Hi3861 V100 / Hi3861L V100 Wi-Fi Software

Development Guide

Issue 01

Date 2020-04-30

HiSilicon (Shanghai) Technologies Co., Ltd.

Address:	New R&D Center, 49 Wuhe Road, Bantian, Longgang District, Shenzhen 518129 P. R. China
Website:	https://www.hisilicon.com/en/
Email:	support@hisilicon.com

About This Document

Purpose

This document describes the functions and development processes of the Wi-Fi software station (STA) and SoftAP interfaces of Hi3861 V100 and Hi3861L V100.

Related Versions

The following table lists the product versions related to this document.

Product Name	Version
Hi3861	V100
Hi3861L	V100

Intended Audience

This document is intended for:

- Technical support engineers
- Software development engineers

Symbol Conventions

The symbols that may be found in this document are defined as follows.

Symbol	Description
⚠ DANGER	Indicates a hazard with a high level of risk which, if not avoided, will result in death or serious injury.
⚠ WARNING	Indicates a hazard with a medium level of risk which, if not avoided, could result in death or serious injury.

Symbol	Description
⚠ CAUTION	Indicates a hazard with a low level of risk which, if not avoided, could result in minor or moderate injury.
NOTICE	Indicates a potentially hazardous situation which, if not avoided, could result in equipment damage, data loss, performance deterioration, or unanticipated results. NOTICE is used to address practices not related to personal injury.
🕮 NOTE	Supplements the important information in the main text. NOTE is used to address information not related to personal injury, equipment damage, and environment deterioration.

Change History

Issue	Date	Change Description
01	2020-04-30	This issue is the first official release. ● In **1 Introduction**, the description of HSO is deleted, and the description of WPA SUPPLICANT is updated, the API description of each module is added. ● In **2.3 Precautions**, the precaution for properly configuring the number of initialized resources is updated. ● In **2.4 Programming Sample**, the code samples 1 and 2 are updated. ● In **Table 3-1** of **3.2 Development Process**, the descriptions of hi_wifi_set_bandwidth, hi_wifi_sta_set_reconnect_policy, hi_wifi_config_callback, hi_wifi_register_event_callback, hi_wifi_sta_advance_scan, and netifapi_dhcp_stop are added. The description of netifapi_dhcp_start is updated. **Development Process** is updated. ● In **3.3 Precautions**, the descriptions that the STA supports the 5 MHz or 10 MHz narrowband mode, scanning with specified parameters, parameters of the network to be connected, and registration event callback function are added. ● In **3.4 Programming Sample**, the code sample is updated. ● In **4.1 Overview**, the description of the SoftAP function is updated. ● In **Table 4-1** of **4.2 Development Process**, the descriptions of hi_wifi_set_bandwidth and netifapi_dhcps_stop are added. **Development Process** is updated. ● In **4.3 Precautions**, the precautions that the SoftAP supports the 5 MHz or 10 MHz narrowband mode, the SoftAP network parameters are not reset when the SoftAP is disabled, and the maximum number of associated users in SoftAP mode is limited are added. ● In **4.4 Programming Sample**, the code sample is updated. ● In **5.3 Precautions**, the precaution that the promiscuous mode consumes a large number of CPU and memory resources is added.

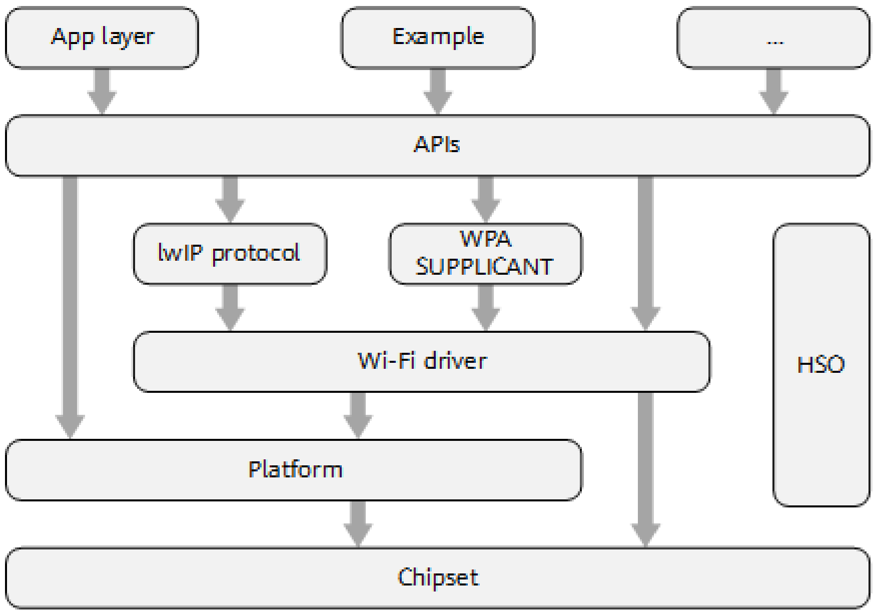
Issue	Date	Change Description
		● In **5.4 Programming Sample**, the code sample is updated. ● In **6.1 Overview**, the description of the PHY layer is deleted. ● In **6.3 Precautions**, the precautions for configuring the CSI sampling and reporting period and the CSI reporting conditions are added. ● In **6.4 Programming Sample**, the code sample is updated. ● **7 Coexistence of STA and SoftAP** is added. ● **8 Coexistence of Wi-Fi and Bluetooth** is added. ● The error codes in this document are changed to return values.
00B03	2020-03-25	● **5 Promiscuous Mode** is added. ● **6 CSI Data Collection** is added.
00B02	2020-03-19	**4.4 Programming Sample** is updated.
00B01	2020-01-15	This issue is the first draft release.

