free it by calling functions`

3.5.39. wifi_softap_dhcps_start

功能	开启 ESP8266 SoftAP DHCP server
注意	• DHCP 默认开启。 • DHCP 与静态 IP 功能 `wifi_set_ip_info` 互相影响，以最后设置的为准： DHCP 开启，则静态 IP 失效；设置静态 IP，则关闭 DHCP。
函数定义	`bool wifi_softap_dhcps_start(void)`

参数	无
返回	`true`：成功 `false`：失败

3.5.40. wifi_softap_dhcps_stop

功能	关闭 ESP8266 SoftAP DHCP server。默认开启 DHCP。
函数定义	`bool wifi_softap_dhcps_stop(void)`
参数	无
返回	`true`：成功 `false`：失败

3.5.41. wifi_softap_set_dhcps_lease

功能	设置 ESP8266 SoftAP DHCP server 分配 IP 地址的范围
注意	• 设置的 IP 分配范围必须与 ESP8266 SoftAP IP 在同一网段。 • 本接口必须在 ESP8266 SoftAP DHCP server 关闭 `wifi_softap_dhcps_stop` 的情况下设置。 • 本设置仅对下一次使能的 DHCP server 生效 `wifi_softap_dhcps_start`，如果 DHCP server 再次被关闭，则需要重新调用本接口设置 IP 范围；否则之后 DHCP server 重新使能，会使用默认的 IP 地址分配范围。
函数定义	`bool wifi_softap_set_dhcps_lease(struct dhcps_lease *please)`
参数	`struct dhcps_lease {` `  struct ip_addr start_ip;` `  struct ip_addr end_ip;` `};`
返回	`true`：成功 `false`：失败

示例	``` void dhcps_lease_test(void) { struct dhcps_lease dhcp_lease; const char* start_ip = “192.168.5.100”; const char* end_ip = “192.168.5.105”; dhcp_lease.start_ip.addr = ipaddr_addr(start_ip); dhcp_lease.end_ip.addr = ipaddr_addr(end_ip); wifi_softap_set_dhcps_lease(&dhcp_lease); } ``` 或者 ``` void dhcps_lease_test(void) { struct dhcps_lease dhcp_lease; IP4_ADDR(&dhcp_lease.start_ip, 192, 168, 5, 100); IP4_ADDR(&dhcp_lease.end_ip, 192, 168, 5, 105); wifi_softap_set_dhcps_lease(&dhcp_lease); } void user_init(void) { struct ip_info info; wifi_set_opmode(STATIONAP_MODE); //Set softAP + station mode wifi_softap_dhcps_stop(); IP4_ADDR(&info.ip, 192, 168, 5, 1); IP4_ADDR(&info.gw, 192, 168, 5, 1); IP4_ADDR(&info.netmask, 255, 255, 255, 0); wifi_set_ip_info(SOFTAP_IF, &info); dhcps_lease_test(); wifi_softap_dhcps_start(); } ```

3.5.42. wifi_softap_get_dhcps_lease

功能	查询 ESP8266 SoftAP DHCP server 分配 IP 地址的范围
注意	本接口仅支持在 ESP8266 SoftAP DHCP server 使能的情况下查询。
函数定义	`bool wifi_softap_get_dhcps_lease(struct dhcps_lease *please)`
返回	`true`：成功 `false`：失败

3.5.43. wifi_softap_set_dhcps_lease_time

功能	设置 ESP8266 SoftAP DHCP server 的租约时间。默认为 120 分钟。
注意	本接口仅支持在 ESP8266 SoftAP DHCP server 使能的情况下查询。
函数定义	`bool wifi_softap_set_dhcps_lease_time(uint32 minute)`

参数	`uint32 minute`：租约时间，单位：分钟，取值范围：[1，2880]
返回	`true`：成功 `false`：失败

3.5.44. wifi_softap_get_dhcps_lease_time

功能	查询 ESP8266 SoftAP DHCP server 的租约时间。
注意	本接口仅支持在 ESP8266 SoftAP DHCP server 使能的情况下查询。
函数定义	`uint32 wifi_softap_get_dhcps_lease_time(void)`
返回	租约时间，单位：分钟

3.5.45. wifi_softap_reset_dhcps_lease_time

功能	复位 ESP8266 SoftAP DHCP server 的租约时间。恢复到 120 分钟。
注意	本接口仅支持在 ESP8266 SoftAP DHCP server 使能的情况下查询。
函数定义	`bool wifi_softap_reset_dhcps_lease_time(void)`
返回	`true`：成功 `false`：失败

3.5.46. wifi_softap_dhcps_status

功能	获取 ESP8266 SoftAP DHCP server 状态
函数定义	`enum dhcp_status wifi_softap_dhcps_status(void)`
参数	无
返回	`enum dhcp_status {` `  DHCP_STOPPED,` `  DHCP_STARTED` `};`

3.5.47. wifi_softap_set_dhcps_offer_option

功能	设置 ESP8266 SoftAP DHCP server 属性
结构体	`enum dhcps_offer_option{` `  OFFER_START = 0x00,` `  OFFER_ROUTER = 0x01,` `  OFFER_END` `};`
函数定义	`bool wifi_softap_set_dhcps_offer_option(uint8 level, void* optarg)`

参数	• `uint8 level`：OFFER_ROUTER，设置 router 信息 • `void* optarg`：bit0, 0 禁用 router 信息；bit0, 1 启用 router 信息；默认为 1
返回	`true`：成功 `false`：失败
示例	`uint8 mode = 0 ;` `wifi_softap_set_dhcps_offer_option(OFFER_ROUTER, &mode);`

3.5.48. wifi_set_phy_mode

功能	设置 ESP8266 物理层模式 (802.11 b/g/n)
函数定义	`bool wifi_set_phy_mode(enum phy_mode mode)`
参数	`enum phy_mode mode : 物理层模式` `enum phy_mode {` `  PHY_MODE_11B = 1,` `  PHY_MODE_11G = 2,` `  PHY_MODE_11N = 3` `};`
返回	`true`：成功 `false`：失败

3.5.49. wifi_get_phy_mode

功能	查询 ESP8266 物理层模式 (802.11 b/g/n)
函数定义	`enum phy_mode wifi_get_phy_mode(void)`
参数	无
返回	`enum phy_mode{` `  PHY_MODE_11B = 1,` `  PHY_MODE_11G = 2,` `  PHY_MODE_11N = 3` `};`

3.5.50. wifi_get_ip_info

功能	查询 Wi-Fi Station 接口或者 SoftAP 接口的 IP 地址
注意	在 `user_init` 中，由于初始化尚未完成，无法通过本接口查询到有效 IP 地址。
函数定义	`bool wifi_get_ip_info(` `  uint8 if_index,` `  struct ip_info *info` `)`
参数	`uint8 if_index`：获取 Station 或者 SoftAP 接口的信息 `  #define STATION_IF  0x00` `  #define SOFTAP_IF  0x01` `struct ip_info *info`：获取到的 IP 信息

返回	true：成功 false：失败

3.5.51. wifi_set_ip_info

功能	设置 Wi-Fi Station 或者 SoftAP 的 IP 地址
注意	• 本接口设置静态 IP，请先关闭对应 DHCP 功能 wifi_station_dhcpc_stop 或者 wifi_softap_dhcps_stop • 设置静态 IP，则关闭 DHCP；DHCP 开启，则静态 IP 失效。
函数定义	bool wifi_set_ip_info(uint8 if_index, struct ip_info *info)
参数	uint8 if_index：设置 Station 或者 SoftAP 接口 #define STATION_IF 0x00 #define SOFTAP_IF 0x01 struct ip_info *info：获取到的 IP 信息
返回	true：成功 false：失败
示例	wifi_set_opmode(STATIONAP_MODE); //Set softAP + station mode struct ip_info info; wifi_station_dhcpc_stop(); wifi_softap_dhcps_stop(); IP4_ADDR(&info.ip, 192, 168, 3, 200); IP4_ADDR(&info.gw, 192, 168, 3, 1); IP4_ADDR(&info.netmask, 255, 255, 255, 0); wifi_set_ip_info(STATION_IF, &info); IP4_ADDR(&info.ip, 10, 10, 10, 1); IP4_ADDR(&info.gw, 10, 10, 10, 1); IP4_ADDR(&info.netmask, 255, 255, 255, 0); wifi_set_ip_info(SOFTAP_IF, &info); wifi_softap_dhcps_start();

3.5.52. wifi_set_macaddr

功能	设置 MAC 地址
注意	• 本接口必须在 user_init 中调用 • ESP8266 SoftAP 和 Station MAC 地址不同，请勿将两者设置为同一 MAC 地址 • ESP8266 MAC 地址第一个字节的 bit 0 不能为 1。例如，MAC 地址可以设置为 1a:XX:XX:XX:XX:XX，但不能设置为 15:XX:XX:XX:XX:XX。
函数定义	bool wifi_set_macaddr(uint8 if_index, uint8 *macaddr)

参数	uint8 if_index：设置 Station 或者 SoftAP 接口 #define STATION_IF 0x00 #define SOFTAP_IF 0x01 uint8 *macaddr：MAC 地址
返回	true：成功 false：失败
示例	wifi_set_opmode(STATIONAP_MODE); char sofap_mac[6] = {0x16, 0x34, 0x56, 0x78, 0x90, 0xab}; char sta_mac[6] = {0x12, 0x34, 0x56, 0x78, 0x90, 0xab}; wifi_set_macaddr(SOFTAP_IF, sofap_mac); wifi_set_macaddr(STATION_IF, sta_mac);

3.5.53. wifi_get_macaddr

功能	查询 MAC 地址
函数定义	bool wifi_get_macaddr(uint8 if_index, uint8 *macaddr)
参数	uint8 if_index：查询 Station 或者 SoftAP 接口 #define STATION_IF 0x00 #define SOFTAP_IF 0x01 uint8 *macaddr：MAC 地址
返回	true：成功 false：失败

3.5.54. wifi_set_sleep_type

功能	设置省电模式。设置为 NONE_SLEEP_T，则关闭省电模式。
注意	• 默认为 Modem-sleep 模式。 • Light Sleep 为了降低功耗，将 TCP timer tick 由原本的 250ms 改为了 3s，这将导致 TCP timer 超时时间相应增加；如果用户对 TCP timer 的准确度有要求，请使用 modem sleep 或者 deep sleep 模式。
函数定义	bool wifi_set_sleep_type(enum sleep_type type)
参数	enum sleep_type type：省电模式
返回	true：成功 false：失败

3.5.55. wifi_get_sleep_type

功能	查询省电模式。
函数定义	enum sleep_type wifi_get_sleep_type(void)

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

3.5.56. wifi_status_led_install

功能	注册 Wi-Fi 状态 LED。
函数定义	`void wifi_status_led_install (` `  uint8 gpio_id,` `  uint32 gpio_name,` `  uint8 gpio_func` `)`
参数	• `uint8 gpio_id`：GPIO ID • `uint8 gpio_name`：GPIO MUX 名称 • `uint8 gpio_func`：GPIO 功能
返回	无
示例	使用 GPIO0 作为 Wi-Fi 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)`

3.5.57. wifi_status_led_uninstall

功能	注销 Wi-Fi 状态 LED
函数定义	`void wifi_status_led_uninstall ()`
参数	无
返回	无

3.5.58. wifi_set_broadcast_if

功能	设置 ESP8266 发送 UDP 广播包时，从 Station 接口还是 SoftAP 接口发送。默认从 SoftAP 接口发送。
注意	如果设置仅从 Station 接口发 UDP 广播包，会影响 ESP8266 SoftAP 的功能，DHCP server 无法使用。需要使能 SoftAP 的广播包功能，才可正常使用 ESP8266 SoftAP。
函数定义	`bool wifi_set_broadcast_if (uint8 interface)`
参数	`uint8 interface` • `1`：station • `2`：SoftAP • `3`：Station 和 SoftAP 接口均发送

返回	`true`：成功 `false`：失败

3.5.59. wifi_get_broadcast_if

功能	查询 ESP8266 发送 UDP 广播包时，从 Station 接口还是 SoftAP 接口发送。
函数定义	`uint8 wifi_get_broadcast_if (void)`
参数	无
返回	• `1`：Station • `2`：SoftAP • `3`：Station 和 SoftAP 接口均发送

3.5.60. wifi_set_event_handler_cb

功能	注册 Wi-Fi event 处理回调
函数定义	`void wifi_set_event_handler_cb(wifi_event_handler_cb_t cb)`
参数	`wifi_event_handler_cb_t cb`：回调函数
返回	无

示例

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

3.5.61. wifi_wps_enable

功能	使能 Wi-Fi WPS 功能
注意	WPS 功能必须在 ESP8266 Station 使能的情况下调用。

结构体	typedef enum wps_type { WPS_TYPE_DISABLE=0, WPS_TYPE_PBC, WPS_TYPE_PIN, WPS_TYPE_DISPLAY, WPS_TYPE_MAX, }WPS_TYPE_t;
函数定义	bool wifi_wps_enable(WPS_TYPE_t wps_type)
参数	WPS_TYPE_t wps_type：WPS 的类型，目前仅支持 WPS_TYPE_PBC
返回	true：成功 false：失败

3.5.62. wifi_wps_disable

功能	关闭 Wi-Fi WPS 功能，释放占用的资源。
函数定义	bool wifi_wps_disable(void)
参数	无
返回	true：成功 false：失败

3.5.63. wifi_wps_start

功能	WPS 开始进行交互
注意	WPS 功能必须在 ESP8266 Station 使能的情况下调用。
函数定义	bool wifi_wps_start(void)
参数	无
返回	true：成功开始交互，并不表示 WPS 成功完成 false：失败

3.5.64. wifi_set_wps_cb

功能	设置 WPS 回调函数，回调函数中将传入 WPS 运行状态。WPS 不支持 WEP 加密方式。
回调及参数结构体	typedef void (*wps_st_cb_t)(int status); enum wps_cb_status { WPS_CB_ST_SUCCESS = 0, WPS_CB_ST_FAILED, WPS_CB_ST_TIMEOUT, WPS_CB_ST_WEP, // WPS failed because that WEP is not supported. WPS_CB_ST_SCAN_ERR, // can not find the target WPS AP };

注意	• 如果回调函数的传入参数状态为 `WPS_CB_ST_SUCCESS`，表示成功获得 AP 密钥，请调用 `wifi_wps_disable` 关闭 WPS 功能释放资源，并调用 `wifi_station_connect` 连接 AP。 • 否则，表示 WPS 失败，可以创建一个定时器，间隔一段时间后调用 `wifi_wps_start` 再次尝试 WPS，或者调用 `wifi_wps_disable` 关闭 WPS 并释放资源。
函数定义	`bool wifi_set_wps_cb(wps_st_cb_t cb)`
参数	`wps_st_cb_t cb`：回调函数
返回	`true`：成功 `false`：失败

3.5.65. wifi_register_send_pkt_freedom_cb

功能	注册 freedom 发包的回调函数。freedom 发包功能，即支持发送用户自定义 802.11 的包。
注意	• freedom 发包必须等前一个包发送完毕，进入发包回调 `freedom_outside_cb_t` 之后，才能发下一个包。 • 设置发送回调函数可以用来判别包是否发送成功（IEEE802.11 MAC 底层是否发送成功）。使用发送回调函数请注意如下情况： ▸ 针对单播包： - 回调函数状态显示成功时，对方应用层实际没有收到的状况。原因： 1. 存在流氓设备进行攻击 2. 加密密钥设置错误 3. 应用层丢包 若需要更强地发包保证发包成功率，请在应用层实现发包握手机制。 - 回调函数状态显示失败时，对方应用层实际已收到的状况。原因： 1. 信道繁忙，未收到对方ACK。 请注意应用层发包重传，接收方需要检测重传包。 ▸ 针对组播包（包括广播包）： - 回调函数状态显示成功，表示组播包已成功发送 - 回调函数状态显示失败，表示组播包发送失败
回调函数定义	`typedef void (*freedom_outside_cb_t)(uint8 status);` `status`：0，发包成功；其他值，发包失败。
参数	`freedom_outside_cb_t cb`：回调函数
返回	`0`：注册成功 `-1`：注册失败

3.5.66. wifi_unregister_send_pkt_freedom_cb

功能	注销 freedom 发包的回调函数。
函数定义	`void wifi_unregister_send_pkt_freedom_cb(void)`

参数	无
返回	无

3.5.67. wifi_send_pkt_freedom

功能	发包函数。
注意	• 发送包必须是完整的 802.11 包，长度不包含 FCS。发包长度必须大于最小 802.11 头，即 24 字节，且不能超过 1400 字节，否则返回发包失败。 • duration 域填写无效，由 ESP8266 底层程序决定，自动填充。 • 发包速率限制成管理包速率，与系统的发包速率一致。 • 支持发送：非加密的数据包，非加密的 beacon/probe req/probe resp。 • 不支持发送：所有加密包 (即包头中的加密 bit 必须为 0，否则返回发包失败)，控制包，除 beacon/probe req/probe resp 以外的其他管理包。 • freedom 发包必须等前一个包发送完毕，进入发包回调之后，才能发下一个包。
函数定义	`int wifi_send_pkt_freedom(uint8 *buf, int len, bool sys_seq)`
参数	• `uint8 *buf`：数据包指针 • `int len`：数据包长度 • `bool sys_seq`：是否跟随系统的 802.11 包 sequence number，如果跟随系统，将会在每次发包后自加 1
返回	`0`：成功 `-1`：失败

3.5.68. wifi_rfid_locp_recv_open

功能	开启 RFID LOCP (Location Control Protocol) 功能，用于接收 WDS 类型的包。
函数定义	`int wifi_rfid_locp_recv_open(void)`
参数	无
返回	`0`：成功 其他值：失败

3.5.69. wifi_rfid_locp_recv_close

功能	关闭 RFID LOCP (Location Control Protocol) 功能。
函数定义	`void wifi_rfid_locp_recv_close(void)`
参数	无
返回	无

3.5.70. wifi_register_rfid_locp_recv_cb

功能	注册 WDS 收包回调。仅在收到的 WDS 包的第一个 MAC 地址为组播地址时，才会进入回调函数。
回调函数定义	`typedef void (*rfid_locp_cb_t)(uint8 *frm, int len, int rssi);`
参数	• `uint8 *frm`：指向 802.11 包头的指针 • `int len`：数据包长度 • `int rssi`：信号强度
返回	`0`：成功 其他值：失败

