GigaDevice Semiconductor Inc.

GD32VW553 Wi-Fi 开发指南

应用笔记

AN158

1.1 版本

（2024 年 7 月）

目录

图索引

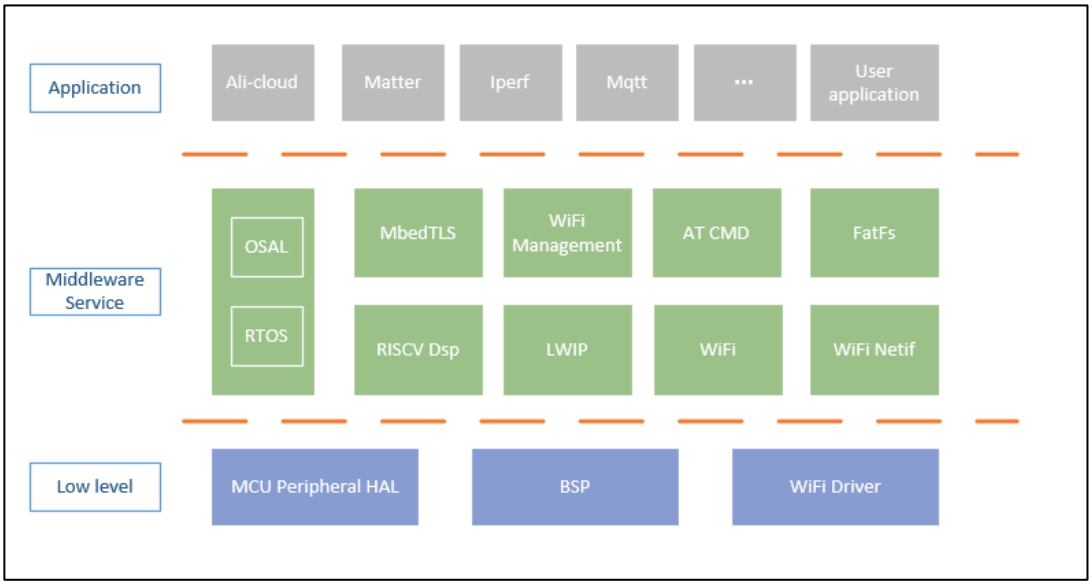
表索引

1. Wi-Fi SDK 概述

GD32VW553 系列芯片是以 RISC-V 为内核的 32 位微控制器（MCU），包含了 Wi-Fi4/ Wi-Fi6 及 BLE5.3 连接技术。GD32VW553 Wi-Fi+BLE SDK 集成 Wi-Fi 驱动、BLE 驱动、LwIP TCP/IP 协议栈、MbedTLS 等组件，可使开发者基于 GD32VW553 快速开发物联网应用程序。本应用指南描述了 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，BLE 相关内容请参考《AN152 GD32VW553 BLE 开发指南》。

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，GD32VW553 Wi-Fi SDK 的软件框架由 Low level、Middleware Service、Application 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 UART、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 Driver 可通过 Middleware Service 层的组件访问。

Middleware Service 层由多个组件构成，为应用提供加密、网络通信等服务。其中 RISC-V Dsp、MbedTLS、LwIP 等是第三方组件，它们的使用可以参考其官方文档。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。本文第 2 章 *OSAL API* 介绍 OSAL 的 API 使用。WiFi 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 3 章 *WiFi Netif API* 介绍 WiFi Netif 的 API 使用。WiFi API 组件是 Wi-Fi 管理相关操作的集合。开发者可以获取或设置 WiFi 相关参数和信息，如 WiFi 状态，WiFi IP 地址等，也可以通

过 WiFi Management 执行扫描无线网络、连接 AP、启动软 AP 等操作。WiFi Management 基于 Netif 和 event loop 实现，采用了状态机和事件管理组件，可以让开发者监控 WiFi Driver 事件的发生，第 4 章 ***WiFi API*** 会介绍其使用方法，开发者可以进行客制化开发。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《GD32VW553 AT 指令用户指南》文档进行开发。

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。

2. OSAL API

头文件 MSDK\rtos\rtos_wrapper\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。

输入参数：size，需要分配内存的长度。

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，将内存初始化为 0。

输入参数：count，分配的个数。

size，需要分配内存的长度。

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。

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_heap_info

原型：void sys_heap_info(int *total_size, int *free_size, int *min_free_size)

功能：获取堆的信息。

输入参数：无。

输出参数：total_size，指向堆总空间大小的指针。

free_size，指向堆空闲的空间大小的指针。

min_free_size，指向堆最小空闲的空间大小的指针。

返回：无。

2.1.9. 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。

2.1.10. 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。

2.1.11. 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：dst，目的内存地址。

返回：无。

2.1.12. 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。

2.1.13. sys_add_heap_region

原型：void sys_add_heap_region(uint32_t ucStartAddress, uint32_t xSizeInBytes)

功能：增加堆区域。

输入参数：ucStartAddress，起始地址。

xSizeInBytes，区域大小，单位 bytes。。

输出参数：无。

返回：无。

2.1.14. sys_remove_heap_region

原型：void sys_remove_heap_region(uint32_t ucStartAddress, uint32_t xSizeInBytes)

功能：移除堆区域。

输入参数：ucStartAddress，起始地址。

xSizeInBytes，区域大小，单位 bytes。。

输出参数：无。

返回：无。

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 queue_item_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。

queue_item_size，消息队列中每个数据的大小。

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_create_dynamic

原型：#define sys_task_create_dynamic(name, stack_size, priority, func, ctx) sys_task_create(NULL, name, NULL, stack_size, 0, 0, priority, func, ctx)

2.2.3. sys_task_name_get

原型：char* sys_task_name_get(void *task)

功能：获取任务名字。

输入参数：task，任务句柄，NULL 则返回当前任务的名字。

输出参数：无。

返回：任务名字。

2.2.4. 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。

2.2.5. 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。

2.2.6. sys_current_task_handle_get

原型：os_task_t sys_current_task_handle_get(void)

功能：获取当前任务的句柄。

输入参数：无。

输出参数：无。

返回：当前任务句柄。

2.2.7. 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。

2.2.8. sys_current_task_stack_depth

原型：int32_t sys_current_task_stack_depth(unsigned long cur_sp)

功能：获取任务栈的栈深度。

输入参数：cur_sp，堆栈指针。

输出参数：无。

返回：任务栈的栈深度。

2.2.9. sys_stack_free_get

原型：uint32_t sys_stack_free_get(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。

2.2.10. sys_task_wait_notification

原型：int sys_task_wait_notification(int timeout)

功能：task 挂起直到收到通知或是超时。

输入参数：timeout，等待通知超时时间。其中，0 表示不等待直接返回，-1 表示一直等待。

输出参数：无。

返回：超时返回 0，其他返回通知值。

2.2.11. sys_task_notify

原型：void sys_task_notify(void *task, bool isr)

功能：给 task 发送通知。

输入参数：task，任务句柄。

isr，指示是否由中断调用。

输出参数：无。

返回：无。

2.2.12. sys_priority_set

原型：void sys_priority_set(void *task, os_prio_t priority)

功能：改变 task 的优先级。

输入参数：task，任务句柄。

priority，待设置的优先级。

输出参数：无。

返回：无。

2.2.13. sys_priority_get

原型：os_prio_t sys_priority_get(void *task)

功能：获取 task 的优先级。

输入参数：task，任务句柄。

输出参数：无。

返回：task 的优先级。

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_ext

原型：int32_t sys_sema_init_ext(os_sema_t *sema, int max_count, int init_count)