Contents

1 Introduction

Hi3861 V100 and Hi3861L V100 provide developers with application programming interfaces (APIs) for the development and application of Wi-Fi functions including chip initialization, resource configuration, station creation and configuration, scanning, association and disassociation, and status query, as shown in **Figure 1-1**.

Figure 1-1 API control flow of Hi3861/Hi3861L



The functional modules are described as follows:

- App layer: secondary development based on APIs
- Example: function development example of the SDK
- APIs: universal interfaces based on the SDK
- lwIP protocol stack: network protocol stack
- WPA SUPPLICANT (including HOSTAPD): Wi-Fi management module
- Wi-Fi driver: 802.11 protocol implementation module

- Platform: board support package for the system-on-a-chip (SoC), including the chip and peripheral drivers, operating system (OS), and system management

NOTE

This document describes the basic process of each module. For details about the APIs, see the *Hi3861 V100/Hi3861L V100 API Development Reference*.

2 Wi-Fi Driver Loading and Unloading

2.1 Overview

After the chip is powered on, a driver is loaded to implement the initial configuration of chip registers, read and write of calibration parameters, and application and configuration of software resources. Software resources are released when the driver is unloaded.

2.2 Development Process

Application Scenario

The Wi-Fi driver initialization, as the first step for implementing the Wi-Fi function, provides basic resource configuration and chip initialization. To configure the Wi-Fi function, you must initialize the driver first. After the Wi-Fi function is used, you can deinitialize the driver or soft reset the driver to release resources.

Function

Table 2-1 describes the APIs for loading and unloading the Wi-Fi driver.

Table 2-1 APIs for loading and unloading the Wi-Fi driver

API Name	Description
hi_wifi_init	Initializes the Wi-Fi driver.
hi_wifi_deinit	Deinitializes the Wi-Fi driver.

Development Process

The typical process of loading and unloading the Wi-Fi driver is as follows:

Step 1 Initialize the Wi-Fi driver by calling **hi_wifi_init**.

Step 2 Configure the Wi-Fi function by referring to **3 STA Function** or **4 SoftAP Function**.

Step 3 Initialize the Wi-Fi driver by calling **hi_wifi_deinit**.

----End

Return Value

Table 2-2 describes the return values when the Wi-Fi driver is loaded or unloaded.

Table 2-2 Return values for loading and unloading the Wi-Fi driver

No.	Definition	Actual Value	Description
1	HISI_OK	0	Initialization succeeded.
2	HISI_FAIL	–1	Initialization failed.

2.3 Precautions

- The driver resource configuration cannot be modified during running. You must unload the driver, modify the configuration, and then initialize the driver again.
- To ensure the continuity of Wi-Fi services, the driver pre-allocates memory based on the number of VAPs and users during startup. One VAP needs pre-allocation of about 5 KB memory, and one user needs pre-allocation of about 7 KB memory. Configure the number of initialized resources based on the scenario requirements. Currently, the SoftAP and STA need to coexist only during network configuration. It is recommended that two VAPs and two STAs be configured. If the SoftAP can be disabled before the STA is disassociated during network configuration, the number of VAPs and users can be both set to **1**.

2.4 Programming Sample

Sample 1: Based on the **app_main** function of Huawei LiteOS, the Wi-Fi driver is automatically loaded during system initialization. In this mode, the driver does not need to be unloaded during development. The driver is automatically unloaded and loaded during system reboot.

Sample code

```
#define APP_INIT_VAP_NUM    2
#define APP_INIT_USR_NUM    2

hi_void app_main(hi_void)
{
    hi_u32 ret;
    const hi_uchar wifi_vap_res_num = APP_INIT_VAP_NUM;
    const hi_uchar wifi_user_res_num = APP_INIT_USR_NUM;

    ret = hi_wifi_init(wifi_vap_res_num, wifi_user_res_num);
    if (ret != 0) {
        printf("fail to init wifi\n");
    } else {
        printf("wifi init success\n");
    }

}
```

Verification

```
#reboot
ready to OS start
system init status:0x0
FileSystem mount ok.
wifi init success
#
```

Sample 2: After system boot, run the **shell** command to load and unload the Wi-Fi driver.

Sample code

```
#define APP_INIT_VAP_NUM    2
#define APP_INIT_USR_NUM    2

void cmd_wifi_init(int argc, const char* argv[])
{
    hi_u32 ret;
    const hi_uchar wifi_vap_res_num = APP_INIT_VAP_NUM;
    const hi_uchar wifi_user_res_num = APP_INIT_USR_NUM;

    ret = hi_wifi_init(wifi_vap_res_num, wifi_user_res_num);
    if (ret != 0) {
        printf("fail to init wifi\n");
    } else {
        printf("wifi init success\n");
    }
}

void cmd_wifi_deinit(int argc, const char* argv[])
{
    int ret;

    ret = hi_wifi_deinit();
    if (ret != 0) {
        printf("fail to deinit wifi\n");
    } else {
        printf("deinit wifi success\n");
    }
}

void hisi_wifi_shell_cmd_register(void)
{
OsCmdReg(CMD_TYPE_EX, "wifi_init", XARGS, (CmdCallBackFunc)cmd_wifi_init);
OsCmdReg(CMD_TYPE_EX, "wifi_deinit", XARGS, (CmdCallBackFunc)cmd_wifi_deinit);
}

hi_void app_main(hi_void)
```

```
{
    hisi_wifi_shell_cmd_register();
}
```

Verification

```
#wifi_init
wifi init success
#wifi_deinit
deinit wifi success
#
```

3 STA Function

3.1 Overview

The station (STA) function, also the NON-AP station function, is used to implement the creation, scanning, association, and Dynamic Host Configuration Protocol (DHCP) of driver STA VAPs and establishes communication links. Before developing the STA function, the driver must be loaded.