3.5.71. wifi_unregister_rfid_locp_recv_cb

功能	注销 WDS 收包回调。
函数定义	`void wifi_unregister_rfid_locp_recv_cb(void)`
参数	无
返回	无

3.5.72. wifi_enable_gpio_wakeup

功能	使能 GPIO 唤醒 Light-sleep 模式的功能。
注意	在自动 Light-sleep 休眠 `wifi_set_sleep_type(LIGHT_SLEEP_T);` 的情况下，由 GPIO 触发 ESP8266 从 Light-sleep 唤醒之后，如需再次进入休眠时，将判断唤醒 GPIO 的状态: • 如果 GPIO 仍然处于唤醒状态，则进入 Modem-sleep 休眠； • 如果 GPIO 不处于唤醒状态，则进入 Light-sleep 休眠。
函数定义	`void wifi_enable_gpio_wakeup(uint32 i, GPIO_INT_TYPE intr_status)`
参数	`uint32 i`：GPIO 号，取值范围：[0, 15] `GPIO_INT_TYPE intr_status`：GPIO 触发唤醒的状态
返回	无
示例	设置 GPIO12 低电平时，将 ESP8266 从 Light-sleep 模式唤醒。 `GPIO_DIS_OUTPUT(12);` `PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U, FUNC_GPIO12);` `wifi_enable_gpio_wakeup(12, GPIO_PIN_INTR_LOLEVEL);`

3.5.73. wifi_disable_gpio_wakeup

功能	取消 GPIO 唤醒 Light-sleep 模式的功能。

函数定义	`void wifi_disable_gpio_wakeup(void)`
参数	无
返回	无

3.5.74. wifi_set_country

功能	设置 WiFi 国家码
函数定义	`bool wifi_set_country(wifi_country_t *country)`
参数	`wifi_country_t *country`: 国家码信息
注意	• 默认国家码为 {.cc="CN", .schan=1, .nchan=13, policy=WIFI_COUNTRY_POLICY_AUTO} • 当 policy=WIFI_COUNTRY_POLICY_AUTO，ESP8266 的国家码会在连上 AP 后，自动更改为与 AP 一致；当与 AP 断开连接后，又回到原设置值。 • 当 policy=WIFI_COUNTRY_POLICY_MANUAL，ESP8266 的国家码将始终保持为设置值。 • 在 station+softAP 模式下，如果 ESP8266 station 的国家码信息改变，softAP 端 probe response/beacon 中的 country IE 国家码信息也会同样改变。 • 国家码信息不保存在 flash，重新上电后，需要重新配置。
返回	true：成功 false：失败

3.5.75. wifi_get_country

功能	获取当前 WiFi 国家码
函数定义	`bool wifi_get_country(wifi_country_t *country)`
参数	`wifi_country_t *country`: 国家码信息
返回	true：成功 false：失败

3.5.76. wifi_set_sleep_level

功能	设置 modem sleep 和 light sleep 的 sleep level
函数定义	`bool wifi_set_sleep_level(enum sleep_level level)`
参数	`level`: modem sleep 和 light sleep 的 sleep level
返回	true：成功 false：失败

说明	• 如需设置，请在 `wifi_set_sleep_type` 调用本接口。 • modem sleep 和 light sleep 支持两种 sleep level： - 最小 sleep level：ESP station 在每个 DTIM 都会醒来接收 beacon；广播数据会在 DTIM 之后传输，因此不会丢失；但如果 AP 的 DTIM 设置较短，则此配置无法达到较好的低功耗效果。 - 最大 sleep level：ESP station 在每个 listen interval 都会醒来接收 beacon；这样在 DTIM 时 ESP station 可能在休眠中，广播数据会因此丢失；如果将 listen interval 设置的越长，低功耗效果越好，但也会因此丢失更多的广播数据。 - 默认为最小 sleep level。 - listen interval 可以通过 API `wifi_set_listen_interval` 设置。

3.5.77. wifi_get_sleep_level

功能	查询 modem sleep 和 light sleep 的 sleep level
函数定义	`enum sleep_level wifi_get_sleep_level(void)`
参数	无
返回	sleep level 值

3.5.78. wifi_set_listen_interval

功能	设置 modem sleep 和 light sleep 最大 sleep level 下的 listen interval
函数定义	`bool wifi_set_listen_interval(uint8 interval)`
参数	`interval`: modem sleep 和 light sleep 最大 sleep level 下的侦听间隔，单位是 AP 的一个 beacon interval，取值范围: [1, 10]
返回	true：成功 false：失败
说明	• 本设置仅针对 modem sleep 和 light sleep 的最大 sleep level（`MAX_SLEEP_T`）。 • 如需设置，应按照以下顺序调用： - `wifi_set_sleep_level(MAX_SLEEP_T)` - `wifi_set_listen_interval` - `wifi_set_sleep_type`

3.5.79. wifi_get_listen_interval

功能	查询 modem sleep 和 light sleep 最大 sleep level 下的 listen interval
函数定义	`uint8 wifi_get_listen_interval(void)`
参数	无

返回	modem sleep 和 light sleep 最大 sleep level 下的侦听间隔

3.6. Rate Control 接口

Wi-Fi Rate Control 接口位于 ***/ESP8266_NONOS_SDK/include/user_interface.h***。

3.6.1. wifi_set_user_fixed_rate

功能	设置 ESP8266 Station 或 SoftAP 发数据的固定 rate 和 mask
参数定义	`enum FIXED_RATE {` `PHY_RATE_48 = 0x8,` `PHY_RATE_24 = 0x9,` `PHY_RATE_12 = 0xA,` `PHY_RATE_6 = 0xB,` `PHY_RATE_54 = 0xC,` `PHY_RATE_36 = 0xD,` `PHY_RATE_18 = 0xE,` `PHY_RATE_9 = 0xF,` `}` `#define FIXED_RATE_MASK_NONE (0x00)` `#define FIXED_RATE_MASK_STA (0x01)` `#define FIXED_RATE_MASK_AP (0x02)` `#define FIXED_RATE_MASK_ALL (0x03)`
注意	• 当 `enable_mask` 的对应 bit 为 1，ESP8266 Station 或 SoftAP 才会以固定 rate 发送数据。 • 如果 `enable_mask` 设置成 0，则 ESP8266 Station 和 SoftAP 均不会以固定 rate 发送数据。 • ESP8266 Station 和 SoftAP 共享同一个 rate，不支持分别设置为不同 rate 值。
函数定义	`int wifi_set_user_fixed_rate(uint8 enable_mask, uint8 rate)`
参数	• `uint8 enable_mask` - `0x00`：禁用固定 rate - `0x01`：固定 rate 用于 ESP8266 Station 接口 - `0x02`：固定 rate 用于 ESP8266 SoftAP 接口 - `0x03`：固定 rate 用于 ESP8266 Station+SoftAP • `uint8 rate`：固定 rate 值
返回	`0`：成功 其他：失败

3.6.2. wifi_get_user_fixed_rate

功能	获取已经设置的固定 rate 的 mask 和 rate 值
函数定义	`int wifi_get_user_fixed_rate(uint8 *enable_mask, uint8 *rate)`
参数	• `uint8 *enable_mask`：mask 的指针 • `uint8 *rate`：rate 的指针
返回	`0`：成功 其他：失败

3.6.3. wifi_set_user_sup_rate

功能	设置 ESP8266 beacon、probe req/resp 等包里的 support rate 的 IE 中支持的 rate 范围。用于将 ESP8266 支持的通信速率告知通信对方，以限制对方设备的发包速率。
注意	本接口目前仅支持 802.11g 模式，后续会增加支持 802.11b。
参数定义	`enum support_rate {` `RATE_11B5M = 0,` `RATE_11B11M = 1,` `RATE_11B1M = 2,` `RATE_11B2M = 3,` `RATE_11G6M = 4,` `RATE_11G12M = 5,` `RATE_11G24M = 6,` `RATE_11G48M = 7,` `RATE_11G54M = 8,` `RATE_11G9M = 9,` `RATE_11G18M = 10,` `RATE_11G36M = 11,` `};`
函数定义	`int wifi_set_user_sup_rate(uint8 min, uint8 max)`
参数	• `uint8 min`：support rate 下限值，仅支持从 enum support_rate 中取值。 • `uint8 max`：support rate 上限值，仅支持从 enum support_rate 中取值。
返回	`0`：成功 其他：失败
示例	`wifi_set_user_sup_rate(RATE_11G6M, RATE_11G24M);`

3.6.4. wifi_set_user_rate_limit

功能	设置 ESP8266 发包的初始速率范围。重传速率则不受此接口限制。
参数定义	`enum RATE_11B_ID {` `RATE_11B_B11M = 0,` `RATE_11B_B5M = 1,` `RATE_11B_B2M = 2,` `RATE_11B_B1M = 3,` `}` `enum RATE_11G_ID {` `RATE_11G_G54M = 0,` `RATE_11G_G48M = 1,` `RATE_11G_G36M = 2,` `RATE_11G_G24M = 3,` `RATE_11G_G18M = 4,` `RATE_11G_G12M = 5,` `RATE_11G_G9M = 6,` `RATE_11G_G6M = 7` `RATE_11G_B5M = 8,` `RATE_11G_B2M = 9,` `RATE_11G_B1M = 10` `}` `enum RATE_11N_ID {` `RATE_11N_MCS7S = 0,` `RATE_11N_MCS7 = 1,` `RATE_11N_MCS6 = 2,` `RATE_11N_MCS5 = 3,` `RATE_11N_MCS4 = 4,` `RATE_11N_MCS3 = 5,` `RATE_11N_MCS2 = 6,` `RATE_11N_MCS1 = 7,` `RATE_11N_MCS0 = 8,` `RATE_11N_B5M = 9,` `RATE_11N_B2M = 10,` `RATE_11N_B1M = 11` `}`
函数定义	`bool wifi_set_user_rate_limit(uint8 mode, uint8 ifidx, uint8 max, uint8 min)`
参数	• `uint8 mode`：设置模式 `#define RC_LIMIT_11B 0` `#define RC_LIMIT_11G 1` `#define RC_LIMIT_11N 2` • `uint8 ifidx`：设置接口 `0x00 - ESP8266 station 接口` `0x01 - ESP8266 soft-AP 接口` • `uint8 max`：速率上限。请从第一个参数 mode 对应的速率枚举中取值。 • `uint8 min`：速率下限。请从第一个参数 mode 对应的速率枚举中取值。
返回	`true`：成功 `false`：失败
示例	设置 11G 模式下的 ESP8266 station 接口的速率，限制为最大 18M，最小 6M。 `wifi_set_user_rate_limit(RC_LIMIT_11G, 0, RATE_11G_G18M, RATE_11G_G6M);`

3.6.5. wifi_set_user_limit_rate_mask

功能	设置使能受 `wifi_set_user_rate_limit` 限制速率的接口。
参数定义	`#define LIMIT_RATE_MASK_NONE (0x00)` `#define LIMIT_RATE_MASK_STA (0x01)` `#define LIMIT_RATE_MASK_AP (0x02)` `#define LIMIT_RATE_MASK_ALL (0x03)`
函数定义	`bool wifi_ set_user_limit_rate_mask(uint8 enable_mask)`
参数	`uint8 enable_mask` • `0x00`：ESP8266 Station+SoftAP 接口均不受限制 • `0x01`：ESP8266 Station 接口开启限制 • `0x02`：ESP8266 SoftAP 接口开启限制 • `0x03`：ESP8266 Station+SoftAP 接口均开启限制
返回	`true`：成功 `false`：失败

3.6.6. wifi_get_user_limit_rate_mask

功能	查询当前受 `wifi_set_user_rate_limit` 限制速率的接口。
函数定义	`uint8 wifi_get_user_limit_rate_mask(void)`
参数	无
返回	• `0x00`：ESP8266 Station+SoftAP 接口均不受限制 • `0x01`：ESP8266 Station 接口开启限制 • `0x02`：ESP8266 SoftAP 接口开启限制 • `0x03`：ESP8266 Station+SoftAP 接口均开启限制

3.7. 强制休眠接口

强制休眠接口位于 */ESP8266_NONOS_SDK/include/user_interface.h*。

使用强制休眠功能，必须先设置 Wi-Fi 工作模式为 NULL_MODE。从强制休眠中唤醒 ESP8266，或者休眠时间到，进入唤醒回调 (由 `wifi_fpm_set_wakeup_cb` 注册)后，先关闭强制休眠功能，才能再设置 Wi-Fi 工作模式为 Station、SoftAP 或 Station+SoftAP 的正常工作模式运行，具体可参考本章节后文提供的“示例代码”。

注意，

- 定时器会影响进入 Light-sleep 模式，如需 Light-sleep 休眠，请先将定时器关闭。
- Light Sleep 为了降低功耗，将 TCP timer tick 由原本的 250ms 改为了 3s，这将导致 TCP timer 超时时间相应增加；如果用户对 TCP timer 的准确度有要求，请使用 modem sleep 或者 deep sleep 模式。

3.7.1. wifi_fpm_open

功能	开启强制休眠功能
函数定义	`void wifi_fpm_open (void)`
默认值	强制 sleep 功能关闭
返回	无

3.7.2. wifi_fpm_close

功能	关闭强制休眠功能
函数定义	`void wifi_fpm_close (void)`
参数	无
返回	无

3.7.3. wifi_fpm_do_wakeup

功能	唤醒 `MODEM_SLEEP_T` 类型的强制休眠
注意	本接口仅支持在强制休眠功能开启的情况下调用，可在 `wifi_fpm_open` 之后调用；在 `wifi_fpm_close` 之后，不可以调用。
函数定义	`void wifi_fpm_do_wakeup (void)`
返回	无

3.7.4. wifi_fpm_set_wakeup_cb

功能	设置强制休眠的定时唤醒功能超时，系统醒来后的回调函数
注意	• 本接口仅支持在强制休眠功能开启的情况下调用，可在 wifi_fpm_open 之后调用；在 wifi_fpm_close 之后，不可以调用。 • 仅在定时唤醒 wifi_fpm_do_sleep 且参数不为 0xFFFFFFF 功能的定时时间到，系统醒来，才会进入唤醒回调 fpm_wakeup_cb_func。 • MODEM_SLEEP_T 类型的强制休眠被 wifi_fpm_do_wakeup 唤醒，并不会进入唤醒回调。
函数定义	void wifi_fpm_set_wakeup_cb(void (*fpm_wakeup_cb_func)(void))
参数	void (*fpm_wakeup_cb_func)(void)：回调函数
返回	无

3.7.5. wifi_fpm_do_sleep

功能	让系统强制休眠，休眠时间到后，系统将自动醒来。
注意	• 本接口仅支持在强制休眠功能开启的情况下调用，可在 wifi_fpm_open 之后调用；在 wifi_fpm_close 之后，不可以调用。 • 本接口返回 0 表示休眠设置成功，但并不表示立即进入休眠状态。系统会在进入底层相关任务处理时，进行休眠。请勿在调用本接口后，立即调用其他 Wi-Fi 相关操作。
函数定义	int8 wifi_fpm_do_sleep (uint32 sleep_time_in_us)
参数	uint32 sleep_time_in_us：休眠时间，单位：us，取值范围：10000 ~ 268435455(0xFFFFFFF) 如果参数设置为 0xFFFFFFF，则系统将一直休眠，直至： • 若 wifi_fpm_set_sleep_type 设置为 LIGHT_SLEEP_T，可被 GPIO 唤醒。 • 若 wifi_fpm_set_sleep_type 设置为 MODEM_SLEEP_T，可被 wifi_fpm_do_wakeup 唤醒。
返回	0：休眠设置成功 -1：强制休眠的状态错误，休眠失败 -2：强制休眠功能未开启，休眠失败

3.7.6. wifi_fpm_set_sleep_type

功能	设置系统强制休眠的休眠类型。
注意	如需调用本接口，请在 wifi_fpm_open 之前调用。
函数定义	void wifi_fpm_set_sleep_type (enum sleep_type type)

参数	enum sleep_type{ NONE_SLEEP_T = 0, LIGHT_SLEEP_T, MODEM_SLEEP_T, };
返回	无

3.7.7. wifi_fpm_get_sleep_type

功能	查询系统强制休眠的休眠类型。
函数定义	`enum sleep_type wifi_fpm_get_sleep_type (void)`
参数	无
返回	enum sleep_type{ NONE_SLEEP_T = 0, LIGHT_SLEEP_T, MODEM_SLEEP_T, };

3.7.8. wifi_fpm_auto_sleep_set_in_null_mode

功能	设置在关闭 Wi-Fi 模式 `wifi_set_opmode(NULL_MODE)` 的情况下，是否自动进入 Modem-sleep 模式。
函数定义	`void wifi_fpm_auto_sleep_set_in_null_mode (uint8 req)`
参数	uint8 req • 0：关闭 Wi-Fi 模式后，不自动进入 Modem-sleep 模式 • 1：关闭 Wi-Fi 模式后，自动进入 Modem-sleep 模式。
返回	无

3.7.9. 示例代码

调用强制休眠接口，在需要的情况下强制关闭 RF 电路以降低功耗。

> ⚠ **注意：**
>
> 强制休眠接口调用后，并不会立即休眠，而是等到系统 *idle task* 执行时才进入休眠。请参考下述示例使用。

示例一：Modem-sleep 模式

强制进入 Modem-sleep 模式，即强制关闭 RF。

```
        #define FPM_SLEEP_MAX_TIME      0xFFFFFFF

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

void user_func(...)
{
   wifi_station_disconnect();
   wifi_set_opmode(NULL_MODE);              // set WiFi mode to null mode.
   wifi_fpm_set_sleep_type(MODEM_SLEEP_T);      // modem sleep
   wifi_fpm_open();                         // enable force sleep
#ifdef SLEEP_MAX
        /* For modem sleep, FPM_SLEEP_MAX_TIME can only be wakened by calling
wifi_fpm_do_wakeup. */
   wifi_fpm_do_sleep(FPM_SLEEP_MAX_TIME);
#else
        // wakeup automatically when timeout.
   wifi_fpm_set_wakeup_cb(fpm_wakup_cb_func1) ;  // Set wakeup callback
   wifi_fpm_do_sleep(50*1000);
#endif
}

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

示例二：Light-sleep 模式

强制进入 Light-sleep 模式，即强制关闭 RF 和 CPU，需要设置一个回调函数，以便唤醒后程序继续运行。注意，定时器会影响进入 Light-sleep 模式，如需休眠，请先将定时器关闭。

