GigaDevice Semiconductor Inc.

# GD32W51x Wi-Fi 开发指南

应用笔记

**AN100**

# 目录

# 图索引

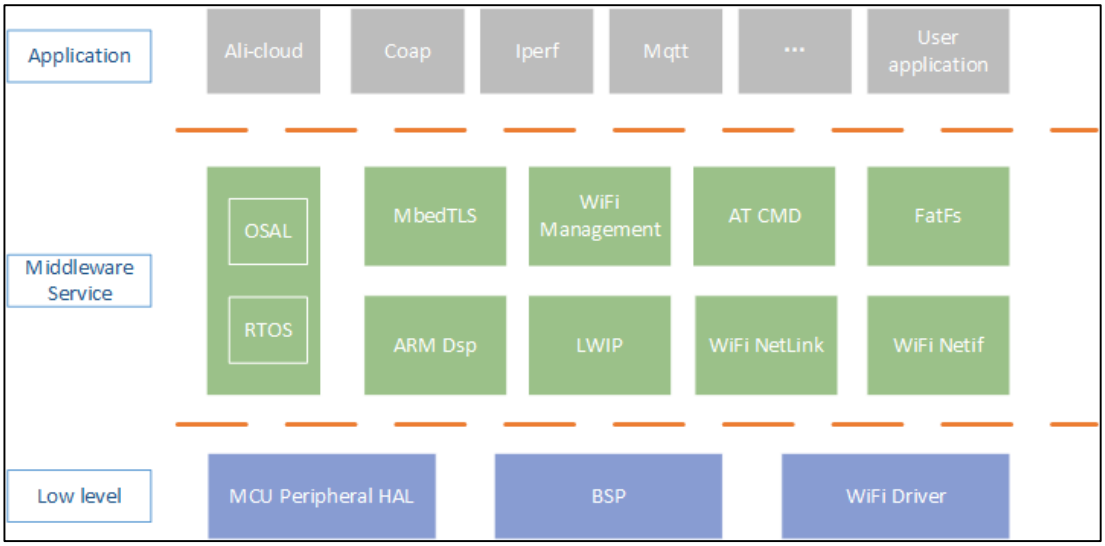
# 表索引

# 1. Wi-Fi SDK 概述

GD32W51x 系列芯片是以 Arm® Cortex®-M33 为内核的 32 位微控制器（MCU），支持 Arm® Cortex®-M33 的 TrustZone、FPU 特性，并包含了 Wi-Fi4 连接技术。GD32W51x Wi-Fi SDK 集成 Wi-Fi 驱动、Lwip TCP / IP 协议栈、MbedTLS 等组件，可使开发者基于 GD32W51x 快速开发物联网应用程序。本应用笔记描述了 SDK 框架、启动过程、Wi-Fi 及相关组件应用程序接口，旨在帮助开发者熟悉 SDK 并使用 API 开发自己的应用程序。

## 1.1. Wi-Fi SDK 软件框架

**图 1-1. Wi-Fi SDK 框架图**




如*图 1-1. Wi-Fi SDK 框架图*所示，GD32W51x Wi-Fi SDK 的软件框架由 Low level（底层）、Middleware Service（中间服务层）、Application（应用层）三层构成。

Low level 层接近硬件，可直接进行硬件外设相关操作，包含了 MCU 的外设硬件抽象层（HAL）、板级包（BSP）、Wi-Fi 驱动。开发者可通过 HAL 操作 USART、I2C、SPI 等 MCU 的外设，BSP 则可进行板级的初始化、使能 PMU、使能硬件加密引擎等操作。Wi-Fi 驱动可通过中间服务层的组件访问。

中间服务层由多个组件构成，为应用提供加密、网络通信等服务。其中 Arm Dsp、Mbedtls、Lwip 等是第三方组件，它们的使用方法可以参考其官方文档说明。OSAL（操作系统抽象层）是对 RTOS 内核函数的封装，开发者可通过 OSAL 操作 RTOS。由于 OSAL 的存在，开发者可根据需要选择自己的 RTOS，而不会影响应用程序及其他组件。本文 ***OSAL API*** 章节介绍了 OSAL 的 API 使用。Wi-Fi Netlink 组件是 Wi-Fi 驱动的操作集合，开发者通过 Netlink 可以获取或设置 Wi-Fi 相关参数和信息，如信道，进入混杂模式等，第 3 章 ***Wi-Fi Netlink API*** 列出了 API。Wi-Fi Netif 组件基于 Lwip 的封装，是对 Wi-Fi 设备的网络接口操作集合，开发者可对网络接口进行网络地址设置，获取接口的网络地址、网关等信息，第 4 章 ***Wi-Fi Netif API*** 介绍 Wi-Fi Netif 的 API 使用。AT CMD 组件是 AT 命令的集合，适合熟悉 AT 命令的开发者使用，可以参考《GD32W51x AT 指令用户指南》文档进行开发。Wi-Fi Management 是基于 Netif 和

Netlink 实现的 Wi-Fi 连接和漫游管理服务，开发者通过该服务可以执行扫描无线网络、连接 AP、启动软 AP 等操作。Wi-Fi Management 的软件实现上采用了状态机和事件管理组件，可以让开发者监控 Wi-Fi Driver 事件的发生，第 5 章 *Wi-Fi Management API* 会介绍其使用方法，开发者可以进行自定义开发。

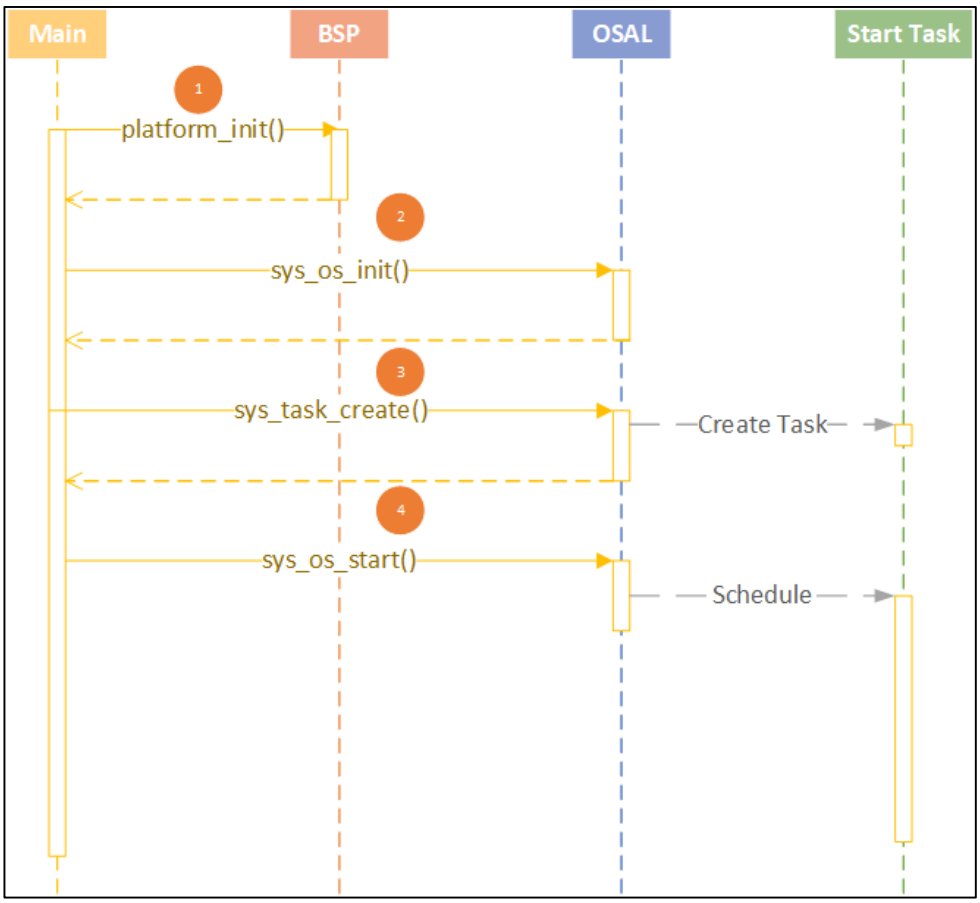
Application 层是多个应用程序的集合，例如基于阿里云 iotkit 配网及云服务程序 Ali-Cloud，性能测试程序 iperf3，以及开发者自定义的应用程序等等。

## 1.2. Wi-Fi SDK 软件启动过程

本节描述了 GD32W51x Wi-Fi SDK 启动过程，有助于开发者理解 SDK 各组件关系。

SDK 软件启动过程分为两个阶段，第一阶段如*图 1-2. Wi-Fi SDK 启动第一阶段*所示，从 Main 函数板级初始化到 RTOS 开始调度 Start Task 为止。