功能：创建并初始化信号量。

输入参数：max_count，信号量最大值。

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_init

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

功能：创建并初始化信号量。

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。

2.3.7. 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。

2.3.8. 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。

2.3.9. 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。

2.3.10. 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，0 表示一直等待。

输出参数：无。

返回：0，成功；非 0，失败。

2.3.11. sys_sema_get_count

原型：int sys_sema_get_count(os_sema_t *sema)

功能：获取信号量值。

输入参数：sema，信号量句柄。

输出参数：无。

返回：信号量值。

2.3.12. sys_mutex_init

原型：void sys_mutex_init(os_mutex_t *mutex)

功能：创建互斥锁。

输入参数：无。

输出参数：mutex，互斥锁句柄。

返回：无。

2.3.13. 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。

2.3.14. sys_mutex_try_get

原型：int32_t sys_mutex_try_get(os_mutex_t *mutex, int timeout)

功能：获取互斥锁。

输入参数：mutex，互斥锁句柄。

timeout，等待时间，单位 ms。0 表示不等待，-1 表示一直等待。

输出参数：无。

返回：0，获取锁成功；-1，失败。

2.3.15. sys_mutex_get

原型：int32_t 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：0，获取锁成功；-1，失败。

2.3.16. 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。

2.3.17. 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。

2.3.18. 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。

2.3.19. 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。

2.3.20. sys_queue_post_with_timeout

原型：int32_t sys_queue_post_with_timeout(os_queue_t *queue, void *msg, int32_t timeout_ms)

功能：向队列发送消息，等待超时则返回

输入参数：queue，队列句柄。

msg，消息指针。

timeout_ms，等待超时时间，单位毫秒。

输出参数：无。

返回：0，发送成功；-1，失败。

2.3.21. 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。

2.3.22. sys_queue_is_empty

原型：bool sys_queue_is_empty(os_queue_t *queue)

功能：检测消息队列是否为空。

输入参数：queue，队列句柄。

输出参数：无。

返回：bool 类型，true 表示队列为空，false 表示非空。

2.3.23. sys_queue_cnt

原型：int sys_queue_cnt(os_queue_t *queue)

功能：获取消息队列中排队的消息个数。

输入参数：queue，队列句柄。

输出参数：无。

返回：消息队列中排队的消息个数。

2.3.24. sys_queue_write

原型：int sys_queue_write(os_queue_t *queue, void *msg, int timeout, bool isr)

功能：将消息写入消息队列的最后。

输入参数：queue，队列句柄。

msg，消息指针。

timeout，等待时间。0 表示不等待，-1 表示一直等待。

isr，是否来自 ISR。如果是，忽视 timeout 参数。

输出参数：无。

返回：0，成功；非 0，失败。

2.3.25. sys_queue_read

原型：int sys_queue_read(os_queue_t *queue, void *msg, int timeout, bool isr)

功能：从消息队列中读取消息。

输入参数：queue，队列句柄。

timeout，等待时间。0 表示不等待，-1 表示一直等待。

isr，是否来自 ISR。如果是，忽视 timeout 参数。

输出参数：msg，消息指针。

返回：0，成功；非 0，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 boot up 以来的时间。

输入参数：无。

输出参数：无。

返回：系统 boot up 以来的时间，单位毫秒。

2.4.2. sys_time_get

原型：uint32_t sys_time_get(void *p)

功能：获取系统 boot up 以来的时间。

输入参数：p，实际未使用。

输出参数：无。

返回：系统 boot up 以来的时间，单位毫秒。

2.4.3. sys_ms_sleep

原型：void sys_ms_sleep(int 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。

2.4.4. 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。

2.4.5. 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。

2.4.6. 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。

2.4.7. 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。

2.4.8. 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。

2.4.9. 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。

2.4.10. 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。

2.4.11. sys_os_now

原型：uint32_t sys_os_now(bool isr)

功能：获取当前 RTOS 的时间。

输入参数：isr，是否在 ISR 调用。

输出参数：无。

返回：当前 RTOS 的时间，单位 ticks。

2.4.12. sys_cpu_sleep_time_get

原型：void sys_cpu_sleep_time_get(uint32_t *stats_ms, uint32_t *sleep_ms)

功能：获取 CPU 的统计时间和休眠时间。

输入参数：无。

输出参数：stats_ms，统计时间，单位毫秒。

sleep_ms，休眠时间，单位毫秒。

返回：无。

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。

2.5.8. sys_in_critical

原型：uint32_t sys_in_critical(void)

功能：获取 RTOS 临界嵌套中的中断状态。

输入参数：无。

输出参数：无。

返回：RTOS 临界嵌套中的中断状态。

2.5.9. sys_enter_critical

原型：void sys_enter_critical(void)

功能：RTOS 进入临界态。

输入参数：无。

输出参数：无。

返回：无。

2.5.10. sys_exit_critical

原型：void sys_exit_critical(void)

功能：RTOS 退出临界态。

输入参数：无。

输出参数：无。

返回：无。

2.5.11. sys_ps_set

原型：void sys_ps_set(uint8_t mode)

功能：配置 RTOS 的省电模式。

输入参数：mode，省电模式。0：退出省电模式；1：CPU Deep Sleep 模式。

输出参数：无。

返回：无。

2.5.12. sys_ps_get

原型：uint8_t sys_ps_get(void)

功能：获取当前 RTOS 的省电模式。

输入参数：无。

输出参数：无。

返回：当前 RTOS 的省电模式。

2.5.13. sys_cpu_stats

原型：void sys_cpu_stats(void)

功能：输出每个任务的 CPU 利用率。

输入参数：无。

输出参数：无。

返回：无。

3. WiFi Netif API

MSDK\lwip\lwip-2.1.2\port\wifi_netif.h

3.1. WiFi LwIP 网络接口 API

3.1.1. net_ip_chksum

原型：uint16_t net_ip_chksum(const void *dataptr, int len)

功能：计算数据的校验和。

输入参数：dataptr，指向 buffer 的指针，该 buffer 保存着待计算校验和的数据。

len，dataptr 的长度，单位 bytes。

输出参数：无

返回值：计算得到的校验和。

3.1.2. net_if_add

原型：int net_if_add(void *net_if, const uint8_t *mac_addr, const uint32_t *ipaddr, const uint32_t *netmask, const uint32_t *gw, void *vif_priv)

功能：向 LwIP 注册 WiFi 网络接口。

输入参数：net_if，net_if 结构体指针，指向待注册的网络接口。

mac_addr，指向 MAC 地址的指针。

ipaddr，指向 IPv4 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。

vif_priv，wifi_vif_tag 结构体指针，指向 WiFi VIF。

输出参数：无

返回值：执行成功返回 0，失败返回-1。

3.1.3. net_if_remove

原型：int net_if_remove(void *net_if)

功能：移除 WiFi 网络接口。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

输出参数：无

返回值：执行成功返回 0，失败返回非 0 值。

3.1.4. net_if_get_mac_addr

原型：const uint8_t *net_if_get_mac_addr(void *net_if)

功能：获取 WiFi 网络接口的 MAC 地址。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

输出参数：无

返回值：指向 WiFi 网络接口 MAC 地址的指针。

3.1.5. net_if_find_from_name

原型：void *net_if_find_from_name(const char *name)

功能：通过 WiFi 网络接口的名字获取 WiFi 网络接口。

输入参数：指向 WiFi 网络接口名字的指针。

输出参数：无

返回值：执行成功返回指向网络接口的指针，失败返回 NULL。

3.1.6. net_if_get_name

原型：int net_if_get_name(void *net_if, char *buf, int len)