```
                #define FPM_SLEEP_MAX_TIME      0xFFFFFFF
void fpm_wakup_cb_func1(void)
{
   wifi_fpm_close();                    // disable force sleep function
   wifi_set_opmode(STATION_MODE);       // set station mode
   wifi_station_connect();              // connect to AP
}
```

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

   PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U, FUNC_GPIO12);
      wifi_enable_gpio_wakeup(12, GPIO_PIN_INTR_LOLEVEL);

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

3.8. ESP-NOW 接口

ESP-NOW 接口位于 ***/ESP8266_NONOS_SDK/include/espnow.h***。

ESP-NOW 详细介绍请参考文档 ESP-NOW 用户指南，软件接口使用时的注意事项如下：

- ESP-NOW 目前不支持组播包；
- ESP8266_NONOS_SDK_V2.1 及之后版本，ESP-NOW 支持发送广播包，但请注意，仅支持不加密的广播包；
- 建议 slave 和 combo 角色对应 ESP8266 SoftAP 模式或者 SoftAP+Station 共存模式；controller 角色对应 Station 模式；
- 当 ESP8266 处于 SoftAP+Station 共存模式时，若作为 slave 或 combo 角色，将从 SoftAP 接口通信；若作为 controller 角色，将从 Station 接口通信；
- ESP-NOW 不实现休眠唤醒功能，因此如果通信对方的 ESP8266 Station 正处于休眠状态，ESP-NOW 发包将会失败；
- ESP8266 Station 模式下，最多可设置 10 个加密的 ESP-NOW peer，加上不加密的设备，总数不超过 20 个；
- ESP8266 SoftAP 模式或者 SoftAP+station 模式下，最多设置 6 个加密的 ESP-NOW peer，加上不加密的设备，总数不超过 20 个。

3.8.1. 结构体

```
enum esp_now_role {
        ESP_NOW_ROLE_IDLE = 0,
        ESP_NOW_ROLE_CONTROLLER,
        ESP_NOW_ROLE_SLAVE,
        ESP_NOW_ROLE_COMBO, // both slave and controller
        ESP_NOW_ROLE_MAX,
};
```

3.8.2. esp_now_init

功能	初始化 ESP-NOW 功能
函数定义	`init esp_now_init(void)`
参数	无
返回	`0`：成功 其它：失败

3.8.3. esp_now_deinit

功能	卸载 ESP-NOW 功能
函数定义	`int esp_now_deinit(void)`
参数	无
返回	`0`：成功 其它：失败

3.8.4. esp_now_register_recv_cb

功能	注册 ESP-NOW 收包的回调函数
注意	当收到 ESP-NOW 的数据包，进入收包回调函数 `typedef void (*esp_now_recv_cb_t)(u8 *mac_addr, u8 *data, u8 len)` 回调函数的 3 个参数分别为： • `u8 *mac_add`：发包方的 MAC 地址 • `u8 *data`：收到的数据 • `u8 len`：数据长度
函数定义	`int esp_now_register_recv_cb(esp_now_recv_cb_t cb)`
参数	`esp_now_recv_cb_t cb`：回调函数
返回	`0`：成功 其它：失败

3.8.5. esp_now_unregister_recv_cb

功能	注销 ESP-NOW 收包的回调函数
函数定义	`int esp_now_unregister_recv_cb(void)`
参数	无
返回	`0`：成功 其它：失败

3.8.6. esp_now_register_send_cb

功能	设置 ESP-NOW 发包回调函数
注意	• 当发送了 ESP-NOW 的数据包，进入收包回调函数 `void esp_now_send_cb_t(u8 *mac_addr, u8 status)` • 回调函数的 2 个参数分别为： - `u8 *mac_addr`：发包对方的目标 MAC 地址 - `u8 status`：发包状态；0，成功；否则，失败。对应结构体： `mt_tx_status { MT_TX_STATUS_OK = 0, MT_TX_STATUS_FAILED, }` • 发包回调函数不判断密钥是否匹配，如果使用密钥加密，请自行确保密钥正确。 • 设置发送回调函数可以用来判别包是否发送成功（IEEE802.11 MAC 底层是否发送成功）。使用发送回调函数请注意如下情况： ▸ 针对单播包： - 回调函数状态显示成功时，对方应用层实际没有收到的状况。原因： 1. 存在流氓设备进行攻击 2. 加密密钥设置错误 3. 应用层丢包 若需要更强地发包保证发包成功率，请在应用层实现发包握手机制。 - 回调函数状态显示失败时，对方应用层实际已收到的状况。原因： 1. 信道繁忙，未收到对方ACK。 请注意应用层发包重传，接收方需要检测重传包。 ▸ 针对组播包（包括广播包）： - 回调函数状态显示成功，表示组播包已成功发送 - 回调函数状态显示失败，表示组播包发送失败
函数定义	`u8 esp_now_register_send_cb(esp_now_send_cb_t cb)`
参数	`esp_now_send_cb_t cb`：回调函数
返回	`0`：成功 其它：失败

3.8.7. esp_now_unregister_send_cb

功能	注销 ESP-NOW 发包的回调函数，不再报告发包状态。
函数定义	`int esp_now_unregister_send_cb(void)`
参数	无
返回	`0`：成功 其它：失败

3.8.8. esp_now_send

功能	发送 ESP-NOW 数据包
函数定义	`int esp_now_send(u8 *da, u8 *data, int len)`
参数	• `u8 *da`：目的 MAC 地址；如果为 NULL，则遍历 ESP-NOW 维护的所有 MAC 地址进行发送，否则，向指定 MAC 地址发送 • `u8 *data`：要发送的数据 • `int len`：数据长度
返回	`0`：成功 其它：失败

3.8.9. esp_now_add_peer

功能	增加 ESP-NOW 匹配设备，将设备 MAC 地址存入 ESP-NOW 维护的列表。
函数定义	`int esp_now_add_peer(u8 *mac_addr, u8 role,u8 channel, u8 *key, u8 key_len)`
参数	• `u8 *mac_addr`：匹配设备的 MAC 地址 • `u8 channel`：匹配设备的信道值 • `u8 *key`：与该匹配设备通信时，需使用的密钥，目前仅支持 16 字节的密钥 • `u8 key_len`：密钥长度，目前长度仅支持 16 字节
返回	`0`：成功 其它：失败

3.8.10. esp_now_del_peer

功能	删除 ESP-NOW 匹配设备，将设备 MAC 地址从 ESP-NOW 维护的列表中删除。
函数定义	`int esp_now_del_peer(u8 *mac_addr)`
参数	`u8 *mac_addr`：要删除设备的 MAC 地址
返回	`0`：成功 其它：失败

3.8.11. esp_now_set_self_role

功能	设置自身 ESP-NOW 的角色
函数定义	`int esp_now_set_self_role(u8 role)`
参数	`u8 role`：角色类型，详见 `esp_now_role`
返回	`0`：成功 其它：失败

3.8.12. esp_now_get_self_role

功能	查询自身 ESP-NOW 的角色
函数定义	`u8 esp_now_get_self_role(void)`
参数	无
返回	角色类型

3.8.13. esp_now_set_peer_role

功能	设置指定匹配设备的 ESP-NOW 角色。如果重复设置，新设置会覆盖原有设置。
函数定义	`int esp_now_set_peer_role(u8 *mac_addr, u8 role)`
参数	• `u8 *mac_addr`：指定设备的 MAC 地址 • `u8 role`：角色类型，详见 `esp_now_role`
返回	`0`：成功 其它：失败

3.8.14. esp_now_get_peer_role

功能	查询指定匹配设备的 ESP-NOW 角色
函数定义	`int esp_now_get_peer_role(u8 *mac_addr)`
参数	`u8 *mac_addr`：指定设备的 MAC 地址
返回	角色类型，详见 `esp_now_role` 否则，失败

3.8.15. esp_now_set_peer_key

功能	设置指定匹配设备的 ESP-NOW 密钥。如果重复设置，新设置会覆盖原有设置。
函数定义	`int esp_now_set_peer_key(u8 *mac_addr, u8 *key, u8 key_len)`
参数	• `u8 *mac_addr`：指定设备的 MAC 地址 • `u8 *key`：密钥指针，目前仅支持 16 字节的密钥；如果传 NULL，则清除当前密钥 • `u8 key_len`：密钥长度，目前长度仅支持 16 字节
返回	`0`：成功 其它：失败

3.8.16. esp_now_get_peer_key

功能	查询指定匹配设备的 ESP-NOW 密钥
函数定义	`int esp_now_set_peer_key(u8 *mac_addr, u8 *key, u8 *key_len)`
参数	• `u8 *mac_addr`：指定设备的 MAC 地址 • `u8 *key`：查询到的密钥指针，请使用 16 字节的 buffer 保存密钥 • `u8 key_len`：查询到的密钥长度
返回	• `0`：成功 • >0：找到目标设备，但未获得 key • <0：失败

3.8.17. esp_now_set_peer_channel

功能	记录指定匹配设备的信道值。 当与该指定设备进行 ESP-NOW 通信时， • 先调用 `esp_now_get_peer_channel` 查询该设备所在信道； • 再调用 wifi_set_channel 与该设备切换到同一信道进行通信； • 通信完成后，请注意切换回原所在信道。
函数定义	`int esp_now_set_peer_channel(u8 *mac_addr, u8 channel)`
参数	• `u8 *mac_addr`：指定设备的 MAC 地址 • `u8 channel`：信道值，一般为 1 ~ 13，部分地区可能用到 14
返回	`0`：成功 其它：失败

3.8.18. esp_now_get_peer_channel

功能	查询指定匹配设备的信道值。ESP-NOW 要求切换到同一信道进行通信。
函数定义	`int esp_now_get_peer_channel(u8 *mac_addr)`
参数	`u8 *mac_addr`：指定设备的 MAC 地址
返回	1 ~ 13（部分地区可能到 14），成功 否则，失败

3.8.19. esp_now_is_peer_exist

功能	根据 MAC 地址判断设备是否存在
函数定义	`int esp_now_is_peer_exist(u8 *mac_addr)`
参数	`u8 *mac_addr`：指定设备的 MAC 地址

返回	• 0：设备不存在 • >0：出错，查询失败 • <0：设备存在

3.8.20. esp_now_fetch_peer

功能	查询当前指向的 ESP-NOW 配对设备的 MAC 地址，并将内部游标指向 ESP-NOW 维护列表的后一个设备或重新指向 ESP-NOW 维护列表的第一个设备。
注意	• 本接口不可重入。 • 第一次调用本接口时，参数必须为 true，让内部游标指向 ESP-NOW 维护列表的第一个设备。
函数定义	`u8 *esp_now_fetch_peer(bool restart)`
参数	`bool restart` • true：将内部游标重新指向 ESP-NOW 维护列表的第一个设备 • false：将内部游标指向 ESP-NOW 维护列表的后一个设备
返回	• NULL：不存在已关联的 ESP-NOW 设备 • 否则，当前指向的 ESP-NOW 配对设备的 MAC 地址指针

3.8.21. esp_now_get_cnt_info

功能	查询已经匹配的全部设备总数和加密的设备总数。
函数定义	`int esp_now_get_cnt_info(u8 *all_cnt, u8 *encryp_cnt)`
参数	• u8 *all_cnt：已经匹配的全部设备总数 • u8 *encryp_cnt：加密的设备总数
返回	0：成功 其它：失败

3.8.22. esp_now_set_kok

功能	设置用于将通信密钥加密的主密钥 (key of key)。所有设备的通信均共享同一主密钥，如不设置，则使用默认主密钥给通信密钥加密。 如需调用本接口，请在 esp_now_add_peer 和 esp_now_set_peer_key 接口之前调用。
函数定义	`int esp_now_set_kok(u8 *key, u8 len)`
参数	• u8 *key：主密钥指针 • u8 len：主密钥长度，目前长度仅支持 16 字节
返回	0：成功 其它：失败

3.9. Simple-Pair 接口

Simple-Pair 接口位于 */ESP8266_NONOS_SDK/include/simple_pair.h* 中。

3.9.1. 结构体

