WLAN API Reference Guide

1. Overview

The WF88-M SDK provides a comprehensive software development environment for the ACH118x series Wi-Fi modules. It abstracts the complexities of the underlying Wi-Fi stack (802.11 b/g/n) protocols, allowing developers to focus on building application logic.

Key Features:

- RTOS Foundation: Built on FreeRTOS, enabling standard real-time operating system features such as multi-tasking, semaphores, queues, and software timers.
- Connectivity: Simplified APIs for Station (STA), Access Point (AP), and Mesh networking modes, along with a full networking stack (LwIP).
- Peripheral Access: Drivers for UART, SPI, GPIO hardware interfaces.
- Application Services: Integrated support for MQTT, and Security protocols (SSL/TLS).

Development Environment:

- Compiler: IAR Embedded Workbench for ARM (v7.2 or higher) is required for compilation.
- Tools: The "Amped RF Term(http://download.ampedrftech.com/)" utility is provided for firmware flashing and serial debugging.
- Resources: The user application has access to approximately 45KB of Flash memory for code storage.

Contents

1. Overview 0
2. Module APIs 4
2.1 Signal Framework 4
2.1.1 Subscription Modes & Workflows 4
2.1.2 Data Structures 6
2.1.3 APIs 9
2.1.4 Usage Example 9
2.2 Common Return Codes 10
2.3 Connect APIs 11
2.3.1 wlan_sta_scan() 11
2.3.2 wlan_sta_join() 12
2.3.3 wlan_sta_status() 14
2.3.4 wlan_sta_unjoin() 14
2.3.5 wlan_sta_connect() 14
2.3.6 wlan_sta_disconnect() (Unsupported) 15
2.3.7 wlan_ap_showNetwork() (Unsupported) 16
2.4 ~~Transmit APIs~~ (Unsupported) 16
2.4.1 ~~wlan_sta_send()~~ (Unsupported) 16
2.5 Configuration APIs 17
2.5.1 wlan_get_device_name() 18
2.5.2 wlan_set_device_name() 18
2.5.3 wlan_get_mac_address() 19
2.5.4 wlan_set_mac_address() 19
2.5.5 wlan_get_dhcp_mode() 20
2.5.6 wlan_set_dhcp_mode() 20
2.5.7 wlan_get_ip_info() 21
2.5.8 wlan_set_ip_info() 22
2.5.9 wlan_get_sleep_mode() (Unsupported) 23
2.5.10 ~~wlan_set_sleep_mode()~~ (Unsupported) 23
2.5.11 wlan_get_operation_mode() 24
2.5.12 wlan_set_operation_mode() 24
2.5.13 wlan_config_info() 25

2.5.14 wlan_get_config_byID() 31
2.6 System APIs 32
2.6.1 wlan_restart() 32
2.6.2 wlan_printf() 32
2.7 Driver APIs 33
2.7.1 wlan_set_uart_connect_mode() (Unsupported) 33
2.7.2 wlan_uart_send() 34
2.7.3 wlan_set_uart_baudrate() (Unsupported) 34
2.7.4 ~~wlan_spi_init()~~ (Unsupported) 35
2.7.5 ~~wlan_spi_snd_bytes()~~ (Unsupported) 35
2.7.6 wlan_spi_sndrcv_bytes() (Unsupported) 35
2.7.7 ~~wlan_spi_rcv_bytes()~~ (Unsupported) 36
2.7.8 wlan_gpio_config() 36
2.7.9 wlan_gpio_set() 37
2.7.10 wlan_gpio_get() 37
2.8 Timer APIs 37
2.8.1 wlan_timer_config() 38
2.8.2 wlan_timer_start() 39
2.8.3 wlan_timer_stop() 39
2.8.4 wlan_timer_destroy() 39
2.8.5 Example: Timer Usage 40
2.9 MQTT APIs 41
2.9.1 wlan_mqtt_set_server_ip() 42
2.9.2 wlan_mqtt_set_server_port() 42
2.9.3 wlan_mqtt_set_client_id() 43
2.9.4 wlan_mqtt_set_keep_alive() 43
2.9.5 wlan_mqtt_set_username() 44
2.9.6 wlan_mqtt_set_password() 44
2.9.7 wlan_mqtt_set_pub_topic() 45
2.9.8 wlan_mqtt_set_sub_topic() 45
2.9.9 wlan_mqtt_set_qos() 46
2.9.10 wlan_mqtt_set_auth_type() 47
2.9.11 wlan_mqtt_set_ca_cert() 47

2.9.12 wlan_mqtt_set_client_cert() 48
2.9.13 wlan_mqtt_set_client_key() 48
2.9.14 wlan_mqtt_connect() 49
2.9.15 wlan_mqtt_disconnect() 50
2.9.16 wlan_mqtt_get_status() 50
2.9.17 Monitoring MQTT Connection 51
2.9.18 wlan_mqtt_publish() 51
2.9.19 wlan_mqtt_subscribe() 52
2.9.20 wlan_mqtt_unsubscribe() 52
2.9.21 MQTT Connection Sequence 53
2.10 Mesh Networking Overview & Configuration 55
2.10.1 Overview 55
2.10.2 Key Configuration Variables 55
2.10.3 wlan_mesh_get_status() 56
2.10.4 Monitoring Mesh Connection 56
2.10.5 Configuration Workflow 57
2.10.6 Example: Mesh Connection 57
3. Demo Application Implementation Guide 58
3.1 Overview 59
3.2 Implementation Logic 59
3.2.1 Initialization 59
3.2.2 Subscribing to Signals 59
3.2.3 The Event Loop 60
3.3 Signal Processing (`msg_processed_for_test`) 60

2. Module APIs

2.1 Signal Framework

In the WF88-M SDK, many operations (such as Wi-Fi scanning, Wi-Fi joining, mesh joining, and MQTT messaging) are completed asynchronously. If applications rely on polling, they add latency and complexity. For this reason, the SDK exposes a unified Signal Framework so your application can react to key events at the right time. This framework is the basis for later sections such as Connect, MQTT, and Mesh.

This section describes the Signal Framework and its Publish-Subscribe model for delivering system events (e.g., Wi-Fi status changes, MQTT messages, UART data) to your application.

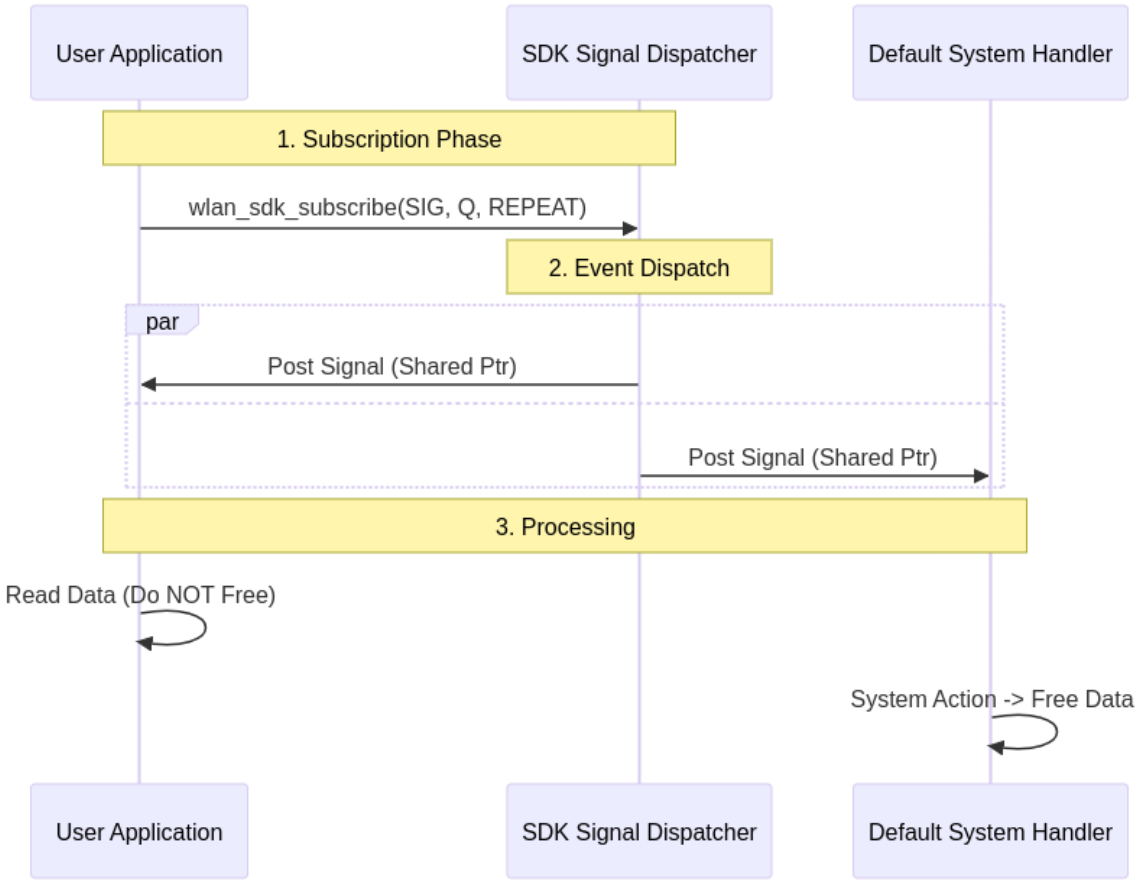
The Signal Framework utilizes FreeRTOS Queues to dispatch events. The application creates a queue and subscribes to specific event IDs. When an event occurs, the SDK posts a sdk_signal_t message to the registered queue, which the application task can then process in an event loop.

2.1.1 Subscription Modes & Workflows

The SDK supports two distinct subscription modes, determining how signals are routed to the application and the system's default handlers.

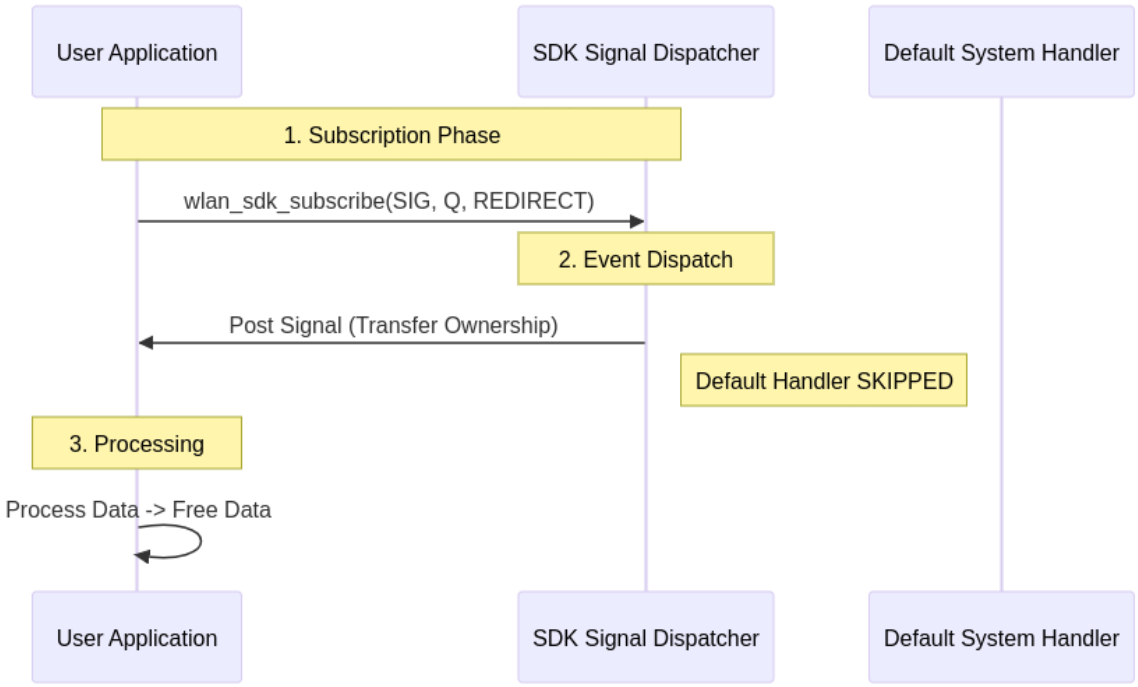
2.1.1.1 REPEAT Mode

- Description: The signal is delivered to the subscriber's queue, and the system also routes it to the default system handler. Both share the same payload pointer.
- Use Case: Non-intrusive monitoring (e.g., logging WiFi status) where the system must still perform its standard operations.
- Memory Management: Do NOT free `msg.data`(see struct sdk_signal_t). The system's default handler owns the data and will free it after processing. ***Freeing it in the application will cause a system crash.***
- Flow: `Event -> User Queue` (App reads) -> `System Queue` (System reads & frees)



2.1.1.2 REDIRECT Mode

- Description: The signal is delivered ONLY to the subscriber's queue. The default system handler is skipped.
- Use Case: Taking full control of a function (e.g., custom MQTT handling), effectively disabling the default system behavior for that event.
- Memory Management: ***App MUST free*** `msg.data`(see struct sdk_signal_t). Since the system handler is skipped, ownership of the allocated payload is transferred entirely to the application.
- Flow: `Event -> User Queue` (System Skipped)



2.1.2 Data Structures

2.1.2.1 struct sdk_signal_t

Represents the message object sent to the application queue.

Field	Type	Description
id	uint16_t	Unique identifier for the event (e.g., SIG_SDK_WLAN_CONNECTED).
source	uint8_t	Reserve For Use
data	void*	Pointer to the event payload (specific structure depends on the Signal ID).
data_len	uint16_t	Length of the payload data in bytes.

2.1.2.2 Signal IDs (Event List)

This section details the available system signals, their trigger conditions, and associated payload data structures.

The following table summarizes the available signals:

Signal ID	Description
SIG_SDK_SYS_INIT_DONE	System initialization is complete.
SIG_SDK_JOIN_SUCESS	Asynchronous AP connection request succeeded (Associated).
SIG_SDK_JOIN_FAILED	Asynchronous AP connection request failed (Timeout/Auth).