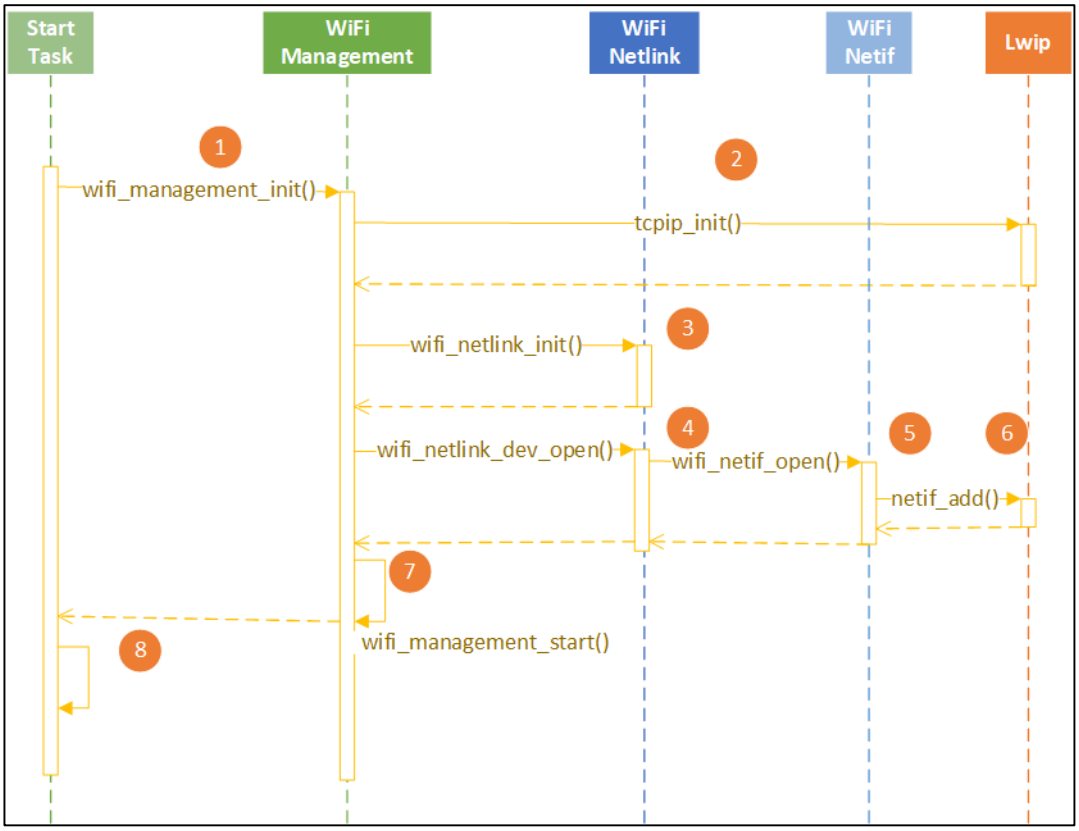
**图 1-2. Wi-Fi SDK 启动第一阶段**




- ① platform_init() 进行板级初始化，在此之后，开发者可以使用串口调试；
- ② sys_os_init() 进行 RTOS 初始化，在此之后，开发者可以创建 RTOS task；
- ③ sys_task_create() 创建 Start Task，为启动第二阶段做准备；
- ④ sys_os_start() RTOS 启动调度，Start Task 开始工作，进入 SDK 启动的第二阶段。

SDK 启动的第二阶段如*图 1-3. Wi-Fi SDK 启动第二阶段*所示，全部在 Start Task 内完成。

图 1-3. Wi-Fi SDK 启动第二阶段




- ① wifi_management_init() 对 wifi 和网络相关组件初始化；
- ② tcpip_init() 初始化 Lwip TCP / IP 协议栈；
- ③ wifi_netlink_init() 初始化 netlink 组件；
- ④ wifi_netlink_dev_open()、⑤ wifi_netif_open 和⑥ netif_add() 打开 wifi 设备，初始化 wifi 驱动，并向 lwip 添加网络接口；
- ⑦ wifi_management_start() 启动连接管理服务；
- ⑧ 最后 SDK 启动完成，开发者可以启动用户程序。SDK 中这里启动 console 程序。

# 2. OSAL API

头文件 SDK \ NSPE \ WIFI_IOT \ common \ wrapper_os.h。

## 2.1. 内存管理

### 2.1.1. sys_malloc

原型：void *sys_malloc (size_t size)

功能：分配长度为 size 的内存块。

输入参数：size，需要分配内存块的长度。

输出参数：无。

返回：分配内存块的指针，失败为 NULL。

### 2.1.2. sys_calloc

原型：void *sys_calloc (size_t count, size_t size)

功能：分配 count 个长度为 size 的连续内存块。

输入参数：count，分配的内存块个数；

size，需要分配内存块的长度。

输出参数：无。

返回：分配内存块的指针，失败为 NULL。

### 2.1.3. sys_mfree

原型：void sys_mfree (void *ptr)

功能：释放内存块。

输入参数：ptr，指向需要释放的内存块。

输出参数：无。

返回：无。

### 2.1.4. sys_realloc

原型：void *sys_realloc (void *mem, size_t size)

功能：扩大已分配的内存。

输入参数：mem，指向需要扩大的内存；

size，新内存块的大小。

输出参数：无。

返回：分配内存块的指针，失败为 NULL。

### 2.1.5. sys_free_heap_size

原型：int32_t sys_free_heap_size (void)

功能：获取堆的空闲大小。

输入参数：无。

输出参数：无。

返回：堆空闲的空间大小。

### 2.1.6. sys_min_free_heap_size

原型：int32_t sys_min_free_heap_size (void)

功能：获取堆的最小空闲大小。

输入参数：无。

输出参数：无。

返回：堆最小空闲的空间大小。

### 2.1.7. sys_heap_block_size

原型：uint16_t sys_heap_block_size (void)

功能：获取堆的块大小。

输入参数：无。

输出参数：无。

返回：堆的块大小。

### 2.1.8. sys_memset

原型：void sys_memset (void *s, uint8_t c, uint32_t count)

功能：初始化内存块。

输入参数：s，初始化的内存块地址；

c，初始化的内容；

count，内存块的大小。

输出参数：无。

返回：无。

### 2.1.9. sys_memcpy

原型：void sys_memcpy (void *des, const void *src, uint32_t n)

功能：内存拷贝。

输入参数：src，源内存地址；

n，拷贝的长度。

输出参数：dst，目的内存地址。

返回：无。

### 2.1.10. sys_memmove

原型：void sys_memmove (void *des, const void *src, uint32_t n)

功能：内存搬移。

输入参数：src，源内存地址；

n，搬移长度。

输出参数：des，目的内存地址。

返回：无。

### 2.1.11. sys_memcmp

原型：int32_t sys_memcmp (const void *buf1, const void *buf2, uint32_t count)

功能：比较两块内存值是否相同。

输入参数：buf1，比较内存地址 1；

buf2，比较内存地址 2；

count，长度。

输出参数：无。

返回：0，相同；非 0，不相同。

## 2.2. 任务管理

### 2.2.1. sys_task_create

原型：void *sys_task_create (void *static_tcb, const uint8_t *name, uint32_t *stack_base, uint32_t stack_size, uint32_t queue_size, uint32_t priority, task_func_t func, void *ctx)

功能：创建任务。

输入参数：static_tcb，静态任务控制块，NULL 则由 OS 分配任务控制块；

name，任务名字；

stack_base，任务栈底，NULL 则由 OS 分配任务栈；

stack_size，栈大小；

queue_size，消息队列大小；

priority，任务优先级；

func，任务函数；

ctx，任务上下文。

输出参数：无。

返回：非 NULL,创建任务成功，返回任务句柄；

NULL，创建任务失败。

### 2.2.2. sys_task_delete

原型：void sys_task_delete (void *task)

功能：删除任务。

输入参数：task，任务句柄，NULL 则删除任务自身。

输出参数：无。

返回：无。

### 2.2.3. sys_task_list

原型：void sys_task_list (char *pwrite_buf)

功能：任务列表。

输入参数：无。

输出参数：pwrite_buf，任务列表内容。

返回：无。

### 2.2.4. sys_idle_task_handle_get

原型：os_task_t *sys_idle_task_handle_get (void)

功能：获取 idle 任务的句柄。

输入参数：无。

输出参数：无。

返回：idle 任务句柄。

### 2.2.5. sys_timer_task_handle_get

原型：os_task_t *sys_timer_task_handle_get (void)

功能：获取 timer 任务的句柄。

输入参数：无。

输出参数：无。

返回：timer 任务句柄。

### 2.2.6. sys_stack_free_get

原型：uint32_t sys_get_stack_free (void *task)

功能：获取任务栈空闲的大小。

输入参数：task，任务句柄。

输出参数：无。

返回：任务栈的空闲大小。

## 2.3. 任务间通信

### 2.3.1. sys_task_wait

原型：int32_t sys_task_wait (uint32_t timeout_ms, void *msg_ptr)

功能：等待任务消息。

输入参数：timeout_ms，等待超时时间，0 代表无限等待。

输出参数：msg_ptr，消息指针。

返回：0，成功；非 0，失败。

### 2.3.2. sys_task_post

原型：int32_t sys_task_post (void *receiver_task, void *msg_ptr, uint8_t from_isr)

