GigaDevice Semiconductor Inc.

GD32VW553 Wi-Fi Development Guide

Application Note
AN158

Revision 1.2

(Mar 2025)

Table of Contents

List of Figures

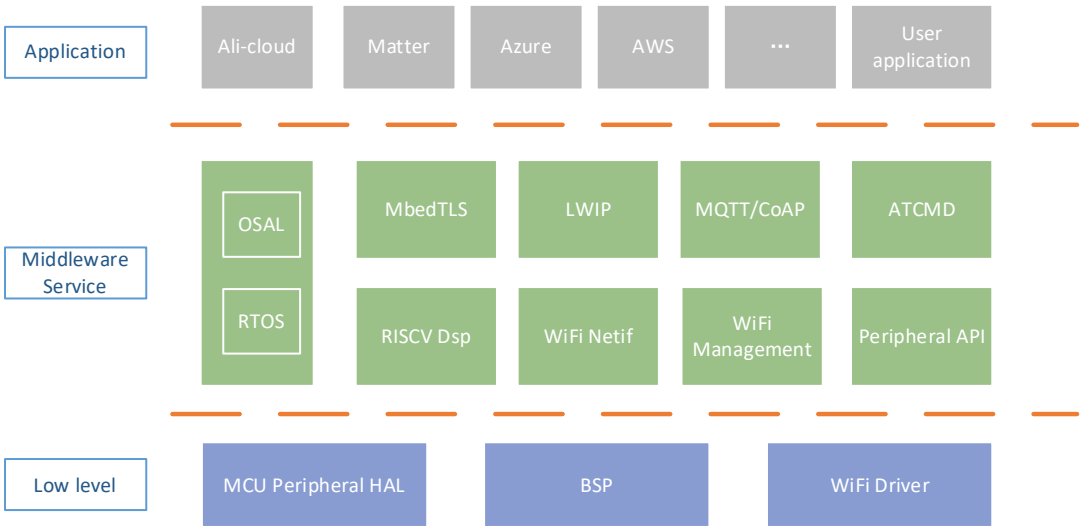
List of Tables

1. Overview of Wi-Fi SDK

The GD32VW553 series chip is a 32-bit microcontroller (MCU) with RISC-V as the core, which contains Wi-Fi4/Wi-Fi6 and BLE5.3 connection technologies. GD32VW553 Wi-Fi+BLE SDK integrates the Wi-Fi driver, BLE driver, LwIP TCP/IP protocol stack, MbedTLS, and other components, allowing developers to quickly develop IoT applications based on GD32VW553. This document describes the SDK framework, boot process, Wi-Fi, and related component application interfaces, which are designed to help developers become familiar with the SDK and use the APIs to develop their own applications. For BLE related information, please refer to the "AN152 GD32VW553 BLE Development Guide".

1.1. Wi-Fi SDK software framework

Figure 1-1. Wi-Fi SDK framework



As shown in ***Figure 1-1. Wi-Fi SDK framework***, the software framework of GD32VW553 Wi-Fi SDK consists of three layers: Low level, Middleware Service, and Application.

The Low level layer is close to the hardware and can directly perform operations related to hardware peripherals, including the peripheral hardware abstraction layer (HAL) of MCU, board support package (BSP), and Wi-Fi Driver. Developers can operate peripherals of MCU such as UART, I2C, and SPI through the HAL, while the BSP can perform board-level initialization, enable PMU, enable hardware encryption engine, and perform other operations. The Wi-Fi Driver is accessible through components in the Middleware Service layer.

The Middleware Service layer consists of multiple components, and provides applications with encryption, network communication, and other services. Components such as RISC-V Dsp, MbedTLS, and LwIP are third-party components. For the usage of such components, please refer to their official documents. The operating system abstraction layer (OSAL) is a package of RTOS kernel functions, and developers can operate RTOS through the OSAL. Thanks to the OSAL, developers can choose their own RTOS as needed without affecting

applications and other components. ***OSAL API*** introduces how to use the APIs of OSAL. The Wi-Fi Netif component is a package based on LwIP and a collection of network interface operations for Wi-Fi devices. Developers can set the network addresses of network interfaces and get the network addresses, gateways, and other information of the interfaces. ***Wi-Fi Netif API*** introduces how to use the APIs of Wi-Fi Netif. The Wi-Fi API component is a collection of Wi-Fi management related operations. Developers can get or set Wi-Fi related parameters and information, such as Wi-Fi status and Wi-Fi IP address, and can also perform operations such as scanning wireless network, connecting to AP, and starting SoftAP through Wi-Fi Management. Wi-Fi Management is implemented based on Netif and event loop, and uses state machines and event management components, which allow developers to monitor the occurrence of Wi-Fi Driver events. ***Wi-Fi API*** introduces how to use it, and developers can carry out customized development. The AT CMD component is a collection of AT commands and is suitable for developers who are familiar with AT commands. Refer to the document "GD32VW553 AT Command User Guide" for development.

The Application layer is a collection of multiple applications, such as Ali-cloud, a distribution network and cloud service program based on Alibaba Cloud iotkit, the performance testing program iperf3, and developer-defined applications.

2. OSAL API

The header file is MSDK\rtos\rtos_wrapper\wrapper_os.h.

2.1. Memory management

2.1.1. sys_malloc

Prototype: void *sys_malloc(size_t size)

Function: Allocate memory whose length is size.

Input parameter: size, the length of memory to be allocated.

Output parameter: None.

Return value: A pointer to the allocated memory block upon success, and NULL upon failure.

2.1.2. sys_calloc

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

Function: Allocate count (representing a number) contiguous memories whose length is size and initialize the memories to 0.

Input parameter: count, the number of memories to be allocated.

size, the length of memories to be allocated.

Output parameter: None.

Return value: A pointer to the allocated memory block upon success, and NULL upon failure.

2.1.3. sys_mfree

Prototype: void sys_mfree(void *ptr)

Function: Release the memory block.

Input parameter: ptr, a pointer to the memory to be released.

Output parameter: None.

Return value: None.

2.1.4. sys_realloc

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

Function: Expand the allocated memory.

Input parameter: mem, a pointer to the memory to be expanded.

size, the size of the new memory block.

Output parameter: None.

Return value: A pointer to the allocated memory block upon success, and NULL upon failure.

2.1.5. sys_free_heap_size

Prototype: int32_t sys_free_heap_size(void)

Function: Get the free size of the heap.

Input parameter: None.

Output parameter: None.

Return value: The free space size of the heap.

2.1.6. sys_min_free_heap_size

Prototype: int32_t sys_min_free_heap_size(void)

Function: Get the minimum free size of the heap.

Input parameter: None.

Output parameter: None.

Return value: The minimum free space size of the heap.

2.1.7. sys_heap_block_size

Prototype: uint16_t sys_heap_block_size(void)

Function: Get the block size of the heap.

Input parameter: None.

Output parameter: None.

Return value: The block size of the heap.

2.1.8. sys_heap_info

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

Function: Get heap information.

Input parameter: None.

Output parameter: total_size, a pointer to the total space size of the heap.

free_size, a pointer to the free space size of the heap.

min_free_size, a pointer to the minimum free space size of the heap.

Return value: None.

2.1.9. sys_memset

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

Function: Initialize the memory block.

Input parameter: s, the address of the memory block to be initialized.

c, initialization content.

count, the size of the memory block.

Output parameter: None.

Return value: None.

2.1.10. sys_memcpy

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

Function: Copy the memory.

Input parameter: src, the source memory address.

n, the length to be copied.

Output parameter: dst, the destination memory address.

Return value: None.

2.1.11. sys_memmove

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

Function: Migrate the memory.

Input parameter: src, the source memory address.

n, the length to be migrated.

Output parameter: dst, the destination memory address.

Return value: None.

2.1.12. sys_memcmp

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

Function: Compare two memory values to check whether they are the same.

Input parameter: buf1, memory address 1 to be compared.

buf2, memory address 2 to be compared.

count, length.

Output parameter: None.

Return value: 0 if they are the same; non-0 if they are different.

2.1.13. sys_add_heap_region

Prototype: void sys_add_heap_region(uint32_t ucStartAddress, uint32_t xSizeInBytes)

Function: Increase the heap area.

Input parameter: ucStartAddress, the start address.

xSizeInBytes, the area size, in bytes.

Output parameter: None.

Return value: None.

2.1.14. sys_remove_heap_region

Prototype: void sys_remove_heap_region(uint32_t ucStartAddress, uint32_t xSizeInBytes)

Function: Remove the heap region.

Input parameter: ucStartAddress, the start address.

xSizeInBytes, the area size, in bytes.

Output parameter: None.

Return value: None.

2.2. Task management

2.2.1. sys_task_create

Prototype: 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)

Function: Create a task.

Input parameter: static_tcb, the static task control block; if it is NULL, the OS will allocate the task control block.

name, the task name.

stack_base, the bottom of the task stack; if it is NULL, the OS will allocate the task stack.

stack_size, the stack size.

queue_size, the message queue size.

queue_item_size, the queue item size.

priority, the task priority.

func, the task function.

ctx, the task context.

Output parameter: None.

Return value: Non-NULL when the task is created successfully, and the task handle is returned;

NULL when the task creation fails.

2.2.2. sys_task_create_dynamic

Prototype: #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

Prototype: char* sys_task_name_get(void *task)

Function: Get RTOS task name.