功能：获取 WiFi 网络接口的名字。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

len，buffer 的长度，该 buffer 用来保存 WiFi 网络接口名字，单位 bytes。

输出参数：buf，指向 buffer 的指针，该 buffer 用来保存 WiFi 网络接口的名字。

返回值：WiFi 网络接口名字的长度，单位 bytes。

3.1.7. net_if_up

原型：void net_if_up(void *net_if)

功能：使能 WiFi 网络接口。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

输出参数：无

返回值：无

3.1.8. net_if_down

原型：void net_if_down(void *net_if)

功能：禁用 WiFi 网络接口。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

输出参数：无

返回值：无。

3.1.9. net_if_input

原型：int net_if_input(net_buf_rx_t *buf, void *net_if, void *addr, uint16_t len, net_buf_free_fn free_fn)

功能：将数据传输到 LWIP。

输入参数：buf，net_buf_rx_t 结构体指针，该结构体用于保存传输到 LWIP 的数据的信息。

net_if，net_if 结构体指针，指向传输数据的 WiFi 网络接口。

addr，指向待传输数据的指针。

len，待传输数据的长度，单位 bytes。

free_fn，数据传输完成后的回调函数，用于释放存储数据的 buffer。

输出参数：无

返回值：执行成功返回 0，失败返回-1。

3.1.10. net_if_vif_info

原型：void *net_if_vif_info(void *net_if)

功能：获取 WiFi 网络接口对应的 WiFi 接口。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

输出参数：无

返回值：指向 WiFi VIF 的指针。

3.1.11. net_buf_tx_alloc

原型：net_buf_tx_t *net_buf_tx_alloc(uint32_t length)

功能：分配一块 buffer，用于保存 TX 的数据。该 buffer 的类型是 PBUF_RAM。

输入参数：length，待 TX 数据的长度，单位 bytes。

输出参数：无

返回值：执行成功返回指向 buffer 的指针，该 buffer 由 net_buf_tx_t 结构体填充；失败返回 NULL。

3.1.12. net_buf_tx_alloc_ref

原型：net_buf_tx_t *net_buf_tx_alloc_ref(uint32_t length)

功能：分配一块 buffer，用于保存 TX 的数据。该 buffer 的类型是 PBUF_REF。

输入参数：length，待 TX 数据的长度，单位 bytes。

输出参数：无

返回值：执行成功返回指向 buffer 的指针，该 buffer 由 net_buf_tx_t 结构体填充；失败返回 NULL。

3.1.13. net_buf_tx_info

原型：void *net_buf_tx_info(net_buf_tx_t *buf, uint16_t *tot_len, int *seg_cnt, uint32_t seg_addr[], uint16_t seg_len[])

功能：从 TX buffer--net_buf_tx_t *buf 中获取信息。

输入参数：net_buf_tx_t 结构体指针，指向 TX buffer。

seg_cnt，预设的 TX buffer 最大可分割片段的数量。

输出参数：tot_len，TX buffer 的总长度，单位 bytes。

seg_cnt，TX buffer 实际可分割片段的数量。

seg_addr[]，保存了每个片段的起始地址。

seg_len[]，保存了每个片段的长度，单位 bytes。

返回值：执行成功返回指向第一个片段的指针，失败返回 NULL。

3.1.14. net_buf_tx_free

原型：void net_buf_tx_free(net_buf_tx_t *buf)

功能：释放 TX buffer。

输入参数：net_buf_tx_t 结构体指针，指向 TX buffer。

输出参数：无。

返回值：无。

3.1.15. net_init

原型：int net_init(void)

功能：初始化 L2 资源。

输入参数：无。

输出参数：无。

返回值：执行成功返回 0，失败返回非 0 值。

3.1.16. net_deinit

原型：void net_deinit(void)

功能：释放 L2 资源。

输入参数：无。

输出参数：无。

返回值：无。

3.1.17. net_l2_socket_create

原型：int net_l2_socket_create(void *net_if, uint16_t ethertype)

功能：为指定包创建一个 L2 (aka ethernet)套接字。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

ethertype，Ethernet type。

输出参数：无。

返回值：执行成功返回套接字描述符，失败返回负值。

3.1.18. net_l2_socket_delete

原型：int net_l2_socket_delete(int sock)

功能：删除 L2 (aka ethernet)套接字。

输入参数：sock，待删除 L2 (aka ethernet)套接字描述符。

输出参数：无。

返回值：执行成功返回 0，失败返回非 0 值。

3.1.19. net_l2_send

原型：int net_l2_send(void *net_if, const uint8_t *data, int data_len, uint16_t ethertype, const uint8_t *dst_addr, bool *ack);

功能：用于发送一个 L2 (aka ethernet)包。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

data，指向待传输数据的指针。

data_len，待传输数据的长度，单位 bytes。

ethertype，待传输数据的 Ethernet type。

dst_addr，指向目的地址的指针。

输出参数：ack，指示发送的状态。

返回值：执行成功返回 0，失败返回-1。

3.1.20. net_if_set_default

原型：void net_if_set_default(void *net_if)

功能：将网络接口设置为默认网络接口。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

输出参数：无。

返回值：无。

3.1.21. net_if_set_ip

原型：void net_if_set_ip(void *net_if, uint32_t ip, uint32_t mask, uint32_t gw)

功能：设置 WiFi 网络接口的 ip 地址、掩码、网关。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

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。

3.1.22. net_if_get_ip

原型：int net_if_get_ip(void *net_if, uint32_t *ip, uint32_t *mask, uint32_t *gw)

功能：获取 WiFi 网络接口的 IP 地址，网络掩码和网关地址。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

输出参数：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。

返回值：执行成功返回 0，失败返回-1。

3.1.23. net_dhcp_start

原型：int net_dhcp_start(void *net_if)

功能：在 WiFi 网络接口上启动 DHCP。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

3.1.24. net_dhcp_stop

原型：void net_dhcp_stop(void *net_if)

功能：在 WiFi 网络接口上停止 DHCP。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

输出参数：无。

返回值：无。

3.1.25. net_dhcp_release

原型：int net_dhcp_release(void *net_if)

功能：在 WiFi 网络接口上释放 DHCP 租约。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

3.1.26. net_dhcp_address_obtained

原型：bool net_dhcp_address_obtained(void *net_if)

功能：检测是否已经通过 DHCP 获取到 IP 地址。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

输出参数：无。

返回值：获取到返回 true(1)，未获取到返回 false (0)。

3.1.27. net_dhcpd_start

原型：int net_dhcpd_start(void *net_if)

功能：在 WiFi 网络接口上启动 DHCPD。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

3.1.28. net_dhcpd_stop

原型：void net_dhcpd_stop(void *net_if)

功能：在 WiFi 网络接口上停止 DHCPD。

输入参数：net_if，net_if 结构体指针，指向 WiFi 网络接口。

输出参数：无。

返回值：无。

3.1.29. net_set_dns

原型：int net_set_dns(uint32_t dns_server)

功能：配置 DNS 服务器 IP 地址（IPv4）。

输入参数：dns_server，DNS 服务器 IPv4 地址。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

3.1.30. net_get_dns

原型：int net_get_dns(uint32_t *dns_server)

功能：获取 DNS 服务器 IP 地址（IPv4）。

输入参数：无。

输出参数：dns_server，指向 DNS 服务器 IPv4 地址的指针。

返回值：执行成功返回 0，失败返回-1。

3.1.31. net_buf_tx_cat

原型：void net_buf_tx_cat(net_buf_tx_t *buf1, net_buf_tx_t *buf2)

功能：连结两个 TX buffer（net_buf_tx_t 类型）。

输入参数：buf1，net_buf_tx_t 结构体指针，指向待连结的 TX buffer。

buf2，net_buf_tx_t 结构体指针，指向待连结的 TX buffer。

输出参数：无。

返回值：无。

3.1.32. net_lpbk_socket_create

原型：int net_lpbk_socket_create(int protocol)

