GigaDevice Semiconductor Inc.

GD32W51x Wi-Fi Development Guide

Application Note
AN100

Table of Contents

List of Figures

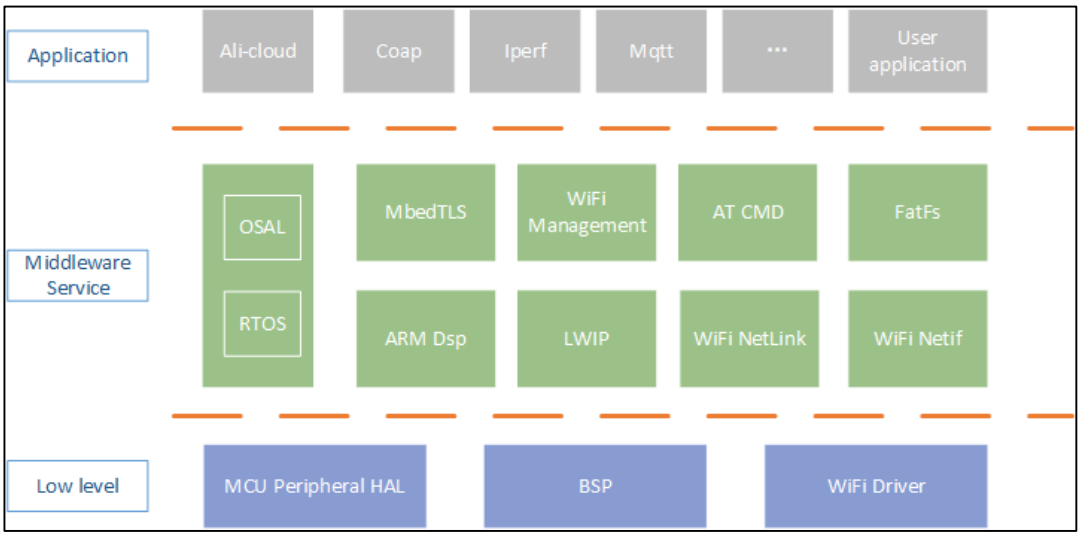
List of Tables

1. Introduction of Wi-Fi SDK

The GD32W51x series devices are 32-bit microcontroller (MCU) based on the Arm® Cortex®-M33 kernel. It supports the TrustZone and FPU features of Arm® Cortex®-M33 and includes Wi-Fi4 connection technology. GD32W51x Wi-Fi SDK integrates Wi-Fi driver, Lwip TCP / IP protocol stack, MbedTLS and other components, enabling developers to rapidly develop Internet of Things (IoT) applications based on GD32W51x. This application note describes the SDK framework, startup process, Wi-Fi, and related component API to help developers understand the SDK and develop applications by using the API.

1.1. Wi-Fi SDK software framework

Figure 1-1. Wi-Fi SDK block diagram



As shown in ***Figure 1-1. Wi-Fi SDK block diagram***, The software framework of GD32W51x Wi-Fi SDK consists of Low level layer, Middleware Service layer, and Application layer.

The Low Level layer is close to hardware and can directly carry out hardware peripherals related operations, including the peripheral Hardware Abstraction Layer (HAL), Board Support Package (BSP) and Wi-Fi driver of MCU. Developers can operate MCU peripherals such as USART, I2C and SPI through HAL, while BSP can perform board-level initialization, enable PMU, and enable hardware encryption engine. The Wi-Fi Driver can be accessed through components at the Middleware Service layer.

The Middleware Service layer consists of several components that provide encryption, network communication, and other services for applications. Among them, Arm Dsp, Mbedtls and Lwip are third-party components, and their use method can be referred to their official documents. The OSAL (Operate System Abstraction Layer) is a seal of RTOS kernel functions, and developers can use the OSAL to operate RTOS. Thanks to OSAL, developers can select their own RTOS as needed without affecting applications and other components. This

application note describes how to use the OSAL API in ***OSAL API*** chapter. The Wi-Fi Netlink component is a collection of Wi-Fi driver operations. Through Netlink, developers can obtain or set Wi-Fi related parameters and information, such as channel, entering hybrid mode, etc. Chapter 3 ***Wi-Fi Netlink API*** lists the API. The Wi-Fi Netif component is a set of network interface operations for Wi-Fi devices based on Lwip encapsulation. Developers can set the network address of the network interface and obtain the network address, gateway and other information of the network interface. Chapter 4 ***Wi-Fi Netif API*** introduces the use of Wi-Fi Netif API. The AT CMD component is a collection of AT commands. It is suitable for developers who are familiar with AT commands. For details, see the "GD32W51x AT Command User Guide". Wi-Fi Management is a Wi-Fi connection and roaming management service based on Netif and Netlink. Developers can scan wireless networks, connect to the AP (Access point), start the soft AP and other operations through this service. The software implementation of Wi-Fi Management adopts state machine and event management components, which allows developers to monitor the occurrence of Wi-Fi Driver events. Chapter 5 ***Wi-Fi Management API*** describes how to use the API, developers can do custom development.

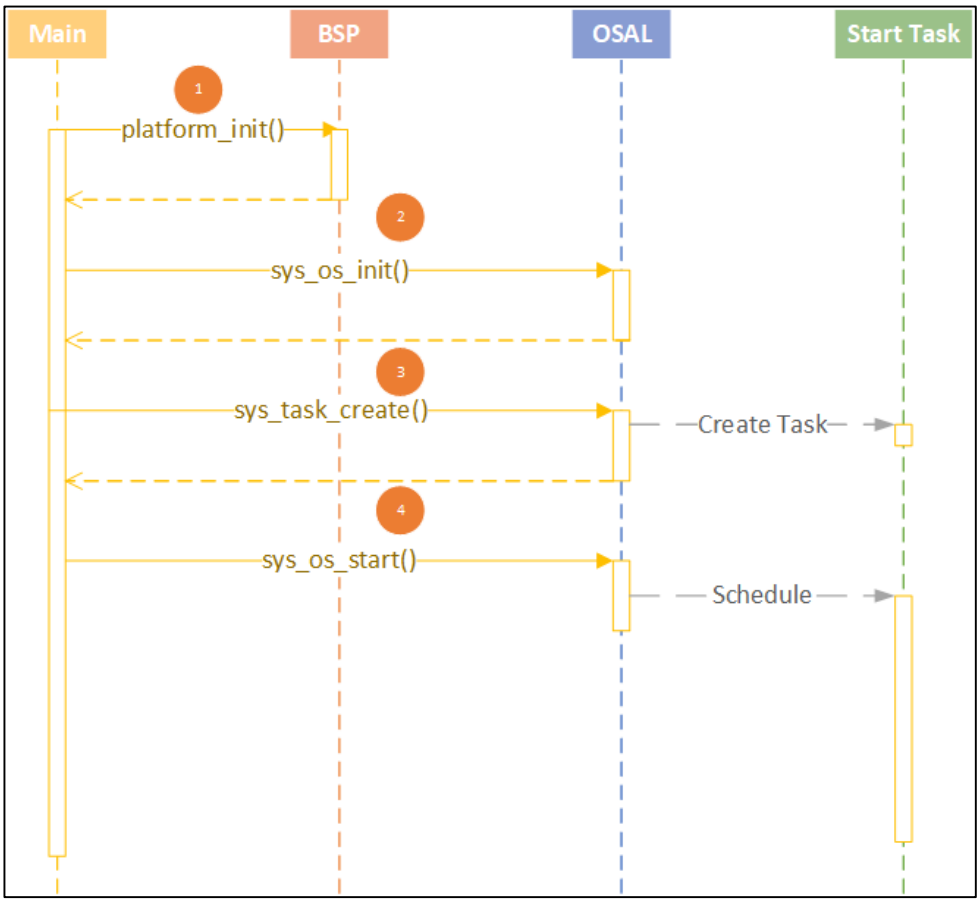
The Application layer is a collection of multiple applications, such as the iotkit distribution network based on Ali-Cloud and cloud service program Ali-Cloud, the performance test program iperf3, and the application customized by the developer and so on.