Input parameter: task, the task handle. If it is NULL, return the name of the task itself.

Output parameter: None.

Return value: RTOS task name.

2.2.4. sys_task_delete

Prototype: void sys_task_delete(void *task)

Function: Delete a task.

Input parameter: task, the task handle. If it is NULL, delete the task itself.

Output parameter: None.

Return value: None.

2.2.5. sys_task_list

Prototype: void sys_task_list(char *pwrite_buf)

Function: Task list.

Input parameter: None.

Output parameter: pwrite_buf, the content of the task list.

Return value: None.

2.2.6. sys_current_task_handle_get

Prototype: os_task_t sys_current_task_handle_get(void)

Function: Get the handle of the current task.

Input parameter: None.

Output parameter: None.

Return value: The current task handle.

2.2.7. sys_timer_task_handle_get

Prototype: os_task_t *sys_timer_task_handle_get(void)

Function: Get the handle of the timer task.

Input parameter: None.

Output parameter: None.

Return value: The timer task handle.

2.2.8. sys_current_task_stack_depth

Prototype: int32_t sys_current_task_stack_depth(unsigned long cur_sp)

Function: Get the RTOS current task stack depth from special sp index.

Input parameter: cur_sp, the sp index.

Output parameter: None.

Return value: RTOS current task stack depth.

2.2.9. sys_stack_free_get

Prototype: uint32_t sys_stack_free_get(void *task)

Function: Get the free size of the task stack.

Input parameter: task, the task handle.

Output parameter: None.

Return value: The free size of the task stack.

2.2.10. sys_task_wait_notification

Prototype: int sys_task_wait_notification(int timeout)

Function: Suspend the task until a notification is received or timeout occurs.

Input parameter: timeout, timeout for notification. 0 means to return directly without waiting, and -1 means to always wait.

Output parameter: None.

Return value: Return 0 if timeout occurs; otherwise, return the notification value.

2.2.11. sys_task_notify

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

Function: Send a notification to the task.

Input parameter: task, the task handle.

isr, indicating whether it is called by an interrupt.

Output parameter: None.

Return value: None.

2.2.12. sys_priority_set

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

Function: Modify the priority of the task.

Input parameter: task, the task handle.

priority, the priority to be set.

Output parameter: None.

Return value: None.

2.2.13. sys_priority_get

Prototype: os_prio_t sys_priority_get(void *task)

Function: Get the priority of the task.

Input parameter: task, the task handle.

Output parameter: None.

Return value: priority of the task.

2.2.14. sys_task_exist

Prototype: uint8_t sys_task_exist(const uint8_t *name)

Function: Check task exist or not.

Input parameter: name, the task name.

Output parameter: None.

Return value: 1, task exist; 0, task not exist.

2.3. Inter-task communication

2.3.1. sys_task_wait

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

Function: Wait for task messages.

Input parameter: timeout_ms, the waiting timeout period; 0 means infinite waiting.

Output parameter: msg_ptr, the message pointer.

Return value: 0 upon success and non-0 upon failure.

2.3.2. sys_task_post

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

Function: Send a task message.

Input parameter: receiver_task, the handle of the receiving task.

msg_ptr, the message pointer.

from_isr, indicating whether it comes from ISR.

Output parameter: None.

Return value: 0 upon success and non-0 upon failure.

2.3.3. sys_task_msg_flush

Prototype: void sys_task_msg_flush(void *task)

Function: Clear the task message queue.

Input parameter: task, the task handle.

Output parameter: None.

Return value: None.

2.3.4. sys_task_msg_num

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

Function: Get the number of current task queue messages.

Input parameter: task, the task handle.

from_isr, indicating whether it comes from ISR.

Output parameter: None.

Return value: The number of messages.

2.3.5. sys_sema_init_ext

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

Function: Create and initialize the semaphore.

Input parameter: max_count, the maximum value of the semaphore.

init_val, the initial value of the semaphore.

Output parameter: sema, the semaphore handle.

Return value: 0 upon successful creation and non-0 upon creation failure.

2.3.6. sys_sema_init

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

Function: Create and initialize the semaphore.

Input parameter: init_val, the initial value of the semaphore.

Output parameter: sema, the semaphore handle.

Return value: 0 upon successful creation and non-0 upon creation failure.

2.3.7. sys_sema_free

Prototype: void sys_sema_free(os_sema_t *sema)

Function: Destroy the semaphore.

Input parameter: sema, the semaphore handle.

Output parameter: None.

Return value: None.

2.3.8. sys_sema_up

Prototype: void sys_sema_up(os_sema_t *sema)

Function: Send a semaphore.

Input parameter: sema, the semaphore handle.

Output parameter: None.

Return value: None.

2.3.9. sys_sema_up_from_isr

Prototype: void sys_sema_up_from_isr(os_sema_t *sema)

Function: Send a semaphore in ISR.

Input parameter: sema, the semaphore handle.

Output parameter: None.

Return value: None.

2.3.10. sys_sema_down

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

Function: Wait for semaphore.

Input parameter: sema, the semaphore handle.

timeout_ms, the waiting timeout period; 0 means always waiting.

Output parameter: None.

Return value: 0 upon success and non-0 upon failure.

2.3.11. sys_sema_get_count

Prototype: int sys_sema_get_count(os_sema_t *sema)

Function: Get the semaphore value.

Input parameter: sema, the semaphore handle.

Output parameter: None.

Return value: The semaphore value.

2.3.12. sys_mutex_init

Prototype: void sys_mutex_init(os_mutex_t *mutex)

Function: Create a mutex.

Input parameter: None.

Output parameter: mutex, the mutex handle.

Return value: None.

2.3.13. sys_mutex_free

Prototype: void sys_mutex_free(os_mutex_t *mutex)

Function: Destroy the mutex.

Input parameter: mutex, the mutex handle.

Output parameter: None.

Return value: None.

2.3.14. sys_mutex_try_get

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

Function: Try to require the mutex.

Input parameter: mutex, the mutex handle.

timeout, maximum duration to wait, in ms. 0 means do not wait and -1 means wait indefinitely.

Output parameter: None.

Return value: 0 upon getting mutex and -1 upon failure.

2.3.15. sys_mutex_get

Prototype: int32_t sys_mutex_get(os_mutex_t *mutex)

Function: Wait for mutex.

Input parameter: mutex, the mutex handle.

Output parameter: None.

Return value: 0 upon getting mutex and -1 upon failure.

2.3.16. sys_mutex_put

Prototype: void sys_mutex_put(os_mutex_t *mutex)

Function: Release the mutex.

Input parameter: mutex, the mutex handle.

Output parameter: None.

Return value: None.

2.3.17. sys_queue_init

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

Function: Create a queue.

Input parameter: queue_size, the size of the queue.

item_size, the size of the queue message.

Output parameter: queue, the queue handle.

Return value: 0 upon successful creation and -1 upon creation failure.

2.3.18. sys_queue_free

Prototype: void sys_queue_free(os_queue_t *queue)

Function: Destroy the message queue.

Input parameter: queue, the queue handle.

Output parameter: None.

Return value: None.

2.3.19. sys_queue_post

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

Function: Send a message to the queue.

Input parameter: queue, the queue handle.

msg, the message pointer.

Output parameter: None.

Return value: 0 upon success and -1 upon failure.

2.3.20. sys_queue_post_with_timeout

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

Function: Send a message to the queue, and return until timeout.

Input parameter: queue, the queue handle.

msg, the message pointer.

timeout_ms, the waiting timeout period in ms.

Output parameter: None.

Return value: 0 upon success and -1 upon failure.

2.3.21. sys_queue_fetch

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

Function: Get a message from the queue.

Input parameter: queue, the queue handle.

timeout_ms, the waiting timeout period.

is_blocking, indicating whether it is a blocking operation.

Output parameter: msg, the message pointer.

Return value: 0 upon success and -1 upon failure.

2.3.22. sys_queue_is_empty

Prototype: bool sys_queue_is_empty(os_queue_t *queue)

Function: Detect whether the message queue is empty.

Input parameter: queue, the queue handle.

Output parameter: None.

Return value: bool type, "true" means that the queue is empty, and "false" means that the queue is not empty.

2.3.23. sys_queue_cnt

Prototype: int sys_queue_cnt(os_queue_t *queue)

Function: Get the number of messages in the message queue.

Input parameter: queue, the queue handle.

Output parameter: None.

Return value: The number of messages in the message queue.

2.3.24. sys_queue_write

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

Function: Write the message to the end of the message queue.

Input parameter: queue, the queue handle.

msg, the message pointer.

timeout, the waiting time. 0 means no waiting and -1 means always waiting.

isr, indicating whether it comes from ISR. If yes, ignore the timeout parameter.

Output parameter: None.

Return value: 0 upon success and non-0 upon failure.

2.3.25. sys_queue_read

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

Function: Read a message from the message queue.

Input parameter: queue, the queue handle.

timeout, the waiting time. 0 means no waiting and -1 means always waiting.

isr, indicating whether it comes from ISR. If yes, ignore the

timeout parameter.

Output parameter: msg, the message pointer.

Return value: 0 upon success and non-0 upon failure.

2.4. Time management

2.4.1. sys_current_time_get

Prototype: uint32_t sys_current_time_get(void)

Function: Get the time since the system boots up.

Input parameter: None.

Output parameter: None.

Return value: The time since the system boots up, in milliseconds.