```
typedef enum {
        SP_ST_STA_FINISH = 0,   // station 端协商结束
        SP_ST_AP_FINISH = 0,    // AP 端协商结束
        SP_ST_AP_RECV_NEG,      // AP 收到 station 发来的协商请求
        SP_ST_STA_AP_REFUSE_NEG,        // station 收到 AP 发来的协商拒绝
        /* definitions below are error codes */
        SP_ST_WAIT_TIMEOUT,     // 错误：协商过程超时
        SP_ST_SEND_ERROR,       // 错误：发送数据出错
        SP_ST_KEY_INSTALL_ERR,  // 错误：密钥安装错误
        SP_ST_KEY_OVERLAP_ERR,  // 错误：同一个 MAC 地址有多个密钥
        SP_ST_OP_ERROR,         // 错误：操作错误
        SP_ST_UNKNOWN_ERROR,    // 错误：未知错误
        SP_ST_MAX,
} SP_ST_t;
```

3.9.2. register_simple_pair_status_cb

功能	注册 Simple-Pair 的状态回调函数
函数定义	`int register_simple_pair_status_cb(simple_pair_status_cb_t cb)`
回调函数定义	`typedef void (*simple_pair_status_cb_t)(u8 *sa, u8 status);` • `u8 *sa`：对方设备的源 MAC 地址 • `u8 status`：状态枚举值，详见 `SP_ST_t` 定义
参数	`simple_pair_status_cb_t cb`：状态回调函数
返回	`0`：成功 其它：失败

3.9.3. unregister_simple_pair_status_cb

功能	注销 Simple-Pair 的状态回调函数
函数定义	`void unregister_simple_pair_status_cb(void)`
参数	无
返回	无

3.9.4. simple_pair_init

功能	初始化 Simple-Pair 功能
函数定义	`int simple_pair_init(void)`
参数	无
返回	`0`：成功 其它：失败

3.9.5. simple_pair_deinit

功能	反初始化 Simple-Pair 功能
函数定义	`void simple_pair_deinit(void)`
参数	无
返回	无

3.9.6. simple_pair_state_reset

功能	重置 Simple-Pair 状态。当需要重新启动 Simple-Pair 时，可调用本接口重置状态。
函数定义	`int simple_pair_state_reset(void)`
参数	无
返回	`0`：成功 其它：失败

3.9.7. simple_pair_ap_enter_announce_mode

功能	Simple-Pair 的 AP 端进入 announce 模式
函数定义	`int simple_pair_ap_enter_announce_mode(void)`
参数	无
返回	`0`：成功 其它：失败

3.9.8. simple_pair_sta_enter_scan_mode

功能	Simple-Pair 的 Station 端进入 scan 模式
函数定义	`int simple_pair_sta_enter_scan_mode(void)`
参数	无
返回	`0`：成功 其它：失败

3.9.9. simple_pair_sta_start_negotiate

功能	Simple-Pair 的 Station 端开始协商
函数定义	`int simple_pair_sta_start_negotiate(void)`
参数	无
返回	`0`：成功 其它：失败

3.9.10. simple_pair_ap_start_negotiate

功能	Simple-Pair 的 AP 端同意协商
函数定义	`int simple_pair_ap_start_negotiate(void)`
参数	无
返回	`0`：成功 其它：失败

3.9.11. simple_pair_ap_refuse_negotiate

功能	Simple-Pair 的 AP 端拒绝协商
函数定义	`int simple_pair_ap_refuse_negotiate(void)`
参数	无
返回	`0`：成功 其它：失败

3.9.12. simple_pair_set_peer_ref

功能	设置需要协商的设备的参数。仅向系统设置信息，不会安装 Key 或进行其他动作。 • 若作为 Station 端，则需要在 `simple_pair_sta_start_negotiate` 之前设置 • 若作为 AP 端，则需要在 `simple_pair_ap_start_negotiate` 或者 `simple_pair_ap_refuse_negotiate` 之前设置。
函数定义	`int simple_pair_set_peer_ref(u8 *peer_mac, u8 *tmp_key, u8 *ex_key)`
参数	• `u8 *peer_mac`：需要协商的对端设备的 MAC 地址，长度为 6 字节。不能为 `NULL` • `u8 *tmp_key`：用于加密 Simple-Pair 通信的临时密钥，长度为 16 字节。不能为 `NULL` • `u8 *ex_key`：需要交换的 Key，长度为 16 字节。如果为 `NULL`，则系统默认使用全 0 为 Key
返回	`0`：成功 其它：失败

3.9.13. simple_pair_get_peer_ref

功能	获取设置的参数值。无需获取的参数，传入 NULL 即可。
函数定义	`int simple_pair_get_peer_ref(u8 *peer_mac, u8 *tmp_key, u8 *ex_key)`
参数	• `u8 *peer_mac`：需要协商的设备 MAC 地址，长度为 6 字节 • `u8 *tmp_key`：用于加密 Simple-Pair 通信的临时密钥，长度为 16 字节 • `u8 *ex_key`：需要交换的 Key，长度为 16 字节。如未设置，则查询到全 `0` 值
返回	`0`：成功 其它：失败

3.10. 云端升级 (FOTA) 接口

云端升级接口位于 */ESP8266_NONOS_SDK/include/user_interface.h* 和 *upgrade.h* 中。

3.10.1. system_upgrade_userbin_check

功能	查询 user bin
函数定义	`uint8 system_upgrade_userbin_check()`
参数	无
返回	`0x00 : UPGRADE_FW_BIN1`, i.e. ***user1.bin*** `0x01 : UPGRADE_FW_BIN2`, i.e. ***user2.bin***

3.10.2. system_upgrade_flag_set

功能	设置升级状态标志
注意	• 若调用 `system_upgrade_start` 升级，本接口无需调用。 • 若用户调用 `spi_flash_write` 自行写 Flash 实现升级，新软件写入完成后，将 flag 置为 `UPGRADE_FLAG_FINISH`， 再调用 `system_upgrade_reboot` 重启运行新软件。
函数定义	`void system_upgrade_flag_set(uint8 flag)`
参数	`uint8 flag:` `#define UPGRADE_FLAG_IDLE 0x00` `#define UPGRADE_FLAG_START 0x01` `#define UPGRADE_FLAG_FINISH 0x02`
返回	无

3.10.3. system_upgrade_flag_check

功能	查询升级状态标志
函数定义	`uint8 system_upgrade_flag_check()`
参数	无
返回	`#define UPGRADE_FLAG_IDLE 0x00` `#define UPGRADE_FLAG_START 0x01` `#define UPGRADE_FLAG_FINISH 0x02`

3.10.4. 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`：升级服务器的相关参数
返回	`true`：开始升级 `false`：已经在升级过程中，无法开始升级

3.10.5. system_upgrade_reboot

功能	重启系统，运行新软件
函数定义	`void system_upgrade_reboot (void)`
参数	无
返回	无

3.11. Sniffer 相关接口

Sniffer 接口位于 */ESP8266_NONOS_SDK/include/user_interface.h*。

3.11.1. wifi_promiscuous_enable

功能	开启混杂模式 (sniffer)
注意	• 仅支持在 ESP8266 单 Station 模式下，开启混杂模式 • 混杂模式中，ESP8266 Station 和 SoftAP 接口均失效 • 若开启混杂模式，请先调用 `wifi_station_disconnect` 确保没有连接 • 混杂模式中请勿调用其他 API，请先调用 `wifi_promiscuous_enable(0)` 退出 sniffer
函数定义	`void wifi_promiscuous_enable(uint8 promiscuous)`
参数	`uint8 promiscuous` • `0`：关闭混杂模式 • `1`：开启混杂模式
返回	无
示例	用户可以向乐鑫申请 sniffer demo

3.11.2. wifi_promiscuous_set_mac

功能	设置 sniffer 模式时的 MAC 地址过滤，可过滤出发给指定 MAC 地址的包（也包含广播包）。
注意	• 本接口需在 `wifi_promiscuous_enable(1)` 使能混杂模式后调用； • MAC 地址过滤仅对当前这次的 sniffer 有效； • 如果停止 sniffer，又再次 sniffer，需要重新设置 MAC 地址过滤。
函数定义	`void wifi_promiscuous_set_mac(const uint8_t *address)`
参数	`const uint8_t *address`：MAC 地址
返回	无
示例	`char ap_mac[6] = {0x16, 0x34, 0x56, 0x78, 0x90, 0xab};` `wifi_promiscuous_set_mac(ap_mac);`

3.11.3. wifi_set_promiscuous_rx_cb

功能	注册混杂模式下的接收数据回调函数，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)`
参数	`wifi_promiscuous_cb_t cb`：回调函数
返回	无

3.11.4. wifi_get_channel

功能	获取信道号
函数定义	`uint8 wifi_get_channel(void)`
参数	无
返回	信道号

3.11.5. wifi_set_channel

功能	设置信道号，用于混杂模式
函数定义	`bool wifi_set_channel (uint8 channel)`
参数	`uint8 channel`：信道号
返回	`true`：成功 `false`：失败

3.12. SmartConfig 接口

Smart Config 接口位于 ***/ESP8266_NONOS_SDK/include/smartconfig.h***。

AirKiss 接口位于 ***/ESP8266_NONOS_SDK/include/airkiss.h***。

开启 SmartConfig 功能前，请先确保 AP 已经开启。

3.12.1. smartconfig_start

功能	开启快连模式，快速连接 ESP8266 Station 到 AP。ESP8266 抓取空中特殊的数据包，包含目标 AP 的 SSID 和 password 信息，同时，用户需要通过手机或者电脑广播加密的 SSID 和 password 信息。
注意	• 仅支持在单 Station 模式下调用本接口 • SmartConfig 过程中，ESP8266 Station 和 SoftAP 失效 • `smartconfig_start` 未完成之前不可重复执行 `smartconfig_start`，请先调用 `smartconfig_stop` 结束本次快连。 • SmartConfig 过程中，请勿调用其他 API；先调用 `smartconfig_stop`，再使用其他 API。
结构体	`typedef enum {` `SC_STATUS_WAIT = 0, // 连接未开始，请勿在此阶段开始连接` `SC_STATUS_FIND_CHANNEL, // 请在此阶段开启 APP 进行配对连接` `SC_STATUS_GETTING_SSID_PSWD,` `SC_STATUS_LINK,` `SC_STATUS_LINK_OVER, // 获取到 IP，连接路由完成` `} sc_status;`
函数定义	`bool smartconfig_start(sc_callback_t cb, uint8 log)`
参数	• `sc_callback_t cb`：SmartConfig 状态发生改变时，进入回调函数。 • 传入回调函数的参数 `status` 表示 SmartConfig 状态： - 当 `status` 为 `SC_STATUS_GETTING_SSID_PSWD` 时，参数 `void *pdata` 为 `sc_type *` 类型的指针变量，表示此次配置是 AirKiss 还是 ESP-TOUCH； - 当 `status` 为 `SC_STATUS_LINK` 时，参数 `void *pdata` 为 `struct station_config` 类型的指针变量； - 当 `status` 为 `SC_STATUS_LINK_OVER` 时，参数 `void *pdata` 是移动端的 IP 地址的指针，4 个字节。（仅支持在 ESP-TOUCH 方式下，其他方式则为 NULL） - 当 `status` 为其他状态时，参数 `void *pdata` 为 `NULL` • `uint8 log` - 1：UART 打印连接过程 - 否则：UART 仅打印连接结果。打印信息仅供调试使用，正常工作时，应避免 SmartConfig 过程中进行串口打印。
返回	`true`：成功 `false`：失败

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

3.12.2. smartconfig_stop

功能	关闭快连模式，释放 smartconfig_start 占用的内存。
注意	• 若快连成功，连上目标 AP 后，调用本接口释放 smartconfig_start 占用的内存。 • 若快连失败，调用本接口退出快连模式，释放占用的内存。
函数定义	`bool smartconfig_stop(void)`
参数	无
返回	true：成功 false：失败

3.12.3. smartconfig_set_type

功能	设置快连模式的协议类型。
注意	如需调用本接口，请在 smartconfig_start 之前调用。

函数定义	`bool smartconfig_set_type(sc_type type)`
参数	`typedef enum {` `  SC_TYPE_ESPTOUCH = 0,` `  SC_TYPE_AIRKISS,` `  SC_TYPE_ESPTOUCH_AIRKISS,` `} sc_type;`
返回	`true`：成功 `false`：失败

3.12.4. airkiss_version

功能	获得 AirKiss 库的版本信息。
注意	版本信息的实际长度未知。
函数定义	`const char* airkiss_version(void)`
参数	无
返回	AirKiss 库的版本信息。

3.12.5. airkiss_lan_recv

功能	用于 AirKiss 内网发现功能，可参考微信官网内网发现功能介绍 http://iot.weixin.qq.com 内网发现功能大致流程为：创建一个 UDP 传输，在 UDP 的 `espconn_recv_callback` 中，将接收到的 UDP 报文传入 `airkiss_lan_recv` 函数，若函数返回 `AIRKISS_LAN_SSDP_REQ`，则调用 `airkiss_lan_pack` 打包响应报文，通过 UDP 传输回复给发送方。 本函数用于接收 AirKiss 发来的 UDP 数据包并解析。
函数定义	`int airkiss_lan_recv(` `const void* body,` `unsigned short length,` `const airkiss_config_t* config)`
参数	• `const void* body`：接收到的 UDP 数据 • `unsigned short length`：有效的数据长度 • `airkiss_config_t* config`：AirKiss 结构体
返回	• >=0：成功 • <0：失败 • 具体可参考 `airkiss_lan_ret_t`

3.12.6. airkiss_lan_pack

功能	用于 AirKiss 内网发现功能，将用户数据组织成 AirKiss 内网探测的 UDP 数据包格式。
函数定义	`int airkiss_lan_pack(` `airkiss_lan_cmdid_t ak_lan_cmdid,` `void* appid,` `void* deviceid,` `void* _datain,` ` unsigned short inlength,` `void* _dataout,` `unsigned short* outlength,` `const airkiss_config_t* config)`
参数	• `airkiss_lan_cmdid_t ak_lan_cmdid`：发包的类型 • `void* appid`：微信公众号，必须从微信获得 • `void* deviceid`：设备 ID 值，必须从微信获得 • `void* _datain`：待组包的用户数据 • `unsigned short inlength`：用户数据长度 • `void* _dataout`：用户数据完成 AirKiss 内网探测组包后的数据 • `unsigned short* outlength`：组包后的数据长度 • `const airkiss_config_t* config`：AirKiss 结构体
返回	• >=0：成功 • <0：失败 • 具体可参考 `airkiss_lan_ret_t`

3.13. SNTP 接口

SNTP 接口位于 */ESP8266_NONOS_SDK/include/sntp.h*。

3.13.1. sntp_setserver

功能	通过 IP 地址设置 SNTP 服务器，一共最多支持设置 3 个 SNTP 服务器
函数定义	`void sntp_setserver(unsigned char idx, ip_addr_t *addr)`
参数	`unsigned char idx`：SNTP 服务器编号，最多支持 3 个 SNTP 服务器（0 ~ 2）；0 号为主服务器，1 号和 2 号为备用服务器。 `ip_addr_t *addr`：IP 地址；用户需自行确保，传入的是合法 SNTP 服务器
返回	无

3.13.2. sntp_getserver

功能	查询 SNTP 服务器的 IP 地址，对应的设置接口为：`sntp_setserver`
函数定义	`ip_addr_t sntp_getserver(unsigned char idx)`
参数	`unsigned char idx`：SNTP 服务器编号，最多支持 3 个 SNTP 服务器（0 ~ 2）
返回	IP 地址

3.13.3. sntp_setservername

功能	通过域名设置 SNTP 服务器，一共最多支持设置 3 个 SNTP 服务器
函数定义	`void sntp_setservername(unsigned char idx, char *server)`
参数	`unsigned char idx`：SNTP 服务器编号，最多支持 3 个 SNTP 服务器（0 ~ 2）；0 号为主服务器，1 号和 2 号为备用服务器。 `char *server`：域名；用户需自行确保，传入的是合法 SNTP 服务器
返回	无

3.13.4. sntp_getservername

功能	查询 SNTP 服务器的域名，仅支持查询通过 `sntp_setservername` 设置的 SNTP 服务器
函数定义	`char * sntp_getservername(unsigned char idx)`
参数	`unsigned char idx`：SNTP 服务器编号，最多支持 3 个 SNTP 服务器（0 ~ 2）
返回	服务器域名

3.13.5. sntp_init

功能	SNTP 初始化

函数定义	`void sntp_init(void)`
参数	无
返回	无

3.13.6. sntp_stop

功能	SNTP 关闭
函数定义	`void sntp_stop(void)`
参数	无
返回	无

3.13.7. sntp_get_current_timestamp

功能	查询当前距离基准时间 (1970.01.01 00：00：00 GMT + 8) 的时间戳，单位：秒
函数定义	`uint32 sntp_get_current_timestamp()`
参数	无
返回	距离基准时间的时间戳

3.13.8. sntp_get_real_time

功能	查询实际时间 (GMT + 8)
函数定义	`char* sntp_get_real_time(long t)`
参数	`long t`：与基准时间相距的时间戳
返回	实际时间

3.13.9. sntp_set_timezone

功能	设置时区信息
注意	调用本接口前，请先调用 `sntp_stop`
函数定义	`bool sntp_set_timezone (sint8 timezone)`
参数	`sint8 timezone`：时区值，参数范围：-11 ~ 13
返回	`true`：成功 `false`：失败
示例	`sntp_stop();` `if( true == sntp_set_timezone(-5) ) {` `  sntp_init();` `}`

3.13.10.sntp_get_timezone

功能	查询时区信息
函数定义	`sint8 sntp_get_timezone (void)`
参数	无
返回	时区值，参数范围：-11 ~ 13

3.13.11.SNTP 示例

Step 1. enable sntp

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

Step 2. set a timer to check sntp timestamp

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

Step 3. timer callback

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

3.14. WPA2-Enterprise 接口

ESP8266 Station 接口支持连接到 WPA2_Enterprise 企业级加密的 AP。

WPA2_Enterprise 接口位于 ***/ESP8266_NONOS_SDK/include/wpa2_enterprise.h***。

3.14.1. wifi_station_set_wpa2_enterprise_auth

功能	使能 WPA2_Enterprise 企业级加密的验证。 • 连接 WPA2_Enterprise AP，需调用 `wifi_station_set_wpa2_enterprise_auth(1);` 使能验证。 • 之后如再需连接普通 AP，则需要先调用 `wifi_station_set_wpa2_enterprise_auth(0);` 清除状态。
函数定义	`int wifi_station_set_wpa2_enterprise_auth(int enable)`
参数	`int enable` • `0`：清除当前 WPA2_Enterprise 状态； • 非 0：使能 WPA2_Enterprise 企业级加密的验证。
返回	`0`：成功 其它：失败

3.14.2. wifi_station_set_enterprise_cert_key

功能	设置 ESP8266 Station 接口连接 WPA2_Enterprise AP 所需的用户证书及密钥，用于 EAP-TLS 认证。
注意	• 支持 WPA2-ENTERPRISE AP 需占用 26 KB 以上的内存，调用本接口时请注意内存是否足够。 • 目前 WPA2-ENTERPRISE 只支持非加密的私钥文件和证书文件，且仅支持 PEM 格式 - 支持的证书文件头信息为：- - - - - BEGIN CERTIFICATE - - - - - - 支持的私钥文件头信息为：- - - - - BEGIN RSA PRIVATE KEY - - - - - 或者 - - - - - BEGIN PRIVATE KEY - - - - - • 请在连接 WPA2_Enterprise AP 之前调用本接口设置私钥文件和证书文件，在成功连接 AP 后先调用 `wifi_station_clear_enterprise_cert_key` 清除内部状态，应用层再释放私钥文件和证书文件信息。 • 如果遇到加密的私钥文件，请使用 `openssl pkey` 命令改为非加密文件使用，或者使用 `openssl rsa` 等命令，对某些私钥文件进行加密-非加密的转换（或起始 TAG 转化）。
函数定义	`int wifi_station_set_enterprise_cert_key(` `u8 *client_cert, int client_cert_len,` `u8 *private_key, int private_key_len,` `u8 *private_key_passwd, int private_key_passwd_len)`

参数	`uint8 *client_cert`：十六进制数组的证书指针 `int client_cert_len`：证书长度 `uint8 *private_key`：十六进制数组的私钥指针，暂不支持超过 2048 的私钥 `int private_key_len`：私钥长度，请勿超过 2048 `uint8 *private_key_passwd`：私钥的提取密码，目前暂不支持，请传入 NULL `int private_key_passwd_len`：提取密码的长度，目前暂不支持，请传入 0
返回	`0`：成功 非 0：失败
示例	假设私钥文件的信息为 - - - - - BEGIN PRIVATE KEY - - - - - … … … … 那么对应的数组为：uint8 key[]={0x2d, 0x2d, 0x2d, 0x2d, 0x2d, 0x42, 0x45, 0x47, … … 0x00 }; 即各字符的 ASCII 码，请注意，数组必须添加 0x00 作为结尾。

3.14.3. wifi_station_clear_enterprise_cert_key

功能	释放连接 WPA2_Enterprise AP 使用用户证书和密钥所占用的资源，并清除相关状态。
函数定义	`void wifi_station_clear_enterprise_cert_key (void)`
参数	无
返回	无

3.14.4. wifi_station_set_enterprise_ca_cert

功能	设置 ESP8266 Station 接口连接 WPA2_Enterprise AP 使用的根证书。EAP-TTLS/PEAP 认证方法可选对根证书进行验证。
函数定义	`int  wifi_station_set_enterprise_ca_cert(u8 *ca_cert, int ca_cert_len)`
参数	• `u8 *ca_cert`：十六进制数组的根证书指针 • `int ca_cert_len`：根证书长度
返回	`0`：成功 其它：失败

3.14.5. wifi_station_clear_enterprise_ca_cert

功能	释放连接 WPA2_Enterprise AP 使用根证书占用的资源，并清除相关状态。
函数定义	`void wifi_station_clear_enterprise_ca_cert (void)`
参数	无
返回	无

3.14.6. wifi_station_set_enterprise_username

功能	设置连接 WPA2_Enterprise AP 时，ESP8266 Station 的用户名。

注意	WPA2_Enterprise 企业级加密方法调用本接口设置用户身份， • 对于 EAP-TTLS 以及 EAP-PEAP 认证必须设置，并且是用在了认证的第二阶段，只有认证服务器支持的用户身份才能通过认证。 • 对于 EAP-TLS 认证方法则为可选项，即使没有设置用户名，也可以通过匿名的身份通过验证。
函数定义	`int wifi_station_set_enterprise_username (u8 *username, int len)`
参数	• `u8 *username`：用户名称 • `int len`：名称长度
返回	`0`：成功 其它：失败

3.14.7. wifi_station_clear_enterprise_username

功能	释放连接 WPA2_Enterprise AP 设置用户名占用的资源，并清除相关状态。
函数定义	`void wifi_station_clear_enterprise_username (void)`
参数	无
返回	无

3.14.8. wifi_station_set_enterprise_password

功能	设置用户密码，用于通过 EAP-TTLS/EAP-PEAP 认证方法。
函数定义	`int wifi_station_set_enterprise_password (u8 *password, int len)`
参数	• `u8 *password`：用户密码 • `int len`：密码长度
返回	`0`：成功 其它：失败

3.14.9. wifi_station_clear_enterprise_password

功能	释放连接 WPA2_Enterprise AP 设置密码占用的资源，并清除相关状态。
函数定义	`void wifi_station_clear_enterprise_password (void)`
参数	无
返回	无

3.14.10.wifi_station_set_enterprise_new_password

功能	设置新用户密码，针对 MSCHAPV2 方法。
函数定义	`int wifi_station_set_enterprise_new_password (u8 *new_password, int len)`

参数	• `u8 *new_password`：新用户密码 • `int len`：密码长度
返回	`0`：成功 其它：失败

3.14.11.wifi_station_clear_enterprise_new_password

功能	释放连接 WPA2_Enterprise AP 设置新用户密码占用的资源，并清除相关状态。
函数定义	`void wifi_station_clear_enterprise_new_password (void)`
参数	无
返回	无

3.14.12.wifi_station_set_enterprise_disable_time_check

功能	设置认证时，是否检查过期时间。默认情况下，认证过程中不检查过期时间。
函数定义	`void wifi_station_set_enterprise_disable_time_check(bool disable)`
参数	`bool disable` • `true`：不检查过期时间 • `false`：检查过期时间，需调用 `wpa2_enterprise_set_user_get_time` 注册回调函数。
返回	无

3.14.13.wifi_station_get_enterprise_disable_time_check

功能	查询认证时，是否检查过期时间。
函数定义	`bool wifi_station_get_enterprise_disable_time_check (void)`
参数	无
返回	`true`：不检查过期时间 `false`：检查过期时间

3.14.14.wpa2_enterprise_set_user_get_time

功能	设置认证过程中从用户获得时间的回调函数。 需调用 `wifi_station_set_enterprise_disable_time_check(false);` 使能检查过期时间。
函数定义	`void wpa2_enterprise_set_user_get_time(get_time_func_t cb)`
参数	`get_time_func_t cb`：回调函数
返回	无

示例	static int sys_get_current_time(struct os_time *t) { t->sec = CURRENT_TIME; // User set current time. return 0; } //Set Callback wpa2_enterprise_set_user_get_time(sys_get_current_time); //Enable Time check wifi_station_set_enterprise_disable_time_check(false);

3.14.15.示例流程

如需连接 WPA2_Enterprise 加密的 AP，流程如下：

1. `wifi_station_set_config` 配置需连接的 AP 信息。
2. `wifi_station_set_wpa2_enterprise_auth(1);` 使能 WPA2_Enterprise 加密验证。
   - 2.1. 若为 EAP-TLS 认证，调用 `wifi_station_set_enterprise_cert_key` 设置证书和密钥。可选调用 `wifi_station_set_enterprise_username` 设置用户名。
   - 2.2. 若为 EAP-TTLS 或 EAP-PEAP 认证，调用 `wifi_station_set_enterprise_username` 以及 `wifi_station_set_enterprise_password`，设置用户名和密码。可选调用 `wifi_station_set_enterprise_ca_cert` 设置根证书。
3. `wifi_station_connnect` 连接 AP。
4. 成功连接 AP 或连接 AP 失败并不再重试后，调用 `wifi_station_clear_enterprise_XXX` 对应接口释放资源。

4. TCP/UDP 接口

位于 *ESP8266_NONOS_SDK/include/espconn.h*。

网络相关接口可分为以下几类：

- **通用接口**：TCP 和 UDP 均可以调用的接口。
- **TCP APIs**：仅建立 TCP 连接时，使用的接口。
- **UDP APIs**：仅收发 UDP 包时，使用的接口。
- **mDNS APIs**：mDNS 相关接口。

4.1. 通用接口

4.1.1. espconn_delete

功能	删除传输连接。
注意	对应创建传输的接口如下： TCP：`espconn_accept` UDP：`espconn_create`
函数定义	`sint8 espconn_delete(struct espconn *espconn)`
参数	`struct espconn *espconn`：对应网络传输的结构体
返回	`0`：成功 其它：失败，返回错误码 • `ESPCONN_ARG`：未找到参数 `espconn` 对应的网络传输 • `ESPCONN_INPROGRESS`：参数 `espconn` 对应的网络连接仍未断开，请先调用 `espconn_disconnect` 断开连接，再进行删除。

4.1.2. espconn_gethostbyname

功能	DNS 功能
函数定义	`err_t espconn_gethostbyname(` `  struct espconn *pespconn,` `  const char *hostname,` `  ip_addr_t *addr,` `  dns_found_callback found` `)`

参数	• `struct espconn *espconn`：对应网络传输的结构体 • `const char *hostname`：域名字符串的指针 • `ip_addr_t *addr`：IP 地址 • `dns_found_callback found`：DNS 回调函数
返回	`err_t` • `ESPCONN_OK`：成功 • `ESPCONN_ISCONN`：失败，错误码含义：已经连接 • `ESPCONN_ARG`：失败，错误码含义：未找到参数 `espconn` 对应的网络传输
示例	请参考 ***IoT_Demo***： `ip_addr_t esp_server_ip;` `LOCAL void ICACHE_FLASH_ATTR` `user_esp_platform_dns_found(const char *name, ip_addr_t *ipaddr, void *arg) {` `  struct espconn *pespconn = (struct espconn *)arg;` `  if (ipaddr != NULL)` `  os_printf(user_esp_platform_dns_found %d.%d.%d.%d/n,` `  *((uint8 *)&ipaddr->addr), *((uint8 *)&ipaddr->addr + 1),` `  *((uint8 *)&ipaddr->addr + 2), *((uint8 *)&ipaddr->addr + 3));` `}` `void dns_test(void) {` `  espconn_gethostbyname(pespconn, “iot.espressif.cn” , &esp_server_ip,` `  user_esp_platform_dns_found);` `}`

4.1.3. espconn_port

功能	获取 ESP8266 可用的端口
函数定义	`uint32 espconn_port(void)`
参数	无
返回	端口号

4.1.4. espconn_regist_sentcb

功能	注册网络数据发送成功的回调函数
函数定义	`sint8 espconn_regist_sentcb(` `  struct espconn *espconn,` `  espconn_sent_callback sent_cb` `)`
参数	`struct espconn *espconn`：对应网络传输的结构体 `espconn_connect_callback connect_cb`：成功接收网络数据的回调函数
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的网络传输

4.1.5. espconn_regist_recvcb

功能	注册成功接收网络数据的回调函数
函数定义	`sint8 espconn_regist_recvcb(` `  struct espconn *espconn,` `  espconn_recv_callback recv_cb` `)`
参数	`struct espconn *espconn`：对应网络传输的结构体 `espconn_connect_callback connect_cb`：成功接收网络数据的回调函数
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的网络传输

4.1.6. espconn_sent_callback

功能	网络数据发送成功的回调函数，由 `espconn_regist_sentcb` 注册
函数定义	`void espconn_sent_callback (void *arg)`
参数	`void *arg`：回调函数的参数，网络传输的结构体 `espconn` 指针。 注意，本指针为底层维护的指针，不同回调传入的指针地址可能不一样，请勿依此判断网络连接。可根据 `espconn` 结构体中的 `remote_ip, remote_port` 判断多连接中的不同网络传输。
返回	无

4.1.7. espconn_recv_callback

功能	成功接收网络数据的回调函数，由 `espconn_regist_recvcb` 注册
函数定义	`void espconn_recv_callback (` `  void *arg,` `  char *pdata,` `  unsigned short len` `)`
参数	• `void *arg`：回调函数的参数，网络传输结构体 `espconn` 指针。注意，本指针为底层维护的指针，不同回调传入的指针地址可能不一样，请勿依此判断网络连接。可根据 `espconn` 结构体中的 `remote_ip`、`remote_port` 判断多连接中的不同网络传输。 • `char *pdata`：接收到的数据 • `unsigned short len`：接收到的数据长度
返回	无

4.1.8. espconn_get_connection_info

功能	查询某个 TCP 连接或者 UDP 传输的远端信息。一般在 `espconn_recv_callback` 中调用。
函数定义	`sint8 espconn_get_connection_info(` `  struct espconn *espconn,` `  remot_info **pcon_info,` `  uint8 typeflags` `)`

参数	• `struct espconn *espconn`：对应网络连接的结构体 • `remot_info **pcon_info`：连接 client 信息 • `uint8 typeflags` - `0`：正常 server - `1`：SSL server