Signal ID	Description
SIG_SDK_WLAN_CONNECTED	Wi-Fi link established (Handshake complete) or Mesh joined.
SIG_SDK_WLAN_DISCONNECTED	Wi-Fi link lost or terminated.
SIG_SDK_WLAN_IP_CHANGED	IP address assigned or updated.
SIG_SDK_MQTT_CONNECT_OK	MQTT Broker connected.
SIG_SDK_MQTT_DISCONNECT	MQTT Broker disconnected.
SIG_SDK_MQTT_DATA_DOWN	MQTT message received.
SIG_SDK_UART_DATA_RX	UART data received.

2.1.2.2.1 SIG_SDK_SYS_INIT_DONE

- Description: Notifies the application that the SDK initialization is fully complete and application-layer logic processing can now begin.
- Trigger: Triggered exactly once when the system startup sequence is finished.
- Payload: `NULL`

2.1.2.2.2 SIG_SDK_JOIN_SUCESS

- Description: (STA Mode Only) Indicates that the asynchronous wlan_sta_join() request has successfully completed and the device has associated with the specified Access Point.
- Trigger: Triggered after calling wlan_sta_join() when the authentication and association handshake with the AP is successful.
- Payload: `NULL`
- Note: This signal confirms the initial connection. For subsequent link maintenance, monitor `SIG_SDK_WLAN_DISCONNECTED`.

2.1.2.2.3 SIG_SDK_JOIN_FAILED

- Description: (STA Mode Only) Indicates that the asynchronous wlan_sta_join() request has failed to establish a connection.
- Trigger: Triggered after calling wlan_sta_join() if the connection attempt fails for any reason, such as authentication failure (incorrect credentials), Access Point not found, or internal timeout.
- Payload: `NULL`
- Note: The SDK does NOT retry automatically. The application layer must explicitly call wlan_sta_join() again to retry.

2.1.2.2.4 SIG_SDK_WLAN_CONNECTED

- Description: Indicates that the Wi-Fi connection has been established.
- Trigger:
    - STA Mode: Triggered after successfully establishing a connection and completing the handshake with an Access Point.
    - Mesh Point (MP) Mode: Triggered only after the device has successfully joined the mesh network and established a connection to the Root Node. Note: This signal will not be triggered if the device connects to a mesh network that does not currently have a root node.

- Payload: `NULL`

2.1.2.2.5 SIG_SDK_WLAN_DISCONNECTED

- Description: Indicates that the Wi-Fi connection has been lost or terminated.
- Trigger:
    - STA Mode: Triggered when the association with the Access Point is terminated (e.g., due to beacon loss, de-authentication by AP, or manual disconnection).
    - Mesh Point (MP) Mode: Triggered when the device loses its connection to the mesh network or its path to the Root Node is severed.
- Payload: `NULL`

2.1.2.2.6 SIG_SDK_WLAN_IP_CHANGED

- Description: The network interface's IP address has been assigned, updated, or lost.
- Trigger: DHCP lease obtained/renewed
- Payload: `NULL` (Use wlan_get_ip_info() to retrieve the current IP configuration).

2.1.2.2.7 SIG_SDK_MQTT_CONNECT_OK

- Description: The MQTT client has successfully established a connection with the Broker.
- Trigger: Received `CONNACK` with success code from the Broker.
- Payload: `NULL`

2.1.2.2.8 SIG_SDK_MQTT_DISCONNECT

- Description: The MQTT client connection has been terminated.
- Trigger: TCP connection loss, keep-alive timeout, or disconnection initiated by the Broker.
- Payload: `NULL`

2.1.2.2.9 SIG_SDK_MQTT_DATA_DOWN

- Description: A message has been received on a subscribed MQTT topic.
- Trigger: Broker publishes a message to the device.
- Payload: Pointer to the message data.

2.1.2.2.10 SIG_SDK_UART_DATA_RX

- Description: Data has been received via a UART interface.
- Trigger: UART receive interrupt or DMA transfer completion.
- Payload: `app_uart_rx_event_t *` (Refer to the definition below).

2.1.2.2.10.1 Struct app_uart_rx_event_t Definition

Member	Type	Description
`len`	`uint16_t`	The number of bytes received in this event.
`data`	`uint8_t[0]`	Flexible array member pointing to the received data buffer.

```
typedef struct {
    uint16_t    len;
```

```
    uint8_t     data[0];
} app_uart_rx_event_t;
```

2.1.3 APIs

2.1.3.1 wlan_sdk_subscribe()

Registers a FreeRTOS queue to receive specific system signals.

Prototype:

```
int wlan_sdk_subscribe(int signal_id, QueueHandle_t queue, int mode)
```

Returns	Parameters
0: Success **Non-zero**: Error	**signal_id**: The Signal ID of the event to subscribe to. **queue**: Handle of the FreeRTOS queue to receive messages. **mode**: Subscription mode (see below).

Subscription Modes:

- `SDK_SUB_MODE_REPEAT`: The application receives the signal, and the SDK's internal default handler continues to process it. App must NOT free `msg.data`. see REPEAT Mode.
- `SDK_SUB_MODE_REDIRECT`: The application receives the signal, but the SDK's default internal handler is skipped. App MUST free `msg.data`. See REDIRECT Mode.

2.1.4 Usage Example

The following example demonstrates how to create an application task, subscribe to system events, and process them in a loop.

```
void vAppMainTask(void *pvParameters)
{
    sdk_signal_t msg;

    // 1. Create a Queue to receive signals (Depth 20 recommended)
    QueueHandle_t xAppQueue = xQueueCreate(20, sizeof(sdk_signal_t));
    if (xAppQueue == NULL) {
        printf("Error: Failed to create App Queue.\n");
        return;
    }

    // 2. Subscribe to interested signals
    // Using REPEAT mode ensures the SDK continues standard background oper
ations
    wlan_sdk_subscribe(SIG_SDK_SYS_INIT_DONE, xAppQueue, SDK_SUB_MODE_REPEA
T);
    wlan_sdk_subscribe(SIG_SDK_WLAN_IP_CHANGED, xAppQueue, SDK_SUB_MODE_REP
EAT);
    wlan_sdk_subscribe(SIG_SDK_MQTT_DATA_DOWN, xAppQueue, SDK_SUB_MODE_REPE
AT);
```

```
    // 3. Event Processing Loop
    for (;;)
    {
        // Block indefinitely until a signal arrives
        if (xQueueReceive(xAppQueue, &msg, portMAX_DELAY) == pdPASS)
        {
            switch (msg.id)
            {
                case SIG_SDK_SYS_INIT_DONE:
                    printf("System Initialized. Starting App logic...\n");
                    break;

                case SIG_SDK_WLAN_IP_CHANGED:
                {
                    ip_info_int_t ip_info;
                    wlan_get_ip_info(&ip_info);
                    if (ip_info.ip.addr != 0) {
                        printf("Network Ready. IP: %08X\n", ip_info.ip.add
r);
                    }
                    break;
                }

                case SIG_SDK_MQTT_DATA_DOWN:
                    printf("Received MQTT Data. Length: %d\n", msg.data_le
n);
                    // msg.data points to the received payload buffer.
                    // REPEAT MODE NOTE: Do NOT free msg.data here. The sys
tem handles it.
                    // REDIRECT MODE NOTE: If using REDIRECT, you MUST call
 os_free(msg.data) here.
                    break;

                default:
                    break;
            }
        }
    }
}
```

2.2 Common Return Codes

Most APIs in this SDK return a status code of type `enum result_type` to indicate success or failure. The following table describes these return values:

Value	Name	Description
0	`NO_ERR`	Success.
1	`PARA_ERR`	Parameter Error. One or more input parameters are invalid.

Value	Name	Description
2	`VALUE_ERR`	Value Error. The configuration value is out of range or unsupported.
3	`STATUS_ERR`	Status Error. The device is in an incorrect state for this operation (e.g., trying to send data when not connected).
4	`CONNECT_ERR`	Connection Error. Failed to establish a connection.
5	`SEND_ERR`	Send Error. Failed to transmit data packet.

```
enum result_type{
    NO_ERR = 0,
    PARA_ERR,
    VALUE_ERR,
    STATUS_ERR,
    CONNECT_ERR,
    SEND_ERR,
};
```

2.3 Connect APIs

The Connect APIs provide the necessary functions to manage Wi-Fi connectivity and data link establishment. This involves multiple phases, from discovering available networks to joining an Access Point and establishing peer-to-peer or socket connections.

Key functionalities include:

- Network Discovery: Use wlan_sta_scan() to find available Access Points in the area.
- AP Association: Connect to a specific AP using wlan_sta_join(), monitor the connection status with wlan_sta_status(), or disconnect with wlan_sta_unjoin().
- AP Management: In AP mode, use ~~`wlan_ap_showNetwork()`~~ to view currently associated client stations. (Unsupported)

The following table summarizes the available connection APIs:

Function	Description
wlan_sta_scan()	Scans for available Access Points.
wlan_sta_join()	Connects the station to a specified Access Point.
wlan_sta_status()	Retrieves the current Wi-Fi connection status.
wlan_sta_unjoin()	Disconnects the station from the current Access Point.

2.3.1 wlan_sta_scan()

Initiates a synchronous scan for available Access Points (APs) in the vicinity.

Functional Description:

- Discovery: Triggers the Wi-Fi hardware to listen for beacons and probe responses across all supported channels.
- Synchronous Execution: The function blocks the caller while the scan is in progress. The registered callback function is executed within the context of this call before the function returns.

- Callback Context: The callback receives a pointer to a linked list of `scan_data_t` structures, each representing an identified AP.

Prototype:

```
u8_t wlan_sta_scan(scan_callback_t *scan_done_cb)
```

Precondition: DeviceMode must be set to `STA` or `AP_STA` (refer to wlan_set_operation_mode).

Returns	Parameters
result_type: `NO_ERR` (0): Scan completed successfully. `STATUS_ERR`: Operation not permitted in current mode (e.g., pure AP mode). `PARA_ERR`: Invalid callback pointer.	**scan_done_cb**: Pointer to a user-defined callback function. Refer to scan_callback_t for details.

Example Usage:

```
void my_scan_cb(const scan_data_t *pScandata) {
    const scan_data_t *curr = pScandata;
    while(curr) {
        wlan_printf("Found AP: %s (RSSI: %d)\n", curr->ssid, curr->signal);
        curr = curr->next;
    }
}

// Start the scan
wlan_sta_scan(my_scan_cb);
```

2.3.1.1 Struct scan_data_t Definition

Member	Type	Description
`signal`	int	Received Signal Strength Indication (RSSI) in dBm.
`freq`	int	Channel frequency in MHz.
`bssid`	uint8_t[6]	BSSID (MAC Address) of the Access Point.
`ssid`	char[32]	SSID (Network Name), null-terminated string.
`next`	void*	Pointer to the next `scan_data_t` entry in the linked list.

```
typedef struct scan_data_s{
    int signal;
    int freq;
    uint8_t bssid[6];
    char ssid[32];
    void *next;
} scan_data_t;
```

2.3.2 wlan_sta_join()

This function initiates an asynchronous request to join a specified Access Point (AP).

Functional Description:

- Asynchronous Request: The function returns immediately after validating parameters and initiating the join sequence. A return value of `NO_ERR` (0) indicates the request has been successfully accepted by the Wi-Fi stack, but the connection is NOT yet established.
- Status Monitoring (Required):
    - Because the operation is asynchronous, the final outcome of the join request is communicated via the Signal Framework:
        - Success: The SIG_SDK_JOIN_SUCESS signal is triggered when the association is successful.
        - Failure: The SIG_SDK_JOIN_FAILED signal is triggered if the connection attempt fails.
    - Alternatively, the application can continue to use wlan_sta_status() (polling) to verify the connection.
- Connection Maintenance (No Auto-Reconnect):
    - The SDK does NOT perform automatic reconnection for this API.
    - If the initial request fails, or if the Wi-Fi link is lost later (detected via polling or the `SIG_SDK_WLAN_DISCONNECTED` signal), the application layer must explicitly call `wlan_sta_join()` again to re-establish connectivity.
- Retry Strategy: If the API returns an error or the connection drops, it is recommended to wait for a backoff period (e.g., 2+ seconds) before retrying.

Prototype:

```
u8_t wlan_sta_join(wlan_ap_t *pAP)
```

Precondition: DeviceMode must be set to `STA` or `AP_STA`.

Returns	Description
`NO_ERR` (0)	Request accepted successfully. Monitor status to confirm connection.
`CONNECT_ERR` (4)	Request failed (e.g., Stack busy). Retry after a delay.
`PARA_ERR`	Invalid parameter.

2.3.2.1 Struct wlan_ap_t Definition

Member	Type	Description
`ssid`	char[32]	The SSID (Network Name) of the target Access Point.
`password`	char[64]	The password/passphrase for WPA/WPA2 authentication.

```
typedef struct wlan_ap_s{
    char ssid[32];
    char password[64];
} wlan_ap_t;
```

2.3.3 wlan_sta_status()

Retrieves the current Wi-Fi connection status.

Functional Description:

- Status Check: Returns the internal state indicating whether the device is currently associated (joined) with an Access Point (AP).
- Polling Usage: Commonly used in a loop (with delay) after calling `wlan_sta_join()` to confirm when the connection is fully established and stable.

Prototype:

```
bool wlan_sta_status(void)
```

Returns	Parameters
`true`: Connected (Joined). `false`: Not connected.	`void`

2.3.4 wlan_sta_unjoin()

Disconnects the station from the currently associated Access Point (AP).

Functional Description:

- Disassociation: Terminates the active Wi-Fi link with the AP and transitions the stack to an idle state.
- Synchronous Execution: This function blocks until the disassociation request is processed by the WLAN stack.
- Status Update: After this function returns, wlan_sta_status will return `false`.

Prototype:

```
void wlan_sta_unjoin(void)
```

Precondition: Device must be currently joined to an AP (refer to wlan_sta_join); otherwise, the function has no effect.

Returns	Parameters
void	void

2.3.5 wlan_sta_connect()

Establishes a logical network connection (UDP or TCP) with a remote station after successfully joining an Access Point.