2.4.2. sys_time_get

Prototype: uint32_t sys_time_get(void *p)

Function: Get the time since the system boots up.

Input parameter: p, not used.

Output parameter: None.

Return value: The time since the system boots up, in milliseconds.

2.4.3. sys_ms_sleep

Prototype: void sys_ms_sleep(int ms)

Function: Switch the task to the sleep mode.

Input parameter: ms, the sleep time.

Output parameter: None.

Return value: None.

2.4.4. sys_us_delay

Prototype: void sys_us_delay(uint32_t nus)

Function: Perform the delay operation.

Input parameter: nus, the delay time, in microseconds.

Output parameter: None.

Return value: None.

2.4.5. sys_timer_init

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

Function: Create a timer.

Input parameter: timer, the timer handle.

name, the timer name.

delay, the timer timeout period.

periodic, indicating whether it is a periodic timer.

func, the timer function.

arg, the timer function parameter.

Output parameter: None.

Return value: None.

2.4.6. sys_timer_delete

Prototype: void sys_timer_delete(os_timer_t *timer)

Function: Destroy the timer.

Input parameter: timer, the timer handle.

Output parameter: None.

Return value: None.

2.4.7. sys_timer_start

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

Function: Start the timer.

Input parameter: timer, the timer handle.

from_isr, indicating whether it is in ISR.

Output parameter: None.

Return value: None.

2.4.8. sys_timer_start_ext

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

Function: Start the timer.

Input parameter: timer, the timer handle.

delay, the reset timer timeout period.

from_isr, indicating whether it is called in ISR.

Output parameter: None.

Return value: None.

2.4.9. sys_timer_stop

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

Function: Stop the timer.

Input parameter: timer, the timer handle.

from_isr, indicating whether it is called in ISR.

Output parameter: None.

Return value: 1 upon success and 0 upon failure.

2.4.10. sys_timer_pending

Prototype: uint8_t sys_timer_pending(os_timer_t *timer)

Function: Determine whether the timer is waiting in the activation queue.

Input parameter: timer, the timer handle.

Output parameter: None.

Return value: 1 when the timer is waiting in activation queue and 0 in other states.

2.4.11. sys_os_now

Prototype: uint32_t sys_os_now(bool isr)

Function: Get the current RTOS time.

Input parameter: isr, indicating whether it is called in ISR.

Output parameter: None.

Return value: The current RTOS time, in ticks.

2.4.12. sys_cpu_sleep_time_get

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

Function: Get cpu sleep time and stats time.

Input parameter: None.

Output parameter: stats_ms, statistics time in ms.

sleep_ms, sleep time in ms.

Return value: None.

2.5. Other system management

2.5.1. sys_os_init

Prototype: void sys_os_init(void)

Function: Initialize the RTOS.

Input parameter: None.

Output parameter: None.

Return value: None.

2.5.2. sys_os_start

Prototype: void sys_os_start(void)

Function: RTOS starts scheduling.

Input parameter: None.

Output parameter: None.

Return value: None.

2.5.3. sys_os_misc_init

Prototype: void sys_os_misc_init(void)

Function: Perform other initializations of RTOS after scheduling (required for some RTOS).

Input parameter: None.

Output parameter: None.

Return value: None.

2.5.4. sys_yield

Prototype: void sys_yield(void)

Function: The task gives up CPU control.

Input parameter: None.

Output parameter: None.

Return value: None.

2.5.5. sys_sched_lock

Prototype: void sys_sched_lock(void)

Function: Pause task scheduling.

Input parameter: None.

Output parameter: None.

Return value: None.

2.5.6. sys_sched_unlock

Prototype: void sys_sched_unlock(void)

Function: Continue task scheduling.

Input parameter: None.

Output parameter: None.

Return value: None.

2.5.7. sys_random_bytes_get

Prototype: int32_t sys_random_bytes_get(void *dst, uint32_t size)

Function: Get random data.

Input parameter: size, the length of random data.

Output parameter: dst, the address where the random data is saved.

Return value: 0 upon success and -1 upon failure.

2.5.8. sys_in_critical

Prototype: uint32_t sys_in_critical(void)

Function: Get the interrupt status in RTOS critical nesting.

Input parameter: None.

Output parameter: None.

Return value: The interrupt status in RTOS critical nesting.

2.5.9. sys_enter_critical

Prototype: void sys_enter_critical(void)

Function: RTOS enters the critical state.

Input parameter: None.

Output parameter: None.

Return value: None.

2.5.10. sys_exit_critical

Prototype: void sys_exit_critical(void)

Function: RTOS exits the critical state.

Input parameter: None.

Output parameter: None.

Return value: None.

2.5.11. sys_ps_set

Prototype: void sys_ps_set(uint8_t mode)

Function: Configure the power save mode of RTOS.

Input parameter: mode, the power save mode. 0: Exit the power save mode; 1: CPU Deep Sleep mode.

Output parameter: None.

Return value: None.

2.5.12. sys_ps_get

Prototype: uint8_t sys_ps_get(void)

Function: Get the power save mode of the current RTOS.

Input parameter: None.

Output parameter: None.

Return value: The power save mode of the current RTOS.

2.5.13. sys_cpu_stats

Prototype: void sys_cpu_stats(void)

Function: Show cpu usage percentage per task.

Input parameter: None.

Output parameter: None.

Return value: None.

2.5.14. sys_int_enter

Prototype: void sys_int_enter(void)

Function: OS IRQ service hook called just after the ISR starts.

Input parameter: None.

Output parameter: None.

Return value: None.

2.5.15. sys_int_exit

Prototype: void sys_int_exit(void)

Function: OS IRQ service hook called before the ISR exits.

Input parameter: None.

Output parameter: None.

Return value: None.

3. Wi-Fi Netif API

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

3.1. Wi-Fi LwIP network interface API

3.1.1. net_ip_chksum

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

Function: Calculate the checksum of data.

Input parameter: dataptr, a pointer to the buffer that stores the data to be calculated for the checksum.

len, the length of dataptr, in bytes.

Output parameter: None

Return value: The calculated checksum.

3.1.2. net_if_add

Prototype: 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)

Function: Register a Wi-Fi network interface with LwIP.

Input parameter: net_if, a net_if structure pointer to the network interface to be registered.

mac_addr, a pointer to the MAC address.

ipaddr, a pointer to the IPv4 address.

netmask, a pointer to the netmask.

gw, a pointer to the gateway address.

vif_priv, a wifi_vif_tag structure pointer to the Wi-Fi VIF.

Output parameter: None

Return value: 0 upon successful execution and -1 upon failure.

3.1.3. net_if_remove

Prototype: int net_if_remove(void *net_if)

Function: Remove the Wi-Fi network interface.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

Output parameter: None

Return value: 0 upon successful execution and non-0 value upon failure.

3.1.4. net_if_get_mac_addr

Prototype: const uint8_t *net_if_get_mac_addr(void *net_if)

Function: Get the MAC address of the Wi-Fi network interface.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

Output parameter: None

Return value: A pointer to the MAC address of the Wi-Fi network interface.

3.1.5. net_if_find_from_name

Prototype: void *net_if_find_from_name(const char *name)

Function: Get the Wi-Fi network interface through its name.

Input parameter: A pointer to the name of the Wi-Fi network interface.

Output parameter: None

Return value: Return a pointer to the network interface upon successful execution and NULL upon failure.

3.1.6. net_if_get_name

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

Function: Get the name of the Wi-Fi network interface.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

len, the length of the buffer, which is used to save the name of the Wi-Fi network interface, in bytes.

Output parameter: buf, a pointer to the buffer, which is used to save the name of the Wi-Fi network interface.

Return value: The length of the Wi-Fi network interface name, in bytes.

3.1.7. net_if_up

Prototype: void net_if_up(void *net_if)

Function: Enable the Wi-Fi network interface.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

Output parameter: None

Return value: None

3.1.8. net_if_down

Prototype: void net_if_down(void *net_if)

Function: Disable the Wi-Fi network interface.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

Output parameter: None

Return value: None.

3.1.9. net_if_input

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

Function: Transfer data to the LWIP.

Input parameter: buf, a net_buf_rx_t structure pointer, which is used to save the data transfered to the LWIP.

net_if, a net_if structure pointer to the Wi-Fi network interface that transfers data.

addr, a pointer to the data to be transferred.

len, the length of the data to be transferred, in bytes.

free_fn, the callback function after the data is transferred, which is used to release the buffer that stores data.

Output parameter: None

Return value: 0 upon successful execution and -1 upon failure.

3.1.10. net_if_vif_info

Prototype: void *net_if_vif_info(void *net_if)

Function: Get the Wi-Fi interface corresponding to the Wi-Fi network interface.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

Output parameter: None

Return value: A pointer to the Wi-Fi VIF.

3.1.11. net_buf_tx_alloc

Prototype: net_buf_tx_t *net_buf_tx_alloc(uint32_t length)

Function: Allocate a buffer to save TX data. The buffer type is PBUF_RAM.

Input parameter: length, the length of the TX data to be saved, in bytes.

Output parameter: None

Return value: Return a pointer to the buffer that is filled by the net_buf_tx_t structure upon successful execution and NULL upon failure.

3.1.12. net_buf_tx_alloc_ref

Prototype: net_buf_tx_t *net_buf_tx_alloc_ref(uint32_t length)

Function: Allocate a buffer to save TX data. The buffer type is PBUF_REF.