1.2. Wi-Fi SDK software startup process

This section describes the GD32W51x Wi-Fi SDK startup process to help developers understand the relationship between SDK components.

The SDK software startup process consists of two phases. The first phase is shown in ***Figure 1-2. The first phase of SDK startup***, from the board-level initialization of Main function to the Start Task scheduling by RTOS.

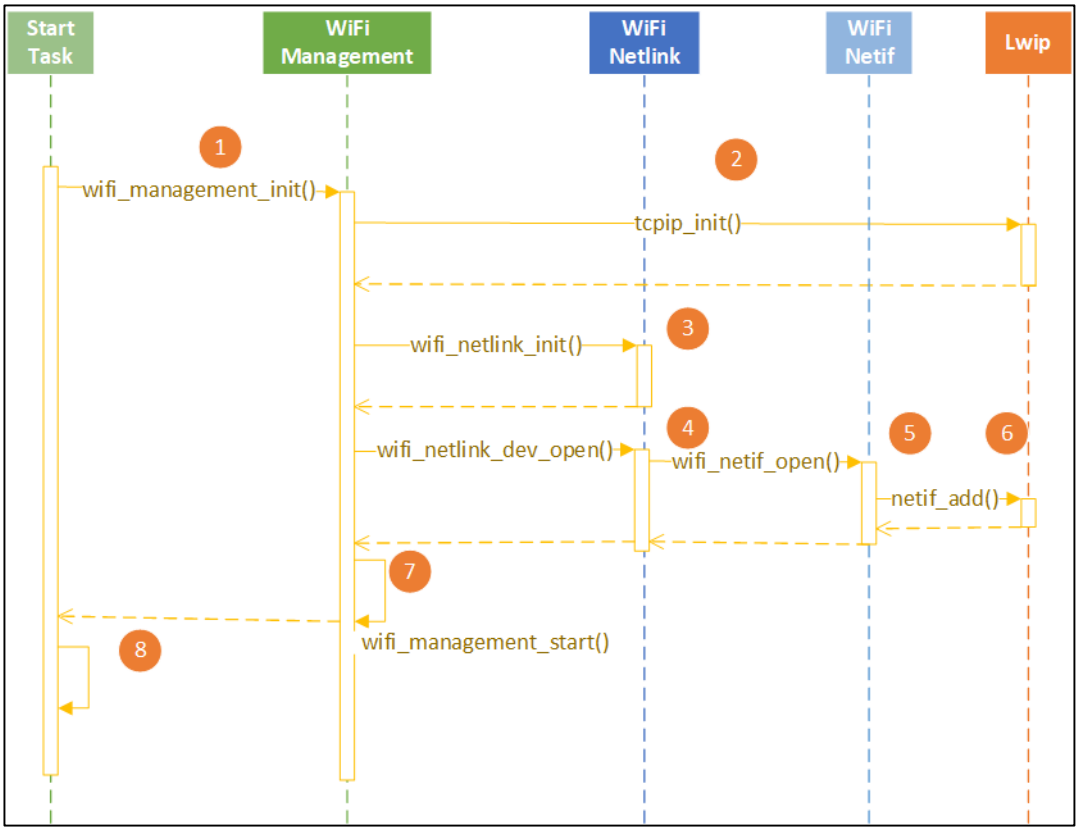
Figure 1-2. The first phase of SDK startup



- ① platform_init () performs the board-level initialization, after which the developer can debug by using the serial port.
- ② sys_os_init () performs RTOS initialization, after which the developer can create RTOS tasks.
- ③ sys_task_create () creates the Start Task in preparation for starting the second phase.
- ④ sys_os_start () starts the RTOS scheduling, Start Task starts working, which enters the second phase of SDK startup.

The second phase of SDK startup, shown in ***Figure 1-3. The second phase of SDK startup***, is all completed within the Start Task.

Figure 1-3. The second phase of SDK startup



- ① wifi_management_init() initializes wifi and network-related components.
- ② tcpip_init() initializes the Lwip TCP / IP protocol stack.
- ③ wifi_netlink_init() Initializes the netlink component.
- ④ wifi_netlink_dev_open(), ⑤ wifi_netif_open() and ⑥ netif_add() open the wifi device, initialize the wifi driver, and add the network interface to lwip.
- ⑦ wifi_management_start() starts the connection management service.
- ⑧ Finally the SDK is launched and the developer can launch the user program. Launch the console program here in the SDK.

2. OSAL API

The header file for the API is SDK \ NSPE \ WIFI_IOT \ common \ wrapper_os.h.

2.1. Memory management

2.1.1. sys_malloc

Function prototype: void *sys_malloc (size_t size)

Function: Allocate a block of memory of length size.

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

Onput parameter: None.

Return value: Return a pointer to allocating memory blocks on success and NULL on failure.

2.1.2. sys_calloc

Function prototype: void *sys_calloc (size_t count, size_t size)

Function: Allocate count contiguous memory of length size.

Input parameter: count, the number of memory blocks allocated.

size, the length of memory to be allocated.

Onput parameter: None.

Return value: Return a pointer to allocating memory blocks on success and NULL on failure.

2.1.3. sys_mfree

Function prototype: void sys_mfree (void *ptr)

Function: Free memory block.

Input parameter: ptr, points to the memory to be freed.

Output parameter: None.

Return value: None.

2.1.4. sys_realloc

Function prototype: void *sys_realloc (void *mem, size_t size)

Function: Expand the allocated memory.

Input parameter: mem, points to memory that needs to be expanded.

size, the size of the new memory block.

Output parameter: None.

Return value: Return a pointer to allocating memory blocks on success and NULL on failure.

2.1.5. sys_free_heap_size

Function prototype: int32_t sys_free_heap_size (void)

Function: Get the free size of the heap.

Input parameter: None.

Output parameter: None.

