Wi-Fi Driver Reference Manual

C API Reference

Generated by Doxygen 1.8.18

Chapter 1

Main Page

1.1 Introduction

NXP's WiFi functionality enables customers to quickly develop applications of interest to add connectivity to different sensors and appliances.

1.1.1 Developer Documentation

This manual provides developer reference documentation for WiFi driver and WLAN Connection Manager.

In addition to the reference documentation in this manual, you can also explore the source code.

Note

> The File Documentation provides documentation for all the APIs that are available in WiFi driver and connection manager.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 cli_command Struct Reference

Data Fields

- const char * name
- const char * help
- void(* function)(int argc, char **argv)

4.1.1 Detailed Description

Structure for registering CLI commands

4.1.2 Field Documentation

4.1.2.1 name

```
const char* cli_command::name
```

The name of the CLI command

4.1.2.2 help

```
const char* cli_command::help
```

The help text associated with the command

4.1.2.3 function

```
void(* cli_command::function) (int argc, char **argv)
```

The function that should be invoked for this command.

The documentation for this struct was generated from the following file:

- cli.h

4.2 ipv4_config Struct Reference

Data Fields

- enum address_types addr_type
- unsigned address
- unsigned gw
- unsigned netmask
- unsigned dns1
- unsigned dns2

4.2.1 Detailed Description

This data structure represents an IPv4 address

4.2.2 Field Documentation

4.2.2.1 addr_type

```
enum address_types ipv4_config::addr_type
```

Set to ADDR_TYPE_DHCP to use DHCP to obtain the IP address or ADDR_TYPE_STATIC to use a static IP. In case of static IP address ip, gw, netmask and dns members must be specified. When using DHCP, the ip, gw, netmask and dns are overwritten by the values obtained from the DHCP server. They should be zeroed out if not used.

4.2.2.2 address

```
unsigned ipv4_config::address
```

The system's IP address in network order.

4.2.2.3 gw

```
unsigned ipv4_config::gw
```

The system's default gateway in network order.

4.2.2.4 netmask

```
unsigned ipv4_config::netmask
```

The system's subnet mask in network order.

4.2.2.5 dns1

```
unsigned ipv4_config::dns1
```

The system's primary dns server in network order.

4.2.2.6 dns2

```
unsigned ipv4_config::dns2
```

The system's secondary dns server in network order.

The documentation for this struct was generated from the following file:

- wlan.h

4.3 ipv6_config Struct Reference

Data Fields

- unsigned address [4]
- unsigned char addr_type
- unsigned char addr_state

4.3.1 Detailed Description

This data structure represents an IPv6 address

4.3.2 Field Documentation

4.3.2.1 address

```
unsigned ipv6_config::address[4]
```

The system's IPv6 address in network order.

4.3.2.2 addr_type

```
unsigned char ipv6_config::addr_type
```

The address type: linklocal, site-local or global.

4.3.2.3 addr_state

```
unsigned char ipv6_config::addr_state
```

The state of IPv6 address (Tentative, Preferred, etc).

The documentation for this struct was generated from the following file:

- wlan.h

4.4 net_ip_config Struct Reference

Data Fields

- struct net_ipv6_config ipv6 [CONFIG_MAX_IPV6_ADDRESSES]
- struct net_ipv4_config ipv4

4.4.1 Detailed Description

Network IP configuration.

This data structure represents the network IP configuration for IPv4 as well as IPv6 addresses

4.4.2 Field Documentation

4.4.2.1 ipv6

```
struct net_ipv6_config net_ip_config::ipv6[CONFIG_MAX_IPV6_ADDRESSES]
```

The network IPv6 address configuration that should be associated with this interface.

4.4.2.2 ipv4

```
struct net_ipv4_config net_ip_config::ipv4
```

The network IPv4 address configuration that should be associated with this interface.

The documentation for this struct was generated from the following file:

- wm_net.h

4.5 net_ipv4_config Struct Reference

Data Fields

- enum net_address_types addr_type
- unsigned address
- unsigned gw
- unsigned netmask
- unsigned dns1
- unsigned dns2

4.5.1 Detailed Description

This data structure represents an IPv4 address

4.5.2 Field Documentation

4.5.2.1 addr_type

```
enum net_address_types net_ipv4_config::addr_type
```

Set to ADDR_TYPE_DHCP to use DHCP to obtain the IP address or ADDR_TYPE_STATIC to use a static IP. In case of static IP address ip, gw, netmask and dns members must be specified. When using DHCP, the ip, gw, netmask and dns are overwritten by the values obtained from the DHCP server. They should be zeroed out if not used.

4.5.2.2 address

```
unsigned net_ipv4_config::address
```

The system's IP address in network order.

4.5.2.3 gw

```
unsigned net_ipv4_config::gw
```

The system's default gateway in network order.

4.5.2.4 netmask

```
unsigned net_ipv4_config::netmask
```

The system's subnet mask in network order.

4.5.2.5 dns1

```
unsigned net_ipv4_config::dns1
```

The system's primary dns server in network order.

4.5.2.6 dns2

```
unsigned net_ipv4_config::dns2
```

The system's secondary dns server in network order.

The documentation for this struct was generated from the following file:

- wm_net.h

4.6 net_ipv6_config Struct Reference

Data Fields

- unsigned address [4]
- unsigned char addr_type
- unsigned char addr_state

4.6.1 Detailed Description

This data structure represents an IPv6 address

4.6.2 Field Documentation

4.6.2.1 address

```
unsigned net_ipv6_config::address[4]
```

The system's IPv6 address in network order.

4.6.2.2 addr_type

```
unsigned char net_ipv6_config::addr_type
```

The address type: linklocal, site-local or global.

4.6.2.3 addr_state

```
unsigned char net_ipv6_config::addr_state
```

The state of IPv6 address (Tentative, Preferred, etc).

The documentation for this struct was generated from the following file:

- wm_net.h

4.7 os_queue_pool_t Struct Reference

Data Fields

- int size

4.7.1 Detailed Description

Structure used for queue definition

4.7.2 Field Documentation

4.7.2.1 size

```
int os_queue_pool_t::size
```

Size of the queue

The documentation for this struct was generated from the following file:

- wm_os.h

4.8 os_thread_stack_t Struct Reference

Data Fields

- size_t size

4.8.1 Detailed Description

Structure to be used during call to the function os_thread_create(). Please use the macro os_thread_stack_define instead of using this structure directly.

4.8.2 Field Documentation

4.8.2.1 size

```
size_t os_thread_stack_t::size
```

Total stack size

The documentation for this struct was generated from the following file:

- wm_os.h

4.9 tx_ampdu_prot_mode_para Struct Reference

Data Fields

- int mode

4.9.1 Detailed Description

tx_ampdu_prot_mode parameters

4.9.2 Field Documentation

4.9.2.1 mode

```
int tx_ampdu_prot_mode_para::mode
```

set prot mode

The documentation for this struct was generated from the following file:

- wlan.h

4.10 txrate_setting Struct Reference

Data Fields

- t_u16 preamble: 2
- t_u16 bandwidth: 3
- t_u16 shortGI: 2
- t_u16 stbc: 1
- t_u16 dcm: 1
- t_u16 adv_coding: 1
- t_u16 doppler: 2
- t_u16 max_pktext: 2
- t_u16 reserverd: 2

4.10.1 Detailed Description

TX Rate Setting

4.10.2 Field Documentation

4.10.2.1 preamble

```
t_u16 txrate_setting::preamble
```

Preamble

4.10.2.2 bandwidth

```
t_u16 txrate_setting::bandwidth
```

Bandwidth

4.10.2.3 shortGI

`t_u16 txrate_setting::shortGI`

Short GI

4.10.2.4 stbc

`t_u16 txrate_setting::stbc`

STBC

4.10.2.5 dcm

`t_u16 txrate_setting::dcm`

DCM

4.10.2.6 adv_coding

`t_u16 txrate_setting::adv_coding`

Adv coding

4.10.2.7 doppler

`t_u16 txrate_setting::doppler`

Doppler

4.10.2.8 max_pktext

`t_u16 txrate_setting::max_pktext`

Max PK text

4.10.2.9 reserverd

`t_u16 txrate_setting::reserverd`

Reserved

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.11 wifi_11ax_config_t Struct Reference

Data Fields

- t_u8 band
- t_u16 id
- t_u16 len
- t_u8 ext_id
- t_u8 he_mac_cap [6]
- t_u8 he_phy_cap [11]
- t_u8 he_txrx_mcs_support [4]
- t_u8 val [4]

4.11.1 Detailed Description

Wi-Fi 11AX Configuration

4.11.2 Field Documentation

4.11.2.1 band

```
t_u8 wifi_11ax_config_t::band
```

Band

4.11.2.2 id

```
t_u16 wifi_11ax_config_t::id
```

tlv id of he capability

4.11.2.3 len

```
t_u16 wifi_11ax_config_t::len
```

length of the payload

4.11.2.4 ext_id

```
t_u8 wifi_11ax_config_t::ext_id
```

extension id

4.11.2.5 he_mac_cap

```
t_u8 wifi_11ax_config_t::he_mac_cap[6]
```

he mac capability info

4.11.2.6 he_phy_cap

```
t_u8 wifi_11ax_config_t::he_phy_cap[11]
```

he phy capability info

4.11.2.7 he_txrx_mcs_support

```
t_u8 wifi_11ax_config_t::he_txrx_mcs_support[4]
```

he txrx mcs support for 80MHz

4.11.2.8 val

```
t_u8 wifi_11ax_config_t::val[4]
```

val for PE thresholds

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.12 wifi_antcfg_t Struct Reference

Data Fields

- t_u32 * ant_mode
- t_u16 * evaluate_time
- t_u16 * current_antenna

4.12.1 Detailed Description

Type definition of wifi_antcfg_t

4.12.2 Field Documentation

4.12.2.1 ant_mode

```
t_u32* wifi_antcfg_t::ant_mode
```

Antenna Mode

4.12.2.2 evaluate_time

```
t_u16* wifi_antcfg_t::evaluate_time
```

Evaluate Time

4.12.2.3 current_antenna

```
t_u16* wifi_antcfg_t::current_antenna
```

Current antenna

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.13 wifi_auto_reconnect_config_t Struct Reference