Functional Description:

- Protocol Support: Supports `WLANCONN_UDP`, `WLANCONN_TCP_CLIENT`, and `WLANCONN_TCP_SERVER`.
- Logical Link: Sets up the internal socket structures and background data receive tasks.
- Synchronous Execution: This function blocks until the logical connection is established or fails.
- Multi-Connection: The SDK supports multiple concurrent socket connections (refer to system limits).

Prototype:

```
u8_t wlan_sta_connect(u8_t protocol, wlan_station_t *pSta)
```

Precondition: Device must be currently joined to an AP (refer to wlan_sta_join); otherwise, the function has no effect and will return an error status.

Returns	Parameters
`NO_ERR` (0): Connected successfully. `CONNECT_ERR` (4): Connection failed. `STATUS_ERR` (3): Device not ready (not joined).	**protocol**: Connection type: `WLANCONN_UDP`, `WLANCONN_TCP_CLIENT`, or `WLANCONN_TCP_SERVER`. **pSta**: Pointer to a `wlan_station_t` structure defining local/remote ports, remote IP, and the receive callback.

2.3.5.1 Struct wlan_station_t Definition

Used for setting up UDP or TCP connections.

Member	Type	Description
`local`	`struct station_info_s`	Local IP address and Port configuration.
`remote`	`struct station_info_s`	Remote IP address and Port configuration.
`rx`	`wlan_callback`	Callback function pointer triggered when data is received.

Sub-struct **`station_info_s`**:

Member	Description
`ipAddr`	IPv4 address (integer format).
`portid`	TCP/UDP Port number.

```
struct station_info_s {
    uint32_t ipAddr;
    uint16_t portid;
};

// Callback prototype: void (* wlan_callback)(uint8_t *pBuf, uint16_t len);

typedef struct wlan_station_s {
    struct station_info_s local;
    struct station_info_s remote;
    wlan_callback rx;
} wlan_station_t;
```

2.3.6 wlan_sta_disconnect() (Unsupported)

Disconnects a specific logical connection (socket) with a remote peer.

Functional Description:

- Targeted Disconnection: Closes the connection identified by the remote IP address and protocol type.
- Resource Release: Frees the internal socket resources associated with the connection.

Prototype:

```
void wlan_sta_disconnect(u32_t ipaddr, u8_t protocol)
```

Precondition: A connection must already be established via wlan_sta_connect.

Returns	Parameters
`void`	**ipaddr**: The IPv4 address of the remote peer (in network byte order). **protocol**: The protocol type of the connection to close (`WLANCONN_UDP`, `WLANCONN_TCP_CLIENT`, `WLANCONN_TCP_SERVER`).

2.3.7 wlan_ap_showNetwork() (Unsupported)

show the station list that already connected to AP

Prototype:

```
void wlan_ap_showNetwork(void)
```

Precondition: DeviceMode must be set to `AP` or `AP_STA`.

Returns	Parameters
void	void

2.4 ~~Transmit APIs~~ (Unsupported)

The following list contains all transmit APIs.

1. wlan_sta_send()

2.4.1 ~~wlan_sta_send()~~ (Unsupported)

Sends a data packet to a connected remote station.

Functional Description:

- Data Transmission: Transmits the provided data buffer to the target IP address using the established protocol (UDP/TCP).
- Routing: The `ipaddr` parameter determines the destination.
- Blocking Behavior: Depending on the protocol and socket state, this call may block until the data is buffered for transmission.

Prototype:

```
u8_t wlan_sta_send(u32_t ipaddr, u8_t *pBuf, u16_t len)
```

Precondition: A socket connection must be established via wlan_sta_connect.

Returns	Parameters
`NO_ERR` (0): Success. `SEND_ERR`: Transmission failed. `PARA_ERR`: Invalid parameter.	**ipaddr**: The IPv4 address of the target recipient (network byte order). **pBuf**: Pointer to the data buffer to send. **len**: Length of the data in bytes.

2.5 Configuration APIs

This section details the APIs used to configure the fundamental parameters of the Wi-Fi module. These APIs allow the application to manage device identity, network settings, power consumption, and operating modes.

The Configuration APIs can be broadly categorized into the following groups:

- Device Identification: APIs to get or set the device name (`wlan_get/set_device_name`) and MAC address (`wlan_get/set_mac_address`).
- Network Configuration: APIs to manage IP settings, including enabling/disabling DHCP (`wlan_get/set_dhcp_mode`) and configuring static IP addresses (`wlan_get/set_ip_info`).
- ~~Power Management: APIs to control the module's sleep behaviors (`wlan_get/set_sleep_mode`) to optimize power consumption for different use cases.~~ (Unsupported)
- Operating Modes: APIs to switch between Station (STA), Access Point (AP), or concurrent (AP_STA) modes (`wlan_get/set_operation_mode`).
- Advanced Configuration: The versatile wlan_config_info() API provides access to a wide range of internal system variables (var_id) for fine-tuning the stack behavior.

Important Note:
Many configuration changes, especially those related to operating modes, MAC addresses, and static IP settings, require a system restart (wlan_restart()) to take effect. Please refer to the specific API descriptions for details.

The following table summarizes the available configuration APIs:

Function	Description
wlan_get_device_name()	Retrieves the module's identification string.
wlan_set_device_name()	Sets the module's identification string.
wlan_get_mac_address()	Retrieves the 6-byte hardware MAC address.
wlan_set_mac_address()	Sets a new MAC address (Requires 12-character HEX string).
wlan_get_dhcp_mode()	Checks if DHCP is enabled or if the module uses static IP settings.
wlan_set_dhcp_mode()	Enables or disables DHCP mode.
wlan_get_ip_info()	Retrieves the current IP, Netmask, and Gateway in integer format.
wlan_set_ip_info()	Configures manual static IP settings when DHCP is disabled.

Function	Description
~~wlan_get_sleep_mode()~~	**(Unsupported)** Retrieves the current power-save sleep mode configuration.
~~wlan_set_sleep_mode()~~	**(Unsupported)** Configures the module's power-save behavior.
wlan_get_operation_mode()	Retrieves the current Wi-Fi mode.
wlan_set_operation_mode()	Configures the Wi-Fi operating mode.
wlan_config_info()	Accesses or modifies internal system variables using specific IDs.
wlan_get_config_byID()	Retrieves the current value of a system configuration variable by its ID.

2.5.1 wlan_get_device_name()

Retrieves the module's identification string (Device Name).

Functional Description:

- Configuration Access: Reads the device name string currently stored in the system configuration variables.
- Identity: This name is often used as the identification string within the system configuration.

Prototype:

```
char* wlan_get_device_name(void)
```

Returns	Parameters
char*: Pointer to the null-terminated string containing the device name.	`void`

2.5.2 wlan_set_device_name()

Sets the identification string (Device Name) for the module.

Functional Description:

- Persistent Storage: Updates the "DeviceName" variable in the non-volatile system configuration.
- Length Limitation: The name string is limited to a maximum of 20 characters (case sensitive).
- Effect: Changes take effect only after a system reboot.

Prototype:

```
u8_t wlan_set_device_name(char* pName)
```

Note: A system restart (wlan_restart) is required for the new name to take effect.

Returns	Parameters
`NO_ERR` (0): Success. `PARA_ERR`: `pName` is NULL.	**pName**: Pointer to a null-terminated string containing the new device name.

Returns	Parameters
`VALUE_ERR`: String too long or storage error.	

Example Usage:

```
if (wlan_set_device_name("MyACHModule") == NO_ERR) {
    wlan_printf("Name set successfully. Rebooting...\n");
    wlan_restart();
}
```

2.5.3 wlan_get_mac_address()

Retrieves the current MAC address of the Wi-Fi interface.

Functional Description:

- Data Retrieval: Copies the 6-byte hardware MAC address from the system into the provided buffer.
- Format: binary array of 6 bytes.

Prototype:

```
void wlan_get_mac_address(u8_t* pMac)
```

Returns	Parameters
`void`	**pMac**: Pointer to a **6-byte** unsigned char array (`u8_t mac[6]`) where the address will be stored.

Example Usage:

```
unsigned char mac[6];
wlan_get_mac_address(mac);
wlan_printf("MAC Address: %02X:%02X:%02X:%02X:%02X:%02X\n",
            mac[0], mac[1], mac[2], mac[3], mac[4], mac[5]);
```

2.5.4 wlan_set_mac_address()

Sets a new MAC address for the Wi-Fi interface using a **hexadecimal string**.

Functional Description:

- Persistent Storage: Configures the hardware MAC address in the system's non-volatile memory.
- Format Requirement: This API expects a 12-character hexadecimal string (e.g., `"00043E112233"`). Do NOT pass a 6-byte binary array.
- Effect: A system restart is required for the new MAC address to be applied to the hardware.

Prototype:

```
void wlan_set_mac_address(u8_t* pMac)
```

Note: This function triggers an automatic system restart. Execution will stop here.

Returns	Parameters
`void`	**pMac**: Pointer to a **12-character string** representing the MAC address in hex (e.g., `"00043E212345"`).

Example Usage:

```
// Set MAC address to 00:04:3E:11:22:33
// Note: The system reboots immediately; the line below will not execute.
wlan_set_mac_address("00043E112233");
wlan_printf("MAC address updated. Rebooting system automatic...\n");
```

2.5.5 wlan_get_dhcp_mode()

Retrieves the current DHCP configuration mode.

Functional Description:

- Returns whether the module is configured to obtain an IP address automatically via DHCP or use a static IP configuration.
- If this returns `false`, The module relies on manual IP configurations assigned via wlan_set_ip_info().

Prototype:

```
bool wlan_get_dhcp_mode(void)
```

Returns	Parameters
`true`: DHCP is enabled (Automatic IP). `false`: DHCP is disabled (Static IP).	`void`

Example Usage:

```
if (wlan_get_dhcp_mode()) {
    wlan_printf("DHCP is enabled.\n");
} else {
    wlan_printf("Using static IP configuration.\n");
}
```

2.5.6 wlan_set_dhcp_mode()

Enables or disables DHCP for automatic IP address retrieval.

Functional Description:

- Mode Switch: Configures the system to either use a DHCP server to obtain IP settings or rely on manually configured static IP values.
- Persistent Storage: The setting is saved to the non-volatile system configuration.
- Effect: A system restart is required for the change to take effect.

Prototype:

```
u8_t wlan_set_dhcp_mode(bool mode)
```

Note: A system restart (wlan_restart) is mandatory for the mode switch to be applied.

Returns	Parameters
`NO_ERR` (0): Success. `VALUE_ERR`: Storage error.	**mode**: `true` to enable DHCP; `false` to disable (use Static IP).

Example Usage:

```
// Disable DHCP to use static IP
if (wlan_set_dhcp_mode(false) == NO_ERR) {
    wlan_printf("DHCP disabled. Restarting to apply static IP...\n");
    wlan_restart();
}
```

2.5.7 wlan_get_ip_info()

Retrieves the current IP configuration (IP address, Netmask, and Gateway) in integer format.

Functional Description:

- Data Access: Copies the current active IP settings into the provided structure.
- Format: The information is returned as 32-bit integers within the ip_info_int_t structure.

Prototype:

```
void wlan_get_ip_info(ip_info_int_t* pIP_int)
```

Returns	Parameters
`void`	**pIP_int**: Pointer to an ip_info_int_t structure where the data will be stored.

Example Usage:

```
#include "lwip/ip_addr.h"

ip_info_int_t ip_info;
wlan_get_ip_info(&ip_info);

// Use LwIP utility function to convert IP to a dotted-decimal string
// Note: ip4addr_ntoa is not thread-safe (uses a static buffer).
wlan_printf("IP: %s\n", ip4addr_ntoa((const ip4_addr_t*)&ip_info.ip));
wlan_printf("Netmask: %s\n", ip4addr_ntoa((const ip4_addr_t*)&ip_info.netma
sk));
wlan_printf("Gateway: %s\n", ip4addr_ntoa((const ip4_addr_t*)&ip_info.gatew
ay));
```

2.5.7.1 Struct ip_info_int_t Definition

Member	Type	Description
`ip`	`struct ip_address`	The IP address of the module.
`netmask`	`struct ip_address`	The subnet mask of the local network.
`gateway`	`struct ip_address`	The default gateway address.

Sub-struct **`ip_address`**:

Member	Type	Description
`addr`	`uint32_t`	32-bit unsigned integer representing the IPv4 address.

```
struct ip_address{
    uint32_t addr;
};

typedef struct ip_info_int_s{
    struct ip_address ip;
    struct ip_address netmask;
    struct ip_address gateway;
} ip_info_int_t;
```

2.5.8 wlan_set_ip_info()

Configures static IP settings (IP address, Netmask, and Gateway) for the module.

Functional Description:

- Static Configuration: Sets the manual IP parameters. These settings are used only when DHCP is disabled.
- Format: Parameters are passed via the `ip_info_str_t` structure as dotted-decimal strings (e.g., "192.168.1.10").
- Persistent Storage: Settings are saved to non-volatile memory.
- Effect: A system restart is required for the new IP settings to be applied.

Precondition: `DHCPMode` must be set to `false` (refer to wlan_set_dhcp_mode()); otherwise, these settings will be ignored. A system restart (wlan_restart()) is required after calling this function.

Prototype:

```
u8_t wlan_set_ip_info(ip_info_str_t ip_str)
```

Returns	Parameters
`NO_ERR` (0): Success. `VALUE_ERR`: Storage error or invalid format.	**ip_str**: An ip_info_str_t structure containing the IP, Netmask, and Gateway strings.