3.2 Development Process

Application Scenario

To access a network and communicate with the network, the STA function must be enabled.

Function

Table 3-1 describes the APIs of the driver STA function.

Table 3-1 APIs of the driver STA function

API Name	Description
hi_wifi_sta_start	Starts the STA.
hi_wifi_set_bandwidth	Sets the STA bandwidth.
hi_wifi_sta_set_reconnect_policy	Sets the automatic reconnection of an STA.

API Name	Description
hi_wifi_config_callback	Configures the method of calling the event callback function.
hi_wifi_register_event_callback	Registers the event callback function of an STA.
hi_wifi_sta_scan	Triggers STA scanning.
hi_wifi_sta_advance_scan	Performs scanning with specific parameters.
hi_wifi_sta_scan_results	Obtains the STA scanning results.
hi_wifi_sta_connect	Enables the STA to connect to the Wi-Fi network.
hi_wifi_sta_get_connect_info	Obtains the status of the network connected to the STA.
netifapi_dhcp_start	Starts the DHCP client and obtain the IP address.
netifapi_dhcp_stop	Stops the DHCP client.
hi_wifi_sta_disconnect	Enables the STA to disconnect from the current network.
hi_wifi_sta_stop	Stops the STA.

Development Process

The typical process of developing the STA function is as follows:

Step 1 Start the STA by calling **hi_wifi_sta_start**.

Step 2 Set the STA bandwidth mode by calling **hi_wifi_set_bandwidth**. If the bandwidth is 20 MHz, it does not need to configured.

Step 3 (Optional) Set automatic reconnection by calling **hi_wifi_sta_set_reconnect_policy**.

Step 4 Trigger STA scanning by calling **hi_wifi_sta_scan** (or perform scanning with specified parameters by calling **hi_wifi_sta_advance_scan**).

Step 5 Obtain the scanning results by calling **hi_wifi_sta_scan_results**.

Step 6 Filter the scanning results based on the access network requirements and connect the STA to the Wi-Fi network by calling **hi_wifi_sta_connect**.

Step 7 Query the Wi-Fi connection status by calling **hi_wifi_sta_get_connect_info**.

Step 8 After the connection is successful, start the DHCP client and obtain the IP address by calling **netifapi_dhcp_start**.

Step 9 Disconnect from the current network by calling **hi_wifi_sta_disconnect**.

Step 10 Stop the DHCP client by calling **netifapi_dhcps_stop**.

Step 11 Stop the STA by calling **hi_wifi_sta_stop**.

----End

Return Value

Table 3-2 describes the return values of the STA function.

Table 3-2 Return values of the STA function

No.	Definition	Actual Value	Description
1	HISI_OK	0	Operation succeeded.
2	HISI_FAIL	–1	Operation failed.

3.3 Precautions

- An STA supports the 5 MHz or 10 MHz narrowband mode. The mode can be set by calling the corresponding API if required. If the API is not called, the STA with the 20 MHz bandwidth is started by default.
- The scanning API is implemented in non-blocking mode. After the scanning command is successfully delivered, the scanning result is obtained after a delay. It is recommended that the delay for full channel scan be set to 1s.
- You can specify the SSID, BSSID, and channel to implement more accurate scanning and shorten the scanning time.
- When the parameters of the network to be connected are known, the connection can be directly initiated by skipping the scanning process.
- The connection API is implemented in non-blocking mode. After the connection command is delivered successfully, you need to run the command to obtain the connection status.
- After an event callback function is registered, the Wi-Fi event is reported to the user through this callback function. You can perform subsequent actions based on the event.
- An STA must be stopped first before it is started again.
- The stopping of an STA is optional. If the network position of the device remains unchanged, the STA does not need to be stopped.

3.4 Programming Sample

Start, scan, and associate an STA, and obtain the IP address.