功能：申请一个 loopback socket 套接字。

输入参数：protocol，socket 使用的协议。

输出参数：无。

返回值：成功返回 socket 描述符，失败返回-1。

3.1.33. net_lpbk_socket_bind

原型：int net_lpbk_socket_bind(int sock_recv, uint32_t port)

功能：用于服务端绑定 socket 套接字与网卡信息。

输入参数：sock_recv，socket 描述符。

port，网卡的端口号。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

3.1.34. net_lpbk_socket_connect

原型：int net_lpbk_socket_connect(int sock_send, uint32_t port)

功能：用于客户端绑定 socket 套接字与远端网卡信息。

输入参数：sock_send，socket 描述符。

port，网卡的端口号。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

3.1.35. net_if_use_static_ip

原型：void net_if_use_static_ip(bool static_ip)

功能：指示是否使用静态 IP。

输入参数：static_ip，bool 类型，指示是否使用静态 IP。

输出参数：无。

返回值：无。

3.1.36. net_if_is_static_ip

原型：bool net_if_is_static_ip(void)

功能：检测当前是否已使用静态 IP。

输入参数：无。

输出参数：无。

返回值：bool 类型，true 表示已使用静态 IP，false 表示未使用静态 IP。

4. WiFi API

此章节介绍 WiFi 管理相关 API。

4.1. WiFi 初始化与 task 管理

头文件 MSDK\wifi_manager\wifi_init.h。

4.1.1. wifi_init

原型：int wifi_init(void)

功能：该函数初始化 WiFi pmu 和 WiFi 相关模块。

输入参数：无。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.1.2. wifi_sw_init

原型：int wifi_sw_init(void)

功能：该函数初始化 WiFi 相关模块。

输入参数：无。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.1.3. wifi_sw_deinit

原型：void wifi_sw_deinit(void)

功能：该函数释放 WiFi 相关模块。

输入参数：无。

输出参数：无。

返回值：无。

4.1.4. wifi_task_ready

原型：void wifi_task_ready(enum wifi_task_id task_id)

功能：该函数指示相关 task 已就绪。

输入参数：task_id，task 序号。

输出参数：无。

返回值：无。

4.1.5. wifi_wait_ready

原型：int wifi_wait_ready(void)

功能：该函数用于等待 WiFi 就绪。

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。

4.1.6. wifi_task_terminated

原型：void wifi_task_terminated(enum wifi_task_id task_id)

功能：该函数用于终止 task。

输入参数：task_id，task 序号。

输出参数：无。

返回值：无。

4.1.7. wifi_wait_terminated

原型：int wifi_wait_terminated(enum wifi_task_id task_id)

功能：该函数用于等待 task 终止。

输入参数：task_id，task 序号。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

4.2. WiFi VIF 管理

头文件 MSDK\wifi_manager\wifi_vif.h。

头文件 MSDK\wifi_manager\wifi_net_ip.h。

4.2.1. wifi_vif_init

原型：void wifi_vif_init(int vif_idx, struct mac_addr *base_mac_addr)

功能：该函数用于初始化 WiFi VIF。

输入参数：vif_idx，WiFi VIF 序号。

base_mac_addr，mac_addr 结构体指针，指向 MAC 地址。

输出参数：无。

返回值：无。

4.2.2. wifi_vifs_init

原型：int wifi_vifs_init(struct mac_addr *base_mac_addr)

功能：该函数用于初始化所有 WiFi VIF。

输入参数：base_mac_addr，mac_addr 结构体指针，指向 MAC 地址。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.2.3. wifi_vifs_deinit

原型：void wifi_vifs_deinit(void)

功能：该函数用于释放所有 WiFi VIF。

输入参数：无。

输出参数：无。

返回值：无。

4.2.4. wifi_vif_type_set

原型：int wifi_vif_type_set(int vif_idx, enum wifi_vif_type type)

功能：该函数用于设置 WiFi VIF 的 type。

输入参数：vif_idx，WiFi VIF 序号。

type，待设置的 WiFi VIF 的 type，在枚举 wifi_vif_type 中列出。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

enum wifi_vif_type {

```
    WVIF_UNKNOWN,
    WVIF_STA,
    WVIF_AP,
    WVIF_MONITOR,
};
```

4.2.5. wifi_vif_name

原型：int wifi_vif_name(int vif_idx, char *name, int len)

功能：该函数用于获取 WiFi VIF 的名字。

输入参数：vif_idx，WiFi VIF 序号。

len，buffer 的长度，该 buffer 用来保存 WiFi VIF 名字，单位 bytes。

输出参数：name，指向 buffer 的指针，该 buffer 用来保存 WiFi VIF 名字。

返回值：执行成功返回 WiFi VIF 名字的长度，单位 bytes；失败返回-1。

4.2.6. wifi_vif_reset

原型：void wifi_vif_reset(int vif_idx, enum wifi_vif_type type)

功能：该函数用于重置 WiFi VIF 配置。

输入参数：vif_idx，WiFi VIF 序号。

type，WiFi VIF 的 type。

输出参数：无。

返回值：无。

4.2.7. vif_idx_to_mac_vif

原型：void *vif_idx_to_mac_vif(uint8_t vif_idx)

功能：该函数用于获取 WiFi VIF 对应的 MAC VIF 信息。

输入参数：vif_idx，WiFi VIF 序号。

输出参数：无。

返回值：执行成功返回指向保存 MAC VIF 信息的结构体的指针，失败返回 NULL。

4.2.8. wvif_to_mac_vif

原型：void *wvif_to_mac_vif(void *wvif)

功能：该函数用于获取 WiFi VIF 对应的 MAC VIF 信息。

输入参数：wvif，指向 WiFi VIF。

输出参数：无。

返回值：执行成功返回指向保存 MAC VIF 信息的结构体的指针，失败返回 NULL。

4.2.9. vif_idx_to_net_if

原型：void *vif_idx_to_net_if(uint8_t vif_idx)

功能：该函数用于获取 WiFi VIF 对应的 Netif VIF 信息。

输入参数：vif_idx，WiFi VIF 序号。

输出参数：无。

返回值：执行成功返回指向保存 Netif VIF 信息的结构体的指针，失败返回 NULL。

4.2.10. vif_idx_to_wvif

原型：void *vif_idx_to_wvif(uint8_t vif_idx)

功能：该函数用于获取 WiFi VIF 信息。

输入参数：vif_idx，WiFi VIF 序号。

输出参数：无。

返回值：执行成功返回指向保存 WiFi VIF 信息的结构体的指针，失败返回 NULL。

4.2.11. wvif_to_vif_idx

原型：int wvif_to_vif_idx(void *wvif)

功能：该函数用于获取 WiFi VIF 的序号。

输入参数：wvif，指向 WiFi VIF。

输出参数：无。

返回值：返回 WiFi VIF 的序号。

4.2.12. wifi_vif_sta_uapsd_get

原型：uint8_t wifi_vif_sta_uapsd_get(int vif_idx)

功能：该函数用于获取 Station 模式下 WiFi VIF 的 UAPSD 队列配置。

输入参数：vif_idx，WiFi VIF 序号。

输出参数：无。

返回值：返回 Station 模式下 WiFi VIF 的 UAPSD 队列配置。

4.2.13. wifi_vif_uapsd_queues_set

原型：int wifi_vif_uapsd_queues_set(int vif_idx, uint8_t uapsd_queues)