Data Fields

- t_u8 reconnect_counter
- t_u8 reconnect_interval
- t_u16 flags

4.13.1 Detailed Description

Auto reconnect structure

4.13.2 Field Documentation

4.13.2.1 reconnect_counter

```
t_u8 wifi_auto_reconnect_config_t::reconnect_counter
```

Reconnect counter

4.13.2.2 reconnect_interval

```
t_u8 wifi_auto_reconnect_config_t::reconnect_interval
```

Reconnect interval

4.13.2.3 flags

```
t_u16 wifi_auto_reconnect_config_t::flags
```

Flags

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.14 wifi_bandcfg_t Struct Reference

Data Fields

- t_u16 config_bands
- t_u16 fw_bands

4.14.1 Detailed Description

Type definition of wifi_bandcfg_t

4.14.2 Field Documentation

4.14.2.1 config_bands

```
t_u16 wifi_bandcfg_t::config_bands
```

Infra band

4.14.2.2 fw_bands

```
t_u16 wifi_bandcfg_t::fw_bands
```

fw supported band

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.15 wifi_btwt_config_t Struct Reference

Data Fields

- t_u16 action
- t_u16 sub_id
- t_u8 nominal_wake
- t_u8 max_sta_support
- t_u16 twt_mantissa
- t_u16 twt_offset
- t_u8 twt_exponent
- t_u8 sp_gap

4.15.1 Detailed Description

Wi-Fi BTWT Configuration

4.15.2 Field Documentation

4.15.2.1 action

```
t_u16 wifi_btwt_config_t::action
```

Only support 1: Set

4.15.2.2 sub_id

```
t_u16 wifi_btwt_config_t::sub_id
```

Broadcast TWT AP config

4.15.2.3 nominal_wake

```
t_u8 wifi_btwt_config_t::nominal_wake
```

Range 64-255

4.15.2.4 max_sta_support

```
t_u8 wifi_btwt_config_t::max_sta_support
```

Max STA Support

4.15.2.5 twt_mantissa

```
t_u16 wifi_btwt_config_t::twt_mantissa
```

TWT Mantissa

4.15.2.6 twt_offset

```
t_u16 wifi_btwt_config_t::twt_offset
```

TWT Offset

4.15.2.7 twt_exponent

```
t_u8 wifi_btwt_config_t::twt_exponent
```

TWT Exponent

4.15.2.8 sp_gap

```
t_u8 wifi_btwt_config_t::sp_gap
```

SP Gap

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.16 wifi_cal_data_t Struct Reference

Data Fields

- t_u16 data_len
- t_u8 * data

4.16.1 Detailed Description

Calibration Data

4.16.2 Field Documentation

4.16.2.1 data_len

`t_u16 wifi_cal_data_t::data_len`

Calibration data length

4.16.2.2 data

`t_u8* wifi_cal_data_t::data`

Calibration data

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.17 wifi_chan_info_t Struct Reference

Data Fields

- t_u8 chan_num
- t_u16 chan_freq
- bool passive_scan_or_radar_detect

4.17.1 Detailed Description

Data structure for Channel attributes

4.17.2 Field Documentation

4.17.2.1 chan_num

`t_u8 wifi_chan_info_t::chan_num`

Channel Number

4.17.2.2 chan_freq

`t_u16 wifi_chan_info_t::chan_freq`

Channel frequency for this channel

4.17.2.3 passive_scan_or_radar_detect

```
bool wifi_chan_info_t::passive_scan_or_radar_detect
```

Passice Scan or RADAR Detect

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.18 wifi_chan_list_param_set_t Struct Reference

Data Fields

- t_u8 no_of_channels
- wifi_chan_scan_param_set_t chan_scan_param [1]

4.18.1 Detailed Description

Channel list parameter set

4.18.2 Field Documentation

4.18.2.1 no_of_channels

```
t_u8 wifi_chan_list_param_set_t::no_of_channels
```

number of channels

4.18.2.2 chan_scan_param

```
wifi_chan_scan_param_set_t wifi_chan_list_param_set_t::chan_scan_param[1]
```

channel scan array

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.19 wifi_chan_scan_param_set_t Struct Reference

Data Fields

- t_u8 chan_number
- t_u16 min_scan_time
- t_u16 max_scan_time

4.19.1 Detailed Description

Channel scan parameters

4.19.2 Field Documentation

4.19.2.1 chan_number

```
t_u8 wifi_chan_scan_param_set_t::chan_number
```

channel number

4.19.2.2 min_scan_time

```
t_u16 wifi_chan_scan_param_set_t::min_scan_time
```

minimum scan time

4.19.2.3 max_scan_time

```
t_u16 wifi_chan_scan_param_set_t::max_scan_time
```

maximum scan time

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.20 wifi_chanlist_t Struct Reference

Data Fields

- t_u8 num_chans
- wifi_chan_info_t chan_info [54]

4.20.1 Detailed Description

Data structure for Channel List Config

4.20.2 Field Documentation

4.20.2.1 num_chans

```
t_u8 wifi_chanlist_t::num_chans
```

Number of Channels

4.20.2.2 chan_info

```
wifi_chan_info_t wifi_chanlist_t::chan_info[54]
```

Channel Info

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.21 wifi_channel_desc_t Struct Reference

Data Fields

- t_u16 start_freq
- t_u8 chan_width
- t_u8 chan_num

4.21.1 Detailed Description

Data structure for Channel descriptor

Set CFG data for Tx power limitation

start_freq: Starting Frequency of the band for this channel
2407, 2414 or 2400 for 2.4 GHz
5000
4000
chan_width: Channel Width
20
chan_num : Channel Number

4.21.2 Field Documentation

4.21.2.1 start_freq

`t_u16 wifi_channel_desc_t::start_freq`

Starting frequency of the band for this channel

4.21.2.2 chan_width

`t_u8 wifi_channel_desc_t::chan_width`

Channel width

4.21.2.3 chan_num

`t_u8 wifi_channel_desc_t::chan_num`

Channel Number

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.22 wifi_clock_sync_gpio_tsf_t Struct Reference

Data Fields

- t_u8 clock_sync_mode
- t_u8 clock_sync_Role
- t_u8 clock_sync_gpio_pin_number
- t_u8 clock_sync_gpio_level_toggle
- t_u16 clock_sync_gpio_pulse_width

4.22.1 Detailed Description

Wi-Fi Clock sync configuration

4.22.2 Field Documentation

4.22.2.1 clock_sync_mode

```
t_u8 wifi_clock_sync_gpio_tsf_t::clock_sync_mode
```

clock sync Mode

4.22.2.2 clock_sync_Role

```
t_u8 wifi_clock_sync_gpio_tsf_t::clock_sync_Role
```

clock sync Role

4.22.2.3 clock_sync_gpio_pin_number

```
t_u8 wifi_clock_sync_gpio_tsf_t::clock_sync_gpio_pin_number
```

clock sync GPIO Pin Number

4.22.2.4 clock_sync_gpio_level_toggle

```
t_u8 wifi_clock_sync_gpio_tsf_t::clock_sync_gpio_level_toggle
```

clock sync GPIO Level or Toggle

4.22.2.5 clock_sync_gpio_pulse_width

```
t_u16 wifi_clock_sync_gpio_tsf_t::clock_sync_gpio_pulse_width
```

clock sync GPIO Pulse Width

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.23 wifi_cloud_keep_alive_t Struct Reference

Data Fields

- t_u8 mkeep_alive_id
- t_u8 enable
- t_u8 reset
- t_u8 cached
- t_u32 send_interval
- t_u16 retry_interval
- t_u16 retry_count
- t_u8 src_mac [MLAN_MAC_ADDR_LENGTH]
- t_u8 dst_mac [MLAN_MAC_ADDR_LENGTH]
- t_u32 src_ip
- t_u32 dst_ip
- t_u16 src_port
- t_u16 dst_port
- t_u16 pkt_len
- t_u8 packet [MKEEP_ALIVE_IP_PKT_MAX]

4.23.1 Detailed Description

Cloud keep alive information

4.23.2 Field Documentation

4.23.2.1 mkeep_alive_id

`t_u8 wifi_cloud_keep_alive_t::mkeep_alive_id`

Keep alive id

4.23.2.2 enable

`t_u8 wifi_cloud_keep_alive_t::enable`

Enable keep alive

4.23.2.3 reset

`t_u8 wifi_cloud_keep_alive_t::reset`

Enable/Disable tcp reset

4.23.2.4 cached

`t_u8 wifi_cloud_keep_alive_t::cached`

Saved in driver

4.23.2.5 send_interval

`t_u32 wifi_cloud_keep_alive_t::send_interval`

Period to send keep alive packet(The unit is milliseconds)

4.23.2.6 retry_interval

`t_u16 wifi_cloud_keep_alive_t::retry_interval`

Period to send retry packet(The unit is milliseconds)

4.23.2.7 retry_count

```
t_u16 wifi_cloud_keep_alive_t::retry_count
```

Count to send retry packet

4.23.2.8 src_mac

```
t_u8 wifi_cloud_keep_alive_t::src_mac[MLAN_MAC_ADDR_LENGTH]
```

Source MAC address

4.23.2.9 dst_mac

```
t_u8 wifi_cloud_keep_alive_t::dst_mac[MLAN_MAC_ADDR_LENGTH]
```

Destination MAC address

4.23.2.10 src_ip

```
t_u32 wifi_cloud_keep_alive_t::src_ip
```

Source IP

4.23.2.11 dst_ip

```
t_u32 wifi_cloud_keep_alive_t::dst_ip
```

Destination IP

4.23.2.12 src_port

```
t_u16 wifi_cloud_keep_alive_t::src_port
```

Source Port

4.23.2.13 dst_port

```
t_u16 wifi_cloud_keep_alive_t::dst_port
```

Destination Port

4.23.2.14 pkt_len

```
t_u16 wifi_cloud_keep_alive_t::pkt_len
```

Packet length

4.23.2.15 packet

```
t_u8 wifi_cloud_keep_alive_t::packet[MKEEP_ALIVE_IP_PKT_MAX]
```