Sample code

```
#include "lwip/netifapi.h"
#include "hi_wifi_api.h"
#include "hi_types.h"
#include "hi_timer.h"

#define WIFI_IFNAME_MAX_SIZE             16
```

```
#define WIFI_SCAN_AP_LIMIT              64

hi_s32 example_get_match_network(hi_wifi_assoc_request *expected_bss)
{
    hi_s32  ret;
    hi_u32  num = 0; /*Number of scanned Wi-Fi networks*/
    hi_char expected_ssid[] = "my_wifi";
    hi_char key[] = "my_password"; /*Password of access to the network to be connected*/
    hi_bool find_ap = HI_FALSE;
    hi_u8   bss_index;
   /*Obtain the scanning results.*/
    hi_u32 scan_len = sizeof(hi_wifi_ap_info) * WIFI_SCAN_AP_LIMIT;
    hi_wifi_ap_info *pst_results = malloc(scan_len);
    if (pst_results == HI_NULL) {
        return HISI_FAIL;
    }
    memset_s(pst_results, scan_len, 0, scan_len);
    ret = hi_wifi_sta_scan_results(pst_results, &num);
    if (ret != HISI_OK) {
        free(pst_results);
        return HISI_FAIL;
    }
   /*Filter the scanned Wi-Fi networks and select the network to be connected. */
    for (bss_index = 0; bss_index < num; bss_index ++) {
        if (strlen(expected_ssid) == strlen(pst_results[bss_index].ssid)) {
            if (memcmp(expected_ssid, pst_results[bss_index].ssid, strlen(expected_ssid)) ==
HISI_OK) {
                find_ap = HI_TRUE;
                break;
            }
        }
    }
   /*The AP to be connected is not found. You can continue scanning or exit. */
    if (find_ap == HI_FALSE) {
        free(pst_results);
        return HISI_FAIL;
    }
   /*Copy the network information and access password after the network is found.*/
    if (memcpy_s(expected_bss->ssid, (HI_WIFI_MAX_SSID_LEN+1), expected_ssid,
strlen(expected_ssid)) != HISI_OK) {
        free(pst_results);
        return HISI_FAIL;
    }
    expected_bss->auth = pst_results[bss_index].auth;
   /* hidden_ssid is fixed at 1.*/
    expected_bss->hidden_ssid = 1;
   /* The pairwise mode is set to 0 by default.*/
    expected_bss->pairwise = 0;
    if (memcpy_s(expected_bss->key, (HI_WIFI_MAX_KEY_LEN+1), key, strlen(key)) != HISI_OK) {
        free(pst_results);
        return HISI_FAIL;
    }
    free(pst_results);
    return HISI_OK;
}

hi_bool example_check_connect_status(hi_void)
{
    hi_u8 index;
    hi_wifi_status wifi_status;
   /*Obtain the network connection status for five times at an interval of 500 ms. */
    for (index = 0; index < 5; index ++) {
        hi_udelay(500000);  /*500 ms delay*/
        memset_s(&wifi_status, sizeof(hi_wifi_status), 0, sizeof(hi_wifi_status));
        if (hi_wifi_sta_get_connect_info(&wifi_status) != HISI_OK) {
            continue;
        }
        if (wifi_status.status == HI_WIFI_CONNECTED) {
            return HI_TRUE; /* The connection is successful and the loop exits. */
        }
    }
    return HI_FALSE;
}

hi_s32 example_sta_function(hi_void)
{
    hi_char ifname[WIFI_IFNAME_MAX_SIZE + 1] = {0}; /* Name of the STA interface*/
    hi_s32  len = WIFI_IFNAME_MAX_SIZE + 1; /* Length of the STA interface name. +1 is used to
store the end character of the string.*/
    hi_wifi_assoc_request expected_bss = {0}; /* Connection request information*/
    struct netif *netif_p = HI_NULL;

    /*Create an STA interface.*/
    if (hi_wifi_sta_start(ifname, &len) != HISI_OK) {
        return HISI_FAIL;
    }
    /*Start STA scanning.*/
    if (hi_wifi_sta_scan() != HISI_OK) {
        return HISI_FAIL;
    }
hi_udelay(1000000); /* Wait for the scanning to complete 1s later. */
   /* Obtain the network to be connected. */
    if (example_get_match_network(&expected_bss) != HISI_OK) {
        return HISI_FAIL;
    }
    /*Start connection.*/
    if (hi_wifi_sta_connect(&expected_bss) != HISI_OK) {
        return HISI_FAIL;
    }
   /*Check whether the network connection is successful.*/
    if (example_check_connect_status() == HI_FALSE) {
        return HISI_FAIL;
    }
   /*Obtain the IP address for the DHCP.*/
    netif_p = netif_find(ifname);
    if (netif_p == HI_NULL) {
        return HISI_FAIL;
    }
    if (netifapi_dhcp_start(netif_p) != HISI_OK) {
        return HISI_FAIL;
    }
   /*The connection is successful.*/
    printf("STA connect success.\n");
    return HISI_OK;
}
```

Verification

```
#STA connect success.
#
```

4 SoftAP Function

4.1 Overview

The SoftAP function provides a network access point for other STAs to access, and provides the DHCP server for the accessed STA.

4.2 Development Process

Application Scenario

To create a network access point for other devices to access and share data on the network, you need to use the SoftAP function.

Function

Table 4-1 describes the APIs of the driver SoftAP function.

Table 4-1 APIs of the driver SoftAP function

API Name	Description
hi_wifi_softap_start	Starts the SoftAP.
hi_wifi_softap_set_protocol_mode	Sets the SoftAP protocol mode.
hi_wifi_softap_get_protocol_mode	Obtains the SoftAP protocol mode.

API Name	Description
hi_wifi_softap_set_beacon_period	Sets the beacon interval of the SoftAP.
hi_wifi_softap_set_dtim_period	Sets the DTIM interval of the SoftAP.
hi_wifi_set_bandwidth	Sets the bandwidth mode of the SoftAP.
netifapi_netif_set_addr	Sets the IP address, subnet mask, and gateway of the DHCP server of the SoftAP.
netifapi_dhcps_start	Starts the DHCP server of the SoftAP.
netifapi_dhcps_stop	Stops the DHCP server of the SoftAP.
hi_wifi_softap_get_connected_sta	Obtains information about the connected STA.
hi_wifi_softap_deauth_sta	Disconnects a specified STA.
hi_wifi_softap_stop	Stops the SoftAP.