功能：该函数用于设置 Station 模式下 WiFi VIF 的 UAPSD 队列配置。

输入参数：vif_idx，WiFi VIF 序号。

uapsd_queues，UAPSD 队列配置。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

4.2.14. wifi_vif_mac_addr_get

原型：uint8_t * wifi_vif_mac_addr_get(int vif_idx)

功能：该函数用于获取 WiFi VIF 的 MAC 地址。

输入参数：vif_idx，WiFi VIF 序号。

输出参数：无。

返回值：执行成功返回指向 WiFi VIF 的 MAC 地址的指针，失败返回 NULL。

4.2.15. wifi_vif_mac_vif_set

原型：void wifi_vif_mac_vif_set(int vif_idx, void *mac_vif)

功能：该函数用于将 WiFi VIF 与 MAC VIF 绑定。

输入参数：vif_idx，WiFi VIF 序号。

mac_vif，指向 MAC VIF 的指针。

输出参数：无。

返回值：无。

4.2.16. wifi_vif_is_sta_connecting

原型：int wifi_vif_is_sta_connecting(int vif_idx)

功能：该函数用于判断 Station 模式下 WiFi VIF 是否处于连接阶段。

输入参数：vif_idx，WiFi VIF 序号。

输出参数：无。

返回值：处于连接阶段返回 true，其他情况返回 false。

4.2.17. wifi_vif_is_sta_handshaked

原型：int wifi_vif_is_sta_handshaked(int vif_idx)

功能：该函数用于判断 Station 模式下 WiFi VIF 是否处于握手阶段。

输入参数：vif_idx，WiFi VIF 序号。

输出参数：无。

返回值：处于握手阶段返回 true，其他情况返回 false。

4.2.18. wifi_vif_is_sta_connected

原型：int wifi_vif_is_sta_connected(int vif_idx)

功能：该函数用于判断 Station 模式下 WiFi VIF 是否已连接到某个 AP。

输入参数：vif_idx，WiFi VIF 序号。

输出参数：无。

返回值：已连接返回 true，其他情况返回 false。

4.2.19. wifi_vif_idx_from_name

原型：int wifi_vif_idx_from_name(const char *name)

功能：该函数用于获取 WiFi VIF 的 id。

输入参数：name，指向 WiFi VIF 名字的指针。

输出参数：无。

返回值：执行成功返回 WiFi VIF 的 id，失败返回-1。

4.2.20. wifi_vif_user_addr_set

原型：void wifi_vif_user_addr_set(uint8_t *user_addr)

功能：该函数用于用户设置 WiFi VIF 的 MAC 地址。

输入参数：user_addr，指向 MAC 地址的指针。

输出参数：无。

返回值：无。

4.2.21. wifi_ip_chksum

原型：uint16_t wifi_ip_chksum(const void *dataptr, int len)

功能：用于在 LwIP 中计算校验和。

输入参数：dataptr，计算校验和的数据。

len，数据长度。

输出参数：无。

返回值：返回计算得到的校验和。

4.2.22. wifi_set_vif_ip

原型：int wifi_set_vif_ip(int vif_idx, struct wifi_ip_addr_cfg *cfg)

功能：该函数用于设置 WiFi VIF 的 IP 地址。

输入参数：vif_idx，WiFi VIF 的 id。

cfg，wifi_vif_ip_addr_cfg 结构体指针，该结构体保存了 WiFi VIF 的 IP 地址信息。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

4.2.23. wifi_get_vif_ip

原型：int wifi_get_vif_ip(int vif_idx, struct wifi_ip_addr_cfg *cfg)

功能：该函数用于获取当前 WiFi VIF 的 IP 地址信息。

输入参数：fvif_idx，WiFi VIF 的 id。

输出参数：cfg，wifi_vif_ip_addr_cfg 结构体指针，该结构体保存了 WiFi VIF 的 IP 地址信息。

返回值：执行成功返回 0，失败返回-1。

4.3. WiFi Netlink API

头文件 MSDK\wifi_manager\wifi_netlink.h。

4.3.1. wifi_netlink_wifi_open

原型：int wifi_netlink_wifi_open(void)

功能：该函数用于启动 WiFi 设备。

输入参数：无。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.3.2. wifi_netlink_wifi_close

原型：void wifi_netlink_wifi_close(void)

功能：该函数用于关闭 WiFi 设备。

输入参数：无。

输出参数：无。

返回值：无。

4.3.3. wifi_netlink_dbg_open

原型：int wifi_netlink_dbg_open(void)

功能：该函数用于打开 WiFi 相关 debug log 信息的打印。

输入参数：无。

输出参数：无。

返回值：直接返回 0。

4.3.4. wifi_netlink_dbg_close

原型：int wifi_netlink_dbg_close(void)

功能：该函数用于关闭 WiFi 相关 debug log 信息的打印。

输入参数：无。

输出参数：无。

返回值：直接返回 0。

4.3.5. wifi_netlink_wireless_mode_print

原型：void wifi_netlink_wireless_mode_print(uint32_t wireless_mode)

功能：该函数用于打印 WiFi 无线模式的名称。

输入参数：wireless_mode，WiFi 无线模式。

输出参数：无。

返回值：无。

4.3.6. wifi_netlink_status_print

原型：int wifi_netlink_status_print(void)

功能：该函数用于打印当前开发板的 WiFi 状态。

输入参数：无。

输出参数：无。

返回值：直接返回 0。

4.3.7. wifi_netlink_scan_set

原型：int wifi_netlink_scan_set(int vif_idx, uint8_t channel)

功能：该函数用于配置并启动 WiFi 扫描。

输入参数：vif_idx，WiFi VIF 序号。

channel，待扫描的信道，0xFF 表示全 channel。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.3.8. wifi_netlink_scan_set_with_ssid

原型：int wifi_netlink_scan_set_with_ssid(int vif_idx, char *ssid, uint8_t channel)

功能：该函数用于配置并启动 WiFi 扫描，扫描指定的 AP。

输入参数：vif_idx，WiFi VIF 序号。

ssid，指向指定 AP 的 ssid，不能为 NULL。

channel，待扫描的信道，0xFF 表示全 channel。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.3.9. wifi_netlink_scan_results_get

原型：int wifi_netlink_scan_results_get(int vif_idx, struct macif_scan_results *results)

功能：该函数用于获取 WiFi 扫描的结果。

输入参数：vif_idx，WiFi VIF 序号。

输出参数：results，macif_scan_results 结构体指针，保存 WiFi 扫描结果。

返回值：执行成功返回 0，失败返回其他。

4.3.10. wifi_netlink_scan_result_print

原型：void wifi_netlink_scan_result_print(int idx, struct mac_scan_result *result)

功能：该函数用于将 WiFi 扫描结果打印出来。

输入参数：idx，扫描到的 AP 序号。

result，macif_scan_results 结构体指针，保存了 WiFi 扫描结果。

输出参数：无。

返回值：无。

4.3.11. wifi_netlink_scan_results_print

原型：int wifi_netlink_scan_results_print(int vif_idx, void (*callback)(int, struct mac_scan_result *))

功能：该函数用于将 WiFi 扫描的全部结果打印出来。

输入参数：vif_idx，WiFi VIF 序号。

callback，打印 WiFi 扫描结果的回调函数。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.3.12. wifi_netlink_candidate_ap_find

原型：int wifi_netlink_candidate_ap_find(int vif_idx, uint8_t *bssid, char *ssid, struct mac_scan_result *candidate)

功能：该函数用于在 WiFi 扫描结果中寻找指定 AP。

输入参数：vif_idx，WiFi VIF 序号。

bssid，指定 AP 的 bssid。

ssid，指定 AP 的 ssid。

candidate，macif_scan_results 结构体指针，保存了 WiFi 扫描结果。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

bssid 与 ssid 不可同时为 NULL；当两者均不为 NULL 时，以 bssid 为准。

4.3.13. wifi_netlink_connect_req

原型：int wifi_netlink_connect_req(int vif_idx, struct sta_cfg *cfg)