Return: The free size of the heap.

2.1.6. sys_min_free_heap_size

Function 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 size of the heap.

2.1.7. sys_heap_block_size

Function 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_memset

Function 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, the contents of the initialization.

count, the size of the memory block.

Output parameter: None.

Return value: None.

2.1.9. sys_memcpy

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

Function: Memory copy.

Input parameter: src, source address.

n, the length of the copy required.

Output parameter: dst, destination address.

Return value: None.

2.1.10. sys_memmove

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

Function: Memory move.

Input parameter: src, source address.

n, the length of memory to be moved.

Output parameter: des, destination address.

Return value: None.

2.1.11. sys_memcmp

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

Function: Compare the contents of two pieces of memory.

Input parameter: buf1, the address of memory block 1.

buf2, the address of memory block 2.

count, length.

Output parameter: None.

Return value: Return 0 if both memory block contents are the same, or non-0 if not.

2.2. Task magagment

2.2.1. sys_task_create

Function 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 priority, task_func_t func, void *ctx)

Function: Create the task

Input parameter: static_tcb, static task control block. If it is set to NULL, the OS allocates the task control block.

name, task name.

stack_base, bottom of task stack. If it is set to NULL, the OS allocates the task stack.

stack_size, size of stack.

queue_size, message queue size.

priority, task priority.

func, task function.

ctx, task context.

Output parameter: None.

Return value: If the task is created successfully, the task handle is returned. If the task fails to be created, NULL is returned.

2.2.2. sys_task_delete

Function prototype: void sys_task_delete (void *task)

Function: Delete the task

Input parameter: task, task handle. If it is set to NULL, the task itself is deleted.

Output parameter: None.

Return value: None.

2.2.3. sys_task_list

Function prototype: void sys_task_list (char *pwrite_buf)

Function: Task list.

Input parameter: None.

Output parameter: pwrite_buf, task list contents.

Return value: None.

2.2.4. sys_idle_task_handle_get

Function prototype: os_task_t *sys_idle_task_handle_get (void)

Function: Get the handle to the idle task.

Input parameter: None.

Output parameter: None.

Return value: the handle to the idle task.

2.2.5. sys_timer_task_handle_get

Function prototype: os_task_t *sys_timer_task_handle_get (void)

Function: Get the handle to the timer task.

Input parameter: None.

Output parameter: None.

Return value: The handle to the timer task.

2.2.6. sys_stack_free_get

Function prototype: uint32_t sys_get_stack_free (void *task)

Function: Get the free size of the task stack.

Input parameter: task, task handle.

Output parameter: None.

Return value: the free size of the task stack.

2.3. Inter-task communication

2.3.1. sys_task_wait

Function prototype:int32_t sys_task_wait (uint32_t timeout_ms, void *msg_ptr).

Function: Wait for a task message.

Input parameter: timeout_ms, the timeout period for waiting,0 stands for infinite wait.

Output parameter: msg_ptr, message pointer.

Return value: 0, success; Non-0, fail.

2.3.2. sys_task_post

Function 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, handle to receive the task.

msg_ptr, message pointer.

from_isr, indicates whether it is from ISR.

Output parameter: None.

Return value: 0, success; Non-0, fail.

2.3.3. sys_task_msg_flush

Function prototype: void sys_task_msg_flush (void *task)

Function: Clear the task message queue.

Input parameter: task, task handle.

Output parameter: None.

Return value: None.

2.3.4. sys_task_msg_num

Function prototype:int32_t sys_task_msg_num (void *task, uint8_t from_isr)

Function: Obtain the number of task queued messages.

Input parameter: task, task handle.

from_isr, task handle.

Output parameter: None.

Return value: Number of messages.

2.3.5. sys_sema_init

Function prototype:int32_t sys_sema_init (os_sema_t *sema, int32_t init_val)

Function: Create a semaphore.

Input parameter: init_val, initial semaphore value.

Output parameter: sema, semaphore handle.

Return value: 0, success; Non-0, fail.

2.3.6. sys_sema_free

Function prototype: void sys_sema_free (os_sema_t *sema)

Function: Destroy the semaphore.

Input parameter: sema, semaphore handle.

Output parameter: None.

Return value: None.

2.3.7. sys_sema_up

Function prototype: void sys_sema_up (os_sema_t *sema)

Function: Send the semaphore.

Input parameter: sema, semaphore handle.

Output parameter: None.

Return value: None.

2.3.8. sys_sema_up_from_isr

Function prototype: void sys_sema_up_from_isr (os_sema_t *sema)

Function: Send the semaphore in ISR.

Input parameter: sema, semaphore handle.

Output parameter: None.

Return value: None.

2.3.9. sys_sema_down

Function prototype:int32_t sys_sema_down (os_sema_t *sema, uint32_t timeout_ms)

Function: Wait for the semaphore.

Input parameter: sema, semaphore handle.

timeout_ms, the timeout period for waiting.

Output parameter: None.

Return value: 0, success; Non-0, fail.

2.3.10. sys_mutex_init

Function prototype: void sys_mutex_init (os_mutex_t *mutex)

Function: Create a mutex.

Input parameter: mutex, mutex handle.

Output parameter: None.

Return value: None.

2.3.11. sys_mutex_free

Function prototype: void sys_mutex_free (os_mutex_t *mutex)

Function: Destroy the mutex.

Input parameter: mutex, mutex handle.

Output parameter: None.

Return value: None.

2.3.12. sys_mutex_get

Function prototype: void sys_mutex_get (os_mutex_t *mutex)

Function: Wait for the mutex.

Input parameter: mutex, mutex handle.

Output parameter: None.

Return value: None.

2.3.13. sys_mutex_put

Function prototype: void sys_mutex_put (os_mutex_t *mutex)

Function: Release the mutex.

Input parameter: mutex, mutex handle.

Output parameter: None.

Return value: None.

2.3.14. sys_queue_init

Function 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, size of queue.

item_size, the size of the queue message.

Output parameter: queue, queue handle.

Return value: 0, created successfully; -1, created failure.

2.3.15. sys_queue_free

Function prototype: void sys_queue_free (os_queue_t *queue)

Function: Destroy message queue.

Input parameter: queue, queue handle.

Output parameter: None.

Return value: None.

2.3.16. sys_queue_post

Function prototype:int32_t sys_queue_post (os_queue_t *queue, void *msg)

Function: Send a message to the queue.

Input parameter: queue, queue handle.

msg, message pointer.

Output parameter: None.

Return value: 0, send successfully; -1, send failure.

2.3.17. sys_queue_fetch

Function 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, queue handle.

timeout_ms, the timeout period for waiting.

is_blocking, indicates whether the operation is a blocking operation.

Output parameter: msg, message pointer.

Return value: 0, send successfully; -1, send failure.

2.4. Time management

2.4.1. sys_current_time_get

Function prototype: uint32_t sys_current_time_get (void)

Function: Get the time since the system bootup.

Input parameter: None.