功能：发送任务消息。

输入参数：receiver_task，接收任务的句柄；

msg_ptr，消息指针；

from_isr，是否来自 ISR。

输出参数：无。

返回：0，成功；非 0，失败。

### 2.3.3. sys_task_msg_flush

原型：void sys_task_msg_flush (void *task)

功能：清空任务消息队列。

输入参数：task，任务句柄。

输出参数：无。

返回：无。

### 2.3.4. sys_task_msg_num

原型：int32_t sys_task_msg_num (void *task, uint8_t from_isr)

功能：获取目前任务排队的消息个数。

输入参数：task，任务句柄；

from_isr，是否来自 ISR。

输出参数：无。

返回：消息的个数。

### 2.3.5. sys_sema_init

原型：int32_t sys_sema_init (os_sema_t *sema, int32_t init_val)

功能：创建信号量。

输入参数：init_val，信号量初始值。

输出参数：sema，信号量句柄。

返回：0，创建成功；非 0，创建失败。

### 2.3.6. sys_sema_free

原型：void sys_sema_free (os_sema_t *sema)

功能：销毁信号量。

输入参数：sema，信号量句柄。

输出参数：无。

返回：无。

### 2.3.7. sys_sema_up

原型：void sys_sema_up (os_sema_t *sema)

功能：发送信号量。

输入参数：sema，信号量句柄。

输出参数：无。

返回：无。

### 2.3.8. sys_sema_up_from_isr

原型：void sys_sema_up_from_isr (os_sema_t *sema)

功能：在 ISR 中发送信号量。

输入参数：sema，信号量句柄。

输出参数：无。

返回：无。

### 2.3.9. sys_sema_down

原型：int32_t sys_sema_down (os_sema_t *sema, uint32_t timeout_ms)

功能：等待信号量。

输入参数：sema，信号量句柄；

timeout_ms，等待超时时间。

输出参数：无。

返回：0，成功；非 0，失败。

### 2.3.10. sys_mutex_init

原型：void sys_mutex_init (os_mutex_t *mutex)

功能：创建互斥锁。

输入参数：mutex，互斥锁句柄。

输出参数：无。

返回：无。

### 2.3.11. sys_mutex_free

原型：void sys_mutex_free (os_mutex_t *mutex)

功能：销毁互斥锁。

输入参数：mutex，互斥锁句柄。

输出参数：无。

返回：无。

### 2.3.12. sys_mutex_get

原型：void sys_mutex_get (os_mutex_t *mutex)

功能：等待互斥锁。

输入参数：mutex，互斥锁句柄。

输出参数：无。

返回：无。

### 2.3.13. sys_mutex_put

原型：void sys_mutex_put (os_mutex_t *mutex)

功能：释放互斥锁。

输入参数：mutex，互斥锁句柄。

输出参数：无。

返回：无。

### 2.3.14. sys_queue_init

原型：int32_t sys_queue_init (os_queue_t *queue, int32_t queue_size, uint32_t item_size)

功能：创建队列。

输入参数：queue_size，队列的大小；

item_size，队列消息的大小。

输出参数：queue，队列句柄。

返回：0，创建成功；-1，创建失败。

### 2.3.15. sys_queue_free

原型：void sys_queue_free (os_queue_t *queue)

功能：销毁消息队列。

输入参数：queue，队列句柄。

输出参数：无。

返回：无。

### 2.3.16. sys_queue_post

原型：int32_t sys_queue_post (os_queue_t *queue, void *msg)

功能：向队列发送消息。

输入参数：queue，队列句柄；

msg，消息指针。

输出参数：无。

返回：0，发送成功；-1，失败。

### 2.3.17. sys_queue_fetch

原型：int32_t sys_queue_fetch (os_queue_t *queue, void *msg, uint32_t timeout_ms, uint8_t is_blocking)

功能：从队列中获取一个消息。

输入参数：queue，队列句柄；

timeout_ms，等待超时时间；

is_blocking，是否是阻塞操作。

输出参数：msg，消息指针。

返回：0，发送成功；-1，失败。

## 2.4. 时间管理

### 2.4.1. sys_current_time_get

原型：uint32_t sys_current_time_get (void)

功能：获取系统 bootup 以来的时间。

输入参数：无。

输出参数：无。

返回：系统 bootup 以来的时间，单位毫秒。

### 2.4.2. sys_ms_sleep

原型：void sys_ms_sleep (uint32_t ms)

功能：让任务进入睡眠。

输入参数：ms，睡眠时间。

输出参数：无。

返回：无。

### 2.4.3. sys_us_delay

原型：void sys_us_delay (uint32_t nus)

功能：延迟操作。

输入参数：nus，延迟时间，单位微秒。

输出参数：无。

返回：无。

### 2.4.4. sys_timer_init

原型：void sys_timer_init (os_timer_t *timer, const uint8_t *name, uint32_t delay, uint8_t periodic, timer_func_t func, void *arg)

功能：创建定时器。

输入参数：timer，定时器句柄；

name，定时器名字；

delay，定时器超时时间；

periodic，是否为周期性定时器；

func，定时器函数；

arg，定时器函数参数。

输出参数：无。

返回：无。

### 2.4.5. sys_timer_delete

原型：void sys_timer_delete (os_timer_t *timer)

功能：销毁定时器。

输入参数：timer，定时器句柄。

输出参数：无。

返回：无。

### 2.4.6. sys_timer_start

原型：void sys_timer_start (os_timer_t *timer, uint8_t from_isr);

功能：启动定时器。

输入参数：timer，定时器句柄；

from_isr，是否在 ISR 中。

输出参数：无。

返回：无。

### 2.4.7. sys_timer_start_ext

原型：void sys_timer_start_ext (os_timer_t *timer, uint32_t delay, uint8_t from_isr)

功能：启动定时器。

输入参数：timer，定时器句柄；

delay，重设定时器超时时间；

from_isr，是否在 ISR 调用。

输出参数：无。

返回：无。

### 2.4.8. sys_timer_stop

原型：uint8_t sys_timer_stop (os_timer_t *timer, uint8_t from_isr)

功能：停止定时器。

输入参数：timer，定时器句柄；

from_isr，是否在 ISR 调用。

输出参数：无。

返回：1，操作成功；0，操作失败。

### 2.4.9. sys_timer_pending

原型：uint8_t sys_timer_pending (os_timer_t *timer)

功能：判断定时器是否在激活队列等待中。

输入参数：timer，定时器句柄。

输出参数：无。

返回：1，在激活队列等待中；0，其他状态。

## 2.5. 其他系统管理

### 2.5.1. sys_os_init

原型：void sys_os_init (void)

功能：RTOS 初始化。

输入参数：无。

输出参数：无。

返回：无。

### 2.5.2. sys_os_start

原型：void sys_os_start (void)

功能：RTOS 开始调度。

输入参数：无。

输出参数：无。

返回：无。

### 2.5.3. sys_os_misc_init

原型：void sys_os_misc_init (void)

功能：RTOS 在调度之后的其他初始化，有些 RTOS 需要。

输入参数：无。

输出参数：无。

返回：无。

### 2.5.4. sys_yield

原型：void sys_yield (void)

功能：任务放弃 CPU 控制权。

输入参数：无。

输出参数：无。

返回：无。

### 2.5.5. sys_sched_lock

原型：void sys_sched_lock (void)

功能：暂停任务调度。

输入参数：无。

输出参数：无。

返回：无。

### 2.5.6. sys_sched_unlock

原型：void sys_sched_unlock (void)

功能：继续任务调度。

输入参数：无。

输出参数：无。

返回：无。

### 2.5.7. sys_random_bytes_get

原型：int32_t sys _random_bytes_get (void *dst, uint32_t size)

功能：获取随机数据。

输入参数：size，随机数据长度。

输出参数：dst，保存随机数的地址。

返回：0，操作成功；-1，获取失败。

# 3. Wi-Fi Netlink API

头文件 SDK \ NSPE \ WIFI_IOT \ wifi \ wifi_netlink.h。

## 3.1. Wi-Fi 消息类型

### 3.1.1. WIFI_MESSAGE_TYPE_E

功能：这个枚举包含了各种 wifi message 的类型。

```
typedef enum {
    WIFI_MESSAGE_NOTIFY_SCAN_RESULT_SUCC = 1,
    WIFI_MESSAGE_NOTIFY_SCAN_RESULT_FAIL,
    WIFI_MESSAGE_INDICATE_CONN_SUCCESS,
    WIFI_MESSAGE_INDICATE_CONN_NO_AP,
    WIFI_MESSAGE_INDICATE_CONN_ASSOC_FAIL,
    WIFI_MESSAGE_INDICATE_CONN_HANDSHAKE_FAIL,
    WIFI_MESSAGE_INDICATE_CONN_FAIL,
    WIFI_MESSAGE_INDICATE_DISCON_REKEY_FAIL,
    WIFI_MESSAGE_INDICATE_DISCON_MIC_FAIL,
    WIFI_MESSAGE_INDICATE_DISCON_RECV_DEAUTH,
    WIFI_MESSAGE_INDICATE_DISCON_NO_BEACON,
    WIFI_MESSAGE_INDICATE_DISCON_AP_CHANGED,
    WIFI_MESSAGE_INDICATE_DISCON_FROM_UI,
    WIFI_MESSAGE_INDICATE_DISCON_UNSPECIFIED,
    WIFI_MESSAGE_SHELL_COMMAND,
    WIFI_MESSAGE_TASK_TERMINATE,
    WIFI_MESSAGE_NUM
} WIFI_MESSAGE_TYPE_E;
```