Packet buffer

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.24 wifi_csi_config_params_t Struct Reference

Data Fields

- t_u16 csi_enable
- t_u32 head_id
- t_u32 tail_id
- t_u8 csi_filter_cnt
- t_u8 chip_id
- t_u8 band_config
- t_u8 channel
- t_u8 csi_monitor_enable
- t_u8 ra4us
- wifi_csi_filter_t csi_filter [CSI_FILTER_MAX]

4.24.1 Detailed Description

Structure of CSI parameters

4.24.2 Field Documentation

4.24.2.1 csi_enable

```
t_u16 wifi_csi_config_params_t::csi_enable
```

CSI enable flag. 1: enable, 2: disable

4.24.2.2 head_id

```
t_u32 wifi_csi_config_params_t::head_id
```

Header ID

4.24.2.3 tail_id

```
t_u32 wifi_csi_config_params_t::tail_id
```

Tail ID

4.24.2.4 csi_filter_cnt

```
t_u8 wifi_csi_config_params_t::csi_filter_cnt
```

Number of CSI filters

4.24.2.5 chip_id

```
t_u8 wifi_csi_config_params_t::chip_id
```

Chip ID

4.24.2.6 band_config

```
t_u8 wifi_csi_config_params_t::band_config
```

band config

4.24.2.7 channel

```
t_u8 wifi_csi_config_params_t::channel
```

Channel num

4.24.2.8 csi_monitor_enable

```
t_u8 wifi_csi_config_params_t::csi_monitor_enable
```

Enable getting CSI data on special channel

4.24.2.9 ra4us

```
t_u8 wifi_csi_config_params_t::ra4us
```

CSI data received in cfg channel with mac addr filter, not only RA is us or other

4.24.2.10 csi_filter

```
wifi_csi_filter_t wifi_csi_config_params_t::csi_filter[CSI_FILTER_MAX]
```

CSI filters

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.25 wifi_csi_filter_t Struct Reference

Data Fields

- t_u8 mac_addr [MLAN_MAC_ADDR_LENGTH]
- t_u8 pkt_type
- t_u8 subtype
- t_u8 flags

4.25.1 Detailed Description

Structure of CSI filters

4.25.2 Field Documentation

4.25.2.1 mac_addr

```
t_u8 wifi_csi_filter_t::mac_addr[MLAN_MAC_ADDR_LENGTH]
```

Source address of the packet to receive

4.25.2.2 pkt_type

```
t_u8 wifi_csi_filter_t::pkt_type
```

Pakcet type of the interested CSI

4.25.2.3 subtype

```
t_u8 wifi_csi_filter_t::subtype
```

Packet subtype of the interested CSI

4.25.2.4 flags

```
t_u8 wifi_csi_filter_t::flags
```

Other filter flags

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.26 wifi_cw_mode_ctrl_t Struct Reference

Data Fields

- t_u8 mode
- t_u8 channel
- t_u8 chanInfo
- t_u16 txPower
- t_u16 pktLength
- t_u32 rateInfo

4.26.1 Detailed Description

CW_MODE_CTRL structure

4.26.2 Field Documentation

4.26.2.1 mode

```
t_u8 wifi_cw_mode_ctrl_t::mode
```

Mode of Operation 0:Disable 1: Tx Continuous Packet 2 : Tx Continuous Wave

4.26.2.2 channel

```
t_u8 wifi_cw_mode_ctrl_t::channel
```

channel

4.26.2.3 chanInfo

```
t_u8 wifi_cw_mode_ctrl_t::chanInfo
```

channel info

4.26.2.4 txPower

```
t_u16 wifi_cw_mode_ctrl_t::txPower
```

Tx Power level in dBm

4.26.2.5 pktLength

```
t_u16 wifi_cw_mode_ctrl_t::pktLength
```

Packet Length

4.26.2.6 rateInfo

```
t_u32 wifi_cw_mode_ctrl_t::rateInfo
```

bit rate info

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.27 wifi_data_rate_t Struct Reference

Data Fields

- t_u32 tx_data_rate
- t_u32 rx_data_rate
- t_u32 tx_bw
- t_u32 tx_gi
- t_u32 rx_bw
- t_u32 rx_gi

4.27.1 Detailed Description

Data structure for cmd get data rate

4.27.2 Field Documentation

4.27.2.1 tx_data_rate

```
t_u32 wifi_data_rate_t::tx_data_rate
```

Tx data rate

4.27.2.2 rx_data_rate

```
t_u32 wifi_data_rate_t::rx_data_rate
```

Rx data rate

4.27.2.3 tx_bw

```
t_u32 wifi_data_rate_t::tx_bw
```

Tx channel bandwidth

4.27.2.4 tx_gi

```
t_u32 wifi_data_rate_t::tx_gi
```

Tx guard interval

4.27.2.5 rx_bw

```
t_u32 wifi_data_rate_t::rx_bw
```

Rx channel bandwidth

4.27.2.6 rx_gi

```
t_u32 wifi_data_rate_t::rx_gi
```

Rx guard interval

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.28 wifi_ds_rate Struct Reference

Data Fields

- enum wifi_ds_command_type sub_command
- union {
  wifi_rate_cfg_t rate_cfg
  wifi_data_rate_t data_rate
  } param

4.28.1 Detailed Description

Type definition of wifi_ds_rate

4.28.2 Field Documentation

4.28.2.1 sub_command

```
enum wifi_ds_command_type wifi_ds_rate::sub_command
```

Sub-command

4.28.2.2 rate_cfg

```
wifi_rate_cfg_t wifi_ds_rate::rate_cfg
```

Rate configuration for MLAN_OID_RATE_CFG

4.28.2.3 data_rate

```
wifi_data_rate_t wifi_ds_rate::data_rate
```

Data rate for MLAN_OID_GET_DATA_RATE

4.28.2.4 param

```
union { ... } wifi_ds_rate::param
```

Rate configuration parameter

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.29 wifi_ed_mac_ctrl_t Struct Reference

Data Fields

- t_u16 ed_ctrl_2g
- t_s16 ed_offset_2g
- t_u16 ed_ctrl_5g
- t_s16 ed_offset_5g

4.29.1 Detailed Description

Type definition of wifi_ed_mac_ctrl_t

4.29.2 Field Documentation

4.29.2.1 ed_ctrl_2g

```
t_u16 wifi_ed_mac_ctrl_t::ed_ctrl_2g
```

ED CTRL 2G

4.29.2.2 ed_offset_2g

```
t_s16 wifi_ed_mac_ctrl_t::ed_offset_2g
```

ED Offset 2G

4.29.2.3 ed_ctrl_5g

```
t_u16 wifi_ed_mac_ctrl_t::ed_ctrl_5g
```

ED CTRL 5G

4.29.2.4 ed_offset_5g

```
t_s16 wifi_ed_mac_ctrl_t::ed_offset_5g
```

ED Offset 5G

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.30 wifi_ext_coex_config_t Struct Reference

Data Fields

- t_u8 Enabled
- t_u8 IgnorePriority
- t_u8 DefaultPriority
- t_u8 EXT_RADIO_REQ_ip_gpio_num
- t_u8 EXT_RADIO_REQ_ip_gpio_polarity
- t_u8 EXT_RADIO_PRI_ip_gpio_num
- t_u8 EXT_RADIO_PRI_ip_gpio_polarity
- t_u8 WLAN_GRANT_op_gpio_num
- t_u8 WLAN_GRANT_op_gpio_polarity
- t_u16 reserved_1
- t_u16 reserved_2

4.30.1 Detailed Description

Type definition of wifi_ext_coex_config_t

4.30.2 Field Documentation

4.30.2.1 Enabled

`t_u8 wifi_ext_coex_config_t::Enabled`

Enable or disable external coexistence

4.30.2.2 IgnorePriority

`t_u8 wifi_ext_coex_config_t::IgnorePriority`

Ignore the priority of the external radio request

4.30.2.3 DefaultPriority

`t_u8 wifi_ext_coex_config_t::DefaultPriority`

Default priority when the priority of the external radio request is ignored

4.30.2.4 EXT_RADIO_REQ_ip_gpio_num

`t_u8 wifi_ext_coex_config_t::EXT_RADIO_REQ_ip_gpio_num`

Input request GPIO pin for EXT_RADIO_REQ signal

4.30.2.5 EXT_RADIO_REQ_ip_gpio_polarity

`t_u8 wifi_ext_coex_config_t::EXT_RADIO_REQ_ip_gpio_polarity`

Input request GPIO polarity for EXT_RADIO_REQ signal

4.30.2.6 EXT_RADIO_PRI_ip_gpio_num

`t_u8 wifi_ext_coex_config_t::EXT_RADIO_PRI_ip_gpio_num`

Input priority GPIO pin for EXT_RADIO_PRI signal

4.30.2.7 EXT_RADIO_PRI_ip_gpio_polarity

```
t_u8 wifi_ext_coex_config_t::EXT_RADIO_PRI_ip_gpio_polarity
```

Input priority GPIO polarity for EXT_RADIO_PRI signal

4.30.2.8 WLAN_GRANT_op_gpio_num

```
t_u8 wifi_ext_coex_config_t::WLAN_GRANT_op_gpio_num
```

Output grant GPIO pin for WLAN_GRANT signal

4.30.2.9 WLAN_GRANT_op_gpio_polarity

```
t_u8 wifi_ext_coex_config_t::WLAN_GRANT_op_gpio_polarity
```

Output grant GPIO polarity of WLAN_GRANT

4.30.2.10 reserved_1

```
t_u16 wifi_ext_coex_config_t::reserved_1
```

Reserved Bytes

4.30.2.11 reserved_2

```
t_u16 wifi_ext_coex_config_t::reserved_2
```

Reserved Bytes

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.31 wifi_ext_coex_stats_t Struct Reference