Input parameter: length, the length of the TX data to be saved, in bytes.

Output parameter: None

Return value: Return a pointer to the buffer that is filled by the net_buf_tx_t structure upon successful execution and NULL upon failure.

3.1.13. net_buf_tx_info

Prototype: 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[])

Function: Get information from TX buffer--net_buf_tx_t *buf.

Input parameter: buf, a net_buf_tx_t structure pointer to the TX buffer.

seg_cnt, the preset maximum number of divisible segments of the TX buffer.

Output parameter: tot_len, the total length of the TX buffer, in bytes.

seg_cnt, the number of actual divisible segments of the TX buffer.

seg_addr[], which saves the start address of each segment.

seg_len[], which saves the length of each segment, in bytes.

Return value: Return a pointer to the first segment upon successful execution and NULL upon failure.

3.1.14. net_buf_tx_free

Prototype: void net_buf_tx_free(net_buf_tx_t *buf)

Function: Release the TX buffer.

Input parameter: buf, a net_buf_tx_t structure pointer to the TX buffer.

Output parameter: None.

Return value: None.

3.1.15. net_init

Prototype: int net_init(void)

Function: Initialize the L2 resources.

Input parameter: None.

Output parameter: None.

Return value: 0 upon successful execution and non-0 value upon failure.

3.1.16. net_deinit

Prototype: void net_deinit(void)

Function: Release the L2 resources.

Input parameter: None.

Output parameter: None.

Return value: None.

3.1.17. net_l2_socket_create

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

Function: Create an L2 (aka ethernet) socket for a designated packet.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

ethertype, Ethernet type.

Output parameter: None.

Return value: socket descriptor upon successful execution and a negative value upon failure

3.1.18. net_l2_socket_delete

Prototype: int net_l2_socket_delete(int sock)

Function: Delete an L2 (aka ethernet) socket.

Input parameter: sock, socket descriptor of L2 (aka ethernet) to be deleted.

Output parameter: None.

Return value: 0 upon successful execution and non-0 value upon failure.

3.1.19. net_l2_send

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

Function: Send an L2 (aka ethernet) packet.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

data, a pointer to the data to be transferred.

data_len, the length of the data to be transferred, in bytes.

ethertype, the Ethernet type of the data to be transferred.

dst_addr, a pointer to the destination address.

Output parameter: ack, indicating the sending status.

Return value: 0 upon successful execution and -1 upon failure.

3.1.20. net_if_set_default

Prototype: void net_if_set_default(void *net_if)

Function: Set the network interface as the default network interface.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

Output parameter: None.

Return value: None.

3.1.21. net_if_set_ip

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

Function: Set the IP address, mask, and gateway of the Wi-Fi network interface.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

ip, a pointer to the IP address.

netmask, a pointer to the netmask.

gw, a pointer to the gateway address.

Output parameter: None.

Return value: None.

3.1.22. net_if_get_ip

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

Function: Get the IP address, netmask, and gateway address of the Wi-Fi network interface.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

Output parameter: ip, a pointer to the IP address.

netmask, a pointer to the netmask.

gw, a pointer to the gateway address.

Return value: 0 upon successful execution and -1 upon failure.

3.1.23. net_if_send_gratuitous_arp

Prototype: void net_if_send_gratuitous_arp(void *net_if)

Function: Send "gratuitous ARP" on a given interface.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

Output parameter: None.

Return value: None.

3.1.24. net_dhcp_start

Prototype: int net_dhcp_start(void *net_if)

Function: Enable DHCP on the Wi-Fi network interface.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

3.1.25. net_dhcp_stop

Prototype: void net_dhcp_stop(void *net_if)

Function: Stop DHCP on the Wi-Fi network interface.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

Output parameter: None.

Return value: None.

3.1.26. net_dhcp_release

Prototype: int net_dhcp_release(void *net_if)

Function: Release DHCP lease on the Wi-Fi network interface.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

3.1.27. net_dhcp_address_obtained

Prototype: bool net_dhcp_address_obtained(void *net_if)

Function: Detect whether the IP address has been obtained through DHCP.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

Output parameter: None.

Return value: Return "true" (1) if obtained and "false" (0) if not obtained.

3.1.28. net_dhcpd_start

Prototype: int net_dhcpd_start(void *net_if)

Function: Enable DHCPD on the Wi-Fi network interface.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

3.1.29. net_dhcpd_stop

Prototype: void net_dhcpd_stop(void *net_if)

Function: Stop DHCPD on the Wi-Fi network interface.

Input parameter: net_if, a net_if structure pointer to the Wi-Fi network interface.

Output parameter: None.

Return value: None.

3.1.30. net_set_dns

Prototype: int net_set_dns(uint32_t dns_server)

Function: Configure the IP address (IPv4) of the DNS server.

Input parameter: dns_server, the IPv4 address of the DNS server.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

3.1.31. net_get_dns

Prototype: int net_get_dns(uint32_t *dns_server)

Function: Get the IP address (IPv4) of the DNS server.

Input parameter: None.

Output parameter: dns_server, a pointer to the IPv4 address of the DNS server.

Return value: 0 upon successful execution and -1 upon failure.

3.1.32. net_buf_tx_cat

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

Function: Connect two TX buffers (net_buf_tx_t type).

Input parameter: buf1, a net_buf_tx_t structure pointer to one TX buffer to be connected.

buf2, a net_buf_tx_t structure pointer to the other TX buffer to be connected.

Output parameter: None.

Return value: None.

3.1.33. net_lpbk_socket_create

Prototype: int net_lpbk_socket_create(int protocol)

Function: Apply for a loopback socket.

Input parameter: protocol, the protocol used by the socket.

Output parameter: None.

Return value: Return the socket descriptor upon success and -1 upon failure.

3.1.34. net_lpbk_socket_bind

Prototype: int net_lpbk_socket_bind(int sock_recv, uint32_t port)

Function: Bind the socket and the network card information on the server.

Input parameter: sock_recv, the socket descriptor.

port, the port number of the network card.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

3.1.35. net_lpbk_socket_connect

Prototype: int net_lpbk_socket_connect(int sock_send, uint32_t port)

Function: Bind the socket and the remote network card information on the client.

Input parameter: sock_send, the socket descriptor.

port, the port number of the network card.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

3.1.36. net_if_use_static_ip

Prototype: void net_if_use_static_ip(bool static_ip)

Function: Indicate whether to use the static IP.

Input parameter: static_ip, the bool type, indicating whether to use the static IP.

Output parameter: None.

Return value: None.

3.1.37. net_if_is_static_ip

Prototype: bool net_if_is_static_ip(void)

Function: Detect whether the static IP is being used.

Input parameter: None.

Output parameter: None.

Return value: The bool type, "true" means that the static IP has been used, and "false" means that the static IP has not been used.

4. Wi-Fi API

This section introduces APIs related to Wi-Fi management.

4.1. Wi-Fi initialization and task management

The header file is MSDK\wifi_manager\wifi_init.h.

4.1.1. wifi_init

Prototype: int wifi_init(void)

Function: Initialize Wi-Fi pmu and Wi-Fi modules.

Input parameter: None.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.1.2. wifi_sw_init

Prototype: int wifi_sw_init(void)

Function: Initialize Wi-Fi related modules.

Input parameter: None.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.1.3. wifi_sw_deinit

Prototype: void wifi_sw_deinit(void)

Function: Release Wi-Fi related modules.

Input parameter: None.

Output parameter: None.

Return value: None.

4.1.4. wifi_task_ready

Prototype: void wifi_task_ready(enum wifi_task_id task_id)

Function: Indicate that the relevant task is ready.

Input parameter: task_id, task ID.

Output parameter: None.

Return value: None.

4.1.5. wifi_wait_ready

Prototype: int wifi_wait_ready(void)

Function: Wait for Wi-Fi to be ready.

Input parameter: None.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

4.1.6. wifi_task_terminated

Prototype: void wifi_task_terminated(enum wifi_task_id task_id)

Function: Terminate the task.

Input parameter: task_id, task ID.

Output parameter: None.

Return value: None.

4.1.7. wifi_wait_terminated

Prototype: int wifi_wait_terminated(enum wifi_task_id task_id)

Function: Wait for the task to be terminated.

Input parameter: task_id, task ID.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

4.2. Wi-Fi VIF management

The header file is MSDK\wifi_manager\wifi_vif.h.

The header file is MSDK\wifi_manager\wifi_net_ip.h.

4.2.1. wifi_vif_init

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

Function: Initialize Wi-Fi VIF.

Input parameter: vif_idx, Wi-Fi VIF index.

base_mac_addr, mac_addr structure pointer to the MAC address.

Output parameter: None.

Return value: None.

4.2.2. wifi_vifs_init

Prototype: int wifi_vifs_init(struct mac_addr *base_mac_addr)

Function: Initialize all Wi-Fi VIFs.

Input parameter: base_mac_addr, mac_addr structure point to MAC address.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.2.3. wifi_vifs_deinit

Prototype: void wifi_vifs_deinit(void)

Function: Release all Wi-Fi VIFs.

Input parameter: None.

Output parameter: None.

Return value: None.

4.2.4. wifi_vif_type_set

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

Function: Set the type of Wi-Fi VIF.