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接
示例	`void user_udp_recv_cb(void *arg, char *pusrdata, unsigned short length)` `{` `struct espconn *pesp_conn = arg;` `remot_info *premot = NULL;` `if (espconn_get_connection_info(pesp_conn,&premot,0) == ESPCONN_OK){` `pesp_conn->proto.tcp->remote_port = premot->remote_port;` `pesp_conn->proto.tcp->remote_ip[0] = premot->remote_ip[0];` `pesp_conn->proto.tcp->remote_ip[1] = premot->remote_ip[1];` `pesp_conn->proto.tcp->remote_ip[2] = premot->remote_ip[2];` `pesp_conn->proto.tcp->remote_ip[3] = premot->remote_ip[3];` `espconn_sent(pesp_conn, pusrdata, os_strlen(pusrdata));` `}` `}`

4.1.9. espconn_send

功能	通过 WiFi 发送数据
注意	• 一般情况，请在前一包数据发送成功，进入 `espconn_sent_callback` 后，再调用 `espconn_send` 发送下一包数据。 • 如果是 UDP 传输，请在每次调用 `espconn_send` 前，设置 `espconn->proto.udp->remote_ip` 和 `remote_port` 参数，因为 UDP 无连接，远端信息可能被更改。
函数定义	`sint8 espconn_send(` `struct espconn *espconn,` `uint8 *psent,` `uint16 length` `)`
参数	`struct espconn *espconn`：对应网络传输的结构体 `uint8 *psent`：发送的数据 `uint16 length`：发送的数据长度

返回	0：成功 其它：失败，返回错误码 • ESPCONN_ARG：未找到参数 espconn 对应的网络传输 • ESPCONN_MEM：空间不足 • ESPCONN_MAXNUM：底层发包缓存已满，发包失败 • ESPCONN_IF：UDP 发包失败

4.1.10. espconn_sent

[@deprecated] 本接口不建议使用，建议使用 espconn_send 代替。

功能	通过 WiFi 发送数据
注意	• 一般情况，请在前一包数据发送成功，进入 espconn_sent_callback 后，再调用 espconn_send 发送下一包数据。 • 如果是 UDP 传输，请在每次调用 espconn_send 前，设置 espconn->proto.udp->remote_ip 和 remote_port 参数，因为 UDP 无连接，远端信息可能被更改。
函数定义	`sint8 espconn_sent(` `  struct espconn *espconn,` `  uint8 *psent,` `  uint16 length` `)`
参数	struct espconn *espconn：对应网络传输的结构体 uint8 *psent：发送的数据 uint16 length：发送的数据长度
返回	0：成功 其它：失败，返回错误码 • ESPCONN_ARG：未找到参数 espconn 对应的网络传输 • ESPCONN_MEM：空间不足 • ESPCONN_MAXNUM：底层发包缓存已满，发包失败 • ESPCONN_IF：UDP 发包失败

4.2. TCP 接口

TCP 接口仅用于 TCP 连接，请勿用于 UDP 传输。

4.2.1. espconn_accept

功能	创建 TCP server，建立侦听
函数定义	`sint8 espconn_accept(struct espconn *espconn)`
参数	`struct espconn *espconn`：对应网络连接的结构体
返回	`0`：成功 其它：失败，返回错误码 • `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接 • `ESPCONN_MEM`：空间不足 • `ESPCONN_ISCONN`：连接已经建立

4.2.2. espconn_regist_time

功能	注册 ESP8266 TCP server 超时时间，时间值仅作参考，并不精确。
注意	• 请在 `espconn_accept` 之后，连接未建立之前，调用本接口。本接口不能用于 SSL 连接。 • 如果超时时间设置为 0，ESP8266 TCP server 将始终不会断开已经不与它通信的 TCP client，不建议这样使用。
函数定义	`sint8 espconn_regist_time(` `  struct espconn *espconn,` `  uint32 interval,` `  uint8 type_flag` `)`
参数	• `struct espconn *espconn`：对应网络连接的结构体 • `uint32 interval`：超时时间，单位：秒，最大值：7200 秒 • `uint8 type_flag` - `0`：对所有 TCP 连接生效 - `1`：仅对某一 TCP 连接生效
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接

4.2.3. espconn_connect

功能	连接 TCP server（ESP8266 作为 TCP client）。
注意	• 如果 `espconn_connect` 失败，返回非零值，连接未建立，不会进入任何 `espconn` callback。 • 建议使用 `espconn_port` 接口，设置一个可用的端口号。
函数定义	`sint8 espconn_connect(struct espconn *espconn)`
参数	`struct espconn *espconn`：对应网络连接的结构体

返回	0：成功 其它：失败，返回错误码 • ESPCONN_ARG：未找到参数 espconn 对应的 TCP 连接 • ESPCONN_MEM：空间不足 • ESPCONN_ISCONN：连接已经建立 • ESPCONN_RTE：路由异常

4.2.4. espconn_regist_connectcb

功能	注册 TCP 连接成功建立后的回调函数。
函数定义	sint8 espconn_regist_connectcb(struct espconn *espconn, espconn_connect_callback connect_cb)
参数	• struct espconn *espconn：对应网络连接的结构体 • espconn_connect_callback connect_cb：成功建立 TCP 连接后的回调函数
返回	0：成功 其它：失败，返回错误码 ESPCONN_ARG：未找到参数 espconn 对应的 TCP 连接

4.2.5. espconn_connect_callback

功能	成功建立 TCP 连接的回调函数，由 espconn_regist_connectcb 注册。ESP8266 作为 TCP server 侦听到 TCP client 连入；或者 ESP8266 作为 TCP client 成功与 TCP server 建立连接。
函数定义	void espconn_connect_callback (void *arg)
参数	• void *arg：回调函数的参数，对应网络连接的结构体 espconn 指针。 • 注意，本指针为底层维护的指针，不同回调传入的指针地址可能不一样，请勿依此判断网络连接。可根据 espconn 结构体中的 remote_ip, remote_port 判断多连接中的不同网络传输。
返回	无

4.2.6. espconn_set_opt

功能	设置 TCP 连接的相关配置，对应清除配置标志位的接口为 espconn_clear_opt
注意	• SSL 连接不支持使用本接口 • 一般情况下，无需调用本接口；如需设置 espconn_set_opt 请在 espconn_connect_callback 中调用
函数定义	sint8 espconn_set_opt(struct espconn *espconn, uint8 opt)

参数	• `struct espconn *espconn`：对应网络连接的结构体 • `uint8 opt`：TCP 连接的相关配置，参考 `espconn_option` - bit 0：1，TCP 连接断开时，及时释放内存，无需等待 2 分钟才释放占用内存； - bit 1：1，关闭 TCP 数据传输时的 nalge 算法； - bit 2：1，使能 write finish callback，进入此回调表示 `espconn_send` 要发送的数据已经写入 2920 字节的 write buffer 等待发送或已经发送； - bit 3：1，使能 keep alive；
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接

4.2.7. espconn_clear_opt

功能	清除 TCP 连接的相关配置
函数定义	`sint8 espconn_clear_opt(` `  struct espconn *espconn,` `  uint8 opt` `)`
结构体	`enum espconn_option{` `  ESPCONN_START = 0x00,` `  ESPCONN_REUSEADDR = 0x01,` `  ESPCONN_NODELAY = 0x02,` `  ESPCONN_COPY = 0x04,` `  ESPCONN_KEEPALIVE = 0x08,` `  ESPCONN_END` `}`
参数	• `struct espconn *espconn`：对应网络连接的结构体 • `uint8 opt`：清除 TCP 连接的相关配置，参考 `espconn_option`
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接

4.2.8. espconn_set_keepalive

功能	设置 TCP keep alive 的参数
注意	• 一般情况下，不需要调用本接口 • 如果设置，请在 `espconn_connect_callback` 中调用，并先设置 `espconn_set_opt` 使能 keep alive
函数定义	`sint8 espconn_set_keepalive(struct espconn *espconn, uint8 level, void* optarg)`

结构体	`enum espconn_level{` `ESPCONN_KEEPIDLE,` `ESPCONN_KEEPINTVL,` `ESPCONN_KEEPCNT` `}`
参数	• `struct espconn *espconn`：对应网络连接的结构体 • `uint8 level`：默认设置为每隔 `ESPCONN_KEEPIDLE` 时长进行一次 keep alive 探查，如果报文无响应，则每隔 `ESPCONN_KEEPINTVL` 时长探查一次，最多探查 `ESPCONN_KEEPCNT` 次；若始终无响应，则认为网络连接断开，释放本地连接相关资源，进入 `espconn_reconnect_callback`。注意，时间间隔设置并不可靠精准，仅供参考，受其他高优先级任务执行的影响。参数说明如下： - `ESPCONN_KEEPIDLE`：设置进行 keep alive 探查的时间间隔，单位：秒 - `ESPCONN_KEEPINTVL`：keep alive 探查过程中，报文的时间间隔，单位：秒 - `ESPCONN_KEEPCNT`：每次 keep alive 探查，发送报文的最大次数 • `void* optarg`：设置参数值
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接

4.2.9. espconn_get_keepalive

功能	查询 TCP keep alive 的参数
函数定义	`sint8 espconn_set_keepalive(struct espconn *espconn, uint8 level, void* optarg)`
结构体	`enum espconn_level{` `ESPCONN_KEEPIDLE,` `ESPCONN_KEEPINTVL,` `ESPCONN_KEEPCNT` `}`
参数	• `struct espconn *espconn`：对应网络连接的结构体 • `uint8 level` - `ESPCONN_KEEPIDLE`：设置进行 keep alive 探查的时间间隔，单位：秒 - `ESPCONN_KEEPINTVL`：keep alive 探查过程中，报文的时间间隔，单位：秒 - `ESPCONN_KEEPCNT`：每次 keep alive 探查，发送报文的最大次数 • `void* optarg`：设置参数值
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接

4.2.10. espconn_reconnect_callback

功能	TCP 连接异常断开时的回调函数，相当于出错处理回调，由 `espconn_regist_reconcb` 注册。
函数定义	`void espconn_reconnect_callback (void *arg, sint8 err)`

参数	• `void *arg`：回调函数的参数，对应网络连接的结构体 `espconn` 指针。注意，本指针为底层维护的指针，不同回调传入的指针地址可能不一样，请勿依此判断网络连接。可根据 `espconn` 结构体中的 `remote_ip`、`remote_port` 判断多连接中的不同网络传输。 • `sint8 err`：异常断开的错误码。 - `ESCONN_TIMEOUT`：超时出错断开 - `ESPCONN_ABRT`：TCP 连接异常断开 - `ESPCONN_RST`：TCP 连接复位断开 - `ESPCONN_CLSD`：TCP 连接在断开过程中出错，异常断开 - `ESPCONN_CONN`：TCP 未连接成功 - `ESPCONN_HANDSHAKE`：TCP SSL 握手失败 - `ESPCONN_PROTO_MSG`：SSL 应用数据处理异常
返回	无

4.2.11. espconn_regist_reconcb

功能	注册 TCP 连接发生异常断开时的回调函数，可以在回调函数中进行重连。
注意	`espconn_reconnect_callback` 功能类似于出错处理回调，任何阶段出错时，均会进入此回调。例如，`espconn_sent` 失败，则认为网络连接异常，也会进入 `espconn_reconnect_callback`；用户可在 `espconn_reconnect_callback` 中自行定义出错处理。
函数定义	`sint8 espconn_regist_reconcb(` `  struct espconn *espconn,` `  espconn_reconnect_callback recon_cb` `)`
参数	• `struct espconn *espconn`：对应网络连接的结构体 • `espconn_reconnect_callback recon_cb`：回调函数
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接

4.2.12. espconn_disconnect

功能	断开 TCP 连接
注意	请勿在 `espconn` 的任何 callback 中调用本接口断开连接。如有需要，可以在 callback 中使用任务触发调用本接口断开连接。
函数定义	`sint8 espconn_disconnect(struct espconn *espconn)`
参数	`struct espconn *espconn`：对应网络连接的结构体
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接

4.2.13. espconn_regist_disconcb

功能	注册 TCP 连接正常断开成功的回调函数
函数定义	`sint8 espconn_regist_disconcb(` `  struct espconn *espconn,` `  espconn_connect_callback discon_cb` `)`
参数	• `struct espconn *espconn`：对应网络连接的结构体 • `espconn_connect_callback connect_cb`：回调函数
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接

4.2.14. espconn_abort

功能	强制断开 TCP 连接
注意	请勿在 `espconn` 的任何 callback 中调用本接口断开连接。如有需要，可以在 callback 中使用任务触发调用本接口断开连接。
函数定义	`sint8 espconn_abort(struct espconn *espconn)`
参数	`struct espconn *espconn`：对应网络连接的结构体
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接

4.2.15. espconn_regist_write_finish

功能	注册所有需发送的数据均成功写入 write buffer 后的回调函数。 请先调用 `espconn_set_opt` 使能 write buffer。
注意	• 本接口不能用于 SSL 连接。 • write buffer 用于缓存 `espconn_send` 将发送的数据，最多缓存 8 包数据，write buffer 的容量为 2920 字节。 • 由 `espconn_set_opt` 设置使能 `write_finish_callback` 回调。 • 对发送速度有要求时，可以在 `write_finish_callback` 中调用 `espconn_send` 发送下一包，无需等到 `espconn_sent_callback`
函数定义	`sint8 espconn_regist_write_finish (` `  struct espconn *espconn,` `  espconn_connect_callback write_finish_fn` `)`
参数	`struct espconn *espconn`：对应网络连接的结构体 `espconn_connect_callback write_finish_fn`：回调函数
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接

4.2.16. espconn_tcp_get_max_con

功能	查询允许的 TCP 最大连接数。
函数定义	`uint8 espconn_tcp_get_max_con(void)`
参数	无
返回	允许的 TCP 最大连接数

4.2.17. espconn_tcp_set_max_con

功能	设置允许的 TCP 最大连接数。在内存足够的情况下，建议不超过 10。默认值为 5。
函数定义	`sint8 espconn_tcp_set_max_con(uint8 num)`
参数	`uint8 num`：允许的 TCP 最大连接数
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接

4.2.18. espconn_tcp_get_max_con_allow

功能	查询 ESP8266 某个 TCP server 最多允许连接的 TCP client 数目
函数定义	`sint8 espconn_tcp_get_max_con_allow(struct espconn *espconn)`
参数	`struct espconn *espconn`：对应 TCP server 的结构体
返回	>0：最多允许连接的 TCP client 数目 <0：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接

4.2.19. espconn_tcp_set_max_con_allow

功能	设置 ESP8266 某个 TCP server 最多允许连接的 TCP client 数目
函数定义	`sint8 espconn_tcp_set_max_con_allow(struct espconn *espconn, uint8 num)`
参数	`struct espconn *espconn`：对应 TCP server 的结构体 `uint8 num`：允许的 TCP 最大连接数
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接

4.2.20. espconn_recv_hold

功能	阻塞 TCP 接收数据
注意	调用本接口会逐渐减小 TCP 的窗口，并不是即时阻塞，因此建议预留 1460*5 字节左右的空间时候调用，且本接口可以反复调用。
函数定义	`sint8 espconn_recv_hold(struct espconn *espconn)`

参数	`struct espconn *espconn`：对应网络连接的结构体
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接

4.2.21. espconn_recv_unhold

功能	解除 TCP 收包阻塞，即对应的阻塞接口 `espconn_recv_hold`
注意	本接口实时生效。
函数定义	`sint8 espconn_recv_unhold(struct espconn *espconn)`
参数	`struct espconn *espconn`：对应网络连接的结构体
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 TCP 连接

4.2.22. espconn_secure_accept

功能	创建 SSL server，侦听 SSL 握手
注意	• 目前仅支持建立一个 SSL server，本接口只能调用一次，并且仅支持连入一个 SSL client。 • 如果 SSL 加密一包数据大于 `espconn_secure_set_size` 设置的缓存空间，ESP8266 无法处理，SSL 连接断开，进入 `espconn_reconnect_callback`。 • SSL 相关接口与普通 TCP 接口底层处理不一致，请不要混用。SSL 连接时，仅支持使用 `espconn_secure_XXX` 系列接口和 `espconn_regist_XXXcb` 系列注册回调函数的接口，以及 `espconn_port` 获得一个空闲端口。 • 如需创建 SSL server，必须先调用 `espconn_secure_set_default_certificate` 和 `espconn_secure_set_default_private_key` 传入证书和密钥。
函数定义	`sint8 espconn_secure_accept(struct espconn *espconn)`
参数	`struct espconn *espconn`：对应网络连接的结构体
返回	`0`：成功 其它：失败，返回错误码 • `ESPCONN_ARG`：未找到参数 `espconn` 对应的 SSL 连接 • `ESPCONN_MEM`：空间不足 • `ESPCONN_ISCONN`：连接已经建立

4.2.23. espconn_secure_delete

功能	删除 ESP8266 作为 SSL server 的连接。
函数定义	`sint8 espconn_secure_delete(struct espconn *espconn)`
参数	`struct espconn *espconn`：对应网络连接的结构体

返回	`0`：成功 其它：失败，返回错误码 • `ESPCONN_ARG`：未找到参数 `espconn` 对应的 SSL 连接 • `ESPCONN_INPROGRESS`：参数 `espconn` 对应的 SSL 连接仍未断开，请先调用 `espconn_secure_disconnect` 断开连接，再进行删除。

4.2.24. espconn_secure_set_size

功能	设置加密 (SSL) 数据缓存空间的大小
注意	默认缓存大小为 2KB；如需更改，请在加密 (SSL) 连接建立前调用： 在 `espconn_secure_accept`（ESP8266 作为 SSL server）之前调用； 或者 `espconn_secure_connect`（ESP8266 作为 SSL client）之前调用
函数定义	`bool espconn_secure_set_size (uint8 level, uint16 size)`
参数	• `uint8 level`：设置 ESP8266 SSL server/client - `0x01`：SSL client - `0x02`：SSL server - `0x03`：SSL client 和 SSL server • `uint16 size`：加密数据缓存的空间大小，取值范围：1 ~ 8192，单位：字节，默认值为 2048
返回	`true`：成功 `false`：失败

4.2.25. espconn_secure_get_size

功能	查询加密 (SSL) 数据缓存空间的大小
函数定义	`sint16 espconn_secure_get_size (uint8 level)`
参数	• `uint8 level`：设置 ESP8266 SSL server/client - `0x01`：SSL client - `0x02`：SSL server - `0x03`：SSL client 和 SSL server
返回	加密 (SSL) 数据缓存空间的大小

4.2.26. espconn_secure_connect

功能	加密 (SSL) 连接到 TCP SSL server（ESP8266 作为 TCP SSL client）
注意	• 如果 `espconn_secure_connect` 失败，返回非零值，连接未建立，不会进入任何 `espconn` callback。 • 目前 ESP8266 作为 SSL client 仅支持一个连接，本接口只能调用一次，或者调用 `espconn_secure_disconnect` 断开前一次连接，才可以再次调用本接口建立 SSL 连接； • 如果 SSL 加密一包数据大于 `espconn_secure_set_size` 设置的缓存空间，ESP8266 无法处理，SSL 连接断开，进入 `espconn_reconnect_callback` • SSL 相关接口与普通 TCP 接口底层处理不一致，请不要混用。SSL 连接时，仅支持使用 `espconn_secure_XXX` 系列接口和 `espconn_regist_XXXcb` 系列注册回调函数的接口，以及 `espconn_port` 获得一个空闲端口。
函数定义	`sint8 espconn_secure_connect (struct espconn *espconn)`
参数	`struct espconn *espconn`：对应网络连接的结构体
返回	`0`：成功 其它：失败，返回错误码 • `ESPCONN_ARG`：未找到参数 `espconn` 对应的 SSL 连接 • `ESPCONN_MEM`：空间不足 • `ESPCONN_ISCONN`：连接已经建立

4.2.27. espconn_secure_send

功能	发送加密数据 (SSL)
注意	• 请在上一包数据发送完成，进入 `espconn_sent_callback` 后，再发下一包数据。 • 每一包数据明文的上限值为 1024 字节，加密后的报文上限值是 1460 字节。
函数定义	`sint8 espconn_secure_send (` `  struct espconn *espconn,` `  uint8 *psent,` `  uint16 length` `)`
参数	`struct espconn *espconn`：对应网络连接的结构体 `uint8 *psent`：发送的数据 `uint16 length`：发送的数据长度
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_ARG`：未找到参数 `espconn` 对应的 SSL 连接

4.2.28. espconn_secure_sent

[@deprecated] 本接口不建议使用，建议使用 espconn_secure_send 代替。

功能	发送加密数据 (SSL)
注意	• 请在上一包数据发送完成，进入 espconn_sent_callback 后，再发下一包数据。 • 每一包数据明文的上限值为 1024 字节，加密后的报文上限值是 1460 字节。
函数定义	sint8 espconn_secure_sent (struct espconn *espconn, uint8 *psent, uint16 length)
参数	struct espconn *espconn：对应网络连接的结构体 uint8 *psent：发送的数据 uint16 length：发送的数据长度
返回	0：成功 其它：失败，返回错误码 ESPCONN_ARG：未找到参数 espconn 对应的 SSL 连接

4.2.29. espconn_secure_disconnect

功能	断开加密连接 (SSL)
注意	请勿在 espconn 的任何 callback 中调用本接口断开连接。如有需要，可以在 callback 中使用任务触发调用本接口断开连接。
函数定义	sint8 espconn_secure_disconnect(struct espconn *espconn)
参数	struct espconn *espconn：对应网络连接的结构体
返回	0：成功 其它：失败，返回错误码 ESPCONN_ARG：未找到参数 espconn 对应的 SSL 连接

4.2.30. espconn_secure_ca_enable

功能	开启 SSL CA 认证功能
注意	• CA 认证功能，默认关闭，详细介绍可参考文档 ESP8266 SSL 加密使用手册。 • 如需调用本接口，请在加密 (SSL) 连接建立前调用： - 在 espconn_secure_accept（ESP8266 作为 SSL server）之前调用； - 或者 espconn_secure_connect（ESP8266 作为 SSL client）之前调用
函数定义	bool espconn_secure_ca_enable (uint8 level, uint32 flash_sector)

参数	• `uint8 level`：设置 ESP8266 SSL server/client - `0x01`：SSL client - `0x02`：SSL server - `0x03`：SSL client 和 SSL server • `uint32 flash_sector`：设置 CA 证书 ***esp_ca_cert.bin*** 烧录到 Flash 的位置。例如，参数传入 0x3B，则对应烧录到 Flash 0x7B000
返回	`true`：成功 `false`：失败

4.2.31. espconn_secure_ca_disable

功能	关闭 SSL CA 认证功能
注意	• CA 认证功能，默认关闭，详细介绍可参考文档 ESP8266 SSL 加密使用手册。 • 如需调用本接口，请在加密 (SSL) 连接建立前调用： - 在 `espconn_secure_accept`（ESP8266 作为 SSL server）之前调用； - 或者 `espconn_secure_connect`（ESP8266 作为 SSL client）之前调用
函数定义	`bool espconn_secure_ca_disable (uint8 level)`
参数	• `uint8 level`：设置 ESP8266 SSL server/client - `0x01`：SSL client - `0x02`：SSL server - `0x03`：SSL client 和 SSL server
返回	`true`：成功 `false`：失败

4.2.32. espconn_secure_cert_req_enable

功能	使能 ESP8266 作为 SSL client 时的证书认证功能
注意	• 证书认证功能，默认关闭。如果服务器端不要求认证证书，则无需调用本接口。 • 如需调用本接口，请在 `espconn_secure_connect` 之前调用。
函数定义	`bool espconn_secure_cert_req_enable (uint8 level, uint32 flash_sector)`
参数	• `uint8 level`：仅支持设置为 `0x01` ESP8266 作为 SSL client • `uint32 flash_sector`：设置密钥 ***esp_cert_private_key.bin*** 烧录到 Flash 的位置，例如，参数传入 0x7A，则对应烧录到 Flash 0x7A000。请注意，不要覆盖了代码或系统参数区域。
返回	`true`：成功 `false`：失败

4.2.33. espconn_secure_cert_req_disable

功能	关闭 ESP8266 作为 SSL client 时的证书认证功能

注意	证书认证功能，默认关闭。
函数定义	`bool espconn_secure_ca_disable (uint8 level)`
参数	`uint8 level`：仅支持设置为 `0x01` ESP8266 作为 SSL client
返回	`true`：成功 `false`：失败

4.2.34. espconn_secure_set_default_certificate

功能	设置 ESP8266 作为 SSL server 时的证书
注意	• *ESP8266_NONOS_SDK/examples/IoT_Demo*中提供使用示例 • 本接口必须在 `espconn_secure_accept` 之前调用，传入证书信息
函数定义	`bool espconn_secure_set_default_certificate (const uint8_t* certificate, uint16_t length)`
参数	`const uint8_t* certificate`：证书指针 `uint16_t length`：证书长度
返回	`true`：成功 `false`：失败

4.2.35. espconn_secure_set_default_private_key

功能	设置 ESP8266 作为 SSL server 时的密钥
注意	• *ESP8266_NONOS_SDK/examples/IoT_Demo* 中提供使用示例 • 本接口必须在 `espconn_secure_accept` 之前调用，传入密钥信息
函数定义	`bool espconn_secure_set_default_private_key (const uint8_t* key, uint16_t length)`
参数	`const uint8_t* key`：密钥指针 `uint16_t length`：密钥长度
返回	`true`：成功 `false`：失败

4.3. UDP 接口

4.3.1. espconn_create

功能	建立 UDP 传输。
注意	请注意设置 `remote_ip` 和 `remote_port` 参数，请勿设置为 0。
函数定义	`sin8 espconn_create(struct espconn *espconn)`
参数	`struct espconn *espconn`：对应网络连接的结构体
返回	`0`：成功 其它：失败，返回错误码 • `ESPCONN_ARG`：未找到参数 `espconn` 对应的 UDP 连接 • `ESPCONN_MEM`：空间不足 • `ESPCONN_ISCONN`：连接已经建立

4.3.2. espconn_sendto

功能	UDP 发包接口
函数定义	`sin16 espconn_sendto(struct espconn *espconn, uint8 *psent, uint16 length)`
参数	`struct espconn *espconn`：对应网络连接的结构体 `uint8 *psent`：待发送的数据 `uint16 length`：发送的数据长度
返回	`0`：成功 其它：失败，返回错误码 • `ESPCONN_ARG`：未找到参数 `espconn` 对应的 UDP 传输 • `ESPCONN_MEM`：空间不足 • `ESPCONN_IF`：UDP 发包失败

4.3.3. espconn_igmp_join

功能	加入多播组
注意	请在 ESP8266 Station 已连入路由的情况下调用。
函数定义	`sint8 espconn_igmp_join(ip_addr_t *host_ip, ip_addr_t *multicast_ip)`
参数	`ip_addr_t *host_ip`：主机 IP `ip_addr_t *multicast_ip`：多播组 IP
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_MEM`：空间不足

4.3.4. espconn_igmp_leave

功能	退出多播组
函数定义	`sint8 espconn_igmp_leave(ip_addr_t *host_ip, ip_addr_t *multicast_ip)`
参数	`ip_addr_t *host_ip`：主机 IP `ip_addr_t *multicast_ip`：多播组 IP
返回	`0`：成功 其它：失败，返回错误码 `ESPCONN_MEM`：空间不足

4.3.5. espconn_dns_setserver

功能	设置默认 DNS server
注意	本接口必须在 ESP8266 DHCP client 关闭 `wifi_station_dhcpc_stop` 的情况下使用。
函数定义	`void espconn_dns_setserver(uint8 numdns, ip_addr_t *dnsserver)`
参数	`uint8 numdns`：DNS server ID，支持设置两个 DNS server，ID 分别为 0 和 1 `ip_addr_t *dnsserver`：DNS server IP
返回	无

4.3.6. espconn_dns_getserver

功能	查询 DNS server IP
函数定义	`ip_addr_t espconn_dns_getserver(uint8 numdns)`
参数	`uint8 numdns`：DNS server ID，支持传入 0 或 1
返回	DNS server IP

4.4. mDNS 接口

4.4.1. espconn_mdns_init

功能	mDNS 初始化
注意	• 若为 SoftAP+Station 模式，请先调用 `wifi_set_broadcast_if(STATIONAP_MODE);` • 若使用 ESP8266 Station 接口，请获得 IP 后，再调用本接口初始化 mDNS • `txt_data` 必须为 `key = value` 的形式
结构体	struct mdns_info{ char *host_name; char *server_name; uint16 server_port; unsigned long ipAddr; char *txt_data[10]; };
函数定义	`void espconn_mdns_init(struct mdns_info *info)`
参数	`struct mdns_info *info`：mDNS 结构体
返回	无

4.4.2. espconn_mdns_close

功能	关闭 mDNS，对应开启 mDNS 的 API：`espconn_mdns_init`
函数定义	`void espconn_mdns_close(void)`
参数	无
返回	无

4.4.3. espconn_mdns_server_register

功能	注册 mDNS 服务器
函数定义	`void espconn_mdns_server_register(void)`
参数	无
返回	无

4.4.4. espconn_mdns_server_unregister

功能	注销 mDNS 服务器
函数定义	`void espconn_mdns_server_unregister(void)`
参数	无
返回	无

4.4.5. espconn_mdns_get_servername

功能	查询 mDNS 服务器名称
函数定义	`char* espconn_mdns_get_servername(void)`
参数	无
返回	服务器名称

4.4.6. espconn_mdns_set_servername

功能	设置 mDNS 服务器名称
函数定义	`void espconn_mdns_set_servername(const char *name)`
参数	`const char *name`：服务器名称
返回	无

4.4.7. espconn_mdns_set_hostname

功能	设置 mDNS 主机名称
函数定义	`void espconn_mdns_set_hostname(char *name)`
参数	`char *name`：主机名称
返回	无

4.4.8. espconn_mdns_get_hostname

功能	查询 mDNS 主机名称
函数定义	`char* espconn_mdns_get_hostname(void)`
参数	无
返回	主机名称

4.4.9. espconn_mdns_disable

功能	去能 mDNS，对应使能 API：espconn_mdns_enable
函数定义	void espconn_mdns_disable(void)
参数	无
返回	无

4.4.10. espconn_mdns_enable

功能	使能 mDNS
函数定义	void espconn_mdns_enable(void)
参数	无
返回	无

4.4.11. mDNS 示例

定义 mDNS 信息时，请注意 host_name 和 server_name 不能包含特殊字符（例如“.”符号），或者协议名称（例如不能定义为“http”）。