Development Process

The typical process of developing the SoftAP function is as follows:

Step 1 Set the network parameters of the SoftAP.

- Set the protocol mode by calling **hi_wifi_softap_set_protocol_mode**.
- Set the beacon interval by calling **hi_wifi_softap_set_beacon_period**.
- Set the DTIM interval by calling **hi_wifi_softap_set_dtim_period**.

Step 2 Start the SoftAP by calling **hi_wifi_softap_start**.

Step 3 Set the SoftAp bandwidth mode by calling **hi_wifi_set_bandwidth**. If the bandwidth is 20 MHz, it does not need to configured.

Step 4 Configure the DHCP server by calling **netifapi_netif_set_addr**.

Step 5 Start the DHCP server by calling **netifapi_dhcps_start**.

Step 6 Stop the DHCP server by calling **netifapi_dhcps_stop**.

Step 7 Stop the SoftAP by calling **hi_wifi_softap_stop**.

----End

Return Value

Table 4-2 describes the return values of the SoftAP function.

Table 4-2 Return values of the SoftAP function

No.	Definition	Actual Value	Description
1	HISI_OK	0	Operation succeeded.
2	HISI_FAIL	–1	Operation failed.

4.3 Precautions

- The network parameters of the SoftAP are optional. You can use the default values unless otherwise specified.
- A SoftAP supports the 5 MHz or 10 MHz narrowband mode. The mode can be set by calling the corresponding API if required. If the API is not called, the SoftAP with the 20 MHz bandwidth is started by default.
- The network parameters of the SoftAP are not reset when the SoftAP is disabled. The previous configurations are inherited. The default values can be restored after the board is restarted.
- Restrictions on the maximum number of associated users in SoftAP mode:
  - The maximum number of associated users must be less than the number of users configured during initialization.
  - A maximum of two associated users are supported.

4.4 Programming Sample

Set the beacon interval of the SoftAP function to 200 ms, start the SoftAP, and set the IP address of the DHCP server to **192.168.43.1**.

Sample code

```
#include "lwip/netifapi.h"
#include "hi_wifi_api.h"
#include "hi_types.h"

#define WIFI_IFNAME_MAX_SIZE             16

hi_s32 example_softap_function(hi_void)
{
   /*SoftAP interface information*/
    hi_wifi_softap_config hapd_conf = {
        "my_wifi", "my_password", 1, 0, HI_WIFI_SECURITY_WPA2PSK,
HI_WIFI_PARIWISE_UNKNOWN};
    hi_char ifname[WIFI_IFNAME_MAX_SIZE + 1] = {0}; /* Name of the SoftAP interface*/
    hi_s32  len = WIFI_IFNAME_MAX_SIZE + 1; /* Length of the SoftAP interface name. +1 is used
to store the end character of the string.*/
    struct netif *netif_p = HI_NULL;
    ip4_addr_t   st_gw;
    ip4_addr_t   st_ipaddr;
    ip4_addr_t   st_netmask;
    IP4_ADDR(&st_gw, 192, 168, 43, 1);
    IP4_ADDR(&st_ipaddr, 192, 168, 43, 1);
    IP4_ADDR(&st_netmask, 255, 255, 255, 0);
```

```
 /*Configure the SoftAP network parameters and set the beacon interval to 200 ms.*/
  if (hi_wifi_softap_set_beacon_period(200) != HISI_OK) {
      return HISI_FAIL;
  }
 /*Start the SoftAP.*/
  if (hi_wifi_softap_start(&hapd_conf, ifname, &len) != HISI_OK) {
      return HISI_FAIL;
  }
 /*Configure the DHCP server.*/
  netif_p = netif_find(ifname);
  if (netif_p == HI_NULL) {
      (hi_void)hi_wifi_softap_stop();
      return HISI_FAIL;
  }
  if (netifapi_netif_set_addr(netif_p, &st_ipaddr, &st_netmask, &st_gw) != HISI_OK) {
      (hi_void)hi_wifi_softap_stop();
      return HISI_FAIL;
  }
  if (netifapi_dhcps_start(netif_p, NULL, 0) != HISI_OK) {
      (hi_void)hi_wifi_softap_stop();
      return HISI_FAIL;
  }
  printf("SoftAp start success.\n");
  return HISI_OK;
}
```

Verification

```
#SoftAp start success.
#
```

5 Promiscuous Mode

5.1 Overview

In promiscuous mode, the Wi-Fi module reports the received packets that meet the requirements to the user through the configured callback interface. In this way, the application layer receives air interface data to discover the peer end and implement network distribution.

5.2 Development Process

Application Scenario

To receive packets over the air interface, set the network interface to the promiscuous mode and configure the type of packets to be received.

Function

Table 5-1 describes the APIs of the promiscuous mode.

Table 5-1 APIs of the promiscuous mode

API Name	Description
hi_wifi_promis_set_rx_callback	Registers the callback interface for reporting data in promiscuous mode.
hi_wifi_promis_enable	Enables the promiscuous mode.

Development Process

The typical process of developing the promiscuous mode is as follows:

Step 1 Create a network interface. For details, see **3 STA Function** or **4 SoftAP Function**. You can create a network interface in either associated or non-associated state.

Step 2 Implement the RX callback interface for the promiscuous mode, and register the callback interface by calling **hi_wifi_promis_set_rx_callback**.

Step 3 Call **hi_wifi_promis_enable** to set the frame filtering parameter and set **enable** to **1** to enable the promiscuous mode.