Input parameter: vif_idx, Wi-Fi VIF index.

type, the type of Wi-Fi VIF to be set, which is listed in enumeration wifi_vif_type.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

```
enum wifi_vif_type {
    WVIF_UNKNOWN,
    WVIF_STA,
    WVIF_AP,
    WVIF_MONITOR,
};
```

4.2.5. wifi_vif_name

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

Function: Get the name of Wi-Fi VIF.

Input parameter: vif_idx, Wi-Fi VIF index.

len, the length of the buffer, which is used to save the name of Wi-Fi VIF, in bytes.

Output parameter: name, a pointer to the buffer, which is used to save the name of Wi-Fi VIF.

Return value: Return the length of the Wi-Fi VIF name in bytes upon successful execution and -1 upon failure.

4.2.6. wifi_vif_reset

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

Function: Reset the configuration of Wi-Fi VIF.

Input parameter: vif_idx, Wi-Fi VIF index.

type, the type of Wi-Fi VIF.

Output parameter: None.

Return value: None.

4.2.7. vif_idx_to_mac_vif

Prototype: void *vif_idx_to_mac_vif(uint8_t vif_idx)

Function: Get the MAC VIF information of Wi-Fi VIF.

Input parameter: vif_idx, Wi-Fi VIF index.

Output parameter: None.

Return value: a structure pointer to the saved MAC VIF information upon successful

execution and NULL upon failure.

4.2.8. wvif_to_mac_vif

Prototype: void *wvif_to_mac_vif(void *wvif)

Function: Get the MAC VIF information of Wi-Fi VIF.

Input parameter: wvif, Wi-Fi VIF

Output parameter: None.

Return value: a structure pointer to the saved MAC VIF information upon successful execution and NULL upon failure.

4.2.9. vif_idx_to_net_if

Prototype: void *vif_idx_to_net_if(uint8_t vif_idx)

Function: Get the Netif VIF information of Wi-Fi VIF.

Input parameter: vif_idx, Wi-Fi VIF index.

Output parameter: None.

Return value: a structure pointer to the saved Netif VIF information upon successful execution and NULL upon failure.

4.2.10. vif_idx_to_wvif

Prototype: void *vif_idx_to_wvif(uint8_t vif_idx)

Function: Get the information of Wi-Fi VIF.

Input parameter: vif_idx, Wi-Fi VIF index.

Output parameter: None.

Return value: a structure pointer to the saved Wi-Fi VIF information upon successful execution and NULL upon failure.

4.2.11. wvif_to_vif_idx

Prototype: int wvif_to_vif_idx(void *wvif)

Function: Get the index of Wi-Fi VIF.

Input parameter: wvif, Wi-Fi VIF

Output parameter: None.

Return value: the index of Wi-Fi VIF.

4.2.12. wifi_vif_sta_uapsd_get

Prototype: uint8_t wifi_vif_sta_uapsd_get(int vif_idx)

Function: Get the UAPSD queue configuration of Wi-Fi VIF in the Station mode.

Input parameter: vif_idx, Wi-Fi VIF index.

Output parameter: None.

Return value: UAPSD queue configuration of Wi-Fi VIF in the Station mode.

4.2.13. wifi_vif_uapsd_queues_set

Prototype: int wifi_vif_uapsd_queues_set(int vif_idx, uint8_t uapsd_queues)

Function: Set the UAPSD queue configuration of Wi-Fi VIF in the Station mode.

Input parameter: vif_idx, Wi-Fi VIF index.

uapsd_queues, UAPSD queue configuration.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

4.2.14. wifi_vif_mac_addr_get

Prototype: uint8_t * wifi_vif_mac_addr_get(int vif_idx)

Function: Get the MAC address of Wi-Fi VIF.

Input parameter: vif_idx, Wi-Fi VIF index.

Output parameter: None.

Return value: a pointer to the MAC address of Wi-Fi VIF upon successful execution and NULL upon failure.

4.2.15. wifi_vif_mac_vif_set

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

Function: Bind Wi-Fi VIF with MAC VIF.

Input parameter: vif_idx, Wi-Fi VIF index.

mac_vif, a pointer to MAC VIF.

Output parameter: None.

Return value: None.

4.2.16. wifi_vif_is_softap

Prototype: int wifi_vif_is_softap(int vif_idx)

Function: Judge whether Wi-Fi VIF is in the SoftAP mode.

Input parameter: vif_idx, Wi-Fi VIF index.

Output parameter: None.

Return value: true for SoftAP mode and false for other conditions.

4.2.17. wifi_vif_is_sta_connecting

Prototype: int wifi_vif_is_sta_connecting(int vif_idx)

Function: Judge whether Wi-Fi VIF is in connection phase in the Station mode.

Input parameter: vif_idx, Wi-Fi VIF index.

Output parameter: None.

Return value: true for connection phase and false for other conditions.

4.2.18. wifi_vif_is_sta_handshaked

Prototype: int wifi_vif_is_sta_handshaked(int vif_idx)

Function: Judge whether Wi-Fi VIF is in handshake phase in the Station mode.

Input parameter: vif_idx, Wi-Fi VIF index.

Output parameter: None.

Return value: true for handshaked condition and false for other conditions.

4.2.19. wifi_vif_is_sta_connected

Prototype: int wifi_vif_is_sta_connected(int vif_idx)

Function: Judge whether Wi-Fi VIF is connected to an AP in the Station mode.

Input parameter: vif_idx, Wi-Fi VIF index.

Output parameter: None.

Return value: true if is connected and false for other conditions.

4.2.20. wifi_vif_idx_from_name

Prototype: int wifi_vif_idx_from_name(const char *name)

Function: Get the index of Wi-Fi VIF.

Input parameter: name, a pointer to the name of Wi-Fi VIF.

Output parameter: None.

Return value: Return the index of Wi-Fi VIF upon successful execution and -1 upon failure.

4.2.21. wifi_vif_user_addr_set

Prototype: void wifi_vif_user_addr_set(uint8_t *user_addr)

Function: Set the MAC address of Wi-Fi VIF.

Input parameter: user_addr, a pointer to MAC address.

Output parameter: None.

Return value: None.

4.2.22. wifi_ip_chksum

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

Function: Calculate the checksum in LwIP.

Input parameter: dataptr, data for checksum calculation.

len, length of data.

Output parameter: None.

Return value: The calculated checksum.

4.2.23. wifi_set_vif_ip

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

Function: Set the IP address of Wi-Fi VIF.

Input parameter: vif_idx, Wi-Fi VIF index.

cfg, wifi_vif_ip_addr_cfg structure pointer, which saves the IP address information of Wi-Fi VIF.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

4.2.24. wifi_get_vif_ip

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

Function: Get the IP address information of the current Wi-Fi VIF.

Input parameter: fvif_idx, Wi-Fi VIF index.

Output parameter: cfg, wifi_vif_ip_addr_cfg structure pointer, which saves the IP address information of Wi-Fi VIF.

Return value: 0 upon successful execution and -1 upon failure.

4.3. Wi-Fi Netlink API

The header file is MSDK\wifi_manager\wifi_netlink.h.

4.3.1. wifi_netlink_wifi_open

Prototype: int wifi_netlink_wifi_open(void)

Function: Turn on the Wi-Fi device.

Input parameter: None.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.3.2. wifi_netlink_wifi_close

Prototype: void wifi_netlink_wifi_close(void)

Function: Turn off the Wi-Fi device.

Input parameter: None.

Output parameter: None.

Return value: None.

4.3.3. wifi_netlink_dbg_open

Prototype: int wifi_netlink_dbg_open(void)

Function: Enable the printing of Wi-Fi debug logs.

Input parameter: None.

Output parameter: None.

Return value: Return 0 directly.

4.3.4. wifi_netlink_dbg_close

Prototype: int wifi_netlink_dbg_close(void)

Function: Disable the printing of Wi-Fi debug logs.

Input parameter: None.

Output parameter: None.

Return value: Return 0 directly.

4.3.5. wifi_netlink_wireless_mode_print

Prototype: void wifi_netlink_wireless_mode_print(uint32_t wireless_mode)

Function: Show the name of wifi wireless mode.

Input parameter: wireless_mode, Wi-Fi wireless mode.

Output parameter: None.

Return value: None.

4.3.6. wifi_netlink_status_print

Prototype: int wifi_netlink_status_print(void)

Function: Print the current Wi-Fi status of the development board.

Input parameter: None.

Output parameter: None.

Return value: Return 0 directly.

4.3.7. wifi_netlink_scan_set

Prototype: int wifi_netlink_scan_set(int vif_idx, uint8_t channel)

Function: Set and enable Wi-Fi scan.

Input parameter: vif_idx, Wi-Fi VIF index.

channel, the channel to be scanned. 0xFF indicates all channels.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.3.8. wifi_netlink_scan_set_with_ssid

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

Function: Set and enable Wi-Fi scan for designated AP.

Input parameter: vif_idx, Wi-Fi VIF index.

ssid, ssid of designated AP, which can not be null.

channel, the channel to be scanned. 0xFF indicates all channels.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.3.9. wifi_netlink_scan_set_with_extraie

Prototype: int wifi_netlink_scan_set_with_extraie(int vif_idx, uint8_t channel,

uint8_t *extra_ie, uint32_t extra_ie_len)

Function: Set and enable Wi-Fi scan, extra_ie can be carried in the probe request frame.