Example Usage:

```
ip_info_str_t my_ip;

// 1. Disable DHCP first
wlan_set_dhcp_mode(false);

// 2. Configure static IP details
strncpy(my_ip.ip, "192.168.1.50", 16);
strncpy(my_ip.netmask, "255.255.255.0", 16);
strncpy(my_ip.gateway, "192.168.1.1", 16);
```

```
// 3. Apply settings
if (wlan_set_ip_info(my_ip) == NO_ERR) {
    wlan_printf("Static IP configured. Restarting...\n");
    wlan_restart();
}
```

2.5.8.1 Struct ip_info_str_t Definition

Member	Type	Description
ip	char[16]	IP address as a string (e.g., "192.168.1.10").
netmask	char[16]	Subnet mask as a string (e.g., "255.255.255.0").
gateway	char[16]	Gateway address as a string (e.g., "192.168.1.1").

```
typedef struct ip_info_str_s{
    char ip[16];
    char netmask[16];
    char gateway[16];
} ip_info_str_t;
```

2.5.9 wlan_get_sleep_mode() (Unsupported)

Get the module sleep mode

Prototype:

```
sleep_mode_t wlan_get_sleep_mode(void)
```

Returns	Parameters
sleep_mode_t	void

2.5.10 ~~wlan_set_sleep_mode()~~ (Unsupported)

Set the module sleep mode

Prototype:

```
u8_t wlan_set_sleep_mode(sleep_mode_t mode)
```

Returns	Parameters
`NO_ERR` (0): Success. `VALUE_ERR`: Storage error or invalid format.	sleep_mode_t (see below)

2.5.10.1 Enum sleep_mode_t Definition

Value	Name	Description
0	`NO_SLEEP`	Active mode. No power saving enabled.
1	`SHALLOW_SLEEP`	Light sleep mode. Fast wake-up latency.
2	`DEEP_SLEEP`	Deep sleep mode. Lowest power consumption.
3	`SHALLOW_DEEP_SLEEP`	Mixed/Adaptive sleep mode.

```
typedef enum sleep_mode_e{
    NO_SLEEP,
    SHALLOW_SLEEP,
    DEEP_SLEEP,
    SHALLOW_DEEP_SLEEP
} sleep_mode_t;
```

2.5.11 wlan_get_operation_mode()

Retrieves the currently active Wi-Fi operating mode.

Functional Description:

- Mode Retrieval: Returns the mode the Wi-Fi stack is currently running in.
- Modes: Possible values include `STA`, `AP`, `AP_STA`, and for Mesh operations, `MP` (Mesh Point) or `AP_MP` (Access Point + Mesh Point).

Prototype:

```
operate_mode_t wlan_get_operation_mode(void)
```

Returns	Parameters
operate_mode_t: The current mode (refer to operate_mode_t).	`void`

Example Usage:

```
operate_mode_t mode = wlan_get_operation_mode();
if (mode == STA) {
    wlan_printf("Module is running in Station mode.\n");
}
```

2.5.12 wlan_set_operation_mode()

Sets the Wi-Fi operating mode for the module.

Functional Description:

- Mode Configuration: Switches the module between Station (`STA`), Access Point (`AP`), Concurrent (`AP_STA`), or Mesh (`MP`/`AP_MP`) modes.
- Persistent Storage: The mode is saved to the system configuration.
- Effect: This function triggers an automatic system restart upon successfully saving the new mode configuration.

Note: The system reboots immediately after a successful call to re-initialize the Wi-Fi stack. Code execution stops at this point.

Prototype:

```
u8_t wlan_set_operation_mode(operate_mode_t mode)
```

Returns	Parameters
`NO_ERR` (0): Success (system will reboot). `VALUE_ERR`: Invalid mode or storage error.	**mode**: Target operate_mode_t value.

Example Usage:

```
// Set to Access Point mode.
// Note: The module reboots automatically; wlan_printf will not execute.
if (wlan_set_operation_mode(AP) == NO_ERR) {
    wlan_printf("This line will not be reached.\n");
}
```

2.5.12.1 Enum operate_mode_t Definition

Value	Name	Description
0	`STA`	Station Mode. Device connects to an Access Point.
1	`AP`	Access Point Mode. Device acts as an AP for others.
2	`AP_STA`	Concurrent Mode. Device acts as both STA and AP.
3	`MP`	Mesh Point Mode. Device participates in a mesh network.
4	`AP_MP`	Concurrent Mode. Device acts as both an AP and a Mesh Point.

```
typedef enum operate_mode_e{
    STA=0,
    AP,
    AP_STA,
    MP,
    AP_MP
} operate_mode_t;
```

2.5.13 wlan_config_info()

A versatile API to access or modify the internal system configuration variables.

Functional Description:

- Dual Mode: This function acts as both a "Getter" and a "Setter" depending on the parameters.
    - Get/Print: If `pconfig_info` is `NULL`, it prints the current value of the specified variable to the debug UART. If var_id is 0, it prints *all* variables.
    - Set: If `pconfig_info` is a valid string, it updates the specified variable with that value.
- System Variables: It operates on the internal configuration table (see list below), covering everything from network settings to hardware parameters.
- Persistence: Changes are saved to non-volatile memory.

Precondition: If critical parameters (e.g. Mode, Channel) are changed, a system restart (wlan_restart()) is required.

Prototype:

```
void wlan_config_info(unsigned char var_id, char* pconfig_info)
```

Returns	Parameters
`void`	**var_id**: The ID of the configuration variable (see table below). **pconfig_info**: - **NULL**: Print the current value of `[var_id](#25131-configuration-`

Returns	Parameters
	`variables-var_id`) (or all if ID=0). - **String**: Pointer to a null-terminated string containing the new value to set.

Example Usage:

```
// 1. Set the Mesh ID
wlan_config_info(AMP_VARID_MESH_ID, "MyNewMeshNetwork");

// 2. Print the current Mesh ID to UART to verify
wlan_config_info(AMP_VARID_MESH_ID, NULL);

// 3. Print ALL configuration variables
wlan_config_info(0, NULL);
```

2.5.13.1 Configuration Variables (var_id)

var_id (Macro)	Details	Restart Required
`AMP_VARID_BUILD_VERSION`	**Name**: BuildVersion **Def**: `151202A` **Desc**: Date code version of the software (read only)	No
`AMP_VARID_DEVICE_NAME`	**Name**: DeviceName **Def**: `Amped WIFI` **Desc**: Up to 20 characters are allowed (case sensitive)	No
`AMP_VARID_STA_MAC_ADDR`	**Name**: STA_MAC_ADDR **Def**: `00043e26002d` **Desc**: MAC address of the station interface (Read Only).	-
`AMP_VARID_DHCP_MODE`	**Name**: DHCPMode **Def**: `true` **Desc**: true=enable DHCP false=disable DHCP DHCP on/off.	No
`AMP_VARID_IP_ADDRESS`	**Name**: IPAddress **Def**: `192.168.0.2` **Desc**: A static IP address, when DHCP off or failed, it will be used	No
`AMP_VARID_NET_MASK`	**Name**: NetMask **Def**: `255.255.255.0` **Desc**: Subnet mask of the local network (e.g., "255.255.255.0").	No
`AMP_VARID_GATE_WAY`	**Name**: GateWay **Def**: `192.168.0.1` **Desc**: Gateway of the network	No
`AMP_VARID_SSID`	**Name**: SSID **Def**: `Amped RF`	No

var_id (Macro)	Details	Restart Required
	Desc: ESSID of the Access Point connection destination	
`AMP_VARID_PASS_PHRASE`	**Name**: PassPhrase **Def**: `12345678` **Desc**: WPA/WPA2 Passphrase for the network.	No
`AMP_VARID_AUTH_TYPE`	**Name**: AuthType **Def**: `1` **Desc**: 0=NONE 1= WPA2-PSK WIFI encryption methods	No
`AMP_VARID_HOST_IP_ADDR`	**Name**: HostIPAddr **Def**: `192.168.0.10` **Desc**: Remote device's IP address	No
`AMP_VARID_IP_PROTOCOL`	**Name**: IPProtocol **Def**: `1` **Desc**: 0=TCP Server 1=UDP 2=TCP client Protocol type	No
`AMP_VARID_HOST_PORT`	**Name**: HostPort **Def**: `2015` **Desc**: Remote device's listen port number.	No
`AMP_VARID_LOCAL_PORT`	**Name**: LocalPort **Def**: `2015` **Desc**: Local listen port number.	No
`AMP_VARID_UART_BAUDRATE`	**Name**: UartBaudrate **Def**: `115200` **Desc**: 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800, 921600	No
`AMP_VARID_UART_PARITY`	**Name**: UartParity **Def**: `none` **Desc**: odd, even, none UART parity. Typical: none	No
`AMP_VARID_UART_DATA_BITS`	**Name**: UartDataBits **Def**: `8` **Desc**: 8, 9 UART data bits per character. Typical:8	No
`AMP_VARID_UART_STOP_BITS`	**Name**: UartStopBits **Def**: `1` **Desc**: 0.5, 1, 1.5, 2 UART number of stop bits.Typical:1	No
`AMP_VARID_UART_FLOW_CONTROL`	**Name**: UartFlowControl **Def**: `false`	No

var_id (Macro)	Details	Restart Required
	Desc: True= enable UART hardware RTS/CTS flow control False= disable RST/CTS flow control	
`AMP_VARID_HARDWARE`	**Name**: Hardware **Def**: `WF88-M` **Desc**: Module hardware type. (read only)	-
`AMP_VARID_CPU_MHZ`	**Name**: CpuMHz **Def**: `42` **Desc**: Module's CPU speed: 42Mhz is supported	Yes
`AMP_VARID_CHANNEL`	**Name**: Channel **Def**: `1` **Desc**: 2.4GHz: 1-13 5GHz: 36-165 Set the WiFi channel for AP mode (no effect in STA mode).	Yes
`AMP_VARID_DEVICE_MODE`	**Name**: DeviceMode **Def**: `MP` **Desc**: STA, AP, AP_STA, MP, or AP_MP Wi-Fi module operation mode	Yes
`AMP_VARID_OUT_MTU_SIZE`	**Name**: OutMtuSize **Def**: `1400` **Desc**: 1 - 1420 Packet size of UART received. Typical:1400	Yes
`AMP_VARID_MAX_STA_COUNT`	**Name**: MaxSTACount **Def**: `5` **Desc**: 1-12 Maxim station number in AP mode. Typical:5	Yes
`AMP_VARID_MP_MODE`	**Name**: MPMode **Def**: `0` **Desc**: 0=Disable; 1=Enable Multiple connections on/off	Yes
`AMP_VARID_KEEP_ALIVE`	**Name**: KeepAlive **Def**: `60` **Desc**: Keep-alive interval in seconds.	No
`AMP_VARID_STATION_INACTIVE`	**Name**: StationInactive **Def**: `120` **Desc**: 15-255second When StationInactive time passed, station didn't send any data, AP will confirm whether station still in region	No
`AMP_VARID_WSM_FIRMWARE`	**Name**: WsmFirmware **Def**: `wsm_V3.2.3.bin`	Yes

var_id (Macro)	Details	Restart Required
	Desc: Filename of the Wi-Fi firmware binary stored in flash.	
`AMP_VARID_WSM_BOOTLOADER`	**Name**: WsmBootloader **Def**: `bootloader.bin` **Desc**: Filename of the Wi-Fi bootloader binary.	Yes
`AMP_VARID_WSM_SDD`	**Name**: WsmSdd **Def**: `sdd_6010.bin` **Desc**: Filename of the Wi-Fi SDD (Configuration) binary.	Yes
`AMP_VARID_MQTT_SERVER_IP`	**Name**: MQTTServerIP **Def**: `192.168.1.76` **Desc**: IP address or Domain Name of the MQTT broker.	No
`AMP_VARID_MQTT_SERVER_PORT`	**Name**: MQTTServerPort **Def**: `1883` **Desc**: Port number of the MQTT broker (e.g., 1883, 8883).	No
`AMP_VARID_MQTT_SERVER_USR_NAME`	**Name**: MQTTServerUsrName **Def**: `admin` **Desc**: Username for MQTT broker authentication.	No
`AMP_VARID_MQTT_SERVER_PASSWD`	**Name**: MQTTServerPasswd **Def**: `password` **Desc**: Password for MQTT broker authentication.	No
`AMP_VARID_MQTT_SUBSCRIBE_TOPIC`	**Name**: MQTTSubscribeTopic **Def**: `testtopic` **Desc**: The default topic the module subscribes to after connecting to the broker.	No
`AMP_VARID_MQTT_PUBLISH_TOPIC`	**Name**: MQTTPubishTopic **Def**: `testtoptic` **Desc**: The default topic used for outgoing MQTT messages.	No
`AMP_VARID_MQTT_QOS`	**Name**: MQTTQoS **Def**: `0` **Desc**: MQTT Quality of Service level (0: At most once, 1: At least once, 2: Exactly once).	No
`AMP_VARID_MQTT_AUTH_TYPE`	**Name**: MQTTAuthType **Def**: `1` **Desc**: MQTT Authentication Mode:	No

var_id (Macro)	Details	Restart Required
	0=User/Pass, 1=Cert, 2=Mutual, 4=None.	
`AMP_VARID_ADDR_TYPE`	**Name**: AddrType **Def**: `0` **Desc**: Selects the IP address type preference: 0 for IPv4, 1 for IPv6.	-
`AMP_VARID_LINK_TYPE`	**Name**: LINKTYPE **Def**: `0` **Desc**: Selects the default link protocol: 0 for TCP, 1 for MQTT.	No
`AMP_VARID_MQTT_CA_CRT`	**Name**: MQTTCaCrt **Def**: `CA.crt` **Desc**: Filename of the MQTT CA certificate.	No
`AMP_VARID_MQTT_CLIENT_CRT`	**Name**: MQTTClinetCrt **Def**: `client.crt` **Desc**: Filename of the MQTT Client certificate.	No
`AMP_VARID_MQTT_CLIENT_KEY`	**Name**: MQTTClinetKey **Def**: `client.key` **Desc**: Filename of the MQTT Client private key.	No
`AMP_VARID_DNS1_V4`	**Name**: DNS1V4 **Def**: `8.8.8.8` **Desc**: Primary IPv4 DNS server address.	No
`AMP_VARID_DNS2_V4`	**Name**: DNS2V4 **Def**: `1.1.1.1` **Desc**: Secondary IPv4 DNS server address.	No
`AMP_VARID_MESH_ID`	**Name**: MESH_ID **Def**: `mymesh12345` **Desc**: Range 1~32 char	Yes
`AMP_VARID_MESH_PASS_PHRASE`	**Name**: MESH_PassPhrase **Def**: `12345678` **Desc**: Range 1~64char	Yes
`AMP_VARID_MESH_AUTH_TYPE`	**Name**: MESH_AuthType **Def**: `2` **Desc**: Authentication method. `0`: Open, `2`: SAE (Secure Authentication of Equals).	Yes
`AMP_VARID_APP_AUTO_START`	**Name**: APP_AutoStart **Def**: `false`	Yes

var_id (Macro)	Details	Restart Required
	Desc: Enable (true) or disable (false) automatic application startup on boot.	
`AMP_VARID_AUTO_SSID`	**Name**: AutoSSID **Def**: `Amped RF` **Desc**: Default SSID used for automatic connection/AP setup.	Yes
`AMP_VARID_AUTO_PASS_PHRASE`	**Name**: PassPhrase **Def**: `12345678` **Desc**: WPA/WPA2 Passphrase for the network.	Yes

2.5.14 wlan_get_config_byID()

Retrieves the current value of a system configuration variable by its ID.