Data Fields

- t_u16 ext_radio_req_count
- t_u16 ext_radio_pri_count
- t_u16 wlan_grant_count

4.31.1 Detailed Description

Type definition of wifi_ext_coex_stats_t

4.31.2 Field Documentation

4.31.2.1 ext_radio_req_count

`t_u16 wifi_ext_coex_stats_t::ext_radio_req_count`

External Radio Request count

4.31.2.2 ext_radio_pri_count

`t_u16 wifi_ext_coex_stats_t::ext_radio_pri_count`

External Radio Priority count

4.31.2.3 wlan_grant_count

`t_u16 wifi_ext_coex_stats_t::wlan_grant_count`

WLAN GRANT count

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.32 wifi_flt_cfg_t Struct Reference

Data Fields

- t_u32 criteria
- t_u16 nentries
- wifi_mef_entry_t mef_entry [MAX_NUM_ENTRIES]

4.32.1 Detailed Description

Wifi filter config struct

4.32.2 Field Documentation

4.32.2.1 criteria

```
t_u32 wifi_flt_cfg_t::criteria
```

Filter Criteria

4.32.2.2 nentries

```
t_u16 wifi_flt_cfg_t::nentries
```

Number of entries

4.32.2.3 mef_entry

```
wifi_mef_entry_t wifi_flt_cfg_t::mef_entry[MAX_NUM_ENTRIES]
```

MEF entry

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.33 wifi_fw_version_ext_t Struct Reference

Data Fields

- uint8_t version_str_sel
- char version_str [MLAN_MAX_VER_STR_LEN]

4.33.1 Detailed Description

Extended Firmware version

4.33.2 Field Documentation

4.33.2.1 version_str_sel

```
uint8_t wifi_fw_version_ext_t::version_str_sel
```

ID for extended version select

4.33.2.2 version_str

```
char wifi_fw_version_ext_t::version_str[MLAN_MAX_VER_STR_LEN]
```

Firmware version string

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.34 wifi_fw_version_t Struct Reference

Data Fields

- char version_str [MLAN_MAX_VER_STR_LEN]

4.34.1 Detailed Description

Firmware version

4.34.2 Field Documentation

4.34.2.1 version_str

```
char wifi_fw_version_t::version_str[MLAN_MAX_VER_STR_LEN]
```

Firmware version string

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.35 wifi_indrst_cfg_t Struct Reference

Data Fields

- t_u8 ir_mode
- t_u8 gpio_pin

4.35.1 Detailed Description

Wi-Fi independent reset config

4.35.2 Field Documentation

4.35.2.1 ir_mode

```
t_u8 wifi_indrst_cfg_t::ir_mode
```

reset mode enable/ disable

4.35.2.2 gpio_pin

```
t_u8 wifi_indrst_cfg_t::gpio_pin
```

gpio pin

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.36 wifi_mac_addr_t Struct Reference

Data Fields

- char mac [MLAN_MAC_ADDR_LENGTH]

4.36.1 Detailed Description

MAC address

4.36.2 Field Documentation

4.36.2.1 mac

```
char wifi_mac_addr_t::mac[MLAN_MAC_ADDR_LENGTH]
```

Mac address array

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.37 wifi_mef_entry_t Struct Reference

Data Fields

- t_u8 mode
- t_u8 action
- t_u8 filter_num
- wifi_mef_filter_t filter_item [MAX_NUM_FILTERS]
- t_u8 rpn [MAX_NUM_FILTERS]

4.37.1 Detailed Description

MEF entry struct

4.37.2 Field Documentation

4.37.2.1 mode

```
t_u8 wifi_mef_entry_t::mode
```

mode: bit0–hostsleep mode; bit1–non hostsleep mode

4.37.2.2 action

```
t_u8 wifi_mef_entry_t::action
```

action: 0–discard and not wake host; 1–discard and wake host; 3–allow and wake host;

4.37.2.3 filter_num

```
t_u8 wifi_mef_entry_t::filter_num
```

filter number

4.37.2.4 filter_item

```
wifi_mef_filter_t wifi_mef_entry_t::filter_item[MAX_NUM_FILTERS]
```

filter array

4.37.2.5 rpn

```
t_u8 wifi_mef_entry_t::rpn[MAX_NUM_FILTERS]
```

rpn array

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.38 wifi_mef_filter_t Struct Reference

Data Fields

- t_u32 fill_flag
- t_u16 type
- t_u32 pattern
- t_u16 offset
- t_u16 num_bytes
- t_u16 repeat
- t_u8 num_byte_seq
- t_u8 byte_seq [MAX_NUM_BYTE_SEQ]
- t_u8 num_mask_seq
- t_u8 mask_seq [MAX_NUM_MASK_SEQ]

4.38.1 Detailed Description

Type definition of filter_item support three match methods: <1>Byte comparison type=0x41 <2>Decimal comparison type=0x42 <3>Bit comparison type=0x43

4.38.2 Field Documentation

4.38.2.1 fill_flag

```
t_u32 wifi_mef_filter_t::fill_flag
```

flag

4.38.2.2 type

```
t_u16 wifi_mef_filter_t::type
```

BYTE 0X41; Decimal 0X42; Bit 0x43

4.38.2.3 pattern

```
t_u32 wifi_mef_filter_t::pattern
```

value

4.38.2.4 offset

```
t_u16 wifi_mef_filter_t::offset
```

offset

4.38.2.5 num_bytes

```
t_u16 wifi_mef_filter_t::num_bytes
```

number of bytes

4.38.2.6 repeat

```
t_u16 wifi_mef_filter_t::repeat
```

repeat

4.38.2.7 num_byte_seq

```
t_u8 wifi_mef_filter_t::num_byte_seq
```

byte number

4.38.2.8 byte_seq

```
t_u8 wifi_mef_filter_t::byte_seq[MAX_NUM_BYTE_SEQ]
```

array

4.38.2.9 num_mask_seq

```
t_u8 wifi_mef_filter_t::num_mask_seq
```

mask numbers

4.38.2.10 mask_seq

```
t_u8 wifi_mef_filter_t::mask_seq[MAX_NUM_MASK_SEQ]
```

array

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.39 wifi_mfg_cmd_generic_cfg_t Struct Reference

Data Fields

- t_u32 mfg_cmd
- t_u16 action
- t_u16 device_id
- t_u32 error
- t_u32 data1
- t_u32 data2
- t_u32 data3

4.39.1 Detailed Description

Configuration for Manufacturing generic command

4.39.2 Field Documentation

4.39.2.1 mfg_cmd

```
t_u32 wifi_mfg_cmd_generic_cfg_t::mfg_cmd
```

MFG command code

4.39.2.2 action

```
t_u16 wifi_mfg_cmd_generic_cfg_t::action
```

Action

4.39.2.3 device_id

```
t_u16 wifi_mfg_cmd_generic_cfg_t::device_id
```

Device ID

4.39.2.4 error

```
t_u32 wifi_mfg_cmd_generic_cfg_t::error
```

MFG Error code

4.39.2.5 data1

```
t_u32 wifi_mfg_cmd_generic_cfg_t::data1
```

value 1

4.39.2.6 data2

```
t_u32 wifi_mfg_cmd_generic_cfg_t::data2
```

value 2

4.39.2.7 data3

```
t_u32 wifi_mfg_cmd_generic_cfg_t::data3
```

value 3

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.40 wifi_mfg_cmd_tx_cont_t Struct Reference

Data Fields

- t_u32 mfg_cmd
- t_u16 action
- t_u16 device_id
- t_u32 error
- t_u32 enable_tx
- t_u32 cw_mode
- t_u32 payload_pattern
- t_u32 cs_mode
- t_u32 act_sub_ch
- t_u32 tx_rate
- t_u32 rsvd

4.40.1 Detailed Description

Configuration for Manufacturing command Tx Continuous

4.40.2 Field Documentation

4.40.2.1 mfg_cmd

```
t_u32 wifi_mfg_cmd_tx_cont_t::mfg_cmd
```

MFG command code

4.40.2.2 action

```
t_u16 wifi_mfg_cmd_tx_cont_t::action
```

Action

4.40.2.3 device_id

```
t_u16 wifi_mfg_cmd_tx_cont_t::device_id
```

Device ID

4.40.2.4 error

```
t_u32 wifi_mfg_cmd_tx_cont_t::error
```

MFG Error code

4.40.2.5 enable_tx

```
t_u32 wifi_mfg_cmd_tx_cont_t::enable_tx
```

enable Tx

4.40.2.6 cw_mode

```
t_u32 wifi_mfg_cmd_tx_cont_t::cw_mode
```

Continuous Wave mode

4.40.2.7 payload_pattern

```
t_u32 wifi_mfg_cmd_tx_cont_t::payload_pattern
```

payload pattern

4.40.2.8 cs_mode

```
t_u32 wifi_mfg_cmd_tx_cont_t::cs_mode
```

CS Mode

4.40.2.9 act_sub_ch

```
t_u32 wifi_mfg_cmd_tx_cont_t::act_sub_ch
```

active sub channel

4.40.2.10 tx_rate

```
t_u32 wifi_mfg_cmd_tx_cont_t::tx_rate
```

Tx rate

4.40.2.11 rsvd

```
t_u32 wifi_mfg_cmd_tx_cont_t::rsvd
```

power id

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.41 wifi_mfg_cmd_tx_frame_t Struct Reference

Data Fields

- t_u32 mfg_cmd
- t_u16 action
- t_u16 device_id
- t_u32 error
- t_u32 enable
- t_u32 data_rate
- t_u32 frame_pattern
- t_u32 frame_length
- t_u8 bssid [MLAN_MAC_ADDR_LENGTH]
- t_u16 adjust_burst_sifs
- t_u32 burst_sifs_in_us
- t_u32 short_preamble
- t_u32 act_sub_ch
- t_u32 short_gi
- t_u32 adv_coding
- t_u32 tx_bf
- t_u32 gf_mode
- t_u32 stbc
- t_u32 rsvd [2]