Input parameter: vif_idx, Wi-Fi VIF index.

channel, the channel to be scanned. 0xFF indicates all channels.

extra_ie, extra ie carried in the probe request frame and can be NULL.

extra_ie_len, the length of extra ie. Set to 0 when extra_ie is NULL.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.3.10. wifi_netlink_scan_results_get

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

Function: Get the Wi-Fi scan results.

Input parameter: vif_idx, Wi-Fi VIF index.

Output parameter: results, macif_scan_results structure pointer, which saves the Wi-Fi scan results.

Return value: 0 upon successful execution and other values upon failure.

4.3.11. wifi_netlink_scan_result_print

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

Function: Print the Wi-Fi scan results.

Input parameter: idx, the index of the scanned AP.

result, macif_scan_results structure pointer, which saves the Wi-Fi scan results.

Output parameter: None.

Return value: None.

4.3.12. wifi_netlink_scan_results_print

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

Function: Print all results of wifi scan.

Input parameter: vif_idx, the index of the wifi vif.

callback, callback func to print result of wifi scan.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.3.13. wifi_netlink_candidate_ap_find

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

Function: Find the designated AP among Wi-Fi scan results.

Input parameter: vif_idx, Wi-Fi VIF index.

bssid, bssid of AP.

ssid, ssid of designated AP.

candidate, macif_scan_results structure pointer, which saves the Wi-Fi scan results.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

The bssid and ssid can not be null at the same time. When neither bssid or ssid is null, bssid prevails.

4.3.14. wifi_netlink_connect_req

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

Function: The function is used to enable Wi-Fi VIF to perform the operation of connecting to an AP in the Station mode.

Input parameter: vif_idx, Wi-Fi VIF index.

cfg, sta_cfg structure pointer, which saves the informations of the AP to be connected to.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.3.15. wifi_netlink_associate_done

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

Function: Indicate Wi-Fi VIF is associated in the Station mode.

Input parameter: vif_idx, Wi-Fi VIF index.

ind_param, connection information.

Output parameter: None.

Return value: Return 0 directly.

4.3.16. wifi_netlink_dhcp_done

Prototype: int wifi_netlink_dhcp_done(int vif_idx)

Function: Indicate dhcp is successful in the Station mode.

Input parameter: vif_idx, Wi-Fi VIF index.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.3.17. wifi_netlink_disconnect_req

Prototype: int wifi_netlink_disconnect_req(int vif_idx)

Function: The function is used to enable Wi-Fi VIF to perform the operation of disconnecting from AP in the Station mode.

Input parameter: vif_idx, Wi-Fi VIF index.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

4.3.18. wifi_netlink_auto_conn_set

Prototype: int wifi_netlink_auto_conn_set(uint8_t auto_conn_enable)

Function: Set enabling or disabling automatic connection for Wi-Fi.

Input parameter: auto_conn_enable: 0 means disable, while 1 means enable.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.3.19. wifi_netlink_auto_conn_get

Prototype: uint8_t wifi_netlink_auto_conn_get(void)

Function: Get the information of enabling or disabling automatic connection for Wi-Fi.

Input parameter: None.

Output parameter: None.

Return value: 0 means disable, while 1 means enable.

4.3.20. wifi_netlink_joined_ap_store

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

Function: Save the information of the AP which connected to Wi-Fi after enabling automatic connection.

Input parameter: cfg, sta_cfg structure pointer, which saves the information of the connected AP.

ip, the IP address of the connected AP.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.3.21. wifi_netlink_joined_ap_load

Prototype: int wifi_netlink_joined_ap_load(int vif_idx)

Function: Get the Wi-Fi connected AP information saved after enabling automatic connection..

Input parameter: vif_idx, Wi-Fi VIF index.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.3.22. wifi_netlink_ps_mode_set

Prototype: int wifi_netlink_ps_mode_set(int vif_idx, uint8_t ps_mode)

Function: Set the power save mode of Wi-Fi.

Input parameter: vif_idx, Wi-Fi VIF index.

ps_mode: 0 for disabled, 1 for normal mode, and 2 for dynamic mode.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

4.3.23. wifi_netlink_enable_vif_ps

Prototype: int wifi_netlink_enable_vif_ps(int vif_idx)

Function: Enable the power save mode of Wi-Fi.

Input parameter: vif_idx, Wi-Fi VIF index.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

4.3.24. wifi_netlink_ap_start

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

Function: The function is used to enable Wi-Fi VIF to start the SoftAP mode.

Input parameter: vif_idx, Wi-Fi VIF index.

cfg, ap_cfg structure pointer, which saves the configuration information of the SoftAP mode to be started.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.3.25. wifi_netlink_ap_stop

Prototype: int wifi_netlink_ap_stop(int vif_idx)

Function: The function is used to enable Wi-Fi VIF to stop the SoftAP mode.

Input parameter: vif_idx, Wi-Fi VIF index.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

4.3.26. wifi_netlink_channel_set

Prototype: int wifi_netlink_channel_set(uint32_t channel)

Function: Set the channel of Wi-Fi VIF.

Input parameter: channel, the channel index.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

4.3.27. wifi_netlink_monitor_start

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

Function: The function is used to enable Wi-Fi VIF to start the MONITOR mode.

Input parameter: vif_idx, Wi-Fi VIF index.

cfg, wifi_monitor structure pointer, which saves the configuration information of the MONITOR mode.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.3.28. wifi_netlink_twt_setup

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

Function: The function is used to enable Wi-Fi VIF to configure and establish TWT connection after TWT is enabled.

Input parameter: vif_idx, Wi-Fi VIF index.

param, macif_twt_setup_t structure pointer, which saves the configuration information of TWT.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

4.3.29. wifi_netlink_twt_teardown

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

Function: The function is used to enable Wi-Fi VIF to delete TWT connection after TWT is enabled.

Input parameter: vif_idx, Wi-Fi VIF index.

id, ID of TWT connection.

neg_type, TWT Negotiation type.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

4.3.30. wifi_netlink_fix_rate_set

Prototype: int wifi_netlink_fix_rate_set(int sta_idx, int fixed_rate_idx)

Function: The function is used to enable Wi-Fi VIF to set the fixed rate.

Input parameter: sta_idx, the station index.

fixed_rate_idx, rate index.

Output parameter: None.

Return value: 0 upon successful execution and 1 upon failure.

4.3.31. wifi_netlink_sys_stats_get

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

Function: The function is used to get wifi doze stats.

Input parameter: None.

Output parameter: doze_time, the doze time in statistic window time in ms.

stats_time, statistic window time in ms.

Return value: 0 upon successful execution and other values upon failure.

4.3.32. wifi_netlink_roaming_rssi_set

Prototype: int wifi_netlink_roaming_rssi_set(int vif_idx, int8_t rssi_thresh)

Function: The function is used to set roaming rssi threshold.

Input parameter: vif_idx, Wi-Fi VIF index.

rssi_thresh, the rssi threshold for roaming.

Output parameter: None.

Return value: 0 upon successful execution and 1 upon failure.

4.3.33. wifi_netlink_roaming_rssi_get

Prototype: int8_t wifi_netlink_roaming_rssi_get(int vif_idx)

Function: The function is used to get roaming rssi threshold.

Input parameter: vif_idx, Wi-Fi VIF index.

Output parameter: None.

Return value: rssi threshold on success and 0 if error occured.

4.3.34. wifi_netlink_listen_interval_set

Prototype: int wifi_netlink_listen_interval_set(uint8_t interval)

Function: The function is used to set the interval at which the hardware listens for beacon frames in low-power mode.

Input parameter: interval, the interval at which the hardware listens for beacon frames.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.3.35. wifi_netlink_status_get

Prototype: uint8_t wifi_netlink_status_get(void)

Function: The function is used to get current Wi-Fi work status.

Input parameter: None.

Output parameter: None.

Return value: Return current Wi-Fi work status.

4.4. Wi-Fi connection management

This section introduces the Wi-Fi connection management API. The header file is MSDK\wifi_manager\ wifi_management.h.

4.4.1. wifi_management_init

Prototype: int wifi_management_init(void)

Function: Initialize LwIP, Wi-Fi event loop, etc, only need to call once

Input parameter: None.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.4.2. wifi_management_deinit

Prototype: void wifi_management_deinit(void)

Function: Terminate Wi-Fi event loop and Wi-Fi Management task.

Input parameter: None.

Output parameter: None.

Return value: None.

4.4.3. wifi_management_scan

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

Function: Start scanning wireless networks.

Input parameter: blocked, 1: Block other operations, 0: No blocking.

ssid, NULL or a pointer to the specified ssid.

Output parameter: None.

Return value: Return 0 upon successful scan and other values on scan failure.

4.4.4. wifi_management_connect

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

Function: Start connecting to AP.

Input parameter: ssid, the network name of AP, with 1-32 characters.

password, the password of the AP, with 8-63 characters, which can be NULL if the encryption method is Open.

blocked, 1: Block other operations, 0: No blocking.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.4.5. wifi_management_connect_with_bssid

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

Function: Start connecting to AP.

Input parameter: bssid, bssid of AP.

password, the password of the AP, with 8-63 characters, which can be NULL if the encryption method is Open.

blocked, 1: Block other operations, 0: No blocking.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.4.6. wifi_management_connect_with_eap_tls

Prototype: 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)

Function: Start connecting to enterprise AP with EAP-TLS authentication.