Functional Description:

- Synchronous Retrieval: Returns the requested configuration value into the provided buffer.
- String Format: Regardless of the internal data type (integer, boolean, or string), all values are returned as null-terminated strings.
- Buffer Safety: The caller must provide a buffer and specify its length (`len`) to prevent memory overflow.

Prototype:

```
void wlan_get_config_byID(char* buf, u16_t len, u16_t id)
```

Parameters	Description
buf	Destination buffer where the configuration string will be stored.
len	Maximum size of the buffer in bytes.
id	The ID of the variable to retrieve (refer to var_id).

Example Usage:
To get the Mesh Point mode status:

```
char mode_buf[8];
// Retrieve var 31 (MPMode)
wlan_get_config_byID(mode_buf, sizeof(mode_buf), 31);

if (strcmp(mode_buf, "1") == 0) {
    printf("Mesh Point mode is ENABLED\n");
}
```

Tip: If unsure about the parameter format for a specific var_id, you can call wlan_config_info(0, NULL). This will output the current values and formats of all variables to the UART console for confirmation.

2.6 System APIs

The following table summarizes the available system APIs:

Function	Description
wlan_restart()	Performs a full hardware-level system reset of the module.
wlan_printf()	Prints formatted strings to the system's primary debug UART interface.

2.6.1 wlan_restart()

Performs a full hardware-level system reset of the module.

Functional Description:

- Hardware Reset: Triggers a chip-wide reset by writing to the system configuration reset registers. This action follows the same sequence as a physical power-on reset.
- Configuration Reload: During the subsequent boot sequence, the module re-reads all non-volatile memory (NV) variables. This is the standard method to apply changes made via wlan_config_info().

Prototype:

```
void wlan_restart(void)
```

Precondition: None.

Returns	Parameters
void	void

Example Usage:

```
wlan_printf("Applying new settings and rebooting...\n");
wlan_restart();
// Code beyond this point is unreachable.
```

2.6.2 wlan_printf()

Prints formatted strings to the system's primary debug UART interface.

Functional Description:

- Operates similarly to the standard C `printf()` function, allowing for formatted output (strings, integers, hex, etc.).
- Output is sent to the system console (typically UART0 or the designated debug port).

Prototype:

```
void wlan_printf(char *fmt, ...)
```

Returns	Parameters
`void`	**fmt**: Pointer to a null-terminated format string. **...**: Optional arguments corresponding to the format specifiers.

Example Usage:

```
int sensor_val = 25;
wlan_printf("System initialized. Current temperature: %d.\n", sensor_val);
```

2.7 Driver APIs

The Driver APIs provide low-level control over the module's hardware interfaces, including UART, SPI, and GPIO. These functions are typically used for data passthrough, peripheral communication, and status signaling.

The following table summarizes the available driver APIs:

Function	Description
~~wlan_set_uart_connect_mode()~~	**(Unsupported)** Configures UART for bypass/connect mode with a receive callback.
wlan_uart_send()	Sends a single byte of data over the specified UART port.
~~wlan_set_uart_baudrate()~~	**(Unsupported)** Sets the operating baud rate for the UART interface.
~~wlan_spi_init()~~	**(Unsupported)** Initializes the SPI interface parameters and roles.
~~wlan_spi_snd_bytes~~	**(Unsupported)** Transmits a buffer of data over the SPI interface.
~~wlan_spi_sndrcv_bytes()~~	**(Unsupported)** Performs simultaneous SPI transmit and receive.
~~wlan_spi_rcv_bytes()~~	**(Unsupported)** Receives a buffer of data over the SPI interface.
wlan_gpio_config()	Configures a GPIO pin's direction.
wlan_gpio_set()	Sets the output level of a GPIO pin.
wlan_gpio_get()	Reads the current level of a GPIO pin.

2.7.1 wlan_set_uart_connect_mode() (Unsupported)

Note: This function is currently NOT SUPPORTED in this SDK version.

Functional Description:

- Intended Use: Historically designed to switch the UART interface from Command Mode (AT mode) to Connect Mode (Data passthrough mode).
- Status: This function is a placeholder and does not contain an active implementation.
- Callback: Intended to register a user callback function of type `uart_callback` to handle incoming UART data.

Prototype:

```
void wlan_set_uart_connect_mode(uart_callback *rx_cb)
```

Precondition: None (Feature not supported).

Returns	Parameters
void	**rx_cb**: Pointer to a callback function of type `void (*)(unsigned char *pBuf, unsigned short len)`.

2.7.2 wlan_uart_send()

Sends a single byte of data through the specified UART port.

Functional Description:

- Single Byte Transmission: Transmits exactly one 8-bit data byte. For sending buffers or strings, this function must be called in a loop.
- Port Selection: Supports multiple UART ports available on the module.

Prototype:

```
void wlan_uart_send(uart_port_t PortNum, u8_t data)
```

Returns	Parameters
void	**PortNum**: Target UART port index (refer to uart_port_t). **data**: The 8-bit byte to transmit.

2.7.2.1 Enum uart_port_t Definition

Used to specify the hardware UART interface.

Value	Name	Description
0	`UART_PORT0`	UART interface 0 (typically the primary AT command / Debug port). Only UART0 is supported.

```
typedef enum uart_port_e
{
    UART_PORT0 //Only UART0 is supported.
} uart_port_t;
```

Example Usage:

```
// Send character 'A' to UART0
wlan_uart_send(UART_PORT0, 'A');

// Send a string
char *msg = "Hello";
while(*msg) {
    wlan_uart_send(UART_PORT0, *msg++);
}
```

2.7.3 wlan_set_uart_baudrate() (Unsupported)

Sets the communication baud rate for the specified UART port.

Functional Description:

- Speed Configuration: Adjusts the data transmission and reception speed (bps) of the hardware UART interface.

- Supported Rates: Supports standard baud rates such as 9600, 115200, and 921600.

Prototype:

```
void wlan_set_uart_baudrate(uart_port_t PortNum, u32_t baudrate)
```

Returns	Parameters
void	**PortNum**: Target UART port index (refer to uart_port_t). **baudrate**: Desired communication speed in bits per second (e.g., 115200).

Example Usage:

```
// Configure UART0 to 115200 bps
wlan_set_uart_baudrate(UART_PORT0, 115200);
```

2.7.4 ~~wlan_spi_init()~~ (Unsupported)

Initialization SPI.

Prototype:

```
void wlan_spi_init(spi_port_t portnum)
```

Returns	Parameters
void	Use tSPIDevice struct to set the spi info

2.7.4.1 Enum spi_port_t Definition

Value	Name	Description
0	SPI_PORT0	SPI Interface 0.
1	SPI_PORT1	SPI Interface 1.
2	SPI_PORT2	SPI Interface 2.

```
typedef enum spi_port_e
{
    SPI_PORT0,
    SPI_PORT1,
    SPI_PORT2
} spi_port_t;
```

2.7.5 ~~wlan_spi_snd_bytes()~~ (Unsupported)

spi send the bytes.

Prototype:

```
void wlan_spi_snd_bytes(u8_t * sndbuf, unsigned short length)
```

Returns	Parameters
void	sndbuf: the buffer that want to send length: the buffer length

2.7.6 wlan_spi_sndrcv_bytes() (Unsupported)

spi send and receive the bytes at the same time.

Prototype:

```
u16_t wlan_spi_sndrcv_bytes(u8_t * sndbuf, u8_t rcvbuf, u16_t length)
```

Returns	Parameters
	sndbuf: the buffer that want to send data rcvbuf: the buffer that want to receive data length: the buffer length

2.7.7 ~~wlan_spi_rcv_bytes()~~ (Unsupported)

slave mode, receive the bytes

Prototype:

```
u16_t wlan_spi_rcv_bytes(u8_t * buf, u16_t length)
```

Returns	Parameters
	rcvbuf: the buffer that want to receive data length: the buffer length

2.7.8 wlan_gpio_config()

Configures the operational direction (Input or Output) for a specific GPIO pin.

Functional Description:

- Direction Control: Sets whether a physical pin acts as a digital input (reading external signals) or a digital output (driving external circuits).
- Initialization: This function should be called before performing any read or write operations on the pin.

Prototype:

```
void wlan_gpio_config(wlan_gpio_t gpio, gpio_direction_t dir)
```

Parameters	Description
gpio	The target GPIO pin index (refer to wlan_gpio_t).
dir	The desired direction: `GPIO_INPUT` (0) or `GPIO_OUTPUT` (1).

Example Usage:

```
// Configure GPIO 2 as an output pin
wlan_gpio_config(WLAN_GPIO_2, GPIO_OUTPUT);
```

2.7.8.1 Enum wlan_gpio_t Definition

Available GPIO pins on the module.

Value	Name
0-9	`WLAN_GPIO_0` to `WLAN_GPIO_9`

```
typedef enum gpio_e
{
    WLAN_GPIO_0,
    WLAN_GPIO_1,
```

```
    // ...
    WLAN_GPIO_9,
} wlan_gpio_t;
```

2.7.8.2 Enum gpio_direction_t Definition

Value	Name	Description
0	GPIO_INPUT	Configure pin as Input.
1	GPIO_OUTPUT	Configure pin as Output.

```
typedef enum gpio_direction_e
{
    GPIO_INPUT,
    GPIO_OUTPUT
} gpio_direction_t;
```

2.7.9 wlan_gpio_set()

Set GPIO value

Prototype:

```
void wlan_gpio_set(wlan_gpio_t gpio, u8_t value)
```

Returns	Parameters
	typedef enum gpio_e (wlan_gpio_t)

2.7.10 wlan_gpio_get()

Get GPIO value.

Prototype:

```
u8_t wlan_gpio_get(wlan_gpio_t gpio)
```

Returns	Parameters
0 or 1	typedef enum gpio_e

2.8 Timer APIs

The Timer APIs provide a simplified interface to the underlying FreeRTOS software timer service. These functions allow applications to schedule periodic or one-shot events without blocking the main execution threads.

Key characteristics include:

- Software-Based: Timers are managed by the FreeRTOS kernel and share the context of the Timer Service Task.
- Efficiency: Suitable for low-frequency housekeeping tasks, timeouts, and state machine updates.
- Capacity: The current SDK implementation supports a fixed pool of 5 user timers (Indices 0–4).

The following table summarizes the available timer APIs:

Function	Description
wlan_timer_config()	Allocates and configures a new software timer instance.
wlan_timer_start()	Starts or restarts a configured timer.
wlan_timer_stop()	Stops a running timer.
wlan_timer_destroy()	Deletes a software timer instance.

2.8.1 wlan_timer_config()

Creates and configures a new software timer instance. The SDK uses an internal array to manage up to 5 concurrent timers.

Functional Description:

- Timer Creation: Allocates a FreeRTOS software timer and assigns it a unique index.
- Capacity: Supports a maximum of 5 timers (Indices 0 to 4).
- Execution Context: The callback function executes within the FreeRTOS Timer Service Task (Daemon Task). Avoid long-running or blocking operations in the callback.
- Reload Behavior: Can be configured as a periodic (auto-reload) or one-shot timer.

Prototype:

```
unsigned char wlan_timer_config(const char * const pTimerName,
                                unsigned long TimerPeriodInTicks,
                                unsigned long uxAutoReload,
                                void * const pvTimerID,
                                timer_callback pxCallbackFunction)
```

Returns	Parameters
unsigned char: The assigned **Timer Index (0-4)**. This index acts as the handle for starting and stopping the timer.	**pTimerName**: A text name for the timer (useful for debugging). **TimerPeriodInTicks**: The timer period in ticks. Use `pdMS_TO_TICKS(ms)` for millisecond conversion. **uxAutoReload**: `1` for periodic mode (auto-reload); `0` for one-shot mode. **pvTimerID**: User identifier assigned to the timer. **pxCallbackFunction**: The function to be executed when the timer expires. Refer to timer_callback.

2.8.1.1 Callback timer_callback Definition

`timer_callback` is the callback function executed when the timer expires.

Name	Signature	Description
`timer_callback`	`void (*func)(TimerHandle_t xTimer)`	A pointer to a function that takes no arguments and returns nothing.

```
typedef void (* timer_callback)(TimerHandle_t xTimer);
```

TimerHandle_t xTimer:

Handle of the software timer that expired and caused this callback to be executed.

The handle can be used to identify the timer instance, retrieve the user-defined timer ID (`pvTimerID`), or perform timer-specific operations.

2.8.2 wlan_timer_start()

Activates a previously configured software timer.

Functional Description:

- Activation: Sends a command to the FreeRTOS timer daemon task to start the timer associated with the given index.
- Behavior:
- If the timer is stopped, it starts running.
- If the timer is already running, this function re-starts the timer (resetting its expiry time relative to the current time).
- Context: Can be called from any user task.

Prototype:

```
void wlan_timer_start(unsigned char index)
```

Precondition: The timer must be configured first using wlan_timer_config to obtain a valid index.