4.41.1 Detailed Description

Configuration for Manufacturing command Tx Frame

4.41.2 Field Documentation

4.41.2.1 mfg_cmd

```
t_u32 wifi_mfg_cmd_tx_frame_t::mfg_cmd
```

MFG command code

4.41.2.2 action

```
t_u16 wifi_mfg_cmd_tx_frame_t::action
```

Action

4.41.2.3 device_id

```
t_u16 wifi_mfg_cmd_tx_frame_t::device_id
```

Device ID

4.41.2.4 error

```
t_u32 wifi_mfg_cmd_tx_frame_t::error
```

MFG Error code

4.41.2.5 enable

```
t_u32 wifi_mfg_cmd_tx_frame_t::enable
```

enable

4.41.2.6 data_rate

```
t_u32 wifi_mfg_cmd_tx_frame_t::data_rate
```

data_rate

4.41.2.7 frame_pattern

```
t_u32 wifi_mfg_cmd_tx_frame_t::frame_pattern
```

frame pattern

4.41.2.8 frame_length

```
t_u32 wifi_mfg_cmd_tx_frame_t::frame_length
```

frame length

4.41.2.9 bssid

```
t_u8 wifi_mfg_cmd_tx_frame_t::bssid[MLAN_MAC_ADDR_LENGTH]
```

BSSID

4.41.2.10 adjust_burst_sifs

```
t_u16 wifi_mfg_cmd_tx_frame_t::adjust_burst_sifs
```

Adjust burst sifs

4.41.2.11 burst_sifs_in_us

```
t_u32 wifi_mfg_cmd_tx_frame_t::burst_sifs_in_us
```

Burst sifs in us

4.41.2.12 short_preamble

```
t_u32 wifi_mfg_cmd_tx_frame_t::short_preamble
```

short preamble

4.41.2.13 act_sub_ch

```
t_u32 wifi_mfg_cmd_tx_frame_t::act_sub_ch
```

active sub channel

4.41.2.14 short_gi

```
t_u32 wifi_mfg_cmd_tx_frame_t::short_gi
```

short GI

4.41.2.15 adv_coding

```
t_u32 wifi_mfg_cmd_tx_frame_t::adv_coding
```

Adv coding

4.41.2.16 tx_bf

```
t_u32 wifi_mfg_cmd_tx_frame_t::tx_bf
```

Tx beamforming

4.41.2.17 gf_mode

```
t_u32 wifi_mfg_cmd_tx_frame_t::gf_mode
```

HT Greenfield Mode

4.41.2.18 stbc

```
t_u32 wifi_mfg_cmd_tx_frame_t::stbc
```

STBC

4.41.2.19 rsvd

```
t_u32 wifi_mfg_cmd_tx_frame_t::rsvd[2]
```

power id

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.42 wifi_mgmt_frame_t Struct Reference

Data Fields

- t_u16 frm_len
- wifi_frame_type_t frame_type
- t_u8 frame_ctrl_flags
- t_u16 duration_id
- t_u8 addr1 [MLAN_MAC_ADDR_LENGTH]
- t_u8 addr2 [MLAN_MAC_ADDR_LENGTH]
- t_u8 addr3 [MLAN_MAC_ADDR_LENGTH]
- t_u16 seq_ctl
- t_u8 addr4 [MLAN_MAC_ADDR_LENGTH]
- t_u8 payload [1]

4.42.1 Detailed Description

802_11_header packet

4.42.2 Field Documentation

4.42.2.1 frm_len

`t_u16 wifi_mgmt_frame_t::frm_len`

Packet Length

4.42.2.2 frame_type

`wifi_frame_type_t wifi_mgmt_frame_t::frame_type`

Frame Type

4.42.2.3 frame_ctrl_flags

`t_u8 wifi_mgmt_frame_t::frame_ctrl_flags`

Frame Control flags

4.42.2.4 duration_id

`t_u16 wifi_mgmt_frame_t::duration_id`

Duration ID

4.42.2.5 addr1

`t_u8 wifi_mgmt_frame_t::addr1[MLAN_MAC_ADDR_LENGTH]`

Address 1

4.42.2.6 addr2

`t_u8 wifi_mgmt_frame_t::addr2[MLAN_MAC_ADDR_LENGTH]`

Address 2

4.42.2.7 addr3

`t_u8 wifi_mgmt_frame_t::addr3[MLAN_MAC_ADDR_LENGTH]`

Address 3

4.42.2.8 seq_ctl

`t_u16 wifi_mgmt_frame_t::seq_ctl`

Sequence Control

4.42.2.9 addr4

`t_u8 wifi_mgmt_frame_t::addr4[MLAN_MAC_ADDR_LENGTH]`

Address 4

4.42.2.10 payload

`t_u8 wifi_mgmt_frame_t::payload[1]`

Frame payload

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.43 wifi_nat_keep_alive_t Struct Reference

Data Fields

- t_u16 interval
- t_u8 dst_mac [MLAN_MAC_ADDR_LENGTH]
- t_u32 dst_ip
- t_u16 dst_port

4.43.1 Detailed Description

TCP nat keep alive information

4.43.2 Field Documentation

4.43.2.1 interval

```
t_u16 wifi_nat_keep_alive_t::interval
```

Keep alive interval

4.43.2.2 dst_mac

```
t_u8 wifi_nat_keep_alive_t::dst_mac[MLAN_MAC_ADDR_LENGTH]
```

Destination MAC address

4.43.2.3 dst_ip

```
t_u32 wifi_nat_keep_alive_t::dst_ip
```

Destination IP

4.43.2.4 dst_port

```
t_u16 wifi_nat_keep_alive_t::dst_port
```

Destination port

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.44 wifi_rate_cfg_t Struct Reference

Data Fields

- mlan_rate_format rate_format
- t_u32 rate_index
- t_u32 rate
- t_u32 nss
- t_u16 rate_setting

4.44.1 Detailed Description

Data structure for cmd txratecfg

4.44.2 Field Documentation

4.44.2.1 rate_format

```
mlan_rate_format wifi_rate_cfg_t::rate_format
```

LG rate: 0, HT rate: 1, VHT rate: 2

4.44.2.2 rate_index

```
t_u32 wifi_rate_cfg_t::rate_index
```

Rate/MCS index (0xFF: auto)

4.44.2.3 rate

```
t_u32 wifi_rate_cfg_t::rate
```

Rate rate

4.44.2.4 nss

```
t_u32 wifi_rate_cfg_t::nss
```

NSS

4.44.2.5 rate_setting

```
t_u16 wifi_rate_cfg_t::rate_setting
```

Rate Setting

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.45 wifi_remain_on_channel_t Struct Reference

Data Fields

- uint16_t remove
- uint8_t status
- uint8_t bandcfg
- uint8_t channel
- uint32_t remain_period

4.45.1 Detailed Description

Remain on channel info structure

4.45.2 Field Documentation

4.45.2.1 remove

`uint16_t wifi_remain_on_channel_t::remove`

Remove

4.45.2.2 status

`uint8_t wifi_remain_on_channel_t::status`

Current status

4.45.2.3 bandcfg

`uint8_t wifi_remain_on_channel_t::bandcfg`

band configuration

4.45.2.4 channel

`uint8_t wifi_remain_on_channel_t::channel`

Channel

4.45.2.5 remain_period

`uint32_t wifi_remain_on_channel_t::remain_period`

Remain on channel period

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.46 wifi_rf_channel_t Struct Reference

Data Fields

- uint16_t current_channel
- uint16_t rf_type

4.46.1 Detailed Description

Rf channel

4.46.2 Field Documentation

4.46.2.1 current_channel

```
uint16_t wifi_rf_channel_t::current_channel
```

Current channel

4.46.2.2 rf_type

```
uint16_t wifi_rf_channel_t::rf_type
```

RF Type

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.47 wifi_rssi_info_t Struct Reference

Data Fields

- int16_t data_rssi_last
- int16_t data_nf_last
- int16_t data_rssi_avg
- int16_t data_nf_avg
- int16_t bcn_snr_last
- int16_t bcn_snr_avg
- int16_t data_snr_last
- int16_t data_snr_avg
- int16_t bcn_rssi_last
- int16_t bcn_nf_last
- int16_t bcn_rssi_avg
- int16_t bcn_nf_avg

4.47.1 Detailed Description

RSSI information

4.47.2 Field Documentation

4.47.2.1 data_rssi_last

`int16_t wifi_rssi_info_t::data_rssi_last`

Data RSSI last

4.47.2.2 data_nf_last

`int16_t wifi_rssi_info_t::data_nf_last`

Data nf last

4.47.2.3 data_rssi_avg

`int16_t wifi_rssi_info_t::data_rssi_avg`

Data RSSI average

4.47.2.4 data_nf_avg

`int16_t wifi_rssi_info_t::data_nf_avg`

Data nf average

4.47.2.5 bcn_snr_last

`int16_t wifi_rssi_info_t::bcn_snr_last`

BCN SNR

4.47.2.6 bcn_snr_avg

`int16_t wifi_rssi_info_t::bcn_snr_avg`

BCN SNR average

4.47.2.7 data_snr_last

```
int16_t wifi_rssi_info_t::data_snr_last
```

Data SNR last

4.47.2.8 data_snr_avg

```
int16_t wifi_rssi_info_t::data_snr_avg
```

Data SNR average

4.47.2.9 bcn_rssi_last

```
int16_t wifi_rssi_info_t::bcn_rssi_last
```

BCN RSSI

4.47.2.10 bcn_nf_last

```
int16_t wifi_rssi_info_t::bcn_nf_last
```

BCN nf

4.47.2.11 bcn_rssi_avg

```
int16_t wifi_rssi_info_t::bcn_rssi_avg
```

BCN RSSI average

4.47.2.12 bcn_nf_avg

```
int16_t wifi_rssi_info_t::bcn_nf_avg
```

BCN nf average

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.48 wifi_rutxpwrlimit_t Struct Reference

Data Fields

- t_u8 num_chans
- wifi_rupwrlimit_config_t rupwrlimit_config [MAX_RUTXPWR_NUM]