```
struct mdns_info info;
void user_mdns_config()
{
        struct ip_info ipconfig;
        wifi_get_ip_info(STATION_IF, &ipconfig);
        info->host_name = “espressif” ;
        info->ipAddr = ipconfig.ip.addr; //ESP8266 station IP
        info->server_name = “iot” ;
        info->server_port = 8080;
        info->txt_data[0] = “version = now” ;
        info->txt_data[1] = “user1 = data1” ;
        info->txt_data[2] = “user2 = data2” ;
        espconn_mdns_init(&info);
}
```

5. 应用相关接口

5.1. AT 接口

AT 接口位于 */ESP8266_NONOS_SDK/include/at_custom.h*。

AT 接口的使用示例，请参考 *ESP8266_NONOS_SDK/examples/at/user/user_main.c*。

5.1.1. at_response_ok

功能	AT 串口 (UART0) 输出 `OK`
函数定义	`void at_response_ok(void)`
参数	无
返回	无

5.1.2. at_response_error

功能	AT 串口 (UART0) 输出 `ERROR`
函数定义	`void at_response_error(void)`
参数	无
返回	无

5.1.3. at_cmd_array_regist

功能	注册用户自定义的 AT 指令。请仅调用一次，将所有用户自定义 AT 指令一并注册。
函数定义	`void at_cmd_array_regist (` `  at_function * custom_at_cmd_arrar,` `  uint32 cmd_num` `)`
参数	`at_function * custom_at_cmd_arrar`：用户自定义的 AT 指令数组 `uint32 cmd_num`：用户自定义的 AT 指令数目
返回	无
示例	请参考 *ESP8266_NONOS_SDK/examples/at/user/user_main.c*

5.1.4. at_get_next_int_dec

功能	从 AT 指令行中解析 int 型数字
函数定义	`bool at_get_next_int_dec (char **p_src,int* result,int* err)`
参数	• `char **p_src`：参数 `*p_src` 为接收到的 AT 指令字符串 • `int* result`：从 AT 指令中解析出的 int 型数字 • `int* err`：解析处理时的错误码 - `1`：数字省略时，返回错误码 1 - `3`：只发现'-'时，返回错误码 3
返回	true：正常解析到数字（数字省略时，仍然返回 `true`，但错误码会为 1）； false：解析异常，返回错误码；异常可能：数字超过 10 bytes，遇到 `\r` 结束符，只发现 `-` 字符。
示例	请参考 ***ESP8266_NONOS_SDK/examples/at/user/user_main.c***

5.1.5. at_data_str_copy

功能	从 AT 指令行中解析字符串
函数定义	`int32 at_data_str_copy (char * p_dest, char ** p_src,int32 max_len)`
参数	• `char * p_dest`：从 AT 指令行中解析到的字符串 • `char **p_src`：参数 `*p_src` 为接收到的 AT 指令字符串 • `int32 max_len`：允许的最大字符串长度
返回	解析到的字符串长度： >=0：成功，则返回解析到的字符串长度 <0：失败，返回 `-1`
示例	请参考 ***ESP8266_NONOS_SDK/examples/at/user/user_main.c***

5.1.6. at_init

功能	AT 初始化
函数定义	`void at_init (void)`
参数	无
返回	无
示例	请参考 ***ESP8266_NONOS_SDK/examples/at/user/user_main.c***

5.1.7. at_port_print

功能	从 AT 串口 (UART0) 输出字符串
函数定义	`void at_port_print(const char *str)`
参数	`const char *str`：字符串
返回	无
示例	请参考 *ESP8266_NONOS_SDK/examples/at/user/user_main.c*

5.1.8. at_set_custom_info

功能	开发者自定义 AT 版本信息，可由指令 AT+GMR 查询到。
函数定义	`void at_set_custom_info (char *info)`
参数	`char *info`：版本信息
返回	无

5.1.9. at_enter_special_state

功能	进入 AT 指令执行态，此时不响应其他 AT 指令，返回 `busy`
函数定义	`void at_enter_special_state (void)`
参数	无
返回	无

5.1.10. at_leave_special_state

功能	退出 AT 指令执行态
函数定义	`void at_leave_special_state (void)`
参数	无
返回	无

5.1.11. at_get_version

功能	查询乐鑫提供的 AT lib 版本号
函数定义	`uint32 at_get_version (void)`
参数	无
返回	乐鑫 AT lib 版本号

5.1.12. at_register_uart_rx_intr

功能	设置 UART0 RX 是由用户使用，还是由 AT 使用。
注意	• 本接口可以重复调用。 • 运行 AT BIN，UART0 RX 默认供 AT 使用。
函数定义	`void at_register_uart_rx_intr (at_custom_uart_rx_intr rx_func)`
参数	`at_custom_uart_rx_intr`：注册用户使用 UART0 的 Rx 中断处理函数；如果传 `NULL`，则切换为 AT 使用 UART0
返回	无
示例	`void user_uart_rx_intr (uint8* data, int32 len)` `{` `  // UART0 rx for user` `  os_printf( "len=%d \r\n" ,len);` `  os_printf(data);` `  // change UART0 for AT` `  at_register_uart_rx_intr(NULL);` `}` `void user_init(void)` `{` `  at_register_uart_rx_intr(user_uart_rx_intr);` `}`

5.1.13. at_response

功能	设置 AT 响应
注意	• 默认情况下，`at_response` 从 UART0 TX 输出，与 `at_port_print` 功能相同。 • 如果调用了 `at_register_response_func`，`at_response` 的字符串成为 `response_func` 的参数，由用户自行处理。
函数定义	`void at_response (const char *str)`
参数	`const char *str`：字符串
返回	无

5.1.14. at_register_response_func

功能	注册 `at_response` 的回调函数。调用了 at_register_response_func，`at_response` 的字符串将传入 `response_func`，由用户自行处理。
函数定义	`void at_register_response_func (at_custom_response_func_type response_func)`
参数	`at_custom_response_func_type`：`at_response` 的回调函数
返回	无

5.1.15. at_fake_uart_enable

功能	使能模拟 UART，开发者可用于实现网络 AT 指令，或者 SDIO AT 指令。
函数定义	`bool at_fake_uart_enable(bool enable, at_fake_uart_tx_func_type func)`
参数	`bool enable`：使能模拟 UART `at_fake_uart_tx_func_type func`：模拟 UART Tx 的回调函数
返回	`true`：成功 `false`：失败

5.1.16. at_fake_uart_rx

功能	模拟 UART RX，开发者可用于实现网络 AT 指令，或者 SDIO AT 指令。
函数定义	`uint32 at_fake_uart_rx(uint8* data, uint32 length)`
参数	`uint8* data`：模拟 UART Rx 收到的数据 `uint32 length`：数据长度
返回	如果执行成功，则返回值与 `length` 相同；否则，执行失败。

5.1.17. at_set_escape_character

功能	设置 AT 指令的转义字符，支持设置为符号 `!`、`#`、`$`、`@`、`&`、`\` 的其中之一，默认转义字符为 `\`。
函数定义	`bool at_set_escape_character(uint8 ch)`
参数	`uint8 ch`：转义字符，支持传入符号 `!`、`#`、`$`、`@`、`&`、`\` 的其中之一。
返回	`true`：成功 `false`：失败

5.2. JSON 接口

位于 *ESP8266_NONOS_SDK/include/json/jsonparse.h & jsontree.h*。

5.2.1. 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` `)`
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

5.2.2. 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)`
参数	`struct jsonparse_state *state`：JSON 解析指针
返回	`int`：解析结果

5.2.3. 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` `)`
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

5.2.4. jsonparse_get_value_as_int

功能	解析JSON 格式为整型数据
函数定义	`int jsonparse_get_value_as_int(struct jsonparse_state *state)`
参数	`struct jsonparse_state *state`：JSON 解析指针

返回	int：解析结果

5.2.5. jsonparse_get_value_as_long

功能	解析 JSON 格式为为长整型数据
函数定义	long jsonparse_get_value_as_long(struct jsonparse_state *state)
参数	struct jsonparse_state *state：JSON 解析指针
返回	int：解析结果

5.2.6. 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)
参数	struct jsonparse_state *state：JSON 解析指针
返回	int：解析长度

5.2.7. 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)
参数	struct jsonparse_state *state：JSON 解析指针
返回	int：JSON 格式数据类型

5.2.8. jsonparse_strcmp_value

功能	比较解析 JSON 数据与特定字符串
函数定义	int jsonparse_strcmp_value(struct jsonparse_state *state, const char *str)
参数	struct jsonparse_state *state：JSON 解析指针 const char *str：字符串缓存
返回	int：比较结果

5.2.9. 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))

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

5.2.10. jsontree_reset

功能	设置 JSON 数
函数定义	`void jsontree_reset(struct jsontree_context *js_ctx)`
参数	`struct jsontree_context *js_ctx`：JSON 格式树指针
返回	无

5.2.11. jsontree_path_name

功能	获取 JSON 树参数
函数定义	`const char *jsontree_path_name(` `  const struct jsontree_cotext *js_ctx,` `  int depth` `)`
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

5.2.12. jsontree_write_int

功能	整型数写入 JSON 树
函数定义	`void jsontree_write_int(` `  const struct jsontree_context *js_ctx,` `  int value` `)`
参数	`struct jsontree_context *js_ctx`：JSON 树指针 `int value`：整型数
返回	无

5.2.13. jsontree_write_int_array

功能	整型数组写入 JSON 树
函数定义	`void jsontree_write_int_array(` `  const struct jsontree_context *js_ctx,` `  const int *text,` `  uint32 length` `)`
参数	`struct jsontree_context *js_ctx`：JSON 树指针 `int *text`：数组入口地址 `uint32 length`：数组长度
返回	无

5.2.14. 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` `)`
参数	`struct jsontree_context *js_ctx`：JSON 格式树指针 `const char* text`：字符串指针
返回	无

5.2.15. jsontree_print_next

功能	获取 JSON 树下一个元素
函数定义	`int jsontree_print_next(struct jsontree_context *js_ctx)`
参数	`struct jsontree_context *js_ctx`：JSON 树指针
返回	`int`：JSON 树深度

5.2.16. 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` `)`
参数	`struct jsontree_context *js_ctx`：JSON 树指针 `int`：类型
返回	`struct jsontree_value *`：JSON 树元素指针

6. 参数结构体和宏定义

6.1. 定时器