### 3.1.2. WIFI_NETLINK_STATUS_E

功能：这个枚举包含了各种 wifi netlink 的状态。

```
typedef enum {
    WIFI_NETLINK_STATUS_NO_LINK = 0,
    WIFI_NETLINK_STATUS_NO_LINK_SCAN,
    WIFI_NETLINK_STATUS_LINKING,
    WIFI_NETLINK_STATUS_ROAMING,
    WIFI_NETLINK_STATUS_LINKED,
    WIFI_NETLINK_STATUS_LINKED_SCAN,
    WIFI_NETLINK_STATUS_LINKED_CONFIGED,
    WIFI_NETLINK_STATUS_NUM
} WIFI_NETLINK_STATUS_E;
```

## 3.2. Netlink 数据结构

### 3.2.1. WIFI_NETLINK_INFO_T

功能：这个结构体用来保存 wifi netlink 的信息。

```
typedef struct wifi_netlink_info {
    uint8_t device_opened;
    uint8_t ap_started;
    WIFI_NETLINK_STATUS_E link_status;
    // WIFI_NETLINK_STATUS_E saved_link_status;
    uint8_t scan_ap_num;
    uint8_t valid_ap_num;
    uint8_t scan_list_ready;
    struct wifi_scan_info scan_list[SUPPORT_MAX_AP_NUM];
    struct wifi_scan_info *scan_list_head;
    struct wifi_ssid_config connect_info;
    struct wifi_connect_result connected_ap_info;
    WIFI_DISCON_REASON_E discon_reason;
    struct wifi_ap_config ap_conf;
```

```
    void (*promisc_callback)(unsigned char *, unsigned short, signed char);
    void (*promisc_mgmt_callback)(unsigned char*, int, signed char, int);
    uint32_t promisc_mgmt_filter;
    uint8_t scan_blocked;
    uint8_t conn_blocked;
    os_sema_t block_sema;
} WIFI_NETLINK_INFO_T;
```

## 3.3. 接口函数

### 3.3.1. wifi_netlink_init

原型：WIFI_NETLINK_INFO_T *wifi_netlink_init (void)

功能：该函数初始化一个 WIFI_NETLINK_INFO_T 结构体，并返回指向该结构体的指针。

输入参数：无。

输出参数：无。

返回值：WIFI_NETLINK_INFO_T 结构体指针。

### 3.3.2. wifi_netlink_dev_open

原型：int wifi_netlink_dev_open (void)

功能：打开 Wi-Fi 设备。

输入参数：无。

输出参数：无。

返回值：执行成功返回 0。

### 3.3.3. wifi_netlink_dev_close

原型：int wifi_netlink_dev_close (void)

功能：关闭 Wi-Fi 设备。

输入参数：无。

输出参数：无。

返回值：执行成功返回 0。

### 3.3.4. wifi_netlink_scan_set

原型：int wifi_netlink_scan_set (void)

功能：该函数用于执行 Wi-Fi 扫描操作。

输入参数：无。

输出参数：无。

返回值：0，执行成功，非 0，失败。

### 3.3.5. wifi_netlink_scan_list_get

原型：int wifi_netlink_scan_list_get (iter_scan_item iterator)

功能：该函数用于获取扫描的结果，常与 wifi_netlink_set_scan 函数搭配使用。

输入参数：iter_scan_item 遍历函数指针，开发者自己实现函数获取扫描的结果，参见 ***iter_scan_item***。

输出参数：无。

返回值：执行成功返回 0。

### 3.3.6. iter_scan_item

原型：typedef void (*iter_scan_item) (struct wifi_scan_info *scan_item)

功能：定义了 iter_scan_item 类型函数。

输入参数：wifi_scan_info 结构体指针，该结构体保存了一个热点信息。

输出参数：无。

### 3.3.7. wifi_netlink_connect_req

原型：int wifi_netlink_connect_req (uint8_t *ssid, uint8_t *password)

功能：该函数用于执行 Wi-Fi 连接操作，连接的 AP 由参数指定。

输入参数：ssid，指向待连接 AP 的 SSID 的指针；

password，指向待连接 AP 的密码的指针。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

### 3.3.8. wifi_netlink_disconnect_req

原型：int wifi_netlink_disconnect_req (void)

功能：该函数用于执行 Wi-Fi 断开连接操作，与连接的 AP 断开。

输入参数：无。

输出参数：无。

返回值：执行成功返回 0。

### 3.3.9. wifi_netlink_status_get

原型：int wifi_netlink_status_get (void)

功能：该函数用于获取当前 Wi-Fi 的状态信息。

输入参数：无。

输出参数：无。

返回值：执行成功返回 0。

### 3.3.10. wifi_netlink_ipaddr_set

原型：int wifi_netlink_ipaddr_set (uint8_t *ipaddr)

功能：该函数用于告知 Wi-Fi driver 连接到 AP 后 AP 分配的 IP 地址。

输入参数：ipaddr，指向 IP 地址的指针。

输出参数：无。

返回值：执行成功返回 0。

### 3.3.11. wifi_netlink_ap_start

原型：int wifi_netlink_ap_start(char *ssid, char *password, uint8_t channel, uint8_t hidden)

功能：该函数用于启用软 AP 模式，软 AP 的配置信息由参数决定。

输入参数：ssid，指向待配置软 AP 的 SSID 的指针；

password，指向待配置软 AP 的密码的指针；

channel，待配置软 AP 的 channel；

hidden，决定该软 AP 的 SSID 是否广播。0 代表广播 SSID，1 代表不广播 SSID。

输出参数：无。

返回值：执行成功返回 0；失败返回-1。

### 3.3.12. wifi_netlink_channel_set

原型：int wifi_netlink_channel_set(uint32_t channel, uint32_t bandwidth, uint32_t ch_offset)

功能：该函数用于设置工作信道，信道的配置信息由参数决定。

输入参数：channel，信道编号。

bandwidth，工作信道的带宽。带宽的取值由枚举类型 CHANNEL_WIDTH 决定，目前只有 CHANNEL_WIDTH_20 和 CHANNEL_WIDTH_40 可用，如下：

```
typedef enum {
    CHANNEL_WIDTH_20 = 0,
    CHANNEL_WIDTH_40 = 1,
    CHANNEL_WIDTH_80 = 2,
    CHANNEL_WIDTH_160 = 3,
    CHANNEL_WIDTH_80_80 = 4,
    CHANNEL_WIDTH_MAX = 5,
}CHANNEL_WIDTH;
```

ch_offset，扩展信道的位置。边带的取值由枚举类型 HT_SEC_CHNL_OFFSET 决定，其中，20M 带宽对应 HT_SEC_CHNL_OFFSET_NONE，40M 带宽对应 HT_SEC_CHNL_OFFSET_ABOVE 与 HT_SEC_CHNL_OFFSET_BELOW，其余暂不可用。

```
typedef enum {
    HT_SEC_CHNL_OFFSET_NONE = 0,
    HT_SEC_CHNL_OFFSET_ABOVE = 1,
    HT_SEC_CHNL_OFFSET_RESV = 2,
    HT_SEC_CHNL_OFFSET_BELOW = 3,
    HT_SEC_CHNL_OFFSET_MAX = 4
} HT_SEC_CHNL_OFFSET;
```

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

### 3.3.13. wifi_netlink_ps_set

原型：int wifi_netlink_ps_set (int ps_mode)

功能：该函数用于设置睡眠模式。

输入参数：ps_mode：

0：Wi-Fi 和 CPU 都关闭睡眠模式；

1：代表 Wi-Fi 睡眠模式 + CPU 睡眠模式；

2：代表 Wi-Fi 睡眠模式 + CPU 深度睡眠模式。

输出参数：无。

返回值：执行成功返回 0。

### 3.3.14. wifi_netlink_ps_get

原型：int wifi_netlink_ps_get (void)

功能：该函数用于获取睡眠模式。

输入参数：无。

输出参数：无。

返回值：睡眠模式：

0：Wi-Fi 和 CPU 都关闭睡眠模式；

1：代表 Wi-Fi 睡眠模式 + CPU 睡眠模式；

2：代表 Wi-Fi 睡眠模式 + CPU 深度睡眠模式。

### 3.3.15. wifi_netlink_bss_rssi_get

原型：int wifi_netlink_bss_rssi_get (void)