Output parameter: None.

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

2.4.2. sys_ms_sleep

Function prototype: void sys_ms_sleep (uint32_t ms)

Function: Put the task to sleep.

Input parameter: ms, duration of sleep.

Output parameter: None.

Return value: None.

2.4.3. sys_us_delay

Function prototype: void sys_us_delay (uint32_t nus)

Function: Delay operation.

Input parameter: nus, delay time, in microseconds.

Output parameter: None.

Return value: None.

2.4.4. sys_timer_init

Function 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, timer handle.

name, timer name.

delay, timer timeout period.

periodic, indicates whether it is a periodic timer.

func, timer function.

arg, timer function parameter.

Output parameter: None.

Return value: None.

2.4.5. sys_timer_delete

Function prototype: void sys_timer_delete (os_timer_t *timer)

Function: Destroy timer.

Input parameter: timer, timer handle.

Output parameter: None.

Return value: None.

2.4.6. sys_timer_start

Function prototype: void sys_timer_start (os_timer_t *timer, uint8_t from_isr)

Function: Start timer

Input parameter: timer, timer handle

from_isr, indicates whether it is from ISR

Output parameter: None.

Return value: None.

2.4.7. sys_timer_start_ext

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

Function: Start timer

Input parameter: timer, timer handle.

delay, reset the timer timeout period.

from_isr, indicates whether to call in ISR.

Output parameter: None.

Return value: None.

2.4.8. sys_timer_stop

Function prototype: uint8_t sys_timer_stop (os_timer_t *timer, uint8_t from_isr)

Function: Stop timer

Input parameter: timer, timer handle.

from_isr, indicates whether to call in ISR.

Output parameter: None.

Return value: 1, success; 0, fail.

2.4.9. sys_timer_pending

Function prototype: uint8_t sys_timer_pending (os_timer_t *timer)

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

Input parameter: timer, timer handle.

Output parameter: None.

Return value: 1, the timer is waiting in the activation queue; 0, other states.

2.5. Other system managment

2.5.1. sys_os_init

Function prototype: void sys_os_init (void)

Function: RTOS initialization.

Input parameter: None.

Output parameter: None.

Return value: None.

2.5.2. sys_os_start

Function prototype: void sys_os_start (void)

Function: The RTOS starts scheduling.

Input parameter: None.

Output parameter: None.

Return value: None.

2.5.3. sys_os_misc_init

Function prototype: void sys_os_misc_init (void)

Function: Other initializations of the RTOS after scheduling, as some RTOS require.

Input parameter: None.

Output parameter: None.

Return value: None.

2.5.4. sys_yield

Function prototype: void sys_yield (void)

Function: Task relinquishes CPU control.

Input parameter: None.

Output parameter: None.

Return value: None.

2.5.5. sys_sched_lock

Function prototype: void sys_sched_lock (void)

Function: Suspending Task Scheduling.

Input parameter: None.

Output parameter: None.

Return value: None.

2.5.6. sys_sched_unlock

Function prototype: void sys_sched_unlock (void)

Function: Continuing task scheduling.

Input parameter: None.

Output parameter: None.

Return value: None.

2.5.7. sys_random_bytes_get

Function prototype: int32_t sys_random_bytes_get (void *dst, uint32_t size)

Function: Get random data.

Input parameter: size, random data length.

Output parameter: dst, save the address of the random data.

Return value: 0, success; -1, fail.

3. Wi-Fi Netlink API

The header file for the API is SDK \ NSPE \ WIFI_IOT \ wifi \ wifi_netlink.h.

3.1. Wi-Fi message type

3.1.1. WIFI_MESSAGE_TYPE_E

Function: This enum contains the various types of wifi messages.

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

3.1.2. WIFI_NETLINK_STATUS_E

Function: This enum contains the various wifi netlink states.

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

3.2. Netlink data structure

3.2.1. WIFI_NETLINK_INFO_T

Function: This struct is used to store wifi netlink information.

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

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

3.3. Interface function

3.3.1. wifi_netlink_init

Function prototype: WIFI_NETLINK_INFO_T *wifi_netlink_init (void)

Function: This function initializes a WIFI_NETLINK_INFO_T struct and returns a pointer to it.

Input parameter: None.

Output parameter: None.

Return value: WIFI_NETLINK_INFO_T struct pointer.

3.3.2. wifi_netlink_dev_open

Function prototype: int wifi_netlink_dev_open (void)

Function: Open the Wi-Fi device.

Input parameter: None.

Output parameter: None.

Return value:0 is returned after successful execution.

3.3.3. wifi_netlink_dev_close

Function prototype: int wifi_netlink_dev_close (void)

Function: Close the WIFI device.

Input parameter: None.

Output parameter: None.

Return value: 0 is returned after successful execution.

3.3.4. wifi_netlink_scan_set

Function prototype: int wifi_netlink_scan_set (void)

Function: This function is used to perform Wi-Fi scanning operations.

Input parameter: None.

Output parameter: None.

Return value: 0, success; Non-0, fail.

3.3.5. wifi_netlink_scan_list_get

Function prototype: int wifi_netlink_scan_list_get (iter_scan_item iterator)

Function: This function is used to obtain scan results and is often used in conjunction with the wifi_netlink_set_scan function.

Input parameter: iter_scan_item，the function pointer for traversal, and the developer implements the function to obtain the result of the scan, refer to ***iter_scan_item***.

Output parameter: None.

Return value:0 is returned after successful execution.

3.3.6. iter_scan_item

Function prototype: typedef void (*iter_scan_item) (struct wifi_scan_info *scan_item)

Function: Defines the function of type iter_scan_item.

Input parameter: wifi_scan_info struct pointer that holds a hotspot information.

Output parameter: None.

3.3.7. wifi_netlink_connect_req

Function prototype: int wifi_netlink_connect_req (uint8_t *ssid, uint8_t *password)

Function: This function is used to perform Wi-Fi connection operations. The AP to be connected is specified by the parameter.

Input parameter: ssid, pointer to the SSID of the AP to be connected.

password, pointer to the password of the AP to be connected.

Output parameter: None.

Return value: 0 is returned on success and -1 is returned on failure.

3.3.8. wifi_netlink_disconnect_req

Function prototype: int wifi_netlink_disconnect_req (void)

Function: This function is used to disconnect the Wi-Fi from the connected AP.

Input parameter: None.

Output parameter: None.

Return value: 0 is returned after successful execution.

3.3.9. wifi_netlink_status_get

Function prototype: int wifi_netlink_status_get (void)

Function: This function is used to obtain the current Wi-Fi status information.

Input parameter: None.

Output parameter: None.

Return value: 0 is returned after successful execution.

3.3.10. wifi_netlink_ipaddr_set

Function prototype: int wifi_netlink_ipaddr_set (uint8_t *ipaddr)

Function: This function is used to inform the IP address assigned by the AP after the Wi-Fi driver connects to the AP.

Input parameter: ipaddr, the pointer to the IP address.