Step 4 Call **hi_wifi_promis_enable** to set **enable** to **0** to disable the promiscuous mode.

----End

Return Value

Table 5-2 describes the return values of the promiscuous mode.

Table 5-2 Return values of the promiscuous mode

No.	Definition	Actual Value	Description
1	HISI_OK	0	Operation succeeded.
2	HISI_FAIL	–1	Operation failed.

5.3 Precautions

- In promiscuous mode, you need to create a network interface first. The STA interface can use the promiscuous mode regardless of whether it is associated or not.
- The promiscuous mode consumes a large number of CPU and memory resources, especially when there are a large number of networks. Therefore, enable the promiscuous mode as required.

5.4 Programming Sample

Enable the promiscuous mode and print the received packet information over the UART port.

Sample code

```
#include "hi_wifi_api.h"
#include "hi_types.h"

#define WIFI_IFNAME_MAX_SIZE             16
int g_recv_cnt = 0;
char ifname[WIFI_IFNAME_MAX_SIZE + 1] = ""; /* +1 is used to store the end character of a
string.*/
*****************************************************************************
```

```
Function description: Report the received packets in promiscuous mode and print the
information.
*******************************************************************************/
int hi_promis_recv(void* recv_buf, int frame_len, signed char rssi)
{
    unsigned int ret;
    hi_wifi_ptype_filter filter = {0};     /*Disable the promiscuous mode. The filter value is
meaningless and set to 0.*/

    g_recv_cnt ++;
    printf("resv buf: %u , len: %d , rssi: %c, cnt:%d.\r\n", (unsigned int)recv_buf, frame_len, rssi,
g_recv_cnt);
    /*Disable the promiscuous mode after receiving 1000 packets.*/
    if (g_recv_cnt == 1000) {
        ret = hi_wifi_promis_enable(ifname, 0, &filter);
        if (ret != HI_ERR_SUCCESS) {
            printf("hi_wifi_promis_enable:: set error!\r\n");
            return ret;
        }
        printf("stop promis SUCCESS!\r\n");
    }
    return HI_ERR_SUCCESS;
}

/*******************************************************************************
Function description: Create a STA, enable the promiscuous mode, receive all management and
data frames, and disable the promiscuous mode after receiving 1000 packets.
*******************************************************************************/
int main(hi_void)
{
    int len = WIFI_IFNAME_MAX_SIZE + 1;
    unsigned int ret;
    hi_wifi_ptype_filter filter = {0};

   /*Create an STA interface.*/
    if (hi_wifi_sta_start(ifname, &len) != HISI_OK) {
        return HISI_FAIL;
    }
    /*Enable the promiscuous mode and report unicast/multicast management frames and data
frames.*/
    filter.mdata_en = 1;
    filter.udata_en = 1;
    filter.mmngt_en = 1;
    filter.umngt_en = 1;
    hi_wifi_promis_set_rx_callback(hi_promis_recv);
    ret = hi_wifi_promis_enable(ifname, 1, &filter);
    if (ret != HI_ERR_SUCCESS) {
        printf("hi_wifi_promis_enable:: set error!\r\n");
        return ret;
    }
    printf("start promis SUCCESS!\r\n");
    return HI_ERR_SUCCESS;
}
```

6 CSI Data Collection

6.1 Overview

Channel state information (CSI) is used to measure channel conditions. It belongs to a subcarrier decoded in the orthogonal frequency division multiplexing (OFDM) system at the PHY layer. The CSI can represent a signal in a finer granularity. By separately analyzing signal transmission statuses of different subchannels, the CSI can avoid impact of multipath effect and noise as much as possible, so as to implement human body perception, fine positioning, and fine-grained action identification.

6.2 Development Process

Application Scenario

If the application layer needs to extract CSI for related applications, enable the CSI function.

Function

Table 6-1 describes the APIs of the CSI function.

Table 6-1 APIs of the CSI function

API Name	Description
hi_wifi_csi_set_config	Configures basic parameters for CSI reporting.

API Name	Description
hi_wifi_csi_register_data_recv_func	Registers the callback function for CSI reporting.
hi_wifi_csi_start	Starts CSI reporting.
hi_wifi_csi_stop	Stops CSI reporting.

Development Process

The typical process of developing the CSI function is as follows:

Step 1 Create a network interface. For details, see **3 STA Function** or **4 SoftAP Function**.

Step 2 Implement the CSI processing function and register the callback function by calling **hi_wifi_csi_register_data_recv_func**.

Step 3 Call **hi_wifi_csi_set_config** to configure parameters for CSI reporting.

Step 4 Call **hi_wifi_csi_start** to start CSI reporting.

Step 5 Call **hi_wifi_csi_stop** to stop CSI reporting.

----End

Return Value

Table 6-2 describes the return values of the CSI function.

Table 6-2 Return values of the CSI function

No.	Definition	Actual Value	Description
1	HISI_OK	0	Operation succeeded.
2	HISI_FAIL	–1	Operation failed.

6.3 Precautions

- The CSI function needs to be used in the associated state because the MAC address of the TX end needs to be verified.
- If the low-power mode has been enabled for the current network interface, the low-power mode will be disabled after the CSI is configured by calling **hi_wifi_csi_set_config**. If the CSI is not enabled temporarily, you can enable the low-power mode again after calling **hi_wifi_csi_stop**.
- The smaller the CSI sampling and reporting period, the greater the impact on system performance. The minimum reporting period is 50 ms. Therefore, it is recommended that the CSI sampling period be set to a value greater than 10. The reporting period needs to be set as required.