功能：该函数用于获取已连接 AP 的 RSSI 值。

输入参数：无。

输出参数：无。

返回值：执行成功则返回获取的 RSSI 值，失败返回 0。

### 3.3.16. wifi_netlink_ap_channel_get

原型：int wifi_netlink_ap_channel_get (uint8_t *bssid)

功能：该函数用于获取某个 AP 所在的信道。

输入参数：bssid，指向待获取信道的 AP 的 bssid 的指针。

输出参数：无。

返回值：执行成功返回对应 AP 的信道，失败返回 0。

### 3.3.17. wifi_netlink_task_stack_get

原型：int wifi_netlink_task_stack_get (void)

功能：该函数用于获取当前每个任务的空闲栈。

输入参数：无。

输出参数：无。

返回值：执行成功返回 0。

### 3.3.18. wifi_netlink_link_state_get

原型：int wifi_netlink_link_state_get (void)

功能：该函数用于获取当前 Wi-Fi 的状态。

输入参数：无。

输出参数：无。

返回值：当前 Wi-Fi 的状态，在枚举 WIFI_NETLINK_STATUS_E 中列出。

### 3.3.19. wifi_netlink_linked_ap_info_get

原型：int wifi_netlink_linked_ap_info_get (uint8_t *ssid, uint8_t *passwd, uint8_t *bssid)

功能：该函数用于获取当前已连接 AP 的信息，包括 SSID，password 以及 BSSID。

输入参数：无。

输出参数：ssid，获取的 SSID 保存到该指针；

passwd，获取的 password 保存到该指针；

bssid，获取的 bssid 保存到该指针。

返回值：执行成功返回 0。

### 3.3.20. wifi_netlink_raw_send

原型：int wifi_netlink_raw_send (uint8_t *buf, uint32_t len)

功能：该函数可用于进行任意数据发送，驱动不会组装 MAC 头，用户需要组装完整的 Wi-Fi 数据帧。

输入参数：buf，指向待发送数据的指针；

len，待发送数据的长度。

输出参数：无。

返回值：执行成功返回 1，失败返回 0。

### 3.3.21. wifi_netlink_promisc_mode_set

原型：int wifi_netlink _promisc_mode_set (uint32_t enable,

void (*callback) (unsigned char*, unsigned short, signed char))

功能：该函数用于设置混杂模式。

输入参数：enable，1 代表使能混杂模式，0 代表不使能；

void (*callback) (unsigned char*, unsigned short, signed char))，处理接收包的回调函数，其中，unsigned char*是指向 data 的指针，unsigned short 是 data 的长度，signed char 是对应的 RSSI。

输出参数：无。

返回值：执行成功返回 0。

### 3.3.22. wifi_netlink_promisc_mgmt_cb_set

原型：int wifi_netlink_promisc_mgmt_cb_set (uint32_t filter_mask,

void (*callback) (unsigned char*, int, signed char, int))

功能：设置混杂模式下管理帧的回调处理函数。

输入参数：filter_mask，过滤对应管理子帧的位码，置 1 接收处理，置 0 过滤；

void (*callback)(unsigned char*, int, signed char, int)，处理接收包的回调函数，其中，unsigned char*是指向 data 的指针，unsigned short 是 data 的长度，signed char 是对应的 RSSI。

输出参数：无。

返回值：执行成功返回 0。

### 3.3.23. wifi_netlink_promisc_filter_set

原型：int wifi_netlink_promisc_filter_set (uint8_t filter_type, uint8_t *filter_value)

功能：混杂模式下，过滤帧。

输入参数：filter_type：

0：过滤超过 filter_value 长度的帧；

1：过滤小于 filter_value 能量的帧；

2：过滤超过 filter_value 长度 a-mpdu 帧。

filter_value，门限值。

输出参数：无。

返回值：执行成功返回 0。

### 3.3.24. wifi_netlink_auto_conn_set

原型：int wifi_netlink_auto_conn_set (uint8_t auto_conn_enable)

功能：该函数用于设置开机是否自动连接 AP。

输入参数：auto_conn_enable，1 代表使能，0 代表不使能。

输出参数：无。

返回值：执行成功返回 0。

### 3.3.25. wifi_netlink_auto_conn_get

原型：uint8_t wifi_netlink_auto_conn_get (void)

功能：该函数用于查看是否设置了开机自动连接 AP。

输入参数：无。

输出参数：无。

返回值：执行成功返回已设置的值，执行失败返回 0。

### 3.3.26. wifi_netlink_joined_ap_store

原型：int wifi_netlink_joined_ap_store (void)

功能：该函数用于将已连接的 AP 信息保存到 flash。

输入参数：无。

输出参数：无。

返回值：执行成功返回 0，-1，SSID 长度错误, -2 密码长度错误。

### 3.3.27. wifi_netlink_joined_ap_load

原型：int wifi_netlink_joined_ap_load (void)

功能：该函数用于使用已保存的 AP 信息连接。

输入参数：无。

输出参数：无。

返回值：执行成功返回 0；失败返回-1。

# 4. Wi-Fi Netif API

头文件 SDK \ NSPE \ WIFI_IOT \ network \ lwip-2.1.2 \ port \ wifi_netif.h。

## 4.1. Wi-Fi Lwip 网络接口 API

### 4.1.1. wifi_netif_open

原型：void wifi_netif_open (void)

功能：向 Lwip 注册 Wi-Fi 网络接口。

输入参数：无。

输出参数：无。

返回值：无。

### 4.1.2. wifi_netif_close

原型：void wifi_netif_close (void)

功能：关闭 Lwip 中的 Wi-Fi 网络接口。

输入参数：无。

输出参数：无。

返回值：无。

### 4.1.3. wifi_netif_set_hwaddr

原型：uint8_t wifi_netif_set_hwaddr (uint8_t *mac_addr)

功能：设置 Wi-Fi 网络接口的 MAC 地址。

输入参数：mac_addr,指向 6 字节地址组的指针。

输出参数：无。

返回值：操作成功为 TRUE，失败为 FALSE。

### 4.1.4. wifi_netif_get_hwaddr

原型：uint8_t *wifi_netif_get_hwaddr (void)

功能：获取 Wi-Fi 网络接口的 MAC 地址。

输入参数：无。

输出参数：无。

返回值：指向网络接口的 MAC 地址指针。

### 4.1.5. ip_addr_t *wifi_netif_get_ip

原型：ip_addr_t *wifi_netif_get_ip (void)

功能：获取 Wi-Fi 网络接口的 ip 地址。

输入参数：无。

输出参数：无。

返回值：指向网络接口的 ip 地址指针。

### 4.1.6. wifi_netif_set_ip

原型：void wifi_netif_set_ip (ip_addr_t *ip, ip_addr_t *netmask, ip_addr_t *gw)

功能：设置 Wi-Fi 网络接口的 ip 地址、掩码、网关。

输入参数：ip，指向 IP 地址的指针；

netmask，指向网络掩码的指针；

gw，指向网关地址的指针。

输出参数：无。

返回值：无。

### 4.1.7. wifi_netif_get_gw

原型：ip_addr_t *wifi_netif_get_gw (void)

功能：获取 Wi-Fi 网络接口的网关地址。

输入参数：无。

输出参数：无。

返回值：指向网络接口的网关地址指针。

### 4.1.8. wifi_netif_get_netmask

原型：ip_addr_t *wifi_netif_get_netmask (void)

功能：获取 Wi-Fi 网络接口的掩码。

输入参数：无。

输出参数：无。

返回值：指向网络接口的掩码指针。

### 4.1.9. wifi_netif_set_up

原型：void wifi_netif_set_up (void)

功能：使能 Wi-Fi 网络接口。

输入参数：无。

输出参数：无。

返回值：无。

### 4.1.10. wifi_netif_set_down

原型：void wifi_netif_set_down (void)

功能：禁用 Wi-Fi 网络接口。

输入参数：无。

输出参数：无。

返回值：无。

### 4.1.11. wifi_netif_is_ip_got

原型：int32_t wifi_netif_is_ip_got (void)

功能：判断 Wi-Fi 网络接口是否已设置 IP。

输入参数：无。

输出参数：无。

返回值：1，网络接口已设置 ip；

0，网络接口没有设置 ip。

### 4.1.12. wifi_netif_start_dhcp

原型：void wifi_netif_start_dhcp (void)

功能：在 Wi-Fi 网络接口上进行 DHCP 获取 ip。

输入参数：无。

输出参数：无。

返回值：无。

### 4.1.13. wifi_netif_polling_dhcp

原型：int32_t wifi_netif_polling_dhcp (void)

功能：轮询 DHCP 是否结束。

输入参数：无。

输出参数：无

返回值：1，DHCP 成功；

0，DHCP 失败。

### 4.1.14. wifi_netif_stop_dhcp

原型：void wifi_netif_stop_dhcp (void)

功能：停止 DHCP。

输入参数：无。

输出参数：无。

返回值：无。

### 4.1.15. wifi_netif_set_ip_mode

原型：void wifi_netif_set_ip_mode (uint8_t ip_mode)