功能：该函数用于 Station 模式下 WiFi VIF 执行连接 AP 操作。

输入参数：vif_idx，WiFi VIF 序号。

cfg，sta_cfg 结构体指针，保存了待连接 AP 的信息。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.3.14. wifi_netlink_associate_done

原型：int wifi_netlink_associate_done(int vif_idx, void *ind_param)

功能：该函数用于指示 Station 模式下 WiFi VIF 已完成 associate 阶段。

输入参数：vif_idx，WiFi VIF 序号。

ind_param，连接信息。

输出参数：无。

返回值：直接返回 0。

4.3.15. wifi_netlink_dhcp_done

原型：int wifi_netlink_dhcp_done(int vif_idx)

功能：该函数用于指示 Station 模式下 WiFi VIF 已完成 DHCP。

输入参数：vif_idx，WiFi VIF 序号。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

4.3.16. wifi_netlink_disconnect_req

原型：int wifi_netlink_disconnect_req(int vif_idx)

功能：该函数用于 Station 模式下 WiFi VIF 执行断开连接 AP 操作。

输入参数：vif_idx，WiFi VIF 序号。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

4.3.17. 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 WiFi 是否使能自动连接。

输入参数：auto_conn_enable，0：禁用，1：使能。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.3.18. wifi_netlink_auto_conn_get

原型：uint8_t wifi_netlink_auto_conn_get(void)

功能：该函数用于获取 WiFi 自动连接是否使能。

输入参数：无。

输出参数：无。

返回值：0：禁用，1：使能。

4.3.19. wifi_netlink_joined_ap_store

原型：int wifi_netlink_joined_ap_store(struct sta_cfg *cfg, uint32_t ip)

功能：该函数用于使能自动连接后，保存 WiFi 已连接的 AP 信息。

输入参数：cfg，sta_cfg 结构体指针，保存了已连接 AP 的信息。

ip，对应已连接 AP 的 IP 地址。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.3.20. wifi_netlink_joined_ap_load

原型：int wifi_netlink_joined_ap_load(int vif_idx)

功能：该函数用于获取使能自动连接后保存的 WiFi 已连接的 AP 信息。

输入参数：vif_idx，WiFi VIF 序号。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.3.21. wifi_netlink_ps_mode_set

原型：int wifi_netlink_ps_mode_set(int vif_idx, uint8_t ps_mode)

功能：该函数用于设置 WiFi 省电模式。

输入参数：vif_idx，WiFi VIF 序号。

ps_mode，0 表示禁用；1 表示 Normal mode，2 表示 Dynamic mode。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

4.3.22. wifi_netlink_ap_start

原型：int wifi_netlink_ap_start(int vif_idx, struct ap_cfg *cfg)

功能：该函数用于 WiFi VIF 启动 softap 模式。

输入参数：vif_idx，WiFi VIF 序号。

cfg，ap_cfg 结构体指针，保存了待启动 softap 配置信息。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.3.23. wifi_netlink_ap_stop

原型：int wifi_netlink_ap_stop(int vif_idx)

功能：该函数用于 WiFi VIF 终止 softap 模式。

输入参数：vif_idx，WiFi VIF 序号。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

4.3.24. wifi_netlink_channel_set

原型：int wifi_netlink_channel_set(uint32_t channel)

功能：该函数用于设置 WiFi VIF 信道。

输入参数：channel，信道序号。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

4.3.25. wifi_netlink_monitor_start

原型：int wifi_netlink_monitor_start(int vif_idx, struct wifi_monitor *cfg)

功能：该函数用于 WiFi VIF 启动 MONITOR 模式。

输入参数：vif_idx，WiFi VIF 序号。

cfg，wifi_monitor 结构体指针，保存了 MONITOR 模式配置信息。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.3.26. wifi_netlink_twt_setup

原型：int wifi_netlink_twt_setup(int vif_idx, struct macif_twt_setup_t *param)

功能：该函数用于 WiFi VIF 配置并建立 TWT 连接，使用前需要使能 TWT。

输入参数：vif_idx，WiFi VIF 序号。

param，macif_twt_setup_t 结构体指针，保存了 TWT 配置信息。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

4.3.27. wifi_netlink_twt_teardown

原型：int wifi_netlink_twt_teardown(int vif_idx, uint8_t id, uint8_t neg_type)

功能：该函数用于 WiFi VIF 删除 TWT 连接，使用前需要使能 TWT。

输入参数：vif_idx，WiFi VIF 序号。

id，TWT 连接 ID。

neg_type，TWT Negotiation 类型。

输出参数：无。

返回值：执行成功返回 0，失败返回-1。

4.3.28. wifi_netlink_fix_rate_set

原型：int wifi_netlink_fix_rate_set(int sta_idx, int fixed_rate_idx)

功能：该函数用于 WiFi VIF 设置 fixed rate。

输入参数：sta_idx，station 序号。

fixed_rate_idx，rate 序号。

输出参数：无。

返回值：执行成功返回 0，失败返回 1。

4.3.29. wifi_netlink_sys_stats_get

原型：int wifi_netlink_sys_stats_get(uint32_t *doze_time, uint32_t *stats_time)

功能：该函数用于获取 WiFi doze 状态统计信息。

输入参数：无。

输出参数：doze_time，WiFi doze 时间，单位 ms。

stats_time，WiFi 统计时间，单位 ms。

返回值：执行成功返回 0，失败返回其他。

4.3.30. wifi_netlink_roaming_rssi_set

原型：int wifi_netlink_roaming_rssi_set(int vif_idx, int8_t rssi_thresh)

功能：该函数用于 WiFi VIF 设置 roaming RSSI 阈值。

输入参数：vif_idx，WiFi VIF 序号。

rssi_thresh，RSSI 阈值。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.3.31. wifi_netlink_roaming_rssi_get

原型：int8_t wifi_netlink_roaming_rssi_get(int vif_idx)

功能：该函数用于获取 roaming RSSI 阈值。

输入参数：vif_idx，WiFi VIF 序号。

输出参数：无。

返回值：执行成功返回 RSSI 阈值，失败返回 0。

4.3.32. wifi_netlink_wps_pbc

原型：int wifi_netlink_wps_pbc(int vif_idx)

功能：该函数用于启动 WPS PBC 模式连接。

输入参数：vif_idx，WiFi VIF 序号。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.3.33. wifi_netlink_wps_pin

原型：int wifi_netlink_wps_pin(int vif_idx, char *pin)

功能：该函数用于启动 WPS PIN 模式连接。

输入参数：vif_idx，WiFi VIF 序号。

pin，PIN 码。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.3.34. wifi_netlink_listen_interval_set

原型：int wifi_netlink_listen_interval_set(uint8_t interval)

功能：该函数用于设置低功耗模式下硬件监听 beacon 帧的间隔。

输入参数：interval，监听 beacon 帧的间隔。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.4. WiFi 连接管理

此节介绍 WiFi Management 连接管理 API，头文件 MSDK\wifi_manager\ wifi_management.h。

4.4.1. wifi_management_init

原型：int wifi_management_init(void)

功能：初始化 LwIP，WiFi event loop 等，只需要调用一次。

输入参数：无。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.4.2. wifi_management_deinit

原型：void wifi_management_deinit(void)

功能：终止 WiFi event loop 与 WiFi Management task。

输入参数：无。

输出参数：无。

返回值：无。

4.4.3. wifi_management_scan

原型：int wifi_management_scan(uint8_t blocked, const uint8_t *ssid)