Input parameter: ssid, ssid of AP.

identity, user identity.

ca_cert, root certificate.

client_key, client key.

client_cert, client certificate.

client_key_password, client cert password.

blocked, 1: Block other operations, 0: No blocking.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.4.7. wifi_management_disconnect

Prototype: int wifi_management_disconnect(void)

Function: Start disconnecting from AP.

Input parameter: None.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.4.8. wifi_management_ap_start

Prototype: int wifi_management_ap_start(char *ssid, char *passwd, uint32_t channel, wifi_ap_auth_mode_t auth_mode, uint32_t hidden)

Function: Start SoftAP, and the SDK enters the SoftAP mode.

Input parameter: ssid, the network name of SoftAP, with 1-32 characters.

passwd, the network password of SoftAP. "NULL" means to start an OPEN SoftAP.

channel, the network channel where the SoftAP is located, with a range of 1-13.

auth_mode, the encryption method of SoftAP, which is WPA2-PSK by default.

hidden, indicating whether to hide the ssid. 0: Broadcast ssid, 1: Hide ssid.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.4.9. wifi_management_ap_delete_client

Prototype: int wifi_management_ap_delete_client(uint8_t *client_mac_addr)

Function: Delete the specified peer client device in SoftAP mode.

Input parameter: client_mac_addr, the mac address of peer client device.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.4.10. wifi_management_ap_stop

Prototype: int wifi_management_ap_stop(void)

Function: Stop SoftAP, and the SDK exits the SoftAP mode.

Input parameter: None.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.4.11. wifi_management_concurrent_set

Prototype: int wifi_management_concurrent_set(int enable)

Function: Control SDK to enter or exit the Wi-Fi concurrent mode.

Input parameter: enable, 0: Exit the Wi-Fi concurrent mode; non-0 value: Enter the Wi-Fi concurrent mode.

Output parameter: None.

Return value: 0 upon successful execution.

4.4.12. wifi_management_concurrent_get

Prototype: int wifi_management_concurrent_get(void)

Function: Get the current Wi-Fi concurrent mode.

Input parameter: None.

Output parameter: None.

Return value: the current Wi-Fi concurrent mode.

4.4.13. wifi_management_sta_start

Prototype: int wifi_management_sta_start(void)

Function: The SDK enters the STA mode.

Input parameter: None.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

4.4.14. wifi_management_monitor_start

Prototype: int wifi_management_monitor_start(uint8_t channel, cb_macif_rx monitor_cb)

Function: The SDK enters the MONITOR mode.

Input parameter: channel, the channel monitored under MONITOR mode

monitor_cb, the callback function when a packet is received in MONITOR mode.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.4.15. wifi_management_roaming_set

Prototype: int wifi_management_roaming_set(uint8_t enable, int8_t rssi_th)

Function: Set whether to enable wifi roaming mechanism.

Input parameter: enable, 0: disble; 1: enable.

rssi_th, the rssi threshold to trigger roaming.

Output parameter: None.

Return value: Return 0 directly.

4.4.16. wifi_management_roaming_get

Prototype: int wifi_management_roaming_get(int8_t *rssi_th)

Function: Get whether to enable wifi roaming mechanism.

Input parameter: None.

Output parameter: rssi_th, the rssi threshold to trigger roaming.

Return value: 0: disble; 1: enable.

4.4.17. wifi_management_wps_start

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

Function: Start WPS mode connection.

Input parameter: is_pbc, bool type, "true" means that use PBC mode, and "false" means that use PIN mode.

pin, PIN code, only valid when using WPS PIN mode and can not be NULL.

blocked, 1: Block other operations, 0: No blocking.

Output parameter: None.

Return value: 0 upon successful execution and other values upon failure.

4.5. Wi-Fi event loop API

This section introduces the event loop component API. The header file is MSDK\wifi_manager\wifi_eloop.h.

4.5.1. eloop_event_handler

Prototype: typedef void (*eloop_event_handler)(void *eloop_data, void *user_ctx);

Function: Define a function of the eloop_event_handler type, which is used as the callback function when a general event is triggered.

Input parameter: eloop_data, the eloop context data used for callback.

user_ctx, the user context data used for callback.

Output parameter: None.

Return value: None.

4.5.2. eloop_timeout_handler

Prototype: typedef void (*eloop_timeout_handler)(void *eloop_data, void *user_ctx);

Function: Define a function of the eloop_timeout_handler type, which is used as the callback function when a timer timeout event occurs.

Input parameter: eloop_data, the eloop context data used for callback.

user_ctx, the user context data used for callback.

Output parameter: None.

Return value: None.

4.5.3. wifi_eloop_init

Prototype: int wifi_eloop_init(void)

Function: Initialize a global event for processing loop data.

Input parameter: None.

Output parameter: None

Return value: Return 0 directly.

4.5.4. eloop_event_register

Prototype: int eloop_event_register(eloop_event_id_t event_id,

eloop_event_handler handler,

void *eloop_data, void *user_data)

Function: Register a function for processing trigger events.

Input parameter: eloop_event_id_t event_id, the event that needs to be processed after being triggered.

handler, the callback function after the event is triggered, which is used to process the event.

eloop_data, a parameter of the callback function.

user_data, a parameter of the callback function.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

4.5.5. eloop_event_unregister

Prototype: void eloop_event_unregister(eloop_event_id_t event_id)

Function: Terminate the handler after an event is triggered, corresponding to eloop_event_register.

Input parameter: event_id, the event whose processing is terminated.

Output parameter: None.

Return value: None.

4.5.6. eloop_event_send

Prototype: int eloop_event_send(uint8_t vif_idx, uint16_t event)

Function: Send an event to the pending queue.

Input parameter: vif_idx, Wi-Fi VIF index.

event, the event to be sent.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

4.5.7. eloop_message_send

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

Function: Send a message to the pending queue.

Input parameter: vif_idx, Wi-Fi VIF index.

event, the event of message.

reason, the reason of message.

param, the parameter of message handler.

len, the length of param.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

4.5.8. eloop_timeout_register

Prototype: int eloop_timeout_register(unsigned int msecs,

eloop_timeout_handler handler,

void *eloop_data, void *user_data)

Function: Register a function for processing the triggered event timeout.

Input parameter: msecs, the timeout period, in ms.

handler, the callback function after timeout, which is used to process the timeout event.

eloop_data, a parameter of the callback function.

user_data, a parameter of the callback function.

Output parameter: None.

Return value: 0 upon successful execution and -1 upon failure.

4.5.9. eloop_timeout_cancel

Prototype: int eloop_timeout_cancel(eloop_timeout_handler handler,

void *eloop_data, void *user_data)

Function: Terminate a timer.

Input parameter: handler, the callback function after timeout that needs to be terminated.

eloop_data, a parameter of the callback function.

user_data, a parameter of the callback function.

Output parameter: None.

Return value: Return the number of terminated timers.

Note: When the value of eloop_data/user_data is ELOOP_ALL_CTX, it represents all timeouts.

4.5.10. eloop_timeout_is_registered

Prototype: int eloop_timeout_is_registered(eloop_timeout_handler handler,

void *eloop_data, void *user_data)

Function: Detect whether the timer has been registered.

Input parameter: eloop_timeout_handler handler, the matching callback function.

eloop_data, the matching eloop_data.

user_data, the matching user_data.

Output parameter: None.

Return value: Return 1 if registered and 0 if not registered.

4.5.11. wifi_eloop_run

Prototype: void wifi_eloop_run(void)

Function: Start the event loop and process events or messages in the queue.

Input parameter: None.

Output parameter: None.

Return value: None.

4.5.12. wifi_eloop_terminate

Prototype: void wifi_eloop_terminate(void)

Function: Terminate the event processing thread.

Input parameter: None.

Output parameter: None.

Return value: None.

4.5.13. wifi_eloop_destroy

Prototype: void wifi_eloop_destroy(void)

Function: Release all resources used for the event loop.

Input parameter: None.

Output parameter: None.

Return value: None.

4.5.14. wifi_eloop_terminated

Prototype: int wifi_eloop_terminated (void)

Function: Detect whether the event loop is terminated.

Input parameter: None.

Output parameter: None.

Return value: Return 1 if terminated and 0 if not terminated.