功能：设置网络接口地址模式，目前支持 DHCP 模式和静态地址模式。

输入参数：ip_mode，IP_MODE_STATIC，静态地址模式；

IP_MODE_DHCP，DHCP 自动获取模式。

输出参数：无。

返回值：无。

### 4.1.16. wifi_netif_is_static_ip_mode

原型：int32_t wifi_netif_is_static_ip_mode (void)

功能：判断网络接口是否是静态地址模式。

输入参数：无。

输出参数：无。

返回值：1，是静态地址模式。

0，不是静态地址模式。

# 5. Wi-Fi Management API

此章节介绍 Wi-Fi Management 连接管理服务 API 及 event loop 组件 API。

## 5.1. Wi-Fi 连接管理服务

头文件 SDK \ NSPE \ WIFI_IOT \ wifi \ wifi_management.h。

### 5.1.1. wifi_management_init

原型：void wifi_management_init (void)

功能：初始化 Lwip，Netlink 等，只需要调用一次。

输入参数：无。

输出参数：无。

返回值：无。

### 5.1.2. wifi_management_scan

原型：int wifi_management_scan (uint8_t blocked)

功能：启动扫描无线网络。

输入参数：blocked，1：阻塞其他操作，0：不阻塞。

输出参数：无。

返回值：0：启动扫描成功；-1：启动扫描失败。

### 5.1.3. wifi_management_connect

原型：int wifi_management_connect (uint8_t *ssid, uint8_t *password, uint8_t blocked)

功能：启动连接 AP。

输入参数：ssid，AP 的网络名称 1 - 32 字符；

Password，AP 的密码 8 – 63 字符，如加密方式为 Open，可以为 NULL；

blocked，1：阻塞其他操作，0：不阻塞。

输出参数：无。

返回值：0：启动连接成功；

-1：Wi-Fi 驱动链路忙；

-2：ssid 参数为 NULL；

-3：ssid 参数错误；

-4：password 参数错误。

### 5.1.4. wifi_management_disconnect

原型：int wifi_management_disconnect (void)

功能：启动断开 AP。

输入参数：无。

输出参数：无。

返回值：0：启动断开 AP 成功；

-1：启动失败。

### 5.1.5. wifi_management_sta_start

原型：void wifi_management_sta_start (void)

功能：SDK 进入 STA mode，如果当前为软 AP mode，软 AP 功能停止。

输入参数：无。

输出参数：无。

返回值：无。

### 5.1.6. wifi_management_ap_start

原型：void wifi_management_ap_start (char *ssid, char *passwd, uint32_t channel, uint32_t hidden)

功能：启动软 AP，SDK 进入软 AP mode。

输入参数：ssid：软 AP 网络名称，1 - 32 字符；

Password：软 AP 网络密码，默认加密方式 WPA2-PSK；

Channel：软 AP 所在的网络信道，1 - 13；

Hidden：是否隐藏 ssid。0：广播 ssid，1：隐藏 ssid。

输出参数：无。

返回值：无。

#### 5.1.7. wifi_management_ap_assoc_info

原型：uint32_t wifi_management_ap_assoc_info (uint8_t *assoc_info)

功能：获取关联软 AP 的客户端列表。

输入参数：无。

输出参数：用于存储客户端 MAC 的 buffer，建议大小为 MAX_STATION_NUM * ETH_ALEN。

返回值：关联软热点的客户端个数。

#### 5.1.8. wifi_management_block_wait

原型：int wifi_management_block_wait ()

功能：阻塞操作，等待扫描完成或连接完成。

输入参数：无。

输出参数：无。

返回值：0，操作成功。

### 5.2. Wi-Fi event loop API

头文件 SDK \ NSPE \ WIFI_IOT \ wifi \ wifi_eloop.h。

#### 5.2.1. eloop_event_handler

原型：typedef void (*eloop_event_handler) (void *eloop_data, void *user_ctx)

功能：定义了 eloop_event_handler 类型函数，用于通用 event 触发时的回调。

输入参数：eloop_data，用于回调的 eloop 上下文数据；

user_ctx，用于回调的用户上下文数据。

输出参数：无。

返回值：无。

#### 5.2.2. eloop_timeout_handler

原型：typedef void (*eloop_timeout_handler) (void *eloop_data, void *user_ctx)

功能：定义了 eloop_timeout_handler 类型函数，用于定时器超时事件发生时的回调。

输入参数：eloop_data，用于回调的 eloop 上下文数据；

user_ctx，用于回调的用户上下文数据。

输出参数：无。

返回值：无。

### 5.2.3. eloop_init

原型：int eloop_init (void)

功能：该函数初始化一个全局事件处理循环数据。

输入参数：无。

输出参数：无。

返回值：执行成功返回 0；失败返回-1。

### 5.2.4. eloop_event_register

原型：int eloop_event_register (eloop_event_t event,

eloop_event_handler handler,

void *eloop_data, void *user_data)

功能：该函数注册一个用于处理触发事件的函数。

输入参数：eloop_event_t event，触发后需要处理的事件；

handler，事件触发后的回调函数，用于处理事件；

eloop_data，回调函数的参数；

user_data，回调函数的参数。

输出参数：无。

返回值：执行成功返回 0；失败返回-1。

### 5.2.5. eloop_event_unregister

原型：void eloop_event_unregister (eloop_event_t event)

功能：该函数用于中止一个事件触发后的处理函数，与 eloop_event_register 对应。

输入参数：event，中止处理的事件。

输出参数；无。

返回值：无。

### 5.2.6. eloop_event_send

原型：int eloop_event_send (eloop_event_t event)

功能：该函数用于将事件发送到待处理队列中。

输入参数：event，待发送的事件。

输出参数：无。

返回值：执行成功返回 0；失败返回-1。

### 5.2.7. eloop_message_send

原型：int eloop_message_send (eloop_message_t message)

功能：该函数用于将消息发送到待处理队列中。

输入参数：message，待发送的消息。

输出参数：无。

返回值：执行成功返回 0；失败返回-1。

### 5.2.8. eloop_timeout_register

原型：int eloop_timeout_register (unsigned int msecs,

eloop_timeout_handler handler,

void *eloop_data, void *user_data)

功能：该函数用于注册一个用于处理触发的 event timeout 的函数。

输入参数：msecs，超时时间，单位 ms；

handler，超时后的回调函数，处理超时事件；

eloop_data，回调函数的参数；

user_data，回调函数的参数。

输出参数：无。

返回值：执行成功返回 0；失败返回-1。

### 5.2.9. eloop_timeout_cancel

原型：int eloop_timeout_cancel (eloop_timeout_handler handler,

void *eloop_data, void *user_data)

功能：该函数用于中止一个定时器。

输入参数：handler，需要中止的超时后的回调函数；

eloop_data，回调函数的参数；

user_data，回调函数的参数。

输出参数：无。

返回值：返回中止定时器的个数。

注：eloop_data / user_data 的值是 ELOOP_ALL_CTX 时代表所有超时。

### 5.2.10. eloop_is_timeout_registered

原型：int eloop_is_timeout_registered (eloop_timeout_handler handler,

void *eloop_data, void *user_data)

功能：该函数用于检测是否注册过定时器

输入参数：eloop_timeout_handler handler，匹配的回调函数；

eloop_data，匹配的 eloop_data；

user_data，匹配的 user_data。

输出参数：无。

返回值：注册过返回 1；未注册返回 0。

### 5.2.11. eloop_run

原型：void eloop_run (void)

功能：该函数用于启动 event 循环。

输入参数：无。

输出参数：无。

返回值：无。

### 5.2.12. eloop_terminate

原型：void eloop_terminate (void)

功能：该函数用于中止 event 处理线程。

输入参数：无。

输出参数：无。

返回值：无。

### 5.2.13. eloop_destroy

原型：void eloop_destroy (void)

功能：该函数用于释放所有用于 event 循环的资源。

输入参数：无。

输出参数：无。

返回值：无。

### 5.2.14. eloop_terminated

原型：int eloop_terminated (void)