```
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;
```

6.2. Wi-Fi 参数

6.2.1. Station 参数

```
typedef struct {
    int8  rssi;
    AUTH_MODE  authmode;
} wifi_fast_scan_threshold_t;

struct station_config {
    uint8 ssid[32];
    uint8 password[64];
    uint8 bssid_set;    // Note: If bssid_set is 1, station will just connect to the
router
                        // with both ssid[] and bssid[] matched. Please check about
this.
    uint8 bssid[6];
    wifi_fast_scan_threshold_t threshold;
};
```

> ⚠ **注意：**
>
> *BSSID* 表示 *AP* 的 *MAC* 地址，用于多个 *AP* 的 *SSID* 相同的情况。如果 `station_config.bssid_set==1`，`station_config.bssid` 必须设置，否则连接失败。一般情况下，`station_config.bssid_set` 设置为 *0*。

6.2.2. SoftAP 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 ssid_len;
    uint8 channel;           // support 1 ~ 13
    uint8 authmode;          // Don't support AUTH_WEP in soft-AP mode
    uint8 ssid_hidden;       // default 0
    uint8 max_connection;    // default 4, max 4
    uint16 beacon_interval;  // 100 ~ 60000 ms, default 100
};
```

> ⚠ **注意：**
>
> 如果 `softap_config.ssid_len==0`，读取 *SSID* 直至结束符，否则，根据 `softap_config.ssid_len` 设置 *SSID* 的长度。

6.2.3. Scan 参数

```
struct scan_config {
    uint8 *ssid;
    uint8 *bssid;
    uint8 channel;
    uint8 show_hidden;          // Scan APs which are hiding their SSID or not.
    wifi_scan_type_t scan_type; // scan type, active or passive
    wifi_scan_time_t scan_time; // scan time per channel
};
struct bss_info {
    STAILQ_ENTRY(bss_info)     next;

    uint8 bssid[6];
    uint8 ssid[32];
    uint8 ssid_len;
    uint8 channel;
    sint8 rssi;
    AUTH_MODE authmode;
    uint8 is_hidden;            // SSID of current AP is hidden or not
    sint16 freq_offset;         // AP's frequency offset
    sint16 freqcal_val;
    uint8 *esp_mesh_ie;
    uint8 simple_pair;
    CIPHER_TYPE pairwise_cipher;
    CIPHER_TYPE group_cipher;
    uint32_t phy_11b:1;
    uint32_t phy_11g:1;
    uint32_t phy_11n:1;
    uint32_t wps:1;
    uint32_t reserved:28;
};
typedef void (* scan_done_cb_t)(void *arg, STATUS status);
```

6.2.4. Wi-Fi Event 结构体

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

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

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

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

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

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

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

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

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

typedef struct {
        uint8 mac[6];
        struct ip_addr ip;
        uint8 aid;
} Event_SoftAPMode_Distribute_Sta_IP_t;

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

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

typedef struct {
        uint8 old_opmode;
        uint8 new_opmode;
} Event_OpMode_Change_t;

typedef union {
        Event_StaMode_Connected_t               connected;
```

```
        Event_StaMode_Disconnected_t            disconnected;
        Event_StaMode_AuthMode_Change_t         auth_change;
        Event_StaMode_Got_IP_t                  got_ip;
        Event_SoftAPMode_StaConnected_t         sta_connected;
        Event_SoftAPMode_Distribute_Sta_IP_t    distribute_sta_ip;
        Event_SoftAPMode_StaDisconnected_t      sta_disconnected;
        Event_SoftAPMode_ProbeReqRecved_t       ap_probereqrecved;
        Event_OpMode_Change_t                   opmode_changed;
    } Event_Info_u;

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

6.2.5. SmartConfig 结构体

```
    typedef enum  {
        SC_STATUS_WAIT = 0,     // 连接未开始，请勿在此阶段开始连接
        SC_STATUS_FIND_CHANNEL, // 请在此阶段开启 APP 进行配对连接
        SC_STATUS_GETTING_SSID_PSWD,
        SC_STATUS_LINK,
        SC_STATUS_LINK_OVER,   // 获取到 IP，连接路由完成
    } sc_status;
    typedef enum  {
        SC_TYPE_ESPTOUCH = 0,
        SC_TYPE_AIRKISS,
        SC_TYPE_ESPTOUCH_AIRKISS,
    } sc_type;
```

6.3. JSON 相关结构体

6.3.1. JSON 结构体

```
    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];
};
```

6.3.2. 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 }
```

6.4. espconn 参数

6.4.1. 回调函数

```
/** 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);
```

6.4.2. espconn

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

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

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

enum espconn_option{
        ESPCONN_START = 0x00,
        ESPCONN_REUSEADDR = 0x01,
        ESPCONN_NODELAY = 0x02,
        ESPCONN_COPY = 0x04,
        ESPCONN_KEEPALIVE = 0x08,
        ESPCONN_MANUALRECV = 0x10,
        ESPCONN_END
}

enum espconn_level{
        ESPCONN_KEEPIDLE,
```

```
        ESPCONN_KEEPINTVL,
        ESPCONN_KEEPCNT
}

/** 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;
    uint8 link_cnt;
    void *reverse; // reversed for customer use
};
```

6.5. 中断相关宏定义

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

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

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

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

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

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

/* enable a interrupt */
#define ETS_INTR_ENABLE(inum)   ets_isr_unmask((1<<inum))
/* disable a interrupt */
#define ETS_INTR_DISABLE(inum)  ets_isr_mask((1<<inum))

/* enable SPI interrupt */
#define ETS_SPI_INTR_ENABLE()   ETS_INTR_ENABLE(ETS_SPI_INUM)

/* enable UART interrupt */
#define ETS_UART_INTR_ENABLE()  ETS_INTR_ENABLE(ETS_UART_INUM)
/* disable UART interrupt */
#define ETS_UART_INTR_DISABLE() ETS_INTR_DISABLE(ETS_UART_INUM)

/* enable frc1 timer interrupt */
#define ETS_FRC1_INTR_ENABLE()  ETS_INTR_ENABLE(ETS_FRC_TIMER1_INUM)
/* disable frc1 timer interrupt */
#define ETS_FRC1_INTR_DISABLE() ETS_INTR_DISABLE(ETS_FRC_TIMER1_INUM)