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。

ssid，NULL 或者指向指定 ssid 的指针。

输出参数：无。

返回值：启动扫描成功返回 0，启动扫描失败返回其他。

4.4.4. 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：执行成功返回 0，失败返回其他。

4.4.5. wifi_management_connect_with_bssid

原型：int wifi_management_connect_with_bssid(uint8_t *bssid, char *password, uint8_t blocked)

功能：启动连接 AP。

输入参数：bssid，AP 的 bssid。

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：执行成功返回 0，失败返回其他。

4.4.6. wifi_management_connect_with_eap_tls

原型：int wifi_management_connect_with_eap_tls(char *ssid, const char *identity, const char *ca_cert, const char *client_key, const char *client_cert, const char *client_key_password, uint8_t blocked)

功能：使用 EAP-TLS 认证连接企业级 AP。

输入参数：ssid，AP 的 ssid。

identity，用户的 ID。

ca_cert，根证书。

client_key，客户端密钥。

client_cert，客户端证书。

client_key_password，客户端证书口令。

blocked，1：阻塞其他操作，0：不阻塞。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.4.7. 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：执行成功返回 0，失败返回其他。

4.4.8. wifi_management_ap_start

原型：int wifi_management_ap_start(char *ssid, char *passwd, uint32_t channel, wifi_ap_auth_mode_t auth_mode, 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。

passwd，软 AP 网络密码。当为字符串”NULL”时表示启动一个 OPEN 软 AP。

channel，软 AP 所在的网络信道，1-13。

auth_mode，软 AP 加密方式，默认加密方式 WPA2-PSK。

hidden，是否隐藏 ssid。0：广播 ssid，1：隐藏 ssid。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.4.9. wifi_management_ap_stop

原型：int wifi_management_ap_stop(void)

功能：停止软 AP，SDK 退出软 AP mode。

输入参数：无。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.4.10. wifi_management_concurrent_set

原型：int wifi_management_concurrent_set(int enable)

功能：控制 SDK 进入或退出 WiFi concurrent 模式。

输入参数：enable，0：退出 WiFi concurrent 模式；非 0：进入 WiFi concurrent 模式。

输出参数：无。

返回值：执行成功返回 0。

4.4.11. wifi_management_concurrent_get

原型：int wifi_management_concurrent_get(void)

功能：获取当前 WiFi concurrent 模式。

输入参数：无。

输出参数：无。

返回值：返回当前 WiFi concurrent 模式。

4.4.12. wifi_management_sta_start

原型：int wifi_management_sta_start(void)

功能：SDK 进入 STA mode。

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。

4.4.13. wifi_management_monitor_start

原型：int wifi_management_monitor_start(uint8_t channel, cb_macif_rx monitor_cb)

功能：SDK 进入 MONITOR mode。

输入参数：channel，MONITOR mode 下监听的信道。

monitor_cb，MONITOR mode 下收到 packet 时的回调函数。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.4.14. wifi_management_roaming_set

原型：int wifi_management_roaming_set(uint8_t enable, int8_t rssi_th)

功能：设置是否使能 WiFi roaming 机制。

输入参数：enable，1：使能；0：不使能。

rssi_th，使能 WiFi roaming 机制下的 RSSI 检测阈值。

输出参数：无。

返回值：直接返回 0。

4.4.15. wifi_management_roaming_get

原型：int wifi_management_roaming_get(int8_t *rssi_th)

功能：获取 roaming RSSI 阈值及当前是否使能 WiFi roaming 机制。

输入参数：无。

输出参数：rssi_th，RSSI 阈值。

返回值：是否使能 WiFi roaming 机制，1：使能；0：不使能。

4.4.16. wifi_management_wps_start

原型：int wifi_management_wps_start(bool is_pbc, char *pin, uint8_t blocked)

功能：启动 WPS 模式连接。

输入参数：is_pbc，bool 类型。true 表示使用 PBC 模式；false 表示使用 PIN 模式。

pin，PIN 码，只有 PIN 模式下有效，此时不能为 NULL。

blocked，1：阻塞其他操作，0：不阻塞。

输出参数：无。

返回值：执行成功返回 0，失败返回其他。

4.5. WiFi event loop API

此节介绍 event loop 组件 API，头文件 MSDK\wifi_manager\wifi_eloop.h。

4.5.1. 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。

4.5.2. 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。

4.5.3. wifi_eloop_init

原型：int wifi_eloop_init(void)

功能：该函数初始化一个全局事件用于处理循环数据。

输入参数：无。

输出参数：无

返回值：直接返回 0。

4.5.4. eloop_event_register

原型：int eloop_event_register(eloop_event_id_t event_id,

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_id_t event_id，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。

4.5.5. eloop_event_unregister

原型：void eloop_event_unregister(eloop_event_id_t event_id)

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_id，中止处理的事件。

输出参数；无。

返回值：无。

4.5.6. eloop_event_send

原型：int eloop_event_send(uint8_t vif_idx, uint16_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：vif_idx，WiFi VIF 序号。

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。

4.5.7. eloop_message_send

原型：int eloop_message_send(uint8_t vif_idx, uint16_t event, int reason, uint8_t *param, uint32_t len)

功能：该函数用于将消息发送到待处理队列中。

输入参数：vif_idx，WiFi VIF 序号。

event，消息中的事件。

reason，消息中的原由。

param，消息处理需要的参数。

len，param 的长度。

输出参数：无。

返回值：执行成功返回 0；失败返回-1。

4.5.8. 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。

4.5.9. 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。

4.5.10. eloop_timeout_is_registered

原型：int eloop_timeout_is_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。

4.5.11. wifi_eloop_run

原型：void wifi_eloop_run(void)

功能：该函数用于启动 event 循环，处理队列中的 event 或 message。

输入参数：无。

输出参数：无。

返回值：无。

4.5.12. wifi_eloop_terminate

原型：void wifi_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。

4.5.13. wifi_eloop_destroy

原型：void wifi_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。

4.5.14. wifi_eloop_terminated

原型：int wifi_eloop_terminated (void)

功能：该函数用于检测事件循环是否终止。

输入参数：无。

输出参数：无。

返回值：终止返回 1；未终止返回 0。

4.6. WiFi 管理相关宏

4.6.1. WiFi 管理事件类型

表 4-1. WiFi 管理事件类型

```
Typedef enum {
    WIFI_MGMT_EVENT_START = ELOOP_EVENT_MAX,

      /* For both STA and SoftAP */
    WIFI_MGMT_EVENT_INIT, //5
    WIFI_MGMT_EVENT_SWITCH_MODE_CMD,
    WIFI_MGMT_EVENT_RX_MGMT,
    WIFI_MGMT_EVENT_RX_EAPOL,
```

```
    /* For STA only */
    WIFI_MGMT_EVENT_SCAN_CMD,
    WIFI_MGMT_EVENT_CONNECT_CMD,    //10
    WIFI_MGMT_EVENT_DISCONNECT_CMD,
    WIFI_MGMT_EVENT_AUTO_CONNECT_CMD,

    WIFI_MGMT_EVENT_SCAN_DONE,
    WIFI_MGMT_EVENT_SCAN_FAIL,
    WIFI_MGMT_EVENT_SCAN_RESULT,    //15

    WIFI_MGMT_EVENT_EXTERNAL_AUTH_REQUIRED,    //16

    WIFI_MGMT_EVENT_ASSOC_SUCCESS, //17

    WIFI_MGMT_EVENT_DHCP_START,
    WIFI_MGMT_EVENT_DHCP_SUCCESS,
    WIFI_MGMT_EVENT_DHCP_FAIL, //20

    WIFI_MGMT_EVENT_CONNECT_SUCCESS,
    WIFI_MGMT_EVENT_CONNECT_FAIL,

    WIFI_MGMT_EVENT_DISCONNECT,
    WIFI_MGMT_EVENT_ROAMING_START,

    /* For SoftAP only */
    WIFI_MGMT_EVENT_START_AP_CMD,    //25
    WIFI_MGMT_EVENT_STOP_AP_CMD,
    WIFI_MGMT_EVENT_AP_SWITCH_CHNL_CMD,

    WIFI_MGMT_EVENT_TX_MGMT_DONE, //28
    WIFI_MGMT_EVENT_CLIENT_ADDED,
    WIFI_MGMT_EVENT_CLIENT_REMOVED, //30

    /* For Monitor only */
    WIFI_MGMT_EVENT_MONITOR_START_CMD,

    WIFI_MGMT_EVENT_MAX,
    WIFI_MGMT_EVENT_NUM = WIFI_MGMT_EVENT_MAX - WIFI_MGMT_EVENT_START - 1,

} wifi_management_event_t;
```