功能：该函数用于检测是否存在事件循环。

输入参数：无。

输出参数：无。

返回值：存在返回 1；不存在返回 0。

## 5.3. 宏的类型

### 5.3.1. Wi-Fi 事件类型

WIFI_MGMT_EVENT_SCAN_DONE

WIFI_MGMT_EVENT_SCAN_FAIL

WIFI_MGMT_EVENT_CONNECT_SUCCESS

WIFI_MGMT_EVENT_CONNECT_FAIL

WIFI_MGMT_EVENT_DISCONNECT

WIFI_MGMT_EVENT_DHCP_SUCCESS

WIFI_MGMT_EVENT_DHCP_FAIL

WIFI_MGMT_EVENT_AUTO_CONNECT

### 5.3.2. 配置宏

WIFI_MGMT_ROAMING_RETRY_LIMIT // Wi-Fi 漫游重试的次数

WIFI_MGMT_ROAMING_RETRY_INTERVAL // 漫游重试的时间间隔

WIFI_MGMT_DHCP_POLLING_LIMIT // 轮询 DHCP 成功的次数，

WIFI_MGMT_DHCP_POLLING_INTERVAL // 轮询 DHCP 成功的间隔

WIFI_MGMT_LINK_POLLING_INTERVAL // 轮询 Wi-Fi 连接质量的间隔

WIFI_MGMT_TRIGGER_ROAMING_RSSI_THRESHOLD // 触发漫游信号质量的门限值

WIFI_MGMT_START_ROAMING_RSSI_THRESHOLD_1 // 侯选漫游 AP 超过平均信号质量

WIFI_MGMT_START_ROAMING_RSSI_THRESHOLD_2 // 侯选漫游 AP 超过平均信号质量

WIFI_MGMT_START_SCAN_THROTTLE_INTERVAL // 慢速扫描的时间间隔

WIFI_MGMT_START_SCAN_FAST_INTERVAL // 快速扫描的时间间隔

# 6. 应用举例

在 ***Wi-Fi SDK 软件启动过程***小节 SDK 启动完成之后，开发者就可以使用组件进行 Wi-Fi 应用开发了。下面简单举例如何用组件的 API 完成扫描无线网络、连接 AP、启动软 AP 和接入阿里云等操作。

## 6.1. 扫描无线网络

### 6.1.1. 阻塞模式扫描

此例中 **scan_wireless_network** 启动扫描之后，阻塞等待扫描完成，并打印出扫描的结果。

```
#include "app_cfg.h"
#include "app_type.h"
#include "bsp_inc.h"
#include "osal_types.h"
#include "wlan_debug.h"
#include "wrapper_os.h"
#include "debug_print.h"
#include "malloc.h"
#include "wifi_netlink.h"
#include "wifi_netif.h"
#include "wifi_management.h"
void print_scan_info (struct wifi_scan_info *item)
{
  char *encrypt, *cipher;
  switch (item->encryp_protocol) {
    case WIFI_ENCRYPT_PROTOCOL_OPENSYS:
      encrypt = "Open";
      break;
    case WIFI_ENCRYPT_PROTOCOL_WEP:
      encrypt = "WEP";
      break;
    case WIFI_ENCRYPT_PROTOCOL_WPA:
      encrypt = "WPA";
      break;
    case WIFI_ENCRYPT_PROTOCOL_WPA2:
      encrypt = "WPA2";
      break;
    case WIFI_ENCRYPT_PROTOCOL_WAPI:
      encrypt = "WAPI";
      break;
```

```
        case WIFI_ENCRYPT_PROTOCOL_WPA3_TRANSITION:
            encrypt = "WPA2/WPA3";
            break;
        case WIFI_ENCRYPT_PROTOCOL_WPA3_ONLY:
            encrypt = "WPA3";
            break;
        default:
            encrypt = "";
            break;
    }
    switch (item->pairwise_cipher) {
        case WIFI_CIPHER_TKIP:
            cipher = "-TKIP";
            break;
        case WIFI_CIPHER_CCMP:
            cipher = "-CCMP";
            break;
        default:
            cipher = "";
            break;
    }
    printf("AP: %s,\t\t %d, "MAC_FMT", %d, %s%s\r\n",
        item->ssid.ssid, item->rssi, MAC_ARG(item->bssid_info.bssid),
        item->channel, encrypt, cipher);
}
            encrypt = "WAPI";
            break;
        case WIFI_ENCRYPT_PROTOCOL_WPA3_TRANSITION:
            encrypt = "WPA2/WPA3";
            break;
        case WIFI_ENCRYPT_PROTOCOL_WPA3_ONLY:
            encrypt = "WPA3";
            break;
        default:
            encrypt = "";
            break;
    }
    switch (item->pairwise_cipher) {
        case WIFI_CIPHER_TKIP:
            cipher = "-TKIP";
            break;
        case WIFI_CIPHER_CCMP:
            cipher = "-CCMP";
```

```
            break;
        default:
            cipher = "";
            break;
    }
    printf("AP: %s,\t\t %d, "MAC_FMT", %d, %s%s\r\n",
            item->ssid.ssid, item->rssi, MAC_ARG(item->bssid_info.bssid),
            item->channel, encrypt, cipher);
}
int scan_wireless_network(int argc, char **argv)
{
    if (wifi_management_scan(TRUE) != 0) {
        return -1;
    }
    wifi_management_block_wait();
    if (p_wifi_netlink->scan_list_ready != 1)
        return -1
    wifi_netlink_scan_list_get(print_scan_info);
        return 0;
}
```

### 6.1.2. 非阻塞式扫描

此例中，**scan_wireless_network** 启动扫描，注册扫描完成事件。事件触发之后，获取扫描结果并打印。

```
#include "app_cfg.h"
#include "app_type.h"
#include "bsp_inc.h"
#include "osal_types.h"
#include "wlan_debug.h"
#include "wrapper_os.h"
#include "debug_print.h"
#include "malloc.h"
#include "wifi_netlink.h"
#include "wifi_netif.h"
#include "wifi_management.h"
void print_scan_info(struct wifi_scan_info *item)
{
    char *encrypt, *cipher;
    switch (item->encryp_protocol) {
    case WIFI_ENCRYPT_PROTOCOL_OPENSYS:
        encrypt = "Open";
        break;
```

```
        case WIFI_ENCRYPT_PROTOCOL_WEP:
            encrypt = "WEP";
            break;
        case WIFI_ENCRYPT_PROTOCOL_WPA:
            encrypt = "WPA";
            break;
        case WIFI_ENCRYPT_PROTOCOL_WPA2:
            encrypt = "WPA2";
            break;
        case WIFI_ENCRYPT_PROTOCOL_WAPI:
            encrypt = "WAPI";
            break;
        case WIFI_ENCRYPT_PROTOCOL_WPA3_TRANSITION:
            encrypt = "WPA2/WPA3";
            break;
        case WIFI_ENCRYPT_PROTOCOL_WPA3_ONLY:
            encrypt = "WPA3";
            break;
        default:
            encrypt = "";
            break;
    }
    switch (item->pairwise_cipher) {
        case WIFI_CIPHER_TKIP:
            cipher = "-TKIP";
            break;
        case WIFI_CIPHER_CCMP:
            cipher = "-CCMP";
            break;
        default:
            cipher = "";
            break;
    }
    printf("AP: %s,\t\t %d, "MAC_FMT", %d, %s%s\r\n",
           item->ssid.ssid, item->rssi, MAC_ARG(item->bssid_info.bssid),
           item->channel, encrypt, cipher);
}
void cb_scan_done(void *eloop_data, void *user_ctx)
{
    printf("[Scanned AP list]\r\n");
    wifi_netlink_scan_list_get(print_scan_info);
    eloop_event_unregister(WIFI_MGMT_EVENT_SCAN_DONE);
    eloop_event_unregister(WIFI_MGMT_EVENT_SCAN_FAIL);
```

```
}
void cb_scan_fail(void *eloop_data, void *user_ctx)
{
    printf("WIFI_SCAN: failed\r\n");
    eloop_event_unregister(WIFI_MGMT_EVENT_SCAN_DONE);
    eloop_event_unregister(WIFI_MGMT_EVENT_SCAN_FAIL);
}
int scan_wireless_network()
{
    eloop_event_register(WIFI_MGMT_EVENT_SCAN_DONE, cb_scan_done, NULL, NULL);
    eloop_event_register(WIFI_MGMT_EVENT_SCAN_FAIL, cb_scan_fail, NULL, NULL);
  if (wifi_management_scan(FALSE) != 0) {
    eloop_event_unregister(WIFI_MGMT_EVENT_SCAN_DONE);
    eloop_event_unregister(WIFI_MGMT_EVENT_SCAN_FAIL);
    printf("start wifi_scan failed\r\n");
    return -1;
}
  return 0;
}
```

## 6.2. 连接 AP

此例中 **wifi_connect_ap** 连接名为“test”，密码为“12345678”的 AP，并监控连接成功、失败、断连的事件。连接成功之后，**cb_connect_done** 中调用 wifi_netlink_store_joined_ap，将保存已连接 AP 的名称与网络密码到 flash。系统重新 boot 之后，SDK 会自动连接已保存的 AP。

```
#include "app_cfg.h"
#include "app_type.h"
#include "bsp_inc.h"
#include "osal_types.h"
#include "wlan_debug.h"
#include "wrapper_os.h"
#include "debug_print.h"
#include "malloc.h"
#include "wifi_netlink.h"
#include "wifi_netif.h"
#include "wifi_management.h"
void cb_connect_done(void *eloop_data, void *user_ctx)
{
    printf("WIFI connect: connect AP \r\n");
    wifi_netlink_auto_conn_set(1);
    wifi_netlink_joined_ap_store();
    eloop_event_unregister(WIFI_MGMT_EVENT_CONNECT_SUCCESS);
```

```
}
void cb_connect_failed(void *eloop_data, void *user_ctx)
{
    printf("WIFI connect: connect AP failed\r\n");
    eloop_event_unregister(WIFI_MGMT_EVENT_CONNECT_FAIL);
}

void cb_disconnect(void *eloop_data, void *user_ctx)
{
    printf("WIFI connect: AP disconnected!\r\n");
    eloop_event_unregister(WIFI_MGMT_EVENT_DISCONNECT);
}
void wifi_connect_ap()
{
    int status = 0;
    uint8 *ssid = "test";
    uint8 *password = "12345678";
    wifi_management_sta_start();
    status = wifi_management_connect(ssid, password, FALSE);
if (status != 0)
    printf("wifi connect failed\r\n");
eloop_event_register(WIFI_MGMT_EVENT_CONNECT_FAIL, cb_connect_failed, NULL, NULL);
eloop_event_register(WIFI_MGMT_EVENT_CONNECT_SUCCESS, cb_connect_done, NULL,
NULL);
eloop_event_register(WIFI_MGMT_EVENT_DISCONNECT, cb_disconnect, NULL, NULL);
}
```