/* enable GPIO interrupt */
#define ETS_GPIO_INTR_ENABLE()  ETS_INTR_ENABLE(ETS_GPIO_INUM)
/* disable GPIO interrupt */
#define ETS_GPIO_INTR_DISABLE() ETS_INTR_DISABLE(ETS_GPIO_INUM)
```

7. 外设驱动接口

外围设备驱动可以参考 */ESP8266_NONOS_SDK/driver_lib*。

7.1. GPIO 接口

GPIO 相关接口位于 */ESP8266_NONOS_SDK/include/eagle_soc.h & gpio.h*。

使用示例可参考 */ESP8266_NONOS_SDK/examples/IoT_Demo/user/user_plug.c*。

7.1.1. PIN 相关宏定义

以下宏定义控制 GPIO 管脚状态：

`PIN_PULLUP_DIS(PIN_NAME)`	管脚上拉屏蔽	示例： `// Use MTDI pin as GPIO12.` `PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U, FUNC_GPIO12);`
`PIN_PULLUP_EN(PIN_NAME)`	管脚上拉使能	
`PIN_FUNC_SELECT(PIN_NAME, FUNC)`	管脚功能选择	

7.1.2. gpio_output_set

功能	设置 GPIO 属性
函数定义	`void gpio_output_set(` `  uint32 set_mask,` `  uint32 clear_mask,` `  uint32 enable_mask,` `  uint32 disable_mask` `)`
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
示例	`gpio_output_set(BIT12, 0, BIT12, 0)`：设置 GPIO12 输出高电平； `gpio_output_set(0, BIT12, BIT12, 0)`：设置 GPIO12 输出低电平； `gpio_output_set(BIT12, BIT13, BIT12|BIT13, 0)`：设置 GPIO12 输出高电平， GPIO13 输出低电平； `gpio_output_set(0, 0, 0, BIT12)`：设置 GPIO12 为输入

7.1.3. 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，与上一节的输出高低电平的示例相同。
`GPIO_DIS_OUTPUT(gpio_no)`	设置 `gpio_no` 管脚输入，与上一节的设置输入示例相同。
`GPIO_INPUT_GET(gpio_no)`	获取 `gpio_no` 管脚的电平状态。

7.1.4. GPIO 中断

`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

7.1.5. gpio_pin_intr_state_set

功能	设置 GPIO 中断触发状态
函数定义	`void gpio_pin_intr_state_set(` `  uint32 i,` `  GPIO_INT_TYPE intr_state` `)`
参数	`uint32 i`：GPIO pin ID，例如设置 GPIO14，则为 `GPIO_ID_PIN(14);` `GPIO_INT_TYPE intr_state` ： 中断触发状态： `typedef enum {` `  GPIO_PIN_INTR_DISABLE = 0,` `  GPIO_PIN_INTR_POSEDGE = 1,` `  GPIO_PIN_INTR_NEGEDGE = 2,` `  GPIO_PIN_INTR_ANYEDGE = 3,` `  GPIO_PIN_INTR_LOLEVEL = 4,` `  GPIO_PIN_INTR_HILEVEL = 5` `} GPIO_INT_TYPE;`
返回	无

7.1.6. 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);
```

7.2. UART 接口

默认情况下，UART0 作为系统的打印信息输出接口，当配置为双 UART 时，UART0 作为数据收发接口，UART1 作为打印信息输出接口。**使用时，请确保硬件连接正确。**

关于 UART 的详细介绍，请参考 ESP8266 技术参考。

7.2.1. 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` `)`
参数	`UartBautRate uart0_br`：UART0 波特率 `UartBautRate uart1_br`：UART1 波特率
波特率	`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

7.2.2. uart0_tx_buffer

功能	通过 UART0 输出用户数据
函数定义	`void uart0_tx_buffer(uint8 *buf, uint16 len)`
参数	`uint8 *buf`：数据缓存 `uint16 len`：数据长度
返回	无

7.2.3. 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。
函数定义	`void uart0_rx_intr_handler(void *para)`
参数	`void *para`：指向数据结构 `RcvMsgBuff` 的指针
返回	无

7.2.4. uart_div_modify

功能	设置 UART 波特率。
函数定义	`void uart_div_modify(uint8 uart_no, uint32 DivLatchValue)`
参数	`uint8 uart_no`：UART 号，UART0 或者 UART1 `uint32 DivLatchValue`: 分频参数
返回	无
示例	`void ICACHE_FLASH_ATTR UART_SetBaudrate(uint8 uart_no, uint32 baud_rate)` `{` `  uart_div_modify(uart_no, UART_CLK_FREQ /baud_rate);` `}`

7.3. I2C Master 接口

ESP8266 不能作为 I2C 从设备，但可以作为 I2C 主设备，对其他 I2C 从设备（例如大多数数字传感器）进行控制与读写。

每个 GPIO 管脚内部都可以配置为开漏模式 (open-drain)，从而可以灵活的将 GPIO 口用作 I2C data 或 clock 功能。

同时，芯片内部提供上拉电阻，以节省外部的上拉电阻。

关于 I2C 的详细介绍，请参考 ESP8266 技术参考。

7.3.1. i2c_master_gpio_init

功能	设置 GPIO 为 I2C master 模式
函数定义	`void i2c_master_gpio_init (void)`
参数	无
返回	无

7.3.2. i2c_master_init

功能	初始化 I2C
函数定义	`void i2c_master_init(void)`
参数	无
返回	无

7.3.3. i2c_master_start

功能	设置 I2C 进入发送状态
函数定义	`void i2c_master_start(void)`
参数	无
返回	无

7.3.4. i2c_master_stop

功能	设置 I2C 停止发送
函数定义	`void i2c_master_stop(void)`
参数	无
返回	无

7.3.5. i2c_master_send_ack

功能	发送 I2C ACK
函数定义	`void i2c_master_send_ack (void)`
参数	无
返回	无

7.3.6. i2c_master_send_nack

功能	发送 I2C NACK
函数定义	`void i2c_master_send_nack (void)`
参数	无
返回	无

7.3.7. i2c_master_checkAck

功能	检查 I2C slave 的 ACK
函数定义	`bool i2c_master_checkAck (void)`
参数	无
返回	`true`：获取 I2C slave ACK `false`：获取 I2C slave NACK

7.3.8. i2c_master_readByte

功能	从 I2C slave 读取一个字节
函数定义	`uint8 i2c_master_readByte (void)`
参数	无
返回	`uint8`：读取到的值

7.3.9. i2c_master_writeByte

功能	向 I2C slave 写一个字节
函数定义	`void i2c_master_writeByte (uint8 wrdata)`
参数	`uint8 wrdata`：数据
返回	无

7.4. PWM 接口

本文档仅简单介绍 ***pwm.h*** 中的 PWM 相关接口，详细的 PWM 介绍文档请参考 ESP8266 技术参考。

PWM 驱动接口函数不能跟 ***hw_timer.c*** 的接口同时使用，因为它们共用了同一个硬件定时器。PWM 不支持进入 Deep sleep 模式，也请勿调用 `wifi_set_sleep_type(LIGT_SLEEP);` 将自动睡眠模式设置为 Light-sleep。因为 Light-sleep 在睡眠期间会停 CPU，停 CPU 期间不能响应 NMI 中断。

7.4.1. pwm_init

功能	初始化 PWM，包括 GPIO 选择，周期和占空比。目前仅支持调用一次。
函数定义	`void pwm_init(` `uint32 period,` `uint8 *duty,` `uint32 pwm_channel_num,` `uint32 (*pin_info_list)[3])`
参数	`uint32 period`：PWM 周期 `uint8 *duty`：各路 PWM 的占空比 `uint32 pwm_channel_num`：PWM 通道数 `uint32 (*pin_info_list)[3]`：PWM 各通道的 GPIO 硬件参数。本参数是一个 n * 3 的数组指针，数组中定义了 GPIO 的寄存器，对应 PIN 脚的 IO 复用值和 GPIO 对应的序号
返回	无

示例	初始化一个三通道的 PWM： `uint32 io_info[][3] =` `{{PWM_0_OUT_IO_MUX,PWM_0_OUT_IO_FUNC,PWM_0_OUT_IO_NUM},` `{PWM_1_OUT_IO_MUX,PWM_1_OUT_IO_FUNC,PWM_1_OUT_IO_NUM},` `{PWM_2_OUT_IO_MUX,PWM_2_OUT_IO_FUNC,PWM_2_OUT_IO_NUM}};` `pwm_init(light_param.pwm_period, light_param.pwm_duty, 3, io_info);`

7.4.2. pwm_start

功能	PWM 开始。每次更新 PWM 设置后，都需要重新调用本接口进行计算。
函数定义	`void pwm_start (void)`
参数	无
返回	无

7.4.3. pwm_set_duty

功能	设置 PWM 某个通道信号的占空比。设置各路 PWM 信号高电平所占的时间，`duty` 的范围随 PWM 周期改变，最大值为：Period * 1000 /45。例如，1KHz PWM，`duty` 范围是：0 ~ 22222
注意	设置完成后，需要调用 `pwm_start` 生效。
函数定义	`void pwm_set_duty(uint32 duty, uint8 channel)`
参数	`uint32 duty`：设置高电平时间参数，占空比的值为 (duty*45)/ (period*1000) `uint8 channel`：当前要设置的 PWM 通道，取值范围依据实际使用了几路 PWM，在 ***IOT_Demo*** 中取值在 `#define PWM_CHANNEL` 定义的范围内。
返回	无

7.4.4. pwm_get_duty

功能	获取某路 PWM 信号的 `duty` 参数，占空比的值为 (duty*45)/ (period*1000)
函数定义	`uint8 pwm_get_duty(uint8 channel)`
参数	`uint8 channel`：当前要查询的 PWM 通道，取值范围依据实际使用了几路 PWM，在 ***IOT_Demo*** 中取值在 `#define PWM_CHANNEL` 定义的范围内。
返回	对应某路 PWM 信号的 `duty` 参数

7.4.5. pwm_set_period

功能	设置 PWM 周期，单位：μs。例如，1KHz PWM，参数为 1000 μs。
注意	设置完成后，需要调用 `pwm_start` 生效。
函数定义	`void pwm_set_period(uint32 period)`
参数	`uint32 period`：PWM 周期，单位：μs

返回	无

7.4.6. pwm_get_period

功能	查询 PWM 周期
函数定义	`uint32 pwm_get_period(void)`
参数	无
返回	PWM 周期，单位：μs

7.4.7. get_pwm_version

功能	查询 PWM 版本信息
函数定义	`uint32 get_pwm_version(void)`
参数	无
返回	PWM 版本信息

7.5. SDIO 接口

ESP8266 仅支持作为 SDIO slave。

7.5.1. sdio_slave_init

功能	初始化 SDIO
函数定义	`void sdio_slave_init(void)`
参数	无
返回	无

7.5.2. sdio_load_data

功能	加载数据到 SDIO buffer 中，并通知 SDIO host 读取。
函数定义	`int32 sdio_load_data(const uint8* data, uint32 len)`
参数	`const uint8* data`：待传输的数据 `uint32 len`：数据长度
返回	实际成功加载到 SDIO buffer 中的数据长度。 目前不支持加载部分数据，如果数据超过 SDIO buffer 可加载容量，将返回 `0`，数据加载失败。

7.5.3. sdio_register_recv_cb

功能	注册 SDIO 收到 host 数据的回调函数。
回调函数定义	`typedef void(*sdio_recv_data_callback)(uint8* data, uint32 len)` 注册的回调函数不能放在 cache 中，即回调函数前不能添加 `ICACHE_FLASH_ATTR` 宏定义。
函数定义	`bool sdio_register_recv_cb(sdio_recv_data_callback cb)`
参数	`sdio_recv_data_callback cb`：回调函数
返回	`true`：注册成功 `false`：注册失败

A. 附录

A.1. ESPCONN 编程

可参考乐鑫提供的示例代码：
https://github.com/espressif/esp8266-nonos-sample-code
https://github.com/espressif/esp8266-rtos-sample-code

A.1.1. TCP Client 模式

> ⚠ **注意：**
>
> - *ESP8266* 工作在 *Station* 模式下，需确认 *ESP8266* 已经连接 *AP*（路由）分配到 *IP* 地址，启用 *client* 连接。
> - *ESP8266* 工作在 *SoftAP* 模式下，需确认连接 *ESP8266* 的设备已被分配到 *IP* 地址，启用 *client* 连接。

步骤如下：

1. 依据工作协议初始化 `espconn` 参数；
2. 注册连接成功的回调函数和连接失败重连的回调函数；
   - 调用 `espconn_regist_connectcb` 和 `espconn_regist_reconcb`
3. 调用 `espconn_connect` 建立与 TCP Server 的连接；
4. TCP连接建立成功后，在连接成功的回调函数 `espconn_connect_callback` 中，注册接收数据的回调函数，发送数据成功的回调函数和断开连接的回调函数。
   - 调用 `espconn_regist_recvcb`、`espconn_regist_sentcb` 和 `espconn_regist_disconcb`
5. 在接收数据的回调函数，或者发送数据成功的回调函数中，执行断开连接操作时，建议适当延时一定时间，确保底层函数执行结束。

A.1.2. TCP Server 模式

> ⚠ **注意：**
>
> - *ESP8266* 工作在 *Station* 模式下，需确认 *ESP8266* 已经分配到 *IP* 地址，再启用 *server* 侦听。
> - *ESP8266* 工作在 *SoftAP* 模式下，可以直接启用 *server* 侦听。

步骤如下：

1. 依据工作协议初始化 `espconn` 参数；

2. 注册连接成功的回调函数和连接失败重连的回调函数；
   - 调用 espconn_regist_connectcb 和 espconn_regist_reconcb
3. 调用 espconn_accept 侦听 TCP 连接；
4. TCP 连接建立成功后，在连接成功的回调函数 espconn_connect_callback 中，注册接收数据的回调函数，发送数据成功的回调函数和断开连接的回调函数。
   - 调用 espconn_regist_recvcb、espconn_regist_sentcb 和 espconn_regist_disconcb

A.1.3. espconn Callback

注册函数	回调函数	说明
espconn_regist_connectcb	espconn_connect_callback	TCP 连接建立成功
espconn_regist_reconcb	espconn_reconnect_callback	TCP 连接发生异常而断开
espconn_regist_sentcb	espconn_sent_callback	TCP 或 UDP 数据发送完成
espconn_regist_recvcb	espconn_recv_callback	TCP 或 UDP 数据接收
espconn_regist_write_finish	espconn_write_finish_callback	数据成功写入 TCP 数据缓存
espconn_regist_disconcb	espconn_disconnect_callback	TCP 连接正常断开

⚠ **注意：**

- 回调函数中传入的指针 arg，对应网络连接的结构体 espconn 指针。该指针为 *SDK* 内部维护的指针，不同回调传入的指针地址可能不一样，请勿依此判断网络连接。可根据 espconn 结构体中的 remote_ip、remote_port 判断多连接中的不同网络传输。
- 如果 espconn_connect（或者 espconn_secure_connect）失败，返回非零值，连接未建立，不会进入任何 espconn *callback*。
- 请勿在 espconn 任何回调中调用 espconn_disconnect（或者 espconn_secure_disconnect）断开连接。如有需要，可以在 espconn 回调中使用触发任务的方式（system_os_task 和 system_os_post）调用 espconn_disconnect（或者 espconn_secure_disconnect）断开连接。

A.2. RTC API 使用示例

以下测试示例，可以验证 RTC 时间和系统时间，在 system_restart 时的变化，以及读写 RTC memory。

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

os_timer_t rtc_test_t;
#define  RTC_MAGIC  0x55aaaa55
```

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

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

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

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

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

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

    cal1 = system_rtc_clock_cali_proc();
    os_delay_us(300);

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

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

    rtc_time.time_base = rtc_t2;
```

```
    system_rtc_mem_write(64, &rtc_time, sizeof(rtc_time));
    os_printf("----------------------\r\n");

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

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

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

A.3. Sniffer 说明

关于 sniffer 的详细说明，请参考 ESP8266 技术参考。

A.4. ESP8266 SoftAP 和 Station 信道定义

虽然 ESP8266 支持 SoftAP+Station 共存模式，但是 ESP8266 实际只有一个硬件信道。因此在 SoftAP+Station 模式时，ESP8266 SoftAP 会动态调整信道值与 ESP8266 Station 一致。

这个限制会导致 ESP8266 SoftAP+Station 模式时一些行为上的不便，用户请注意。例如：

情况一

1. 如果 ESP8266 Station 连接到一个路由（假设路由信道号为 6）
2. 通过接口 `wifi_softap_set_config` 设置 ESP8266 SoftAP
3. 若设置值合法有效，该 API 将返回 `true`，但信道号仍然会自动调节成与 ESP8266 Station 接口一致，在这个例子里也就是信道号为 6。因为 ESP8266 在硬件上只有一个信道，由 ESP8266 Station 与 SoftAP 接口共用。

情况二

1. 调用接口 `wifi_softap_set_config` 设置 ESP8266 SoftAP（例如信道号为 5）
2. 其他 Station 连接到 ESP8266 SoftAP

3. 将 ESP8266 Station 连接到路由（假设路由信道号为 6）
4. ESP8266 SoftAP 将自动调整信道号与 ESP8266 Station 一致（信道 6）
5. 由于信道改变，之前连接到 ESP8266 SoftAP 的 Station 的 Wi-Fi 连接断开。

情况三

1. 其他 Station 与 ESP8266 SoftAP 建立连接
2. 如果 ESP8266 Station 一直尝试扫描或连接某路由，可能导致 ESP8266 SoftAP 端的连接断开。
3. 因为 ESP8266 Station 会遍历各个信道查找目标路由，意味着 ESP8266 其实在不停切换信道，ESP8266 SoftAP 的信道也因此在不停更改。这可能导致 ESP8266 SoftAP 端的原有连接断开。
4. 这种情况，用户可以通过设置定时器，超时后调用 `wifi_station_disconnect` 停止 ESP8266 Station 不断连接路由的尝试；或者在初始配置时，调用 `wifi_station_set_reconnect_policy` 和 `wifi_station_set_auto_connect` 禁止 ESP8266 Station 尝试重连路由。

A.5. ESP8266 启动信息说明

ESP8266 启动时，将从 UART0 以波特率 74880 打印如下启动信息：

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



load 0x4010f000, len 1264, room 16

tail 0

chksum 0x42

csum 0x42
```

其中可供用户参考的启动信息说明如下：

启动信息	说明
rst cause	1：上电
	2：外部复位

启动信息	说明
	4：硬件看门狗复位
boot mode 第一个参数	1：ESP8266 处于 UART-down 模式，可通过 UART 下载固件
	3：ESP8266 处于 Flash-boot 模式，从 Flash 启动运行
chksum	chksum 与 csum 值相等，表示启动过程中 Flash 读取正确

A.6. ESP8266 信令测试使用说明

ESP8266_NonOS_SDK_V3.0 增加了信令测试功能，该功能默认关闭。在 user_interface.h 定义了函数 wifi_enable_signaling_measurement() 和 wifi_disable_signaling_measurement() 可用于开启/关闭信令测试功能。建议使用罗德施瓦茨公司的 CMW500 信令测试仪进行测试。具体使用步骤如下：

(1) 在系统启动后，调用 wifi_set_opmode 使能 sta 模式，调用 wifi_enable_signaling_measurement 开启信令测试模式；

(2) 如果需要测试 11n 或者 11b，可以调用 wifi_set_phy_mode 将模式设置为 11n 或者 11b；如果无需测试，则无需调用，ESP8266 默认在 11g 模式；

(3) 调用 wifi_station_connect 连接测试仪。

乐鑫 IOT 团队
www.espressif.com