4.48.1 Detailed Description

Data structure for Channel RU PWR config

For RU PWR support

4.48.2 Field Documentation

4.48.2.1 num_chans

```
t_u8 wifi_rutxpwrlimit_t::num_chans
```

Number of Channels

4.48.2.2 rupwrlimit_config

```
wifi_rupwrlimit_config_t wifi_rutxpwrlimit_t::rupwrlimit_config[MAX_RUTXPWR_NUM]
```

RU PWR config

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.49 wifi_scan_chan_list_t Struct Reference

Data Fields

- uint8_t num_of_chan
- uint8_t chan_number [MLAN_MAX_CHANNEL]

4.49.1 Detailed Description

Channel list structure

4.49.2 Field Documentation

4.49.2.1 num_of_chan

```
uint8_t wifi_scan_chan_list_t::num_of_chan
```

Number of channels

4.49.2.2 chan_number

```
uint8_t wifi_scan_chan_list_t::chan_number[MLAN_MAX_CHANNEL]
```

Channel number

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.50 wifi_scan_channel_list_t Struct Reference

Data Fields

- t_u8 chan_number
- mlan_scan_type scan_type
- t_u16 scan_time

4.50.1 Detailed Description

Scan channel list

4.50.2 Field Documentation

4.50.2.1 chan_number

```
t_u8 wifi_scan_channel_list_t::chan_number
```

Channel numder

4.50.2.2 scan_type

```
mlan_scan_type wifi_scan_channel_list_t::scan_type
```

Scan type Active = 1, Passive = 2

4.50.2.3 scan_time

```
t_u16 wifi_scan_channel_list_t::scan_time
```

Scan time

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.51 wifi_scan_params_v2_t Struct Reference

Data Fields

- t_u8 scan_only
- t_u8 is_bssid
- t_u8 is_ssid
- t_u8 bssid [MLAN_MAC_ADDR_LENGTH]
- char ssid [MAX_NUM_SSID][MLAN_MAX_SSID_LENGTH+1]
- t_u8 num_channels
- wifi_scan_channel_list_t chan_list [MAX_CHANNEL_LIST]
- t_u8 num_probes
- t_u16 scan_chan_gap
- int(∗ cb)(unsigned int count)

4.51.1 Detailed Description

V2 scan parameters

4.51.2 Field Documentation

4.51.2.1 scan_only

```
t_u8 wifi_scan_params_v2_t::scan_only
```

Scan Only

4.51.2.2 is_bssid

```
t_u8 wifi_scan_params_v2_t::is_bssid
```

BSSID present

4.51.2.3 is_ssid

```
t_u8 wifi_scan_params_v2_t::is_ssid
```

SSID present

4.51.2.4 bssid

```
t_u8 wifi_scan_params_v2_t::bssid[MLAN_MAC_ADDR_LENGTH]
```

BSSID to scan

4.51.2.5 ssid

```
char wifi_scan_params_v2_t::ssid[MAX_NUM_SSID][MLAN_MAX_SSID_LENGTH+1]
```

SSID to scan

4.51.2.6 num_channels

```
t_u8 wifi_scan_params_v2_t::num_channels
```

Number of channels

4.51.2.7 chan_list

```
wifi_scan_channel_list_t wifi_scan_params_v2_t::chan_list[MAX_CHANNEL_LIST]
```

Channel list with channel information

4.51.2.8 num_probes

```
t_u8 wifi_scan_params_v2_t::num_probes
```

Number of probes

4.51.2.9 scan_chan_gap

```
t_u16 wifi_scan_params_v2_t::scan_chan_gap
```

scan channel gap

4.51.2.10 cb

```
int(* wifi_scan_params_v2_t::cb) (unsigned int count)
```

Callback to be called when scan is completed

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.52 wifi_scan_result2 Struct Reference

Data Fields

- uint8_t bssid [MLAN_MAC_ADDR_LENGTH]
- bool is_ibss_bit_set
- uint8_t ssid [MLAN_MAX_SSID_LENGTH]
- int ssid_len
- uint8_t Channel
- uint8_t RSSI
- uint16_t beacon_period
- uint16_t dtim_period
- _SecurityMode_t WPA_WPA2_WEP
- _Cipher_t wpa_mcstCipher
- _Cipher_t wpa_ucstCipher
- _Cipher_t rsn_mcstCipher
- _Cipher_t rsn_ucstCipher
- bool is_pmf_required
- t_u8 ap_mfpc
- t_u8 ap_mfpr
- bool phtcap_ie_present
- bool phtinfo_ie_present
- bool pvhtcap_ie_present
- bool phecap_ie_present
- bool wmm_ie_present
- uint16_t band
- bool wps_IE_exist
- uint16_t wps_session
- bool wpa2_entp_IE_exist
- uint8_t trans_mode
- uint8_t trans_bssid [MLAN_MAC_ADDR_LENGTH]
- uint8_t trans_ssid [MLAN_MAX_SSID_LENGTH]
- int trans_ssid_len
- bool mbo_assoc_disallowed
- uint16_t mdid
- bool neighbor_report_supported
- bool bss_transition_supported

4.52.1 Detailed Description

Scan result information

4.52.2 Field Documentation

4.52.2.1 bssid

```
uint8_t wifi_scan_result2::bssid[MLAN_MAC_ADDR_LENGTH]
```

BSSID array

4.52.2.2 is_ibss_bit_set

```
bool wifi_scan_result2::is_ibss_bit_set
```

Is bssid set?

4.52.2.3 ssid

```
uint8_t wifi_scan_result2::ssid[MLAN_MAX_SSID_LENGTH]
```

ssid array

4.52.2.4 ssid_len

```
int wifi_scan_result2::ssid_len
```

SSID length

4.52.2.5 Channel

```
uint8_t wifi_scan_result2::Channel
```

Channel associated to the BSSID

4.52.2.6 RSSI

```
uint8_t wifi_scan_result2::RSSI
```

Received signal strength

4.52.2.7 beacon_period

```
uint16_t wifi_scan_result2::beacon_period
```

Beacon period

4.52.2.8 dtim_period

```
uint16_t wifi_scan_result2::dtim_period
```

DTIM period

4.52.2.9 WPA_WPA2_WEP

```
_SecurityMode_t wifi_scan_result2::WPA_WPA2_WEP
```

Security mode info

4.52.2.10 wpa_mcstCipher

```
_Cipher_t wifi_scan_result2::wpa_mcstCipher
```

WPA multicast cipher

4.52.2.11 wpa_ucstCipher

```
_Cipher_t wifi_scan_result2::wpa_ucstCipher
```

WPA unicast cipher

4.52.2.12 rsn_mcstCipher

```
_Cipher_t wifi_scan_result2::rsn_mcstCipher
```

No security multicast cipher

4.52.2.13 rsn_ucstCipher

```
_Cipher_t wifi_scan_result2::rsn_ucstCipher
```

No security unicast cipher

4.52.2.14 is_pmf_required

```
bool wifi_scan_result2::is_pmf_required
```

Is pmf required flag

4.52.2.15 ap_mfpc

```
t_u8 wifi_scan_result2::ap_mfpc
```

MFPC bit of AP

4.52.2.16 ap_mfpr

```
t_u8 wifi_scan_result2::ap_mfpr
```

MFPR bit of AP WPA_WPA2 = 0 => Security not enabled = 1 => WPA mode = 2 => WPA2 mode = 3 => WEP mode

4.52.2.17 phtcap_ie_present

```
bool wifi_scan_result2::phtcap_ie_present
```

PHT CAP IE present info

4.52.2.18 phtinfo_ie_present

```
bool wifi_scan_result2::phtinfo_ie_present
```

PHT INFO IE present info

4.52.2.19 pvhtcap_ie_present

```
bool wifi_scan_result2::pvhtcap_ie_present
```

11AC VHT capab support

4.52.2.20 phecap_ie_present

```
bool wifi_scan_result2::phecap_ie_present
```

11AX HE capab support

4.52.2.21 wmm_ie_present

```
bool wifi_scan_result2::wmm_ie_present
```

WMM IE present info

4.52.2.22 band

```
uint16_t wifi_scan_result2::band
```

Band info

4.52.2.23 wps_IE_exist

```
bool wifi_scan_result2::wps_IE_exist
```

WPS IE exist info

4.52.2.24 wps_session

```
uint16_t wifi_scan_result2::wps_session
```

WPS session

4.52.2.25 wpa2_entp_IE_exist

```
bool wifi_scan_result2::wpa2_entp_IE_exist
```

WPA2 enterprise IE exist info

4.52.2.26 trans_mode

```
uint8_t wifi_scan_result2::trans_mode
```

Trans mode

4.52.2.27 trans_bssid

```
uint8_t wifi_scan_result2::trans_bssid[MLAN_MAC_ADDR_LENGTH]
```

Trans bssid array

4.52.2.28 trans_ssid

```
uint8_t wifi_scan_result2::trans_ssid[MLAN_MAX_SSID_LENGTH]
```

Trans ssid array

4.52.2.29 trans_ssid_len

```
int wifi_scan_result2::trans_ssid_len
```

Trans bssid length

4.52.2.30 mbo_assoc_disallowed

```
bool wifi_scan_result2::mbo_assoc_disallowed
```

MBO disallowed

4.52.2.31 mdid

```
uint16_t wifi_scan_result2::mdid
```

Mobility domain identifier

4.52.2.32 neighbor_report_supported

```
bool wifi_scan_result2::neighbor_report_supported
```

Neigbort report support

4.52.2.33 bss_transition_supported

```
bool wifi_scan_result2::bss_transition_supported
```

bss transition support

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.53 wifi_sta_info_t Struct Reference

Data Fields

- t_u8 mac [MLAN_MAC_ADDR_LENGTH]
- t_u8 power_mgmt_status
- t_s8 rssi

4.53.1 Detailed Description

Station information structure

4.53.2 Field Documentation

4.53.2.1 mac

```
t_u8 wifi_sta_info_t::mac[MLAN_MAC_ADDR_LENGTH]
```

MAC address buffer

4.53.2.2 power_mgmt_status

```
t_u8 wifi_sta_info_t::power_mgmt_status
```

Power management status 0 = active (not in power save) 1 = in power save status

4.53.2.3 rssi

```
t_s8 wifi_sta_info_t::rssi
```

RSSI: dBm

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.54 wifi_sta_list_t Struct Reference

Data Fields

- int count

4.54.1 Detailed Description

Note: This is variable length structure. The size of array mac_list is equal to count. The caller of the API which returns this structure does not need to separately free the array mac_list. It only needs to free the sta_list_t object after use.