## 6.3. 启动软 AP

此例中 **wifi_start_ap** 启动一个名称为”test”的软 AP，**wifi_get_client** 获取客户端列表。

```
#include "app_cfg.h"
#include "app_type.h"
#include "bsp_inc.h"
#include "osal_types.h"
#include "wlan_debug.h"
#include "wrapper_os.h"
#include "debug_print.h"
#include "malloc.h"
#include "wifi_netlink.h"
#include "wifi_netif.h"
#include "wifi_management.h"
void wifi_get_client()
```

```
{
    uint8_t info[MAX_STATION_NUM * ETH_ALEN];
    uint32_t client_num, i;
    client_num = wifi_management_ap_assoc_info(info);
    for (i = 0; i < client_num; i++) {
      printf("wireless client: [%d] "MAC_FMT"\r\n", i, MAC_ARG(info + i * ETH_ALEN));
    }
}
void wifi_start_ap()
{
    int status = 0;
    uint8 *ssid = "test";
    uint8 *password = "12345678";
    uint8 channel = 1;
    wifi_management_ap_start(ssid, passwd, channel, 0);
}
```

## 6.4. 阿里云接入

本节以阿里云 IOT SDK iotkit-embedded-3.2.0 为例，介绍如何使用上述 Wi-Fi SDK API 适配云上服务。iotkit-embedded-3.2.0 需要适配的 API 大概分为 Wi-Fi 配网、系统、SSL 网络通信三部分，下面分别作简单的介绍。

### 6.4.1. 系统接入

阿里云系统接入包括以下所列函数，可在第 2 章 ***OSAL API*** 组件服务中找到对应 API。

void *HAL_Malloc(uint32_t size);

void HAL_Free(void *ptr);

uint64_t HAL_UptimeMs(void);

void HAL_SleepMs(uint32_t ms);

void HAL_Srandom(uint32_t seed);

int HAL_Snprintf(char *str, const int len, const char *fmt, ...);

int HAL_Vsnprintf(char *str, const int len, const char *format, va_list ap);

void *HAL_SemaphoreCreate(void);

void HAL_SemaphoreDestroy(void *sem);

void HAL_SemaphorePost(void *sem);

int HAL_SemaphoreWait(void *sem, uint32_t timeout_ms);

```
int HAL_ThreadCreate(
                    void **thread_handle,
                    void *(*work_routine)(void *),
                    void *arg,
                   hal_os_thread_param_t *hal_os_thread_param,
                  int *stack_used);
void *HAL_MutexCreate(void);
void HAL_MutexDestroy(void *mutex);
void HAL_MutexLock(void *mutex);
void HAL_MutexUnlock(void *mutex);
```

### 6.4.2. Wi-Fi 配网

阿里云支持的 Wi-Fi 配网方式有多种，从原理上可以分为两类，一类是配网设备发出有编码信息的多播帧或特殊的管理帧，待配网 IOT 设备切换不同信道监听空口的包。当 IOT 设备接收到足够多的编码信息，解析网络名称和密码，就可以连接无线网络。另一类是待配网 IOT 设备开启软 AP，配网设备连接到软 AP 把配网信息告知 IOT 设备，IOT 设备关闭软 AP，连接无线网络。

**表 6-1. 阿里云 SDK 适配接口与 Wi-Fi SDK API 对照表**

| 功能 | 阿里云 SDK 适配接口 | Wi-Fi SDK API |
|---|---|---|
| 设置 Wi-Fi 工作进入监听(Monitor)模式，并在收到 802.11 帧的时候调用被传入的回调函数 | HAL_Awss_Open_Monitor<br>HAL_Awss_Close_Monitor | wifi_netlink_promisc_mode_set |
| 设置 Wi-Fi 切换到指定的信道(channel)上 | HAL_Awss_Switch_Channel | wifi_netlink_channel_set |
| 要求 Wi-Fi 连接指定热点(Access Point)的函数 | HAL_Awss_Connect_Ap | wifi_management_connect |
| Wi-Fi 网络是否已连接网络 | HAL_Sys_Net_Is_Ready | wifi_netif_is_ip_got |
| 在当前信道(channel)上以基本数据速率(1Mbps)发送裸的 802.11 帧(raw 802.11 frame) | HAL_Wifi_Send_80211_Raw_Frame | wifi_netlink_raw_send |
| 在站点(Station)模式 | HAL_Wifi_Enable_Mgmt_Frame_Filter | wifi_netlink_promisc_filter_set |

| 功能 | 阿里云 SDK 适配接口 | Wi-Fi SDK API |
| --- | --- | --- |
| 下使能或禁用对管理帧的过滤 | | wifi_netlink_promisc_mgmt_cb_set |
| 获取所连接的热点(Access Point)的信息 | HAL_Wifi_Get_Ap_Info | wifi_netlink_linked_ap_info_get |
| 打开当前设备热点，并把设备由 Station 模式切换到软 AP 模式 | HAL_Awss_Open_Ap | wifi_management_ap_start |
| 关闭当前设备热点，并把设备由 SoftAP 模式切换到 Station 模式 | HAL_Awss_Close_Ap | wifi_management_sta_start |
| 获取 Wi-Fi 网口的 MAC 地址 | HAL_Wifi_Get_Mac | wifi_netif_get_hwaddr |

### 6.4.3. SSL 网络通信

下列是阿里云需要适配的 SSL 通信接口。Wi-Fi SDK 移植了 Mbedtls2.17.0，在适配阿里云 SSL 接口中直接调用 Mbedtls 的 API。开发者在使用过程中可以参考其官方文档，也可参考 SDK \ NSPE\WIFI_IOT \ cloud \ alicloud \ iotkit-embedded-3.2.0 \ lib_iot_sdk_src \ eng \ wrappers \ wrappers.c.

```
int HAL_SSL_Read(uintptr_t handle, char *buf, int len, int timeout_ms);

int HAL_SSL_Write(uintptr_t handle, const char *buf, int len, int timeout_ms);

int32_t HAL_SSL_Destroy(uintptr_t handle);

uintptr_t HAL_SSL_Establish(const char *host,
                            uint16_t port,
                            const char *ca_crt,
                            uint32_t ca_crt_len);
```

### 6.4.4. 阿里云接入示例

参考 SDK \ NSPE \ WIFI_IOT \ cloud \ alicloud \ linkkit_example_solo.c。

# 7. 版本历史

表 7-1. 版本历史

| 版本号 | 说明 | 日期 |
| --- | --- | --- |
| 1.0 | 首次发布 | 2022 年 12 月 16 日 |

## Important Notice

This document is the property of GigaDevice Semiconductor Inc. and its subsidiaries (the "Company"). This document, including any product of the Company described in this document (the “Product”), is owned by the Company under the intellectual property laws and treaties of the People’s Republic of China and other jurisdictions worldwide. The Company reserves all rights under such laws and treaties and does not grant any license under its patents, copyrights, trademarks, or other intellectual property rights. The names and brands of third party referred thereto (if any) are the property of their respective owner and referred to for identification purposes only.

The Company makes no warranty of any kind, express or implied, with regard to this document or any Product, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The Company does not assume any liability arising out of the application or use of any Product described in this document. Any information provided in this document is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Except for customized products which has been expressly identified in the applicable agreement, the Products are designed, developed, and/or manufactured for ordinary business, industrial, personal, and/or household applications only. The Products are not designed, intended, or authorized for use as components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, atomic energy control instruments, combustion control instruments, airplane or spaceship instruments, transportation instruments, traffic signal instruments, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or Product could cause personal injury, death, property or environmental damage ("Unintended Uses"). Customers shall take any and all actions to ensure using and selling the Products in accordance with the applicable laws and regulations. The Company is not liable, in whole or in part, and customers shall and hereby do release the Company as well as it’s suppliers and/or distributors from any claim, damage, or other liability arising from or related to all Unintended Uses of the Products. Customers shall indemnify and hold the Company as well as it’s suppliers and/or distributors harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of the Products.

Information in this document is provided solely in connection with the Products. The Company reserves the right to make changes, corrections, modifications or improvements to this document and Products and services described herein at any time, without notice.