Returns	Parameters
`void`	**index**: The Timer Index (0-4) returned by `wlan_timer_config`.

2.8.3 wlan_timer_stop()

Stops a running software timer.

Functional Description:

- Deactivation: Transitions the timer associated with the given index to the dormant state.
- Effect: The timer will no longer expire, and its associated callback function will not be executed until the timer is started again.
- Idle Safety: If the timer is already stopped (dormant), calling this function has no effect.

Prototype:

```
void wlan_timer_stop(unsigned char index)
```

Precondition: The timer must be configured first using wlan_timer_config to obtain a valid index.

Returns	Parameters
`void`	**index**: The Timer Index (0-4) returned by `wlan_timer_config`.

2.8.4 wlan_timer_destroy()

Deletes a software timer instance and releases its allocated resources.

Functional Description:

- Resource Cleanup: Permanently deletes the underlying FreeRTOS timer and clears the internal handle in the SDK's timer pool.
- Index Management: Once destroyed, the index (0-4) becomes available for a new `wlan_timer_config()` call.

Prototype:

```
void wlan_timer_destroy(unsigned char index)
```

Precondition: The timer must have been previously configured via wlan_timer_config.

Returns	Parameters
`void`	**index**: The Timer Index (0-4) to be destroyed.

2.8.5 Example: Timer Usage

The following code demonstrates how to configure, start, and use a software timer.

Note: The `timer_setup_example` function is intended to be called as a subroutine (e.g., from `main` or an existing Task), not as a standalone Task. Therefore, it returns naturally.

```
#include "ach118x.h"
#include "FreeRTOS.h"
#include "timers.h"
#include "wlan_driver.h" // For timer APIs
#include <stdio.h>

// Global handle to store the timer index so it can be used later (e.g. to
stop it)
static unsigned char g_my_timer_idx;

// Timer callback function
void my_timer_callback(TimerHandle_t xTimer)
{
    printf("Timer callback executed!\n");
}

// Initialization function (Call this from your Main Task)
void timer_setup_example(void)
{
    unsigned long timer_period_ms = 1000;
    unsigned long auto_reload = 1; // 1 for periodic, 0 for one-shot
    int timer_id = 1;

    // Convert ms to ticks
    unsigned long period_ticks = pdMS_TO_TICKS(timer_period_ms);

    // 1. Configure the timer
    // Returns index 0-4 on success
    g_my_timer_idx = wlan_timer_config("MyTimer",
```

```
                                        period_ticks,
                                        auto_reload,
                                        (void *)(long)timer_id,
                                        my_timer_callback);

    printf("Timer Configured. Index: %d\n", g_my_timer_idx);

    // 2. Start the timer
    // The timer runs in the FreeRTOS Timer Daemon Task, so it continues
    // running even after this function returns.
    wlan_timer_start(g_my_timer_idx);
    printf("Timer Started.\n");
}

void stop_my_timer(void)
{
    // Example of stopping the timer using the stored index
    wlan_timer_stop(g_my_timer_idx);
}
```

2.9 MQTT APIs

The MQTT APIs provide a client implementation of the MQTT protocol. They enable the module to communicate with IoT cloud platforms or local brokers using a lightweight publish/subscribe messaging model.

Key features include:

- APIs to connect, disconnect, manage session.
- Support for SSL/TLS encryption (one-way and mutual authentication) via certificate configuration.
- Flexible publish and subscribe capabilities with Quality of Service (QoS) levels 0, 1, and 2.
- Connection and messaging operations are non-blocking.

The following table summarizes the available MQTT APIs:

Function	Description
wlan_mqtt_set_server_ip()	Configures the Broker IP or Domain Name.
wlan_mqtt_set_server_port()	Sets the Broker Port.
~~wlan_mqtt_set_client_id()~~	**(Unsupported)** Configures the Client ID.
~~wlan_mqtt_set_keep_alive()~~	**(Unsupported)** Configures the Keep Alive interval.
wlan_mqtt_set_username()	Sets the username.
wlan_mqtt_set_password()	Sets the password.
wlan_mqtt_set_pub_topic()	Sets the default publish topic.
wlan_mqtt_set_sub_topic()	Sets the subscription topic.
wlan_mqtt_set_qos()	Configures the QoS level.
wlan_mqtt_set_auth_type()	Selects the authentication mode.

Function	Description
wlan_mqtt_set_ca_cert()	Sets the CA certificate filename.
wlan_mqtt_set_client_cert()	Sets the Client certificate filename.
wlan_mqtt_set_client_key()	Sets the Client private key filename.
wlan_mqtt_connect()	Initiates the connection to the broker.
wlan_mqtt_disconnect()	Terminates the MQTT session.
wlan_mqtt_get_status()	Retrieves connection status.
wlan_mqtt_publish()	Sends a message payload.
wlan_mqtt_subscribe()	Subscribes to a topic.
wlan_mqtt_unsubscribe()	Unsubscribes from a topic.

2.9.1 wlan_mqtt_set_server_ip()

Configures the address of the target MQTT broker.

Functional Description:

- Address Support: Sets the destination broker using either an IPv4 address (e.g., "192.168.1.100") or a fully qualified domain name (e.g., "iot.eclipse.org").
- Persistent Storage: The configuration is saved to the system's non-volatile memory and will persist across reboots.
- Validation: Performs a length check on the input string to ensure system compatibility.

Prototype:

```
enum result_type wlan_mqtt_set_server_ip(const char *ip_or_domain)
```

Returns	Parameters
result_type: `NO_ERR`: Success. `PARA_ERR`: Invalid parameter (e.g., string length > 64). `VALUE_ERR`: Storage failure.	**ip_or_domain**: Pointer to a null-terminated string containing the IP or domain name. **Max length: 64 characters.**

Example Usage:

```
// Configure the MQTT broker address
if (wlan_mqtt_set_server_ip("broker.example.com") == NO_ERR) {
    wlan_printf("MQTT Server IP updated successfully.\n");
}
```

2.9.2 wlan_mqtt_set_server_port()

Configures the broker port for MQTT authentication and connection.

Functional Description:

- Parameter Setup: Sets the broker port used by the MQTT client.
- Persistent Storage: The value is saved to non-volatile memory and persists across system restarts.

- Validation: Verifies the input format and range before saving.

Prototype:

```
enum result_type wlan_mqtt_set_server_port(int port)
```

Returns	Parameters
result_type: `NO_ERR`: Success. `PARA_ERR`: Invalid parameter. `VALUE_ERR`: Storage failure.	**port**: int: Port number (Range: 1-65535, Default: 1883)

Example Usage:

```
if (wlan_mqtt_set_server_port(1883) == NO_ERR) {
    wlan_printf("wlan_mqtt_set_server_port updated successfully.\n");
}
```

2.9.3 wlan_mqtt_set_client_id()

Configures the unique Client Identifier (Client ID) for the MQTT session.

Functional Description:

- Identity: Sets the string used to uniquely identify this client to the broker.
- Uniqueness: It is critical that each device has a unique Client ID (e.g., based on MAC address). If two clients connect with the same ID, the broker will disconnect the earlier one.
- Persistent Storage: Saved to non-volatile memory.

Prototype:

```
enum result_type wlan_mqtt_set_client_id(const char *client_id)
```

Returns	Parameters
result_type: `NO_ERR`: Success. `PARA_ERR`: Invalid parameter (e.g., length > 64).	**client_id**: Pointer to a null-terminated string. **Max length: 64 characters.**

Example Usage:

```
// Use MAC address as Client ID
wlan_mqtt_set_client_id("ACH118x-A1B2C3");
```

2.9.4 wlan_mqtt_set_keep_alive()

Configures the MQTT Keep Alive interval.

Functional Description:

- Heartbeat: Defines the maximum time interval (in seconds) that can elapse between two messages sent by the client.
- Ping: If no data is sent within this period, the client will automatically send a PINGREQ packet to maintain the connection.

- Default: If not set, the default is typically 60 seconds.

Prototype:

```
enum result_type wlan_mqtt_set_keep_alive(int seconds)
```

Returns	Parameters
result_type: `NO_ERR`: Success. `PARA_ERR`: Invalid value (e.g., < 0).	**seconds**: Keep Alive interval in seconds (0-65535). 0 disables the mechanism.

Example Usage:

```
// Set Keep Alive to 120 seconds
wlan_mqtt_set_keep_alive(120);
```

2.9.5 wlan_mqtt_set_username()

Configures the username for MQTT authentication and connection.

Functional Description:

- Parameter Setup: Sets the username used by the MQTT client.
- Persistent Storage: The value is saved to non-volatile memory and persists across system restarts.
- Validation: Verifies the input format and range before saving.

Prototype:

```
enum result_type wlan_mqtt_set_username(const char* username)
```

Returns	Parameters
result_type: `NO_ERR`: Success. `PARA_ERR`: Invalid parameter. `VALUE_ERR`: Storage failure.	**username**: const char*: Pointer to null-terminated string (Max length: 64 characters)

Example Usage:

```
if (wlan_mqtt_set_username("my_user") == NO_ERR) {
    wlan_printf("wlan_mqtt_set_username updated successfully.\n");
}
```

2.9.6 wlan_mqtt_set_password()

Configures the password for MQTT authentication and connection.

Functional Description:

- Parameter Setup: Sets the password used by the MQTT client.
- Persistent Storage: The value is saved to non-volatile memory and persists across system restarts.

- Validation: Verifies the input format and range before saving.

Prototype:

```
enum result_type wlan_mqtt_set_password(const char* password)
```

Returns	Parameters
result_type: `NO_ERR`: Success. `PARA_ERR`: Invalid parameter. `VALUE_ERR`: Storage failure.	**password**: const char*: Pointer to null-terminated string (Max length: 64 characters)

Example Usage:

```
if (wlan_mqtt_set_password("my_secret_pass") == NO_ERR) {
    wlan_printf("wlan_mqtt_set_password updated successfully.\n");
}
```

2.9.7 wlan_mqtt_set_pub_topic()

Configures the default topic used for outgoing (published) MQTT messages.

Functional Description:

- Topic Setup: Sets the primary MQTT topic name that the module will use when sending data.
- Length Limitation: The topic string is strictly limited to a maximum of 20 characters.
- Persistent Storage: The configuration is saved to non-volatile memory and remains active across system restarts.

Prototype:

```
enum result_type wlan_mqtt_set_pub_topic(const char *topic)
```

Returns	Parameters
result_type: `NO_ERR`: Success. `PARA_ERR`: Invalid parameter (e.g., length > 20 characters). `VALUE_ERR`: Storage failure.	**topic**: Pointer to a null-terminated string containing the default publish topic. **Max length: 20 characters.**

Example Usage:

```
// Set the default publish topic
if (wlan_mqtt_set_pub_topic("home/sensor/status") == NO_ERR) {
    wlan_printf("Default publish topic configured.\n");
}
```

2.9.8 wlan_mqtt_set_sub_topic()

Configures the default topic the module will subscribe to upon establishing an MQTT connection.

Functional Description:

- Topic Configuration: Sets the target MQTT topic string for incoming messages.
- Design Limitation: Current SDK architecture supports only one (1) active subscription topic. Multiple calls to this API will overwrite any previously configured topic.
- Persistent Storage: The topic name is saved to non-volatile memory.

Prototype:

```
enum result_type wlan_mqtt_set_sub_topic(const char *topic)
```

Returns	Parameters
result_type: `NO_ERR`: Success. `PARA_ERR`: Invalid parameter (e.g., length > 20 characters). `VALUE_ERR`: Storage failure.	**topic**: Pointer to a null-terminated string containing the subscription topic. **Max length: 20 characters.**

Example Usage:

```
// Configure the single subscription topic
if (wlan_mqtt_set_sub_topic("home/sensor/cmd") == NO_ERR) {
    wlan_printf("Subscription topic set. Note: This overwrites any previous
 topic.\n");
}
```

2.9.9 wlan_mqtt_set_qos()

Configures the Quality of Service (QoS) level for MQTT messages.

Functional Description:

- QoS Levels: Sets the reliability level for publishing and subscribing.
  - `0`: At most once (Fire and forget).
  - `1`: At least once (Acknowledged).
  - `2`: Exactly once (Assured delivery).
- Persistent Storage: Saved to non-volatile memory.

Prototype:

```
enum result_type wlan_mqtt_set_qos(int qos)
```

Returns	Parameters
result_type: `NO_ERR`: Success. `PARA_ERR`: Invalid QoS value (must be 0, 1, or 2).	**qos**: The desired MQTT QoS level (0, 1, or 2).

Example Usage:

```
// Set QoS level to 1 (At least once)
wlan_mqtt_set_qos(1);
```

2.9.10 wlan_mqtt_set_auth_type()

Configures the authentication mode for the MQTT connection.

Functional Description:

- Mode Selection: Defines whether the connection requires a username/password or SSL/TLS certificates.
- Supported Types:
  - `0`: User and Password authentication.
  - `1`: SSL/TLS Server Authentication (One-way).
  - `2`: SSL/TLS Mutual Authentication (Two-way).
  - `4`: No Authentication / Anonymous.
- Persistent Storage: Saved to non-volatile memory.

Prototype:

```
enum result_type wlan_mqtt_set_auth_type(int type)
```

Returns	Parameters
result_type: `NO_ERR`: Success. `PARA_ERR`: Invalid authentication type.	**type**: The authentication mode index (0, 1, 2, or 4).

Example Usage:

```
// Enable Username and Password authentication
wlan_mqtt_set_auth_type(0);
```

2.9.11 wlan_mqtt_set_ca_cert()

Sets the filename for the CA Certificate used in SSL/TLS encrypted MQTT connections.

Functional Description:

- Filename Configuration: Stores the name of the file that contains the CA Certificate data.
- Filesystem Requirement: This API only sets the filename reference. The actual certificate/key file must be pre-uploaded to the module's internal filesystem before establishing a connection.
- Persistent Storage: The filename is saved to the system configuration and persists across reboots.