- CSI reporting conditions:
  - The physical layer completes sampling and successfully reports CSI data.
  - The interval between two reporting times is greater than the configured reporting period.
- The length of the CSI is fixed at 188 bytes. The storage format is 4-byte extended timestamp + 184-byte 64-bit little-endian storage data. For details about the data parsing rules, see **Table 6-3**.

Table 6-3 Description of CSI parsing

DWORD	Bit Field	Name	Description
DWORD 0	bit[31:0]	csi_timestamp[63:32]	CSI collection timestamp, upper 32 bits of the timestamp signal latched at **csi_start**
DWORD 1	bit[31:0]	csi_timestamp[31:0]	CSI collection timestamp, lower 32 bits of the timestamp signal latched at **csi_start**
DWORD 2	bit[31:30]	reserved	Reserved
	bit[29:25]	rx_vap_index	VAP that receives the frame 0: AP0 4: STA0 5: STA1
	bit[24:16]	Antenna AGC code	Antenna AGC code
	bit[15:10]	csi_nss_mcs_rate	MCS rate of the frame corresponding to the collected CSI
	bit[9:6]	csi_frame_type	Type of the frame corresponding to the collected CSI 00: management frame 01: control frame 10: data frame
	bit[5:4]	csi_frame_sub_type	Subtype of the frame corresponding to the collected CSI. For details, see the definition of the 802.11 subframe type.
	bit[3:0]	csi_protocol_mode	Protocol of the frame corresponding to the collected CSI. Upper 2 bits are **0**.
DWORD 3	bit[31:0]	csi_ta	Lower 32 bits of the TX MAC address of the frame corresponding to the collected CSI

DWORD	Bit Field	Name	Description
DWORD 4	bit[29:17]	csi_phase_incr_ant	Antenna frequency offset
	bit[16:8]	csi_snr_ant	Antenna SNR
	bit[7:0]	csi_rssi_ant	Antenna RSSI
DWORD 5	bit[31:0]	csi_ra	Lower 32 bits of the RX MAC address of the frame corresponding to the collected CSI
DWORD 6	bit[31:16]	csi_ta[47:32]	Upper 16 bits of the TX MAC address of the frame corresponding to the collected CSI
	bit[15:0]	csi_ra[47:32]	Upper 16 bits of the RX MAC address of the frame corresponding to the collected CSI
DWORD N (N = 0–12)	bit[31:0]	csi_h_iq	[23:0]: bit[23:0] of H matrix 4xN +0 information [31:24]: bit[7:0] of H matrix 4xN +1 information
DWORD N (N = 0–12)	bit[31:0]	csi_h_iq	[15:0]: bit[23:8] of H matrix 4xN +1 information [31:16]: bit[15:0] of H matrix 4xN+2 information
DWORD N (N = 0–12)	bit[31:0]	csi_h_iq	[7:0]: bit[23:16] of H matrix 4xN +2 information [31:8]: bit[23:0] of H matrix 4xN +3 information

6.4 Programming Sample

Enable CSI reporting and print the received packet information over the UART port.

Sample code

```
/*****************************************************************************
Function description: Print the reported CSI.
*****************************************************************************/
void at_hi_wifi_csi_cb(unsigned char *csi_data, int len)
{
    printf("==========csi data======start======\n");
    for (int i=0; i<len; i++) {
        if(i%23 == 0 && i != 0){
            printf("\n");
        }
        printf("%02x ", csi_data[i]);
    }
    printf("\n==========csi data======end======\n");
}

/********************************************************************************
Function description: Enable CSI data reporting after an STA is associated.
********************************************************************************/
int main(hi_void)
{
    hi_wifi_csi_register_data_recv_func(at_hi_wifi_csi_cb);
    char ifname[] = "wlan0";    /* STA */
    unsigned int interval = 100; /* report interval:100ms */
    hi_wifi_csi_entry csi_entry = {{00,00,00,00,00,00}, 7, 12}; /* mac need change to ap's mac */
    if(hi_wifi_csi_set_config(ifname, interval, &csi_entry, 1) != 0) {

        printf("set csi config failed.\r\n");
        return -1;
    }
    hi_wifi_csi_start();
    return 0;
}
```

7 Coexistence of STA and SoftAP

7.1 Overview

When the STA and SoftAP coexist, both STA and SoftAP functions are enabled. Based on the working channel and bandwidth, the coexistence modes of STA and SoftAP functions are as follows:

- Mode 1: same frequency and same bandwidth
- Mode 2: same frequency and different bandwidths
- Mode 3: different frequencies and same bandwidth
- Mode 4: different frequencies and different bandwidths

In mode 1, the STA and SoftAP use device resources based on the service sequence. The STA and SoftAP are always online, which is a full-time coexistence mode. In other three coexistence modes, the coexistence algorithm needs to be enabled to schedule timeslots to use the STA and SoftAP functions in time-division mode. The two functions work in their respective timeslots, which is a time-division mode. Currently, the time-division mode supports only the average time-division mode. That is, the STA and SoftAP have the same working timeslots, and work alternately.

7.2 Development Process

Application Scenario

During network configuration, the product starts the SoftAP. After a mobile phone associates with the SoftAP, it sends the SSID and password of the home network to the product. After obtaining the connection parameters of the home network, the product starts the STA to associate with the home network. After the product is connected to the network, the SoftAP is disabled. Retain only the STA as the device to maintain the connection for a long time. In other coexistence scenarios, you can use this feature based on the product form and requirements.