Output parameter: None.

Return value: 0 is returned after successful execution.

3.3.11. wifi_netlink_ap_start

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

Function: This function is used to enable the soft AP mode. The soft AP configuration information is determined by parameters.

Input parameter: ssid, the pointer to the SSID of the soft AP to be configured.

password, the pointer to the password of the soft AP to be configured.

channel, the channel of the soft AP to be configured.

hidden, determine whether the SSID of the soft AP is broadcast. 0 indicates

that the SSID is broadcast, and 1 indicates that the SSID is not broadcast.

Output parameter: None.

Return value: 0 is returned on success and -1 is returned on failure.

3.3.12. wifi_netlink_channel_set

Function prototype: int wifi_netlink_channel_set (uint32_t channel, uint32_t bandwidth, uint32_t ch_offset)

Function: This function is used to set the working channel. The configuration information of the channel is determined by the parameters.

Input parameter: channel, channel number.

bandwidth, the bandwidth of the working channel. The bandwidth is determined by the enumerated type of CHANNEL_WIDTH. Currently, only CHANNEL_WIDTH_20 and CHANNEL_WIDTH_40 are available, as follows:

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

ch_offset, the location of the extension channel. The value of the sideband is determined by the enumeration type HT_SEC_CHNL_OFFSET, where the 20M bandwidth corresponds to HT_SEC_CHNL_OFFSET_NONE. The 40M bandwidth corresponds to HT_SEC_CHNL_OFFSET_ABOVE and HT_SEC_CHNL_OFFSET_BELOW, others are temporarily unavailable.

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

}HT_SEC_CHNL_OFFSET;

Output parameter: None.

Return value: 0 is returned on success and -1 is returned on failure.

3.3.13. wifi_netlink_ps_set

Function prototype: int wifi_netlink_ps_set (int ps_mode)

Function: This function is used to set the sleep mode.

Input parameter: ps_mode:

0: Both Wi-Fi and CPU turn off sleep mode.

1: Wi-Fi sleep mode + CPU sleep mode.

2: Wi-Fi sleep mode + CPU deep sleep mode

Output parameter: None.

Return value: 0 is returned after successful execution.

3.3.14. wifi_netlink_ps_get

Function prototype: int wifi_netlink_ps_get (void)

Function: This function is used to get the sleep state.

Input parameter: None.

Output parameter: None.

Return value: Sleep mode

0: Both Wi-Fi and CPU turn off sleep mode.

1: Wi-Fi sleep mode + CPU sleep mode.

2: Wi-Fi sleep mode + CPU deep sleep mode.

3.3.15. wifi_netlink_bss_rssi_get

Function prototype: int wifi_netlink_bss_rssi_get (void)

Function: This function is used to obtain the RSSI value of the connected AP.

Input parameter: None.

Output parameter: None.

Return value: The RSSI value is returned on success, and 0 is returned on failure.

3.3.16. wifi_netlink_ap_channel_get

Function prototype: int wifi_netlink_ap_channel_get (uint8_t *bssid)

Function: This function is used to obtain the channel where the AP resides.

Input parameter: bssid, the pointer to the bssid of the AP that needs to obtain the channel.

Output parameter: None.

Return value: The channel of the AP is returned on success, and 0 is returned on failure.

3.3.17. wifi_netlink_task_stack_get

Function prototype: int wifi_netlink_task_stack_get (void)

Function: This function is used to get the current free stack for each task.

Input parameter: None.

Output parameter: None.

Return value: 0 is returned after successful execution.

3.3.18. wifi_netlink_link_state_get

Function prototype: int wifi_netlink_link_state_get(void)

Function: This function is used to obtain the current Wi-Fi status.

Input parameter: None.

Output parameter: None.

Return value: Current Wi-Fi status, the status types are listed in the enumeration WIFI_NETLINK_STATUS_E.

3.3.19. wifi_netlink_linked_ap_info_get

Function prototype: int wifi_netlink_linked_ap_info_get (uint8_t *ssid, uint8_t *passwd, uint8_t *bssid)

Function: This function is used to obtain information about the connected AP, including the SSID, password, and BSSID.

Input parameter: None.

Output parameter: ssid, the obtained SSID is saved to this pointer.

passwd, the obtained password is saved to this pointer.

bssid, the obtained bssid is saved to this pointer.

Return value: 0 is returned after successful execution.

3.3.20. wifi_netlink_raw_send

Function prototype: int wifi_netlink_raw_send (uint8_t *buf, uint32_t len)

Function: This function can be used to send any data. The driver does not assemble the MAC header. The user needs to assemble the complete Wi-Fi data frame.

Input parameter: buf, the pointer to the data to be sent.

len, length of the data to be sent.

Output parameter: None.

Return value: Returns 1 on success and 0 on failure.

3.3.21. wifi_netlink_promisc_mode_set

Function prototype: int wifi_netlink_promisc_mode_set (uint32_t enable, void (*callback) (unsigned char*, unsigned short, signed char))

Function: This function is used to set promiscuous mode.

Input parameter: enable, 1 indicates that the promiscuous mode is enabled, and 0 indicates that the promiscuous mode is disabled.

void (*callback)(unsigned char*, unsigned short, signed char)), A callback function that handles the received packet, where unsigned char* is a pointer to data, unsigned short is the length of data, and signed char is the corresponding RSSI.

Output parameter: None.

Return value:0 is returned after successful execution.

3.3.22. wifi_netlink_promisc_mgmt_cb_set

Function prototype: int wifi_netlink_promisc_mgmt_cb_set (uint32_t filter_mask,

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

Function: Set the callback function for managing frames in promiscuous mode

Input parameter: filter_mask, Filter corresponds to the bit code of the management subframe, set 1 to receive processing, set 0 to filter.

void (*callback) (unsigned char*, int, signed char, int), A callback function that handles the received packet, where unsigned char* is a pointer to data, unsigned short is the length of data, and signed char is the corresponding

RSSI.

Output parameter: None.

Return value: 0 is returned after successful execution.

3.3.23. wifi_netlink_promisc_filter_set

Function prototype: int wifi_netlink_promisc_filter_set (uint8_t filter_type, uint8_t *filter_value)

Function: Set frame filtering in promiscuous mode.

Input parameter: filter_type

0: Filter frames whose length exceeds filter_value.

1: Filter frames with less energy than filter_value.

2: Filter a-mpdu frames whose length exceeds filter_value.

filter_value, threshold value

Output parameter: None.

Return value: 0 is returned after successful execution.

3.3.24. wifi_netlink_auto_conn_set

Function prototype: int wifi_netlink_auto_conn_set (uint8_t auto_conn_enable)

Function: This function is used to set whether the AP is automatically connected after startup.

Input parameter: auto_conn_enable,1 indicates enable and 0 indicates disable.

Output parameter: None.

Return value:0 is returned after successful execution.