4.6. Wi-Fi management macros

4.6.1. Wi-Fi management event type

Table 4-1. Wi-Fi management event type

```
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_WPS_CMD,


WIFI_MGMT_EVENT_SCAN_DONE,

WIFI_MGMT_EVENT_SCAN_FAIL,

WIFI_MGMT_EVENT_SCAN_RESULT, //16


WIFI_MGMT_EVENT_EXTERNAL_AUTH_REQUIRED, //17


WIFI_MGMT_EVENT_ASSOC_SUCCESS, //18


WIFI_MGMT_EVENT_DHCP_START,
```

```
WIFI_MGMT_EVENT_DHCP_SUCCESS,
WIFI_MGMT_EVENT_DHCP_FAIL,  //21

WIFI_MGMT_EVENT_CONNECT_SUCCESS,
WIFI_MGMT_EVENT_CONNECT_FAIL,

WIFI_MGMT_EVENT_DISCONNECT,
WIFI_MGMT_EVENT_ROAMING_START,

WIFI_MGMT_EVENT_RX_UNPROT_DEAUTH,  //26
WIFI_MGMT_EVENT_RX_ACTION,

/* For STA WPS */
WIFI_MGMT_EVENT_WPS_SUCCESS,  //28
WIFI_MGMT_EVENT_WPS_FAIL,
WIFI_MGMT_EVENT_WPS_CRED,

/* For SoftAP only */
WIFI_MGMT_EVENT_START_AP_CMD,    //31
WIFI_MGMT_EVENT_STOP_AP_CMD,
WIFI_MGMT_EVENT_AP_SWITCH_CHNL_CMD,

WIFI_MGMT_EVENT_TX_MGMT_DONE,  //34
WIFI_MGMT_EVENT_CLIENT_ADDED,
WIFI_MGMT_EVENT_CLIENT_REMOVED,  //36

/* For Monitor only */
WIFI_MGMT_EVENT_MONITOR_START_CMD,
```

```
/* For STA 802.1x EAP */
WIFI_MGMT_EVENT_EAP_SUCCESS,


WIFI_MGMT_EVENT_MAX,
WIFI_MGMT_EVENT_NUM = WIFI_MGMT_EVENT_MAX - WIFI_MGMT_EVENT_START - 1,
} wifi_management_event_t;
```

4.6.2. Configuration macro for Wi-Fi management

WIFI_MGMT_ROAMING_RETRY_LIMIT	// Number of Wi-Fi roaming retries
WIFI_MGMT_ROAMING_RETRY_INTERVAL	// Roaming retry interval
WIFI_MGMT_DHCP_POLLING_LIMIT	// Number of successful DHCP polls,
WIFI_MGMT_DHCP_POLLING_INTERVAL	// Successful DHCP poll interval
WIFI_MGMT_LINK_POLLING_INTERVAL	// Wi-Fi connection quality poll interval

5. Application examples

After the SDK is started, developers can use the components to develop Wi-Fi applications. Here are examples of how to use the API of components to complete operations such as scanning wireless networks, connecting to AP, starting SoftAP, and connecting to Alibaba Cloud.

5.1. Scanning wireless networks

5.1.1. Scanning in blocking mode

In this example, after **scan_wireless_network** starts scanning, it blocks until the scan is completed, and then prints out the scan result.

Table 5-1. Example of code for scanning in blocking mode

```
#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. Scanning in non-blocking mode

In this example, **scan_wireless_network** starts scanning and registers the scan completion event. After the event is triggered, get the scan result and print it.

Table 5-2. Example of code for scanning in non-blocking mode

```
#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. Connect to AP

In this example, **wifi_connect_ap** connects to the AP whose name is "test" and password is "12345678".

Table 5-3. Example of code for connecting to AP

```
#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("wificonnect failed\r\n");

    }

}
```

5.3. Starting SoftAP

In this example, **wifi_start_ap** starts a SoftAP whose name is "test", and **wifi_get_client** gets the client list.

Table 5-4. Example of code for starting SoftAP

```
#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 distribution network

For the BLE distribution network procedure, please refer to the "AN152 GD32VW553 BLE Development Guide".

5.5. Alibaba Cloud access

This section takes Alibaba Cloud ali-smartliving-device-sdk-c-rel_1.6.6 as an example to introduce how to use the above Wi-Fi SDK API to adapt to cloud services. The APIs that ali-smartliving-device-sdk-c-rel_1.6.6 needs to adapt to are roughly divided into four parts: System Access, Wi-Fi distribution network, SSL network communication and OTA firmware upgrade, as described below.

5.5.1. System access

Alibaba Cloud system access includes the functions listed below.

Table 5-5. Examples of system access functions

```
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 distribution network

Alibaba Cloud supports a number of Wi-Fi distribution network methods, which can be divided into two categories in principle. One category is that the distribution network device sends multicast frames or special management frames with encoding information, and the IoT device to be distributed switches to different channels to monitor air interface packets. When the IoT device receives sufficient encoding information and parses the network name and password, it can connect to the wireless network. The other category is that the IoT device to be distributed enables the SoftAP, and the distribution network device connects to the SoftAP and informs the IoT device of the distribution network information. The IoT device disables the SoftAP and connects to the wireless network.

Table 5-6. Comparison of Alibaba Cloud SDK adaptation interfaces and Wi-Fi SDK APIs

Function	Alibaba Cloud SDK adaptation interface	Wi-Fi SDK API
Set the Wi-Fi operation to enter the Monitor mode, and call the passed in callback function when receiving 802.11 frames	HAL_Awss_Open_Monitor HAL_Awss_Close_Monitor	wifi_management_monitor_start
Set Wi-Fi to switch to the specified channel	HAL_Awss_Switch_Channel	wifi_netlink_channel_set
A function requesting Wi-Fi connection to a specified hotspot (Access Point)	HAL_Awss_Connect_Ap	wifi_management_connect
Indicate whether the Wi-Fi network has been connected	HAL_Sys_Net_Is_Ready	wifi_get_vif_ip
Send raw 802.11 frames on the current channel at the basic data rate (1Mbps)	HAL_Wifi_Send_80211_Raw_Frame	wifi_send_80211_frame
Get information of the connected hotspot (Access Point).	HAL_Wifi_Get_Ap_Info	macif_vif_status_get
Open the current device hotspot and switch the device from Station mode to SoftAP mode	HAL_Awss_Open_Ap	wifi_management_ap_start
Disable the current device hotspot	HAL_Awss_Close_Ap	wifi_management_ap_stop
Get the MAC address of the Wi-Fi network interface	HAL_Wifi_Get_Mac	wifi_vif_mac_addr_get

5.5.3. SSL network communication

Here are the SSL communication interfaces that Alibaba Cloud needs to adapt to. Wi-Fi SDK, which has transplanted MbedTLS3.6.2, directly calls MbedTLS API in the SSL interfaces that Alibaba Cloud adapts to. Developers can refer to their official documentsas https://www.alibabacloud.com/help/en/sdk , or refer to 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);

The basic network interfaces including TCP and UDP APIs is implemented, which can be refered to SDK\MSDK\cloud\alicloud\src\ref-impl\hal\os\freertos\hal_tcp_gd.c and SDK\MSDK\cloud\alicloud\src\ref-impl\hal\os\freertos\hal_udp_gd.c.

5.5.4. OTA Firmware upgrade

Alibaba Cloud support firmware upgrades for devices connected to the Alibaba Cloud. The interfaces that are compatible with the Alibaba Cloud SDK API are as follows.

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. Alibaba Cloud access examples

Refer to MSDK\cloud\alicloud\examples\linkkit\living_platform\living_platform_main.c.

6. Revision history

Table 6-1. Revision history

Revision No.	Description	Date
1.0	Initial release	Nov.24.2023
1.1	add api of roaming mechanism; add api of wifi info; adjust api of os; update Alibaba Cloud api.	Jul.12.2024
1.2	add api: sys_int_enter/sys_int_exit/sys_task_exist/wifi_vif_is_softap/ net_if_send_gratuitous_arp/wifi_netlink_scan_set_with_extraie/ wifi_netlink_enable_vif_ps/wifi_management_ap_delete_client; remove api: wifi_netlink_wps_pbc/wifi_netlink_wps_pin; update Wi-Fi management event type; update Alibaba Cloud access demo.	Mar.26.2025

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 according to the laws of the People's Republic of China and other applicable laws. The Company reserves all rights under such laws and no Intellectual Property Rights are transferred (either wholly or partially) or licensed by the Company (either expressly or impliedly) herein. 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.

To the maximum extent permitted by applicable law, the Company makes no representations or warranties of any kind, express or implied, with regard to the merchantability and the fitness for a particular purpose of the Product, nor does the Company assume any liability arising out of the application or use of any Product. Any information provided in this document is provided only for reference purposes. It is the sole responsibility of the user of this document to determine whether the Product is suitable and fit for its applications and products planned, and properly design, program, and test the functionality and safety of its applications and products planned using the Product. The Product is designed, developed, and/or manufactured for ordinary business, industrial, personal, and/or household applications only, and the Product is not designed or intended for use in (i) safety critical applications such as weapons systems, nuclear facilities, atomic energy controller, combustion controller, aeronautic or aerospace applications, traffic signal instruments, pollution control or hazardous substance management; (ii) life-support systems, other medical equipment or systems (including life support equipment and surgical implants); (iii) automotive applications or environments, including but not limited to applications for active and passive safety of automobiles (regardless of front market or aftermarket), for example, EPS, braking, ADAS (camera/fusion), EMS, TCU, BMS, BSG, TPMS, Airbag, Suspension, DMS, ICMS, Domain, ESC, DCDC, e-clutch, advanced-lighting, etc.. Automobile herein means a vehicle propelled by a self-contained motor, engine or the like, such as, without limitation, cars, trucks, motorcycles, electric cars, and other transportation devices; and/or (iv) other uses where the failure of the device or the Product can reasonably be expected to result in personal injury, death, or severe property or environmental damage (collectively "Unintended Uses"). Customers shall take any and all actions to ensure the Product meets the applicable laws and regulations. The Company is not liable for, in whole or in part, and customers shall hereby release the Company as well as its suppliers and/or distributors from, any claim, damage, or other liability arising from or related to all Unintended Uses of the Product. Customers shall indemnify and hold the Company, and its officers, employees, subsidiaries, affiliates as well as its 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 Product.

Information in this document is provided solely in connection with the Product. The Company reserves the right to make changes, corrections, modifications or improvements to this document and the Product described herein at any time without notice. The Company shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2025 GigaDevice Semiconductor Inc. – All rights reserved