Function

The coexistence function uses the APIs of the STA function and SoftAP function. No additional API is required.

Development Process

The typical process of developing the coexistence function in network configuration mode is as follows:

Step 1 Create a SoftAP network interface. For details, see **4 SoftAP Function**.

Step 2 Associate a mobile phone with the SoftAP and send the SSID and password of the home network through the mobile phone app.

Step 3 Create a STA network interface and associate it with the SSID and password. For details, see **3 STA Function**.

Step 4 Stop the SoftAP. For details, see **4 SoftAP Function**.

----End

Return Value

For details about the return values, see the return value description of the corresponding module function.

7.3 Precautions

In time-division coexistence mode, the STA and SoftAP use timeslots in turn. Even if the other module has no data, the same timeslots are obtained. Therefore, the performance is poor in time-division coexistence mode. Therefore, in long-term service scenarios, the time-division coexistence mode is not recommended. Instead, enable the SoftAP to work on the channel where the STA works and use the full-time coexistence mode.

7.4 Programming Sample

For details, see the programming samples of the STA and SoftAP functions. For details, see **3 STA Function** or **4 SoftAP Function**.

8 Coexistence of Wi-Fi and Bluetooth

8.1 Overview

Bluetooth (BT) and Wi-Fi may both work on the 2.4 GHz ISM band, and therefore may interfere with each other. Time division is to use handshake signals between Bluetooth and Wi-Fi to enable Bluetooth and Wi-Fi to work at 2.4 GHz in a time-division manner, so as to avoid noise interference and blocking interference.

IEEE 802.15.2 specifies the arbitration mode and packet traffic arbitration (PTA) framework. When Bluetooth or Wi-Fi is used to receive and send services, an application is submitted to the PTA controller (integrated in Wi-Fi) for permission.

The handshake signals between Bluetooth and Wi-Fi are defined as follows:

- Two-wire coexistence: wlan_active and bt_status
  - Wi-Fi sends the wl_active signal to the PTA: Wi-Fi has transmission and reception services.
  - Bluetooth sends the bt_status signal to the PTA: Bluetooth has high-priority services. Low-priority Bluetooth service access is delayed.
- Three-wire coexistence: wlan_active, bt_active, and bt_status
  - Wi-Fi sends the wl_active signal to the PTA: Wi-Fi has transmission and reception services.
  - Bluetooth sends the bt_active signal to the PTA: Bluetooth has services.
  - Bluetooth sends the bt_status signal to PTA: Bluetooth has services. The bt_status signal is multiplexed as the Bluetooth priority signal.

8.2 Development Process

Application Scenario

Enable Wi-Fi and Bluetooth coexistence when both Wi-Fi and Bluetooth are required.

Function

Table 8-1 describes the Wi-Fi and Bluetooth coexistence API.

Table 8-1 Description of the Wi-Fi and Bluetooth coexistence API

API Name	Description
hi_wifi_btcoex_enable	Enables Wi-Fi and Bluetooth coexistence.

Return Value

Table 8-2 describes the return values of the Wi-Fi and Bluetooth coexistence function.

Table 8-2 Return values of the Wi-Fi and Bluetooth coexistence function

No.	Definition	Actual Value	Description
1	HISI_OK	0	Operation succeeded.
2	HISI_FAIL	–1	Operation failed.

Development Process

The typical process of developing the Wi-Fi and Bluetooth coexistence function is as follows:

Step 1 Create a STA network interface. For details, see **3 STA Function**.

Step 2 Enable Wi-Fi and Bluetooth coexistence by calling **hi_wifi_btcoex_enable()** and use both Wi-Fi and Bluetooth.

Step 3 Enable Wi-Fi and Bluetooth coexistence by calling **hi_wifi_btcoex_enable** and use only the Wi-Fi function.

----End

8.3 Precautions

- The coexistence of Wi-Fi and Bluetooth supports only the STA mode, but not the SoftAP mode.

- The module or product integrates both the Wi-Fi chip and Bluetooth chip. It is recommended that the Wi-Fi and Bluetooth coexistence function be always enabled. If the coexistence function needs to be dynamically enabled based on services, design the API calling policy based on the service scenario.

8.4 Programming Sample

Sample: Enable or disable Wi-Fi and Bluetooth coexistence by calling the **main** function.

```
unsigned int main()
{
   const char *ifname = "wlan0"; /* Wi-Fi STA device name*/
    unsigned char en        = 1;   /* Enable Wi-Fi and Bluetooth coexistence.*/
    unsigned char mode     = 2;    /* Three-wire coexistence mode*/
    unsigned char share_ant = 1;  /* Wi-Fi and BT share the antenna.*/
    unsigned char preempt = 1; /* Send null frames for Wi-Fi and Bluetooth coexistence.*/
    unsigned int ret;
    /*Enable Wi-Fi and Bluetooth coexistence.*/
    ret = hi_wifi_btcoex_enable(ifname, en, mode, share_ant, preempt);
    if (ret != HISI_OK) {
        printf("enable btcoex failed.\n");
    } else {
        printf("enable btcoex success.\n");
    }
    return ret;
}

unsigned int main()
{
    const char *IFNAME = "wlan0";
   /*Disable Wi-Fi and Bluetooth coexistence.*/
    return hi_wifi_btcoex_enable(ifname, 0, 0, 0, 0);
}
```

Verification

```
#enable btcoex success.
#
```