4.54.2 Field Documentation

4.54.2.1 count

```
int wifi_sta_list_t::count
```

Count

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.55 wifi_sub_band_set_t Struct Reference

Data Fields

- t_u8 first_chan
- t_u8 no_of_chan
- t_u8 max_tx_pwr

4.55.1 Detailed Description

Data structure for subband set

For uAP 11d support

4.55.2 Field Documentation

4.55.2.1 first_chan

```
t_u8 wifi_sub_band_set_t::first_chan
```

First channel

4.55.2.2 no_of_chan

```
t_u8 wifi_sub_band_set_t::no_of_chan
```

Number of channels

4.55.2.3 max_tx_pwr

```
t_u8 wifi_sub_band_set_t::max_tx_pwr
```

Maximum Tx power in dBm

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.56 wifi_tbtt_offset_t Struct Reference

Data Fields

- t_u32 min_tbtt_offset
- t_u32 max_tbtt_offset
- t_u32 avg_tbtt_offset

4.56.1 Detailed Description

TBTT offset structure

4.56.2 Field Documentation

4.56.2.1 min_tbtt_offset

```
t_u32 wifi_tbtt_offset_t::min_tbtt_offset
```

Min TBTT offset

4.56.2.2 max_tbtt_offset

```
t_u32 wifi_tbtt_offset_t::max_tbtt_offset
```

Max TBTT offset

4.56.2.3 avg_tbtt_offset

```
t_u32 wifi_tbtt_offset_t::avg_tbtt_offset
```

AVG TBTT offset

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.57 wifi_tcp_keep_alive_t Struct Reference

Data Fields

- t_u8 enable
- t_u8 reset
- t_u32 timeout
- t_u16 interval
- t_u16 max_keep_alives
- t_u8 dst_mac [MLAN_MAC_ADDR_LENGTH]
- t_u32 dst_ip
- t_u16 dst_tcp_port
- t_u16 src_tcp_port
- t_u32 seq_no

4.57.1 Detailed Description

TCP keep alive information

4.57.2 Field Documentation

4.57.2.1 enable

```
t_u8 wifi_tcp_keep_alive_t::enable
```

Enable keep alive

4.57.2.2 reset

```
t_u8 wifi_tcp_keep_alive_t::reset
```

Reset

4.57.2.3 timeout

```
t_u32 wifi_tcp_keep_alive_t::timeout
```

Keep alive timeout

4.57.2.4 interval

```
t_u16 wifi_tcp_keep_alive_t::interval
```

Keep alive interval

4.57.2.5 max_keep_alives

```
t_u16 wifi_tcp_keep_alive_t::max_keep_alives
```

Maximum keep alives

4.57.2.6 dst_mac

```
t_u8 wifi_tcp_keep_alive_t::dst_mac[MLAN_MAC_ADDR_LENGTH]
```

Destination MAC address

4.57.2.7 dst_ip

```
t_u32 wifi_tcp_keep_alive_t::dst_ip
```

Destination IP

4.57.2.8 dst_tcp_port

```
t_u16 wifi_tcp_keep_alive_t::dst_tcp_port
```

Destination TCP port

4.57.2.9 src_tcp_port

```
t_u16 wifi_tcp_keep_alive_t::src_tcp_port
```

Source TCP port

4.57.2.10 seq_no

```
t_u32 wifi_tcp_keep_alive_t::seq_no
```

Sequence number

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.58 wifi_tsf_info_t Struct Reference

Data Fields

- t_u16 tsf_format
- t_u16 tsf_info
- t_u64 tsf
- t_s32 tsf_offset

4.58.1 Detailed Description

Wi-Fi TSF information

4.58.2 Field Documentation

4.58.2.1 tsf_format

```
t_u16 wifi_tsf_info_t::tsf_format
```

get tsf info format

4.58.2.2 tsf_info

```
t_u16 wifi_tsf_info_t::tsf_info
```

tsf info

4.58.2.3 tsf

```
t_u64 wifi_tsf_info_t::tsf
```

tsf

4.58.2.4 tsf_offset

```
t_s32 wifi_tsf_info_t::tsf_offset
```

Positive or negative offset in microsecond from Beacon TSF to GPIO toggle TSF

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.59 wifi_twt_report_t Struct Reference

Data Fields

- t_u8 type
- t_u8 length
- t_u8 reserve [2]
- t_u8 data [WLAN_BTWT_REPORT_LEN ∗WLAN_BTWT_REPORT_MAX_NUM]

4.59.1 Detailed Description

Wi-Fi TWT Report Configuration

4.59.2 Field Documentation

4.59.2.1 type

```
t_u8 wifi_twt_report_t::type
```

TWT report type, 0: BTWT id

4.59.2.2 length

```
t_u8 wifi_twt_report_t::length
```

TWT report length of value in data

4.59.2.3 reserve

```
t_u8 wifi_twt_report_t::reserve[2]
```

Reserved 2

4.59.2.4 data

```
t_u8 wifi_twt_report_t::data[WLAN_BTWT_REPORT_LEN *WLAN_BTWT_REPORT_MAX_NUM]
```

TWT report buffer

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.60 wifi_twt_setup_config_t Struct Reference

Data Fields

- t_u8 implicit
- t_u8 announced
- t_u8 trigger_enabled
- t_u8 twt_info_disabled
- t_u8 negotiation_type
- t_u8 twt_wakeup_duration
- t_u8 flow_identifier
- t_u8 hard_constraint
- t_u8 twt_exponent
- t_u16 twt_mantissa
- t_u8 twt_request

4.60.1 Detailed Description

Wi-Fi TWT setup configuration

4.60.2 Field Documentation

4.60.2.1 implicit

```
t_u8 wifi_twt_setup_config_t::implicit
```

Implicit, 0: TWT session is explicit, 1: Session is implicit

4.60.2.2 announced

```
t_u8 wifi_twt_setup_config_t::announced
```

Announced, 0: Unannounced, 1: Announced TWT

4.60.2.3 trigger_enabled

```
t_u8 wifi_twt_setup_config_t::trigger_enabled
```

Trigger Enabled, 0: Non-Trigger enabled, 1: Trigger enabled TWT

4.60.2.4 twt_info_disabled

```
t_u8 wifi_twt_setup_config_t::twt_info_disabled
```

TWT Information Disabled, 0: TWT info enabled, 1: TWT info disabled

4.60.2.5 negotiation_type

```
t_u8 wifi_twt_setup_config_t::negotiation_type
```

Negotiation Type, 0: Future Individual TWT SP start time, 1: Next Wake TBTT time

4.60.2.6 twt_wakeup_duration

```
t_u8 wifi_twt_setup_config_t::twt_wakeup_duration
```

TWT Wakeup Duration, time after which the TWT requesting STA can transition to doze state

4.60.2.7 flow_identifier

```
t_u8 wifi_twt_setup_config_t::flow_identifier
```

Flow Identifier. Range: [0-7]

4.60.2.8 hard_constraint

```
t_u8 wifi_twt_setup_config_t::hard_constraint
```

Hard Constraint, 0: FW can tweak the TWT setup parameters if it is rejected by AP. 1: Firmware should not tweak any parameters.

4.60.2.9 twt_exponent

```
t_u8 wifi_twt_setup_config_t::twt_exponent
```

TWT Exponent, Range: [0-63]

4.60.2.10 twt_mantissa

```
t_u16 wifi_twt_setup_config_t::twt_mantissa
```

TWT Mantissa Range: [0-sizeof(UINT16)]

4.60.2.11 twt_request

```
t_u8 wifi_twt_setup_config_t::twt_request
```

TWT Request Type, 0: REQUEST_TWT, 1: SUGGEST_TWT

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.61 wifi_twt_teardown_config_t Struct Reference

Data Fields

- t_u8 flow_identifier
- t_u8 negotiation_type
- t_u8 teardown_all_twt

4.61.1 Detailed Description

Wi-Fi Teardown Configuration

4.61.2 Field Documentation

4.61.2.1 flow_identifier

`t_u8 wifi_twt_teardown_config_t::flow_identifier`

TWT Flow Identifier. Range: [0-7]

4.61.2.2 negotiation_type

`t_u8 wifi_twt_teardown_config_t::negotiation_type`

Negotiation Type. 0: Future Individual TWT SP start time, 1: Next Wake TBTT time

4.61.2.3 teardown_all_twt

`t_u8 wifi_twt_teardown_config_t::teardown_all_twt`

Tear down all TWT. 1: To teardown all TWT, 0 otherwise

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.62 wifi_tx_power_t Struct Reference

Data Fields

- uint16_t current_level
- uint8_t max_power
- uint8_t min_power

4.62.1 Detailed Description

Tx power levels

4.62.2 Field Documentation

4.62.2.1 current_level

`uint16_t wifi_tx_power_t::current_level`

Current power level

4.62.2.2 max_power

```
uint8_t wifi_tx_power_t::max_power
```

Maximum power level

4.62.2.3 min_power

```
uint8_t wifi_tx_power_t::min_power
```

Minimum power level

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.63 wifi_txpwrlimit_config_t Struct Reference

Data Fields

- t_u8 num_mod_grps
- wifi_channel_desc_t chan_desc
- wifi_txpwrlimit_entry_t txpwrlimit_entry [20]

4.63.1 Detailed Description

Data structure for TRPC config

For TRPC support

4.63.2 Field Documentation

4.63.2.1 num_mod_grps

```
t_u8 wifi_txpwrlimit_config_t::num_mod_grps
```

Number of modulation groups

4.63.2.2 chan_desc

```
wifi_channel_desc_t wifi_txpwrlimit_config_t::chan_desc
```