3.3.25. wifi_netlink_auto_conn_get

Function prototype: uint8_t wifi_netlink_auto_conn_get (void)

Function: This function is used to check whether the AP is automatically connected upon startup.

Input parameter: None.

Output parameter: None.

Return value: The set value is returned after the execution succeeds. 0 is returned after the execution fails.

3.3.26. wifi_netlink_joined_ap_store

Function prototype: int wifi_netlink_joined_ap_store (void)

Function: This function is used to save connected AP information to the flash.

Input parameter: None.

Output parameter: None.

Return value: Return 0 for success, -1 for SSID length error, and -2 for password length error.

3.3.27. wifi_netlink_joined_ap_load

Function prototype: int wifi_netlink_joined_ap_load (void)

Function: This function is used to connect using saved AP information.

Input parameter: None.

Output parameter: None.

Return value: 0 is returned on success and -1 is returned on failure.

4. Wi-Fi Netif API

The header file for the API is SDK \ NSPE\WIFI_IOT \ network \ lwip-2.1.2 \ port \ wifi_netif.h.

4.1. Wi-Fi Lwip network interface API

4.1.1. wifi_netif_open

Function prototype: void wifi_netif_open (void)

Function: Register the Wi-Fi network interface in Lwip.

Input parameter: None.

Output parameter: None.

Return value: None.

4.1.2. wifi_netif_close

Function prototype: void wifi_netif_close (void)

Function: Close the Wi-Fi network interface in Lwip.

Input parameter: None.

Output parameter: None.

Return value: None.

4.1.3. wifi_netif_set_hwaddr

Function prototype: uint8_t wifi_netif_set_hwaddr (uint8_t *mac_addr)

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

Input parameter: mac_addr, a pointer to a 6-byte address group.

Output parameter: None.

Return value: Returns TRUE on success and FALSE on failure.

4.1.4. wifi_netif_get_hwaddr

Function prototype: uint8_t *wifi_netif_get_hwaddr (void)

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

Input parameter: None.

Output parameter: None.

Return value: the pointer to the MAC address of the network interface.

4.1.5. ip_addr_t *wifi_netif_get_ip

Function prototype: ip_addr_t *wifi_netif_get_ip (void)

Function: Obtain the ip address of the Wi-Fi network interface.

Input parameter: None.

Output parameter: None.

Return value: The pointer to the ip address of the network interface.

4.1.6. wifi_netif_set_ip

Function prototype: void wifi_netif_set_ip (ip_addr_t *ip, ip_addr_t *netmask, ip_addr_t *gw)

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

Input parameter: ip, the pointer to the IP address.

netmask, the pointer to the network mask.

gw, the pointer to the gateway address

Output parameter: None.

Return value: None.

4.1.7. wifi_netif_get_gw

Function prototype: ip_addr_t *wifi_netif_get_gw (void)

Function: Obtain the gateway address of the Wi-Fi network interface.

Input parameter: None.

Output parameter: None.

Return value: The pointer to the gateway address of the network interface.

4.1.8. wifi_netif_get_netmask

Function prototype: ip_addr_t *wifi_netif_get_netmask (void)

Function: Obtain the mask of the Wi-Fi network interface.

Input parameter: None.

Output parameter: None.

Return value: The pointer to the mask of the network interface.

4.1.9. wifi_netif_set_up

Function prototype: void wifi_netif_set_up (void)

Function: Enable the Wi-Fi network interface.

Input parameter: None.

Output parameter: None.

Return value: None.

4.1.10. wifi_netif_set_down

Function prototype: void wifi_netif_set_down(void)

Function: Disable the Wi-Fi network interface.

Input parameter: None.

Output parameter: None.

Return value: None.

4.1.11. wifi_netif_is_ip_got

Function prototype: int32_t wifi_netif_is_ip_got (void)

Function: Check whether an IP address has been configured for the Wi-Fi network interface.

Input parameter: None.

Output parameter: None.

Return value: 1, The ip address of the network interface has been set.

0, No ip address is set for the network interface.

4.1.12. wifi_netif_start_dhcp

Function prototype: void wifi_netif_start_dhcp (void)

Function: Use DHCP to obtain an IP address on the Wi-Fi network interface.

Input parameter: None.

Output parameter: None.

Return value: None.

4.1.13. wifi_netif_polling_dhcp

Function prototype: int32_t wifi_netif_polling_dhcp (void)

Function: Check whether DHCP polling is complete.

Input parameter: None.

Output parameter: None.

Return value: 1, DHCP succeeded; 0, DHCP failed.

4.1.14. wifi_netif_stop_dhcp

Function prototype: void wifi_netif_stop_dhcp (void)

Function: Stop DHCP.

Input parameter: None.

Output parameter: None.

Return value: None.

4.1.15. wifi_netif_set_ip_mode

Function prototype: void wifi_netif_set_ip_mode (uint8_t ip_mode)

Function: Set the address mode of the network interface. DHCP and static address mode are supported.

Input parameter: ip_mode,

IP_MODE_STATIC, static address mode.

IP_MODE_DHCP, DHCP, automatic acquisition mode.

Output parameter: None.

Return value: None.

4.1.16. wifi_netif_is_static_ip_mode

Function prototype: int32_t wifi_netif_is_static_ip_mode (void)

Function: Check whether the network interface is in static address mode.

Input parameter: None.

Output parameter: None.

Return value:1, static address mode; 0, not static address mode.

5. Wi-Fi Management API

This section describes the Wi-Fi Management connection management service API and the event loop component API.

5.1. Wi-Fi connection management service

The header file for the API is SDK \ NSPE\WIFI_IOT \ wifi \ wifi_management.h.

5.1.1. wifi_management_init

Function prototype: void wifi_management_init (void)

Function: Initialize Lwip, Netlink, etc, only need to call once

Input parameter: None.

Output parameter: None.

Return: None.

5.1.2. wifi_management_scan

Function prototype: int wifi_management_scan (uint8_t blocked)

Function: Start scanning the wireless network.

Input parameter: blocked,

1: blocks other operations.

0: does not block.

Output parameter: None.

Return value: 0, starting scan succeeded; -1, starting scan failed.

5.1.3. wifi_management_connect

Function prototype: int wifi_management_connect (uint8_t *ssid, uint8_t *password, uint8_t blocked)

Function: Start connecting to AP.

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

Password, AP password, 8 – 63 characters. If the encryption mode is Open, the value can be NULL.

blocked, 1: blocks other operations. 0: does not block.

Output parameter: None.

Return value: 0: success,

-1: Wi-Fi driver link is busy.

-2: The ssid parameter is NULL.

-3: ssid parameter error.

-4: password parameter error.

5.1.4. wifi_management_disconnect

Function prototype: int wifi_management_disconnect (void)

Function: Disconnect AP.

Input parameter: None.

Output parameter: None.

Return value:0, disconnect AP successfully; -1, disconnect AP failed.

5.1.5. wifi_management_sta_start

Function prototype: void wifi_management_sta_start (void)

Function: The SDK enters STA mode. If the current mode is soft AP mode, the soft AP function stops.