Prototype:

```
enum result_type wlan_mqtt_set_ca_cert(const char *filename)
```

Returns	Parameters
result_type: `NO_ERR`: Success. `PARA_ERR`: Invalid parameter	**filename**: Pointer to a null-terminated string containing the file name. **Max length: 64 characters.**

Returns	Parameters
(e.g., filename length > 64). `VALUE_ERR`: Storage failure.	

Example Usage:

```
// Configure the CA Certificate filename
// Ensure "ca.crt" has been uploaded to the module filesystem.
if (wlan_mqtt_set_ca_cert("ca.crt") == NO_ERR) {
    wlan_printf("CA Certificate filename configured successfully.\n");
}
```

2.9.12 wlan_mqtt_set_client_cert()

Sets the filename for the Client Certificate used in SSL/TLS encrypted MQTT connections.

Functional Description:

- Filename Configuration: Stores the name of the file that contains the Client Certificate data.
- Filesystem Requirement: This API only sets the filename reference. The actual certificate/key file must be pre-uploaded to the module's internal filesystem before establishing a connection.
- Persistent Storage: The filename is saved to the system configuration and persists across reboots.

Prototype:

```
enum result_type wlan_mqtt_set_client_cert(const char *filename)
```

Returns	Parameters
result_type: `NO_ERR`: Success. `PARA_ERR`: Invalid parameter (e.g., filename length > 64). `VALUE_ERR`: Storage failure.	**filename**: Pointer to a null-terminated string containing the file name. **Max length: 64 characters.**

Example Usage:

```
// Configure the Client Certificate filename
// Ensure "client.crt" has been uploaded to the module filesystem.
if (wlan_mqtt_set_client_cert("client.crt") == NO_ERR) {
    wlan_printf("Client Certificate filename configured successfully.\n");
}
```

2.9.13 wlan_mqtt_set_client_key()

Sets the filename for the Client Private Key used in SSL/TLS encrypted MQTT connections.

Functional Description:

- Filename Configuration: Stores the name of the file that contains the Client Private Key data.

- Filesystem Requirement: This API only sets the filename reference. The actual certificate/key file must be pre-uploaded to the module's internal filesystem before establishing a connection.
- Persistent Storage: The filename is saved to the system configuration and persists across reboots.

Prototype:

```
enum result_type wlan_mqtt_set_client_key(const char *filename)
```

Returns	Parameters
result_type: `NO_ERR`: Success. `PARA_ERR`: Invalid parameter (e.g., filename length > 64). `VALUE_ERR`: Storage failure.	**filename**: Pointer to a null-terminated string containing the file name. **Max length: 64 characters.**

Example Usage:

```
// Configure the Client Private Key filename
// Ensure "client.key" has been uploaded to the module filesystem.
if (wlan_mqtt_set_client_key("client.key") == NO_ERR) {
    wlan_printf("Client Private Key filename configured successfully.\n");
}
```

2.9.14 wlan_mqtt_connect()

Initiates a connection request to the configured MQTT broker.

Functional Description:

- Asynchronous Initiation: This function triggers the MQTT connection state machine. It returns immediately after the request is queued and does not wait for the actual network handshake to complete.
- Automatic Handshake: Upon a successful call, the module attempts to establish a TCP connection and perform the MQTT CONNECT exchange using the previously configured server, port, and credentials.
- Status Monitoring: After calling this function, the application can monitor the connection state using wlan_mqtt_get_status() (polling) or the Signal Framework (asynchronous notifications).
- No Auto-Reconnect: If the MQTT connection is lost after being established, the SDK will not attempt to reconnect automatically. The application layer must detect the disconnection (via polling or signals) and explicitly call `wlan_mqtt_connect()` again to re-establish the session.
- UART Bypass Mode: Once the MQTT connection is successfully established, the module's UART0 interface automatically enters Bypass Mode.
    - Behavior: Any raw data received on UART0 from the host is directly encapsulated into MQTT packets and published to the broker (using the default publish topic).
    - Exiting Bypass Mode: To return UART0 to Command Mode (allowing AT commands), the host must send the specific escape sequence: `^#^$^%`.

Precondition:

The device must have a valid network connection (refer to wlan_sta_join).

Protocol Mode Selection: Ensure the module's protocol stack is configured for MQTT operation rather than standard TCP. This is controlled by the `LINKTYPE(0:TCP/1:MQTT)` system variable (`AMP_VARID_LINK_TYPE`, see var_id), which can be adjusted using the wlan_config_info() API. If the system remains in TCP mode (default), the connection attempt will be rejected with the error: *"Current Work Mode is NOT MQTT mode"*.

Broker parameters (IP, Port) and Client credentials (Username, Password) must be configured using the respective `wlan_mqtt_set_*` functions.

Prototype:

```
enum result_type wlan_mqtt_connect(void)
```

Returns	Parameters
`NO_ERR` (0): Connection request successfully initiated.	`void`

2.9.15 wlan_mqtt_disconnect()

Terminates the active MQTT session and closes the network connection to the broker.

Functional Description:

- Session Termination: Sends an MQTT DISCONNECT packet to the broker (if the link is still active) and closes the underlying TCP socket.
- Asynchronous Operation: Like the connect API, this is an asynchronous request. The module will transition to a disconnected state in the background.

Precondition: An MQTT connection have been initiated via wlan_mqtt_connect.

Prototype:

```
enum result_type wlan_mqtt_disconnect(void)
```

Returns	Parameters
`NO_ERR` (0): Disconnect request accepted.	`void`

Example Usage:

```
// Gracefully close the MQTT connection
wlan_mqtt_disconnect();
wlan_printf("MQTT disconnection initiated.\n");
```

2.9.16 wlan_mqtt_get_status()

Retrieves the real-time connection status of the MQTT client.

Functional Description:

- Connectivity Check: Directly queries the MQTT stack to verify if the client is currently connected to the broker.
- Usage: Recommended for business logic to verify link health before publishing or subscribing.

Prototype:

```
u8_t wlan_mqtt_get_status(void)
```

Returns	Parameters
`1`: Connected. `0`: Disconnected.	`void`

2.9.17 Monitoring MQTT Connection

In addition to polling with wlan_mqtt_get_status(), the application can use the **Signal Framework** to receive asynchronous notifications about the MQTT connection state.

- SIG_SDK_MQTT_CONNECT_OK: Triggered when the client successfully connects to the Broker.
- SIG_SDK_MQTT_DISCONNECT: Triggered when the connection to the Broker is lost or terminated.

For details on how to subscribe to and handle these events, refer to the Signal Framework section.

2.9.18 wlan_mqtt_publish()

Publishes a message payload to a specific MQTT topic.

Functional Description:

- Data Transmission: Packages the provided message and sends it to the broker under the specified topic name.
- Topic Routing: The `topic` string determines which subscribers will receive the message.
- Payload: Supports arbitrary binary or text data via the `message` pointer and `len` parameter.

Important Constraints:

- Topic Length: The topic string must not exceed 20 characters.
- Single Subscription Only: The SDK supports only one active subscription topic; calling this API replaces the previous topic.
- Payload Size: While the SDK handles various sizes, it is recommended to keep the message payload under 64 bytes for optimal performance on this module.

Prototype:

```
enum result_type wlan_mqtt_publish(const char *topic, const char *message,
int len)
```

Precondition: An MQTT connection must be established via wlan_mqtt_connect.

Returns	Parameters
result_type: `NO_ERR`: Success. `SEND_ERR`: Failed to transmit the packet. `PARA_ERR`: Invalid topic (length > 20) or NULL pointer.	**topic**: Pointer to a null-terminated string (Max 20 chars). **message**: Pointer to the data payload. **len**: Length of the message payload in bytes.

Example Usage:

```
char *payload = "Temp: 24C";
if (wlan_mqtt_publish("sensors/data", payload, strlen(payload)) == NO_ERR)
{
    wlan_printf("Data published successfully.\n");
}
```

2.9.19 wlan_mqtt_subscribe()

Subscribes to a specific MQTT topic to receive messages.

Functional Description:

- Subscription: Sends a SUBSCRIBE packet to the broker for the specified topic.
- Message Reception: Once subscribed, messages published to this topic by other clients will be received by the module.
- Topic Length: The topic string must not exceed 20 characters.

Prototype:

```
enum result_type wlan_mqtt_subscribe(const char *topic, int rfu)
```

Precondition: An MQTT connection must be established via wlan_mqtt_connect.

Returns	Parameters
result_type: `NO_ERR`: Success. `SEND_ERR`: Subscription request failed. `PARA_ERR`: Invalid topic or NULL pointer.	**topic**: Pointer to a null-terminated string (Max 20 chars). **rfu**: Reserved for Future Use. Pass `0`.

Example Usage:

```
// Subscribe to "home/cmd" (rfu = 0)
if (wlan_mqtt_subscribe("home/cmd", 0) == NO_ERR) {
    wlan_printf("Subscribed to home/cmd successfully.\n");
}
```

2.9.20 wlan_mqtt_unsubscribe()

Unsubscribes from a specific MQTT topic.

Functional Description:

- Unsubscription: Sends an UNSUBSCRIBE packet to the broker.
- Effect: The module will stop receiving messages for this topic.

Prototype:

```
enum result_type wlan_mqtt_unsubscribe(const char *topic)
```

Precondition: An MQTT connection must be established.

Returns	Parameters
result_type: `NO_ERR`: Success. `SEND_ERR`: Request failed. `PARA_ERR`: Invalid topic.	**topic**: Pointer to the topic string to unsubscribe from.

Example Usage:

```
wlan_mqtt_unsubscribe("home/cmd");
```

2.9.21 MQTT Connection Sequence

The following steps describe how to establish an MQTT connection (assuming the device is already connected to an AP or Mesh network):

Switch Protocol Mode: Enable the MQTT stack by setting the `LINKTYPE` variable (var 55) to `1` using the wlan_config_info() API.

Configure MQTT Server information:

- `wlan_mqtt_set_server_ip` to set Server IP or Domain.
- `wlan_mqtt_set_server_port` to set Server Port.

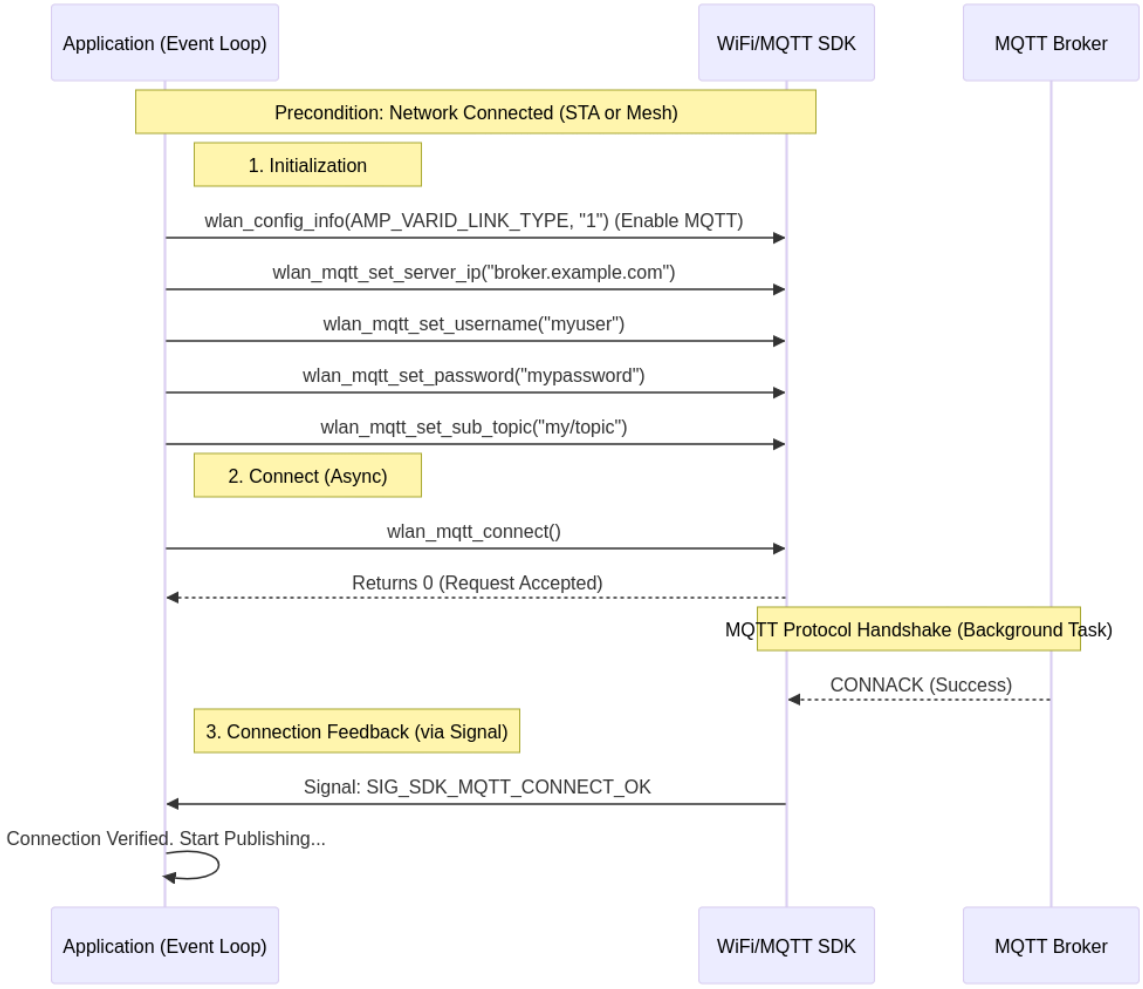
Configure MQTT Client information (Username, Password).

Configure MQTT QoS level.

Subscribe to MQTT topic.

Enable connection by calling `wlan_mqtt_connect`.