4.6.2. WiFi 管理配置宏

```
WIFI_MGMT_ROAMING_RETRY_LIMIT          // WiFi 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 WiFi 连接质量的间隔
```

5. 应用举例

在 SDK 启动完成之后，开发者就可以使用组件进行 WiFi 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。

5.1. 扫描无线网络

5.1.1. 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。

表 5-1. 阻塞模式扫描示例代码

```
#include "mac_types.h"
#include "wifi_management.h"

int scan_wireless_network(int argc, char **argv)

{

    uint8_t *ssid = NULL;

    if (wifi_management_scan(true, ssid) == -1) {

        return -1;

    }

    wifi_netlink_scan_results_print(WIFI_VIF_INDEX_DEFAULT, wifi_netlink_scan_result_print);

    return 0;

}
```

5.1.2. 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。

表 5-2. 非阻塞式扫描代码示例

```
#include "mac_types.h"

#include "wifi_management.h"

void cb_scan_done(void *eloop_data, void *user_ctx)

{

    app_print("WIFI_SCAN: done\r\n");
```

```
    wifi_netlink_scan_results_print(WIFI_VIF_INDEX_DEFAULT, wifi_netlink_scan_result_print);
    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, ssid) == -1) {
        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;
}
```

5.2. 连接 AP

此例中 **wifi_connect_ap** 连接名为“test”，密码为“12345678”的 AP。

表 5-3. 连接 AP 代码示例

```
#include "wifi_management.h"

void wifi_connect_ap(void)
```

```
{
    int status = 0;
    uint8_t *ssid = "test";
    uint8_t *password = "12345678";
    status = wifi_management_connect(ssid, password, true);
    if (status != 0) {
        printf("wifi connect failed\r\n");
    }
}
```

5.3. 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。

表 5-4. 启动软 AP 代码示例

```
#include "mac_types.h"
#include "debug_print.h"
#include "dhcpd.h"
#include "macif_vif.h"
#include "wifi_management.h"
void wifi_get_client()
{
    struct mac_addr cli_mac[CFG_STA_NUM];
    int cli_num;
    int j;
    struct co_list_hdr *cli_list_hdr;
    struct mac_addr *cli_mac;
    cli_num = macif_vif_ap_assoc_info_get(i, (uint16_t *)&cli_mac);
    for (j = 0; j < cli_num; j++) {
        printf("\t Client[%d]: "MAC_FMT" "IP_FMT"\r\n", j, MAC_ARG(cli_mac[j].array),
IP_ARG(dhcpd_find_ipaddr_by_macaddr((uint8_t *)cli_mac->array)));
```

```
    }
}
void wifi_start_ap()
{
    char *ssid = "test";
    char *password = "12345678";
    uint32_t channel = 1;
    char *akm = "wpa2";
    uint32_t is_hidden = 0;
    if (wifi_management_ap_start(ssid, password, channel, akm, is_hidden)) {
        printf("Failed to start AP, check your configuration.\r\n");
    }
}
```

5.4. BLE 配网

BLE 配网例程请参考《AN152 GD32VW553 BLE 开发指南》。

5.5. 阿里云接入

本节以阿里云 ali-smartliving-device-sdk-c-rel_1.6.6 为例，介绍如何使用上述 WiFi SDK API 适配云上服务。ali-smartliving-device-sdk-c-rel_1.6.6 需要适配的 API 大概分为系统接入、WiFi 配网、SSL 网络通信和 OTA 固件升级四部分，下面分别作简单的介绍。

5.5.1. 系统接入

阿里云系统接入包括以下所列函数。

表 5-5. 系统接入函数示例

```
void *HAL_Malloc(uint32_t size);
void HAL_Free(void *ptr);
uint64_t HAL_UptimeMs(void);
void HAL_SleepMs(uint32_t ms);
uint32_t HAL_Random(uint32_t region);
void HAL_Srandom(uint32_t seed);
void HAL_Printf(const char *fmt, ...);
```

```
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_Reboot();
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_ThreadDelete(_IN_ void *thread_handle);
void *HAL_MutexCreate(void);
void HAL_MutexDestroy(void *mutex);
void HAL_MutexLock(void *mutex);
void HAL_MutexUnlock(void *mutex);
void HAL_UTC_Set(long long ms);
long long HAL_UTC_Get(void);
void *HAL_Timer_Create_Ex(const char *name, void (*func)(void *),
void *user_data, char repeat);
void *HAL_Timer_Create(const char *name, void (*func)(void *), void *user_data);
int HAL_Timer_Delete(void *timer);
int HAL_Timer_Start(void *timer, int ms);
int HAL_Timer_Stop(void *timer);
```

5.5.2. Wi-Fi 配网

阿里云支持的 WiFi 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 5-6. 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 HAL_Awss_Close_Monitor	wifi_management_monitor_start
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_get_vif_ip

功能	阿里云 SDK 适配接口	Wi-Fi SDK API
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_send_80211_frame
获取所连接的热点(Access Point)的信息	HAL_Wifi_Get_Ap_Info	macif_vif_status_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	HAL_Awss_Close_Ap	wifi_management_ap_stop
获取 Wi-Fi 网口的 MAC 地址	HAL_Wifi_Get_Mac	wifi_vif_mac_addr_get

5.5.3. 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阿里云官方文档，参见 https://help.aliyun.com/product/123207.html， 也可参考 SDK\MSDK\cloud\alicloud\src\ref-impl\hal\os\freertos\hal_tls_gd.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);

其中适配阿里云 SDK 的 TCP 和 UDP 网络通信的 API，可参考 SDK\MSDK\cloud\alicloud\src\ref-impl\hal\os\freertos\hal_tcp_gd.c 和 SDK\MSDK\cloud\alicloud\src\ref-impl\hal\os\freertos\hal_udp_gd.c。

5.5.4. OTA 固件升级

阿里云 SDK 支持通过云端对接入云的设备进行固件升级。适配阿里云 OTA 功能的 API 如下：

void HAL_Firmware_Persistence_Start(void);

int HAL_Firmware_Persistence_Stop(void);

int HAL_Firmware_Persistence_Write(char *buffer, uint32_t length);

5.5.5. 阿里云接入示例

参考 SDK\ MSDK\cloud\alicloud\examples\linkkit\living_platform\living_platform_main.c。

6. 版本历史

表 6-1. 版本历史

版本号.	说明	日期
1.0	首次发布	2023 年 10 月 17 日
1.1	增加 roaming 机制相关 api；增加 WiFi 信息获取 api；调整 os api；更新阿里云相关 api。	2024 年 7 月 12 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.