Input parameter: None.

Output parameter: None.

Return value: None.

5.1.6. wifi_management_ap_start

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

Function: Start the soft AP and the SDK enters the soft AP mode.

Input parameter: ssid: Soft AP network name,1 - 32 characters.

Password: Soft AP network password. The default encryption mode is WPA2-PSK.

Channel: Network channel where the soft AP resides,1-13.

Hidden: Whether to hide the ssid.0: Broadcast ssid. 1: Hide ssid.

Output parameter: None.

Return value: None.

5.1.7. wifi_management_ap_assoc_info

Function prototype: uint32_t wifi_management_ap_assoc_info (uint8_t *assoc_info)

Function: Obtain the client list associated with the soft AP.

Input parameter: None.

Output parameter: The buffer used to store the client MAC. The recommended size is MAX_STATION_NUM x ETH_ALEN.

Return value: Number of clients associated with soft hotspot.

5.1.8. wifi_management_block_wait

Function prototype: int wifi_management_block_wait ()

Function: Block the operation until the scan completes or the connection completes.

Input parameter: None.

Output parameter: None.

Return value: 0, success.

5.2. Wi-Fi event loop API

The header file for the API is SDK \ NSPE \ WIFI_IOT \ wifi \ wifi_eloop.h.

5.2.1. eloop_event_handler

Function prototype: typedef void (*eloop_event_handler) (void *eloop_data, void *user_ctx);

Function: Defines a function of type eloop_event_handler that is used to call back when a general event is triggered.

Input parameter: eloop_data, eloop context data for callbacks.

user_ctx, the user context data for callbacks.

Output parameter: None.

Return value: None.

5.2.2. eloop_timeout_handler

Function prototype: typedef void (*eloop_timeout_handler) (void *eloop_data, void *user_ctx);

Function: Defines a function of type eloop_timeout_handler that is used to call back when a timer timeout event occurs.

Input parameter: eloop_data, eloop context data for callbacks.

user_ctx, the user context data for callbacks.

Output parameter: None.

Return value: None.

5.2.3. eloop_init

Function prototype: int eloop_init(void)

Function: This function initializes a global event handling loop data.

Input parameter: None.

Output parameter: None

Return value: 0 is returned after successful execution. -1 is returned on failure.

5.2.4. eloop_event_register

Function prototype: int eloop_event_register (eloop_event_t event,

eloop_event_handler handler,

void *eloop_data, void *user_data)

Function: This function registers a function to handle the trigger event.

Input parameter: eloop_event_t event, Events that need to be handled after triggering.

handler, a callback function that handles the event after it is triggered.

eloop_data, the parameter to the callback function.

user_data, the parameter to the callback function.

Output parameter: None.

Return value: 0 is returned after successful execution. -1 is returned on failure.

5.2.5. eloop_event_unregister

Function prototype: void eloop_event_unregister (eloop_event_t event)

Function: This function aborts a handler function after an event is triggered, corresponding to eloop_event_register.

Input parameter: event, the event that aborted processing.

Output parameter: None.

Return value: None.

5.2.6. eloop_event_send

Function prototype: int eloop_event_send (eloop_event_t event)

Function: This function is used to send events to the queue to be processed.

Input parameter: event, the event to be sent.

Output parameter: None.

Return value: 0 is returned after successful execution. -1 is returned on failure.

5.2.7. eloop_message_send

Function prototype: int eloop_message_send (eloop_message_t message)

Function: This function is used to send messages to the queue to be processed.

Input parameter: message, the message to be sent.

Output parameter: None.

Return value: 0 is returned after successful execution. -1 is returned on failure.

5.2.8. eloop_timeout_register

Function prototype: int eloop_timeout_register(unsigned int msecs,

eloop_timeout_handler handler,

void *eloop_data, void *user_data)

Function: This function registers a function to handle the event timeout that is triggered.

Input parameter: msecs, timeout time, in milliseconds.

handler, the callback function that handles timeout events.

eloop_data, the parameter to the callback function.

user_data, the parameter to the callback function.

Output parameter: None.

Return value: 0 is returned after successful execution. -1 is returned on failure.

5.2.9. eloop_timeout_cancel

Function prototype: int eloop_timeout_cancel (eloop_timeout_handler handler,

void *eloop_data, void *user_data)

Function: This function is used to stop timers.

Input parameter: handler, callback function after a timeout that needs to be aborts.

eloop_data, the parameter to the callback function.

user_data, the parameter to the callback function.

Output parameter: None.

Return value: Returns the number of stop timers.

Note: If the value of eloop_data / user_data is ELOOP_ALL_CTX, it indicates all timeouts.

5.2.10. eloop_is_timeout_registered

Function prototype: int eloop_is_timeout_registered (eloop_timeout_handler handler,

void *eloop_data, void *user_data)

Function: This function detects 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; Return 0 if not registered.

5.2.11. eloop_run

Function prototype: void eloop_run (void)

Function: This function is used to start the event loop.

Input parameter: None.

Output parameter: None.

Return value: None.

5.2.12. eloop_terminate

Function prototype: void eloop_terminate (void)

Function: This function is used to abort the event handler thread.

Input parameter: None.

Output parameter: None.

Return value: None.

5.2.13. eloop_destroy

Function prototype: void eloop_destroy (void)

Function: This function frees all resources for the event loop.

Input parameter: None.

Output parameter: None.

Return value: None.

5.2.14. eloop_terminated

Function prototype: int eloop_terminated (void)

Function: This function is used to detect if an event loop exists.

Input parameter: None.

Output parameter: None.

Return value: Return 1 if existed; Return 0 if none existed.

5.3. Types of macros

5.3.1. Wi-Fi event type

WIFI_MGMT_EVENT_SCAN_DONE

WIFI_MGMT_EVENT_SCAN_FAIL

WIFI_MGMT_EVENT_CONNECT_SUCCESS

WIFI_MGMT_EVENT_CONNECT_FAIL

WIFI_MGMT_EVENT_DISCONNECT

WIFI_MGMT_EVENT_DHCP_SUCCESS

WIFI_MGMT_EVENT_DHCP_FAIL

WIFI_MGMT_EVENT_AUTO_CONNECT

5.3.2. Configure macros

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 polling times

WIFI_MGMT_DHCP_POLLING_INTERVAL // Interval for successful DHCP polling

WIFI_MGMT_LINK_POLLING_INTERVAL // Interval for polling Wi-Fi connection quality

WIFI_MGMT_TRIGGER_ROAMING_RSSI_THRESHOLD // Threshold value for triggering roaming signal quality

WIFI_MGMT_START_ROAMING_RSSI_THRESHOLD_1 // Candidate roaming AP exceeds average signal quality

WIFI_MGMT_START_ROAMING_RSSI_THRESHOLD_2 // Candidate roaming AP exceeds average signal quality

WIFI_MGMT_START_SCAN_THROTTLE_INTERVAL // Slow scan interval

WIFI_MGMT_START_SCAN_FAST_INTERVAL // Fast scan interval

6. Application examples

In chapter ***Wi-Fi SDK software startup process***, after the SDK is launched, developers can use the components for Wi-Fi application development. The following is a simple example of how to use the API of the component to scan the wireless network, connect the AP, start the soft AP and access Ali Cloud.