Sequence Diagram:



2.9.21.1 Example: MQTT Connection

The following C code demonstrates the sequence to configure and connect the MQTT client, assuming the device has already successfully joined a network.

```
void mqtt_start_sequence(void)
{
    // 0. Precondition: Ensure Wi-Fi is connected
    if (wlan_sta_status() == false) {
        wlan_printf("Error: Network not connected.");
        return;
    }

    wlan_printf("Starting MQTT Configuration...");

    // 1. Switch Protocol Stack to MQTT (LinkType = 1)
    // This is CRITICAL. Without this, the system stays in TCP mode.
    wlan_config_info(AMP_VARID_LINK_TYPE, "1");

    // 2. Configure Broker Settings
```

```
    wlan_mqtt_set_server_ip("broker.example.com");
    wlan_mqtt_set_server_port(1883);

    // 3. Configure Client Credentials
    wlan_mqtt_set_username("myuser");
    wlan_mqtt_set_password("mypassword");

    // 4. Configure Session Parameters
    wlan_mqtt_set_qos(1);                 // QoS Level 1
    wlan_mqtt_set_sub_topic("cmd/topic"); // Topic to subscribe on connect
    wlan_mqtt_set_pub_topic("data/topic");// Default publish topic

    // 5. Initiate Connection (Async)
    // To confirm success, ensure you have subscribed to SIG_SDK_MQTT_CONNE
CT_OK
    // via wlan_sdk_subscribe().
    if (wlan_mqtt_connect() == NO_ERR) {
        wlan_printf("MQTT Connect Request Sent. Waiting for Signal...\n");
    } else {
        wlan_printf("Failed to initiate connection.\n");
    }
}
```

2.10 Mesh Networking Overview & Configuration

This section provides a guide to configuring and using the Mesh networking capabilities (based on IEEE 802.11s) provided by the SDK. Unlike standard Station (STA) or Access Point (AP) modes, Mesh networking allows devices to form a self-organizing, multi-hop network.

2.10.1 Overview

To establish a Mesh network, devices must be configured to operate in Mesh Point (MP) mode. All configuration is performed by modifying system variables using the wlan_config_info() API. Because these settings modify the underlying Wi-Fi stack behavior, a system restart is mandatory for the changes to take effect.

Once the system has restarted, the application can use the wlan_mesh_get_status() API to programmatically monitor the association state and verify the link to the Mesh network gateway.

2.10.2 Key Configuration Variables

The table below details the specific system variables required to configure a Mesh node. To apply these settings, pass the corresponding `var_id` macros to the `wlan_config_info()` function.

For a Mesh network to form, the `MESH_ID`, `Channel`, and `AuthType` **must be identical** across all participating nodes.

var_id	Name	Value / Description
`AMP_VARID_DEVICE_MODE`	`DeviceMode`	Must be set to **`MP`** (Mesh Point) or `AP_MP` (Concurrent AP + Mesh).
`AMP_VARID_CHANNEL`	`Channel`	All Mesh nodes **must** be on the same radio channel (e.g., `1`, `6`, `161`).
`AMP_VARID_MESH_ID`	`MESH_ID`	A string identifier for the Mesh network (similar to SSID). Max 32 chars.
`AMP_VARID_MESH_AUTH_TYPE`	`MESH_AuthType`	Authentication method. `0`: Open, `2`: SAE (Secure Authentication of Equals).
`AMP_VARID_MESH_PASS_PHRASE`	`MESH_PassPhrase`	The password string for the Mesh network (required if AuthType is SAE).
`AMP_VARID_MP_MODE`	`MPMode`	Set to `1` to enable multi-point forwarding features.
`AMP_VARID_IP_ADDRESS`	`IPAddress`	Static IP address for the node (e.g., `192.168.10.x`).

2.10.3 wlan_mesh_get_status()

Checks the association status of the device within the Mesh network.

Functional Description:

- Gateway Verification: Confirms if the device is in a valid Mesh mode (`MP` or `AP_MP`) and has successfully established a link to a Mesh gateway.
- Path Discovery: Returns success only if at least one Mesh gate is visible in the routing table.

Prototype:

```
u8_t wlan_mesh_get_status(void)
```

Precondition: Device must be operating in Mesh mode (refer to Mesh Overview).

Returns	Parameters
`1`: Connected to mesh gateway. `0`: Not connected or mode mismatch.	`void`

2.10.4 Monitoring Mesh Connection

In addition to polling with wlan_mesh_get_status(), the application can use the Signal Framework to receive asynchronous notifications about the Mesh connection state.

- SIG_SDK_WLAN_CONNECTED: Triggered when the device successfully joins the Mesh network and finds a path to the Root Node.
- SIG_SDK_WLAN_DISCONNECTED: Triggered when the link to the Mesh network or the Root Node is lost.

For details on how to subscribe to and handle these events, refer to the Signal Framework section.

2.10.5 Configuration Workflow

To successfully start a Mesh node, the application should follow this strict sequence:

- Set Mode: Configure `DeviceMode` to `MP`.
- Set Channel: Ensure the `Channel` is set to the designated Mesh frequency.
- Configure Identity: Set the `MESH_ID` to the target network name.
- Configure Security: Set `MESH_AuthType` and `MESH_PassPhrase` to match the network credentials.
- Configure Network: Assign a unique static `IPAddress` and common `GateWay` to the device.
- Apply Changes: Call wlan_restart() to reboot the module and initialize the Mesh stack with the new settings.

Note: Mesh networking does not typically use DHCP for self-addressing in this SDK version. Static IP assignment is recommended to ensure reachability.

2.10.6 Example: Mesh Connection

The following steps describe the correct procedure to configure and enable a Mesh network connection using wlan_config_info(). **Note: A system restart is required for these changes to take effect.**

Configuration Steps:

- Configure Device Mode: Set to "MP" (Mesh Point) (AMP_VARID_DEVICE_MODE).
- Configure Channel: Set the operating channel number (AMP_VARID_CHANNEL).
- Configure Mesh ID: Set the Mesh Network ID string (AMP_VARID_MESH_ID).
- Configure Mesh Auth Type: Set authentication type (0/2) (AMP_VARID_MESH_AUTH_TYPE).
- Configure Mesh Passphrase: Set the mesh password (AMP_VARID_MESH_PASS_PHRASE).
- Configure IP Address: Set the static IP address (AMP_VARID_IP_ADDRESS).
- Configure NetMask: Set the subnet mask (AMP_VARID_NET_MASK).
- Configure Gateway: Set the gateway IP address (AMP_VARID_GATE_WAY).
- Restart System: Call wlan_restart() to apply the new configuration.

Code Example:

```
#include "wlan_config.h"
#include "wlan_com.h"
#include "wlan_sys.h"
#include <stdio.h>

int test_mesh_setup_example(int word_count, char **words)
{
    wlan_printf("Starting Mesh Configuration...\n");

    // 1. Configure Device Mode to "MP"
```

```
    wlan_config_info(AMP_VARID_DEVICE_MODE, "MP");
    printf("Set Device Mode(var %d) to MP\n", AMP_VARID_DEVICE_MODE);
    // 2. Configure Static IP Address
    wlan_config_info(AMP_VARID_IP_ADDRESS, "192.168.0.90");
    printf("Set Static IP Address to 192.168.0.90\n");

    // 3. Configure NetMask
    wlan_config_info(AMP_VARID_NET_MASK, "255.255.255.0");
    printf("Set NetMask to 255.255.255.0\n");
    // 4. Configure Gateway IP
    wlan_config_info(AMP_VARID_GATE_WAY, "192.168.0.1");
    printf("Set Gateway IP to 192.168.0.1\n");
    // 5. Configure Channel (e.g., Channel 161)

    wlan_config_info(AMP_VARID_CHANNEL, "161");

    printf("Set Channel to 161\n");

    // 6. Configure Mesh ID
    wlan_config_info(AMP_VARID_MESH_ID, "mymesh123456789");
    printf("Set Mesh ID to mymesh123456789\n");


    // 7. Configure Mesh Auth Type (e.g., 2 for Auth)
    wlan_config_info(AMP_VARID_MESH_AUTH_TYPE, "2");
    printf("Set Mesh Auth Type to 2\n");

    // 8. Configure Mesh Passphrase
    wlan_config_info(AMP_VARID_MESH_PASS_PHRASE, "12345678");
    printf("Set Mesh Passphrase to 12345678\n");

    printf("Waiting 1 second before restart...\n");
    vTaskDelay(pdMS_TO_TICKS(1000));

    // 9. Restart System to Apply Changes
    wlan_printf("Configuration complete. Restarting system...\n");
    wlan_restart();
}
```

3. Demo Application Implementation Guide

This document explains the implementation of the demo application logic within `vAppMainTask`, focusing on how it leverages the **Signal Framework** to handle system events asynchronously. This pattern serves as a reference for developing user applications on the WiFi SDK.

3.1 Overview

The `vAppMainTask` is the central orchestration task for the application. Instead of polling for status updates, it uses a **message queue** to receive real-time events (Signals) from the SDK. This design ensures the application is responsive and efficient.

Key Components:

- FreeRTOS Queue: A buffer to hold incoming events.
- Signal Subscriptions: Registering interest in specific system events (e.g., WiFi connection, MQTT data).
- Event Loop: A continuous loop that blocks until an event arrives, then processes it.

3.2 Implementation Logic

The implementation is found in `Application/src/amp_example.c` (referenced by `Application/src/wifi_app.c`).

3.2.1 Initialization

First, the task creates a FreeRTOS queue capable of holding `sdk_signal_t` structures.

```
void vAppMainTask(void *pvParameters)
{
    // Create a queue with a depth of 20 messages
    QueueHandle_t xAppQueue = xQueueCreate(20, sizeof(sdk_signal_t));
    if (xAppQueue == NULL) {
        // Handle error...
        return;
    }
```

3.2.2 Subscribing to Signals

The application explicitly subscribes to the events it needs. It uses `SDK_SUB_MODE_REPEAT` mode, which allows the application to "listen in" on events while letting the SDK continues its standard background processing.

> **Tip**: Since the application and the SDK share the same payload pointer in `REPEAT` mode, the application should **not** modify the original data. If modification is required for business logic, the application must create a local copy of the data first.

```
    // Subscribe to System Initialization event
    wlan_sdk_subscribe(SIG_SDK_SYS_INIT_DONE, xAppQueue, SDK_SUB_MODE_REPEA
T);

    // Subscribe to WiFi Join Events (Async Join Result)
    wlan_sdk_subscribe(SIG_SDK_JOIN_SUCESS, xAppQueue, SDK_SUB_MODE_REPEA
T);
    wlan_sdk_subscribe(SIG_SDK_JOIN_FAILED, xAppQueue, SDK_SUB_MODE_REPEA
T);

    // Subscribe to WiFi Connectivity events
    wlan_sdk_subscribe(SIG_SDK_WLAN_CONNECTED, xAppQueue, SDK_SUB_MODE_REPE
```

```
AT);
    wlan_sdk_subscribe(SIG_SDK_WLAN_DISCONNECTED, xAppQueue, SDK_SUB_MODE_R
EPEAT);
    wlan_sdk_subscribe(SIG_SDK_WLAN_IP_CHANGED, xAppQueue, SDK_SUB_MODE_REP
EAT);

    // Subscribe to MQTT events
    wlan_sdk_subscribe(SIG_SDK_MQTT_CONNECT_OK, xAppQueue, SDK_SUB_MODE_REP
EAT);
    wlan_sdk_subscribe(SIG_SDK_MQTT_DATA_DOWN, xAppQueue, SDK_SUB_MODE_REPE
AT);
```

3.2.3 The Event Loop

The core of the task is an infinite loop that waits for signals.

```
    for (;;) {
        // Block indefinitely (portMAX_DELAY) until a signal arrives
        if (xQueueReceive(xAppQueue, &msg, portMAX_DELAY) == pdPASS) {

            // Dispatch the signal to the handler function
            msg_processed_for_test(&msg, &net_status);

            // NOTE: In REPEAT mode, DO NOT free msg.data.
            // The system owns the data and will free it automatically.
        }
    }
}
```

3.3 Signal Processing (`msg_processed_for_test`)

The `msg_processed_for_test` function acts as a dispatcher, executing specific business logic based on the `msg.id`.

- `SIG_SDK_SYS_INIT_DONE`:
  - Action: Calls `start_wifi_example()`.
  - Logic:
    - STA Mode: Initiates the connection sequence by calling `test_wlan_sta_example()`, which configures the device and calls `wlan_sta_join()` with default credentials.
    - MP Mode: Validates Mesh parameters. If valid, the SDK will automatically initiate the Mesh joining process. Otherwise, re-configuration of Mesh parameters is required.
- `SIG_SDK_JOIN_SUCESS`:
  - Action: Calls `on_join_success()`.
  - Logic: `NULL`.
- `SIG_SDK_JOIN_FAILED`:

  - Action: Calls `on_join_fail()`.
  - Logic: Explicitly call `start_wifi_example` to initiate the reconnection process.

- `SIG_SDK_WLAN_CONNECTED`:
  - Action: Calls `on_wlan_connected()`.
  - Logic:
    - STA Mode: Logs the event. Waits for IP address assignment (via DHCP).
    - MP Mode: Immediately starts the MQTT application task (`mqtt_example_task`) as Mesh usually uses static IP.

- `SIG_SDK_WLAN_IP_CHANGED`:
  - Action: Calls `on_wlan_ip_changed()`.
  - Logic: Retrieves the new IP info. If the IP is valid, it starts the MQTT application task (`mqtt_example_task`).

- `SIG_SDK_WLAN_DISCONNECTED`:
  - Action: Calls `on_wlan_disconnected()`.
  - Logic:

    Disconnects MQTT (`wlan_mqtt_disconnect`).

    Explicitly calls `wlan_sta_unjoin()` to clean up.

    Calls `start_wifi_example()` to restart the connection loop.

- `SIG_SDK_MQTT_DATA_DOWN`:
  - Action: Calls `on_mqtt_data_down()`.
  - Logic: Parses the incoming MQTT message.

Note on Automatic Reconnection:

- STA Mode: If the application is intended to automatically restore connectivity with the AP, it must explicitly call `wlan_sta_join()` upon receiving either `SIG_SDK_JOIN_FAILED` or `SIG_SDK_WLAN_DISCONNECTED` signals.
- Mesh Point (MP) Mode: Upon receiving `SIG_SDK_WLAN_DISCONNECTED`, the SDK will automatically attempt to reconnect to the mesh network. No explicit application-level intervention is required.