6.1. Scan the wireless network

6.1.1. Blocking mode scanning

In this example, after scan_wireless_network starts the scan, it blocks until the scan is complete and prints the scan result.

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

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

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

6.1.2. Non-blocking scanning

In this example, **scan_wireless_network** starts the scan and registers the scan completion event. After the event is triggered, get the scan results and print them.

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

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

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

6.2. Connect the AP

In this example, **wifi_connect_ap** connects to an AP whose name is “test” and password is 12345678, and monitors the connection success, failure, and disconnection events. After the connection is successful, wifi_netlink_store_joined_ap is called in **cb_connect_done** to save the name and network password of the connected AP to flash. After the system is restarted, the SDK automatically connects to the saved AP.

```
#include "app_cfg.h"
#include "app_type.h"
#include "bsp_inc.h"
#include "osal_types.h"
#include "wlan_debug.h"
#include "wrapper_os.h"
#include "debug_print.h"
#include "malloc.h"
#include "wifi_netlink.h"
#include "wifi_netif.h"
#include "wifi_management.h"
void cb_connect_done(void *eloop_data, void *user_ctx)
{
```

```
    printf("WIFI connect: connect AP \r\n");
    wifi_netlink_auto_conn_set(1);
    wifi_netlink_joined_ap_store();
    eloop_event_unregister(WIFI_MGMT_EVENT_CONNECT_SUCCESS);
}
void cb_connect_failed(void *eloop_data, void *user_ctx)
{
    printf("WIFI connect: connect AP failed\r\n");
    eloop_event_unregister(WIFI_MGMT_EVENT_CONNECT_FAIL);
}

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

6.3. Start the soft AP

In this example, **wifi_start_ap** starts a soft AP named "test" and **wifi_get_client** obtains the client list.

```
#include "app_cfg.h"
#include "app_type.h"
#include "bsp_inc.h"
#include "osal_types.h"
#include "wlan_debug.h"
#include "wrapper_os.h"
#include "debug_print.h"
```

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

6.4. Ali-Cloud access

This section uses Ali-Cloud IOT SDK iotkit-Embedd-3.2.0 as an example to describe how to use the Wi-Fi SDK API to adapt cloud services. iotkit-embedded-3.2.0 API needs to be adapted into three parts: Wi-Fi distribution network, system and SSL network communication, which are briefly introduced below.

6.4.1. System access

Ali-Cloud system access includes the functions listed below.The corresponding API can be found in chapter ***OSAL API***.

void *HAL_Malloc(uint32_t size);

void HAL_Free(void *ptr);

uint64_t HAL_UptimeMs(void);

void HAL_SleepMs(uint32_t ms);

void HAL_Srandom(uint32_t seed);

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

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

void *HAL_SemaphoreCreate(void);

void HAL_SemaphoreDestroy(void *sem);

void HAL_SemaphorePost(void *sem);

int HAL_SemaphoreWait(void *sem, uint32_t timeout_ms);

int HAL_ThreadCreate(
                    void **thread_handle,
                    void *(*work_routine)(void *),
                    void *arg,
                    hal_os_thread_param_t *hal_os_thread_param,
                    int *stack_used);

void *HAL_MutexCreate(void);

void HAL_MutexDestroy(void *mutex);

void HAL_MutexLock(void *mutex);

void HAL_MutexUnlock(void *mutex);
```

6.4.2. Wi-Fi distribution network

There are various Wi-Fi distribution modes supported by Ali-Cloud, which can be divided into two categories in principle. One is that the distribution network device sends out multicast frames or special management frames with encoded information, and the IoT device to be distributed switches to different channels to listen to the packets on the air port. When the IoT device receive enough encoded information to parse network names and passwords, it can connect to the wireless network. In the other type, the IoT device of the distribution network enable the soft AP, and the distribution network device connects to the soft AP to inform the IoT devices of the distribution network information, and the IoT device closes the soft AP and connect to the wireless network.

Table 6-1. Mapping table between Ali-Cloud SDK adaptation interface and Wi-Fi SDK API

Function	Ali-Cloud SDK adaptation interface	Wi-Fi SDK API
Set the Wi-Fi to work in Monitor mode and call the incoming callback function when an 802.11 frame is	HAL_Awss_Open_Monitor HAL_Awss_Close_Monitor	wifi_netlink_promisc_mode_set

Function	Ali-Cloud SDK adaptation interface	Wi-Fi SDK API
received.		
Set the Wi-Fi to a specified channel	HAL_Awss_Switch_Channel	wifi_netlink_channel_set
Set the Wi-Fi connect to a specified AP (Access Point).	HAL_Awss_Connect_Ap	wifi_management_connect
Whether Wi-Fi network is connected to the network.	HAL_Sys_Net_Is_Ready	wifi_netif_is_ip_got
Send raw 802.11 frames on the current channel at the basic data rate of 1Mbps.	HAL_Wifi_Send_80211_Raw_Frame	wifi_netlink_raw_send
Enable or disable the filtering of admin frames in Station mode.	HAL_Wifi_Enable_Mgmt_Frame_Filter	wifi_netlink_promisc_filter_set wifi_netlink_promisc_mgmt_cb_set
Obtain information about the connected AP（Access Point）.	HAL_Wifi_Get_Ap_Info	wifi_netlink_linked_ap_info_get
Open the current device hotspot and switch the device from Station mode to soft AP mode.	HAL_Awss_Open_Ap	wifi_management_ap_start
Close the current device hotspot and switch the device from SoftAP mode to Station mode.	HAL_Awss_Close_Ap	wifi_management_sta_start
Obtain the MAC address of the Wi-Fi network.	HAL_Wifi_Get_Mac	wifi_netif_get_hwaddr

6.4.3. SSL network communication

The SSL communication interfaces that Ali-Cloud needs to adapt are listed below. The Wi-Fi SDK migrates Mbedtls2.17.0 to directly invoke the API of Mbedtls in adapting Ali-Cloud SSL

interface. Developers can refer to their official documentation during use, also refer to SDK \ NSPE\WIFI_IOT \ cloud \ alicloud \ iotkit-embedded-3.2.0 \ lib_iot_sdk_src \ eng \ wrappers \ wrappers.c.

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

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

int32_t HAL_SSL_Destroy(uintptr_t handle);

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

6.4.4. Ali-Cloud access example

Refer to SDK \ NSPE \ WIFI_IOT \ cloud \ alicloud \ linkkit_example_solo.c.

7. Revision history

Table 7-1. Revision history

Revision No.	Description	Date
1.0	Initial Release	Dec.16, 2022

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.