Chnannel descriptor

4.63.2.3 txpwrlimit_entry

`wifi_txpwrlimit_entry_t wifi_txpwrlimit_config_t::txpwrlimit_entry[20]`

Channel Modulation groups

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.64 wifi_txpwrlimit_entry_t Struct Reference

Data Fields

- t_u8 mod_group
- t_u8 tx_power

4.64.1 Detailed Description

Data structure for Modulation Group

mod_group : ModulationGroup
0: CCK (1,2,5.5,11 Mbps)
1: OFDM (6,9,12,18 Mbps)
2: OFDM (24,36 Mbps)
3: OFDM (48,54 Mbps)
4: HT20 (0,1,2)
5: HT20 (3,4)
6: HT20 (5,6,7)
7: HT40 (0,1,2)
8: HT40 (3,4)
9: HT40 (5,6,7)
10: HT2_20 (8,9,10)
11: HT2_20 (11,12)
12: HT2_20 (13,14,15)
tx_power : Power Limit in dBm

4.64.2 Field Documentation

4.64.2.1 mod_group

`t_u8 wifi_txpwrlimit_entry_t::mod_group`

Modulation group

4.64.2.2 tx_power

```
t_u8 wifi_txpwrlimit_entry_t::tx_power
```

Tx Power

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.65 wifi_txpwrlimit_t Struct Reference

Data Fields

- wifi_SubBand_t subband
- t_u8 num_chans
- wifi_txpwrlimit_config_t txpwrlimit_config [43]

4.65.1 Detailed Description

Data structure for Channel TRPC config

For TRPC support

4.65.2 Field Documentation

4.65.2.1 subband

```
wifi_SubBand_t wifi_txpwrlimit_t::subband
```

SubBand

4.65.2.2 num_chans

```
t_u8 wifi_txpwrlimit_t::num_chans
```

Number of Channels

4.65.2.3 txpwrlimit_config

```
wifi_txpwrlimit_config_t wifi_txpwrlimit_t::txpwrlimit_config[43]
```

TRPC config

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.66 wifi_wowlan_ptn_cfg_t Struct Reference

Data Fields

- t_u8 enable
- t_u8 n_patterns
- wifi_wowlan_pattern_t patterns [MAX_NUM_FILTERS]

4.66.1 Detailed Description

Wowlan Pattern config struct

4.66.2 Field Documentation

4.66.2.1 enable

```
t_u8 wifi_wowlan_ptn_cfg_t::enable
```

Enable user defined pattern

4.66.2.2 n_patterns

```
t_u8 wifi_wowlan_ptn_cfg_t::n_patterns
```

number of patterns

4.66.2.3 patterns

```
wifi_wowlan_pattern_t wifi_wowlan_ptn_cfg_t::patterns[MAX_NUM_FILTERS]
```

user define pattern

The documentation for this struct was generated from the following file:

- wifi-decl.h

4.67 wlan_cipher Struct Reference

Data Fields

- uint16_t none: 1
- uint16_t wep40: 1
- uint16_t wep104: 1
- uint16_t tkip: 1
- uint16_t ccmp: 1
- uint16_t aes_128_cmac: 1
- uint16_t gcmp: 1
- uint16_t sms4: 1
- uint16_t gcmp_256: 1
- uint16_t ccmp_256: 1
- uint16_t rsvd: 1
- uint16_t bip_gmac_128: 1
- uint16_t bip_gmac_256: 1
- uint16_t bip_cmac_256: 1
- uint16_t gtk_not_used: 1
- uint16_t rsvd2: 2

4.67.1 Detailed Description

Wlan Cipher structure

4.67.2 Field Documentation

4.67.2.1 none

```
uint16_t wlan_cipher::none
```

1 bit value can be set for none

4.67.2.2 wep40

```
uint16_t wlan_cipher::wep40
```

1 bit value can be set for wep40

4.67.2.3 wep104

```
uint16_t wlan_cipher::wep104
```

1 bit value can be set for wep104

4.67.2.4 tkip

```
uint16_t wlan_cipher::tkip
```

1 bit value can be set for tkip

4.67.2.5 ccmp

```
uint16_t wlan_cipher::ccmp
```

1 bit valuecan be set for ccmp

4.67.2.6 aes_128_cmac

```
uint16_t wlan_cipher::aes_128_cmac
```

1 bit valuecan be set for aes 128 cmac

4.67.2.7 gcmp

```
uint16_t wlan_cipher::gcmp
```

1 bit value can be set for gcmp

4.67.2.8 sms4

```
uint16_t wlan_cipher::sms4
```

1 bit value can be set for sms4

4.67.2.9 gcmp_256

```
uint16_t wlan_cipher::gcmp_256
```

1 bit value can be set for gcmp 256

4.67.2.10 ccmp_256

```
uint16_t wlan_cipher::ccmp_256
```

1 bit valuecan be set for ccmp 256

4.67.2.11 rsvd

```
uint16_t wlan_cipher::rsvd
```

1 bit is reserved

4.67.2.12 bip_gmac_128

```
uint16_t wlan_cipher::bip_gmac_128
```

1 bit value can be set for bip gmac 128

4.67.2.13 bip_gmac_256

```
uint16_t wlan_cipher::bip_gmac_256
```

1 bit value can be set for bip gmac 256

4.67.2.14 bip_cmac_256

```
uint16_t wlan_cipher::bip_cmac_256
```

1 bit value can be set for bip cmac 256

4.67.2.15 gtk_not_used

```
uint16_t wlan_cipher::gtk_not_used
```

1 bit valuecan be set for gtk not used

4.67.2.16 rsvd2

```
uint16_t wlan_cipher::rsvd2
```

4 bits are reserved

The documentation for this struct was generated from the following file:

- wlan.h

4.68 wlan_ip_config Struct Reference

Data Fields

- struct ipv6_config ipv6 [CONFIG_MAX_IPV6_ADDRESSES]
- struct ipv4_config ipv4

4.68.1 Detailed Description

Network IP configuration.

This data structure represents the network IP configuration for IPv4 as well as IPv6 addresses

4.68.2 Field Documentation

4.68.2.1 ipv6

```
struct ipv6_config wlan_ip_config::ipv6[CONFIG_MAX_IPV6_ADDRESSES]
```

The network IPv6 address configuration that should be associated with this interface.

4.68.2.2 ipv4

```
struct ipv4_config wlan_ip_config::ipv4
```

The network IPv4 address configuration that should be associated with this interface.

The documentation for this struct was generated from the following file:

- wlan.h

4.69 wlan_network Struct Reference

Data Fields

- int id
- int **wps_network**
- char name [WLAN_NETWORK_NAME_MAX_LENGTH+1]
- char ssid [IEEEtypes_SSID_SIZE+1]
- char bssid [IEEEtypes_ADDRESS_SIZE]
- unsigned int channel
- uint8_t sec_channel_offset
- uint16_t acs_band
- int rssi
- unsigned short ht_capab
- unsigned int vht_capab
- unsigned char vht_oper_chwidth
- unsigned char he_oper_chwidth
- enum wlan_bss_type type
- enum wlan_bss_role role
- struct wlan_network_security security
- struct wlan_ip_config ip
- unsigned ssid_specific: 1
- unsigned trans_ssid_specific: 1
- unsigned bssid_specific: 1
- unsigned channel_specific: 1
- unsigned security_specific: 1
- unsigned dot11n: 1
- unsigned dot11ac: 1
- unsigned dot11ax: 1

- uint16_t mdid
- unsigned ft_1x: 1
- unsigned ft_psk: 1
- unsigned ft_sae: 1
- unsigned int owe_trans_mode
- char trans_ssid [IEEEtypes_SSID_SIZE+1]
- unsigned int trans_ssid_len
- uint16_t beacon_period
- uint8_t dtim_period
- uint8_t wlan_capa
- uint8_t btm_mode
- bool bss_transition_supported
- bool neighbor_report_supported

4.69.1 Detailed Description

WLAN Network Profile

This data structure represents a WLAN network profile. It consists of an arbitrary name, WiFi configuration, and IP address configuration.

Every network profile is associated with one of the two interfaces. The network profile can be used for the station interface (i.e. to connect to an Access Point) by setting the role field to WLAN_BSS_ROLE_STA. The network profile can be used for the micro-AP interface (i.e. to start a network of our own.) by setting the mode field to WLAN_BSS_ROLE_UAP.

If the mode field is WLAN_BSS_ROLE_STA, either of the SSID or BSSID fields are used to identify the network, while the other members like channel and security settings characterize the network.

If the mode field is WLAN_BSS_ROLE_UAP, the SSID, channel and security fields are used to define the network to be started.

In both the above cases, the address field is used to determine the type of address assignment to be used for this interface.

4.69.2 Field Documentation

4.69.2.1 id

```
int wlan_network::id
```

Identifier for network profile

4.69.2.2 name

```
char wlan_network::name[WLAN_NETWORK_NAME_MAX_LENGTH+1]
```

The name of this network profile. Each network profile that is added to the WLAN Connection Manager must have a unique name.

4.69.2.3 ssid

```
char wlan_network::ssid[IEEEtypes_SSID_SIZE+1]
```

The network SSID, represented as a C string of up to 32 characters in length. If this profile is used in the micro-AP mode, this field is used as the SSID of the network. If this profile is used in the station mode, this field is used to identify the network. Set the first byte of the SSID to NULL (a 0-length string) to use only the BSSID to find the network.

4.69.2.4 bssid

```
char wlan_network::bssid[IEEEtypes_ADDRESS_SIZE]
```

The network BSSID, represented as a 6-byte array. If this profile is used in the micro-AP mode, this field is ignored. If this profile is used in the station mode, this field is used to identify the network. Set all 6 bytes to 0 to use any BSSID, in which case only the SSID will be used to find the network.

4.69.2.5 channel

```
unsigned int wlan_network::channel
```

The channel for this network.

If this profile is used in micro-AP mode, this field specifies the channel to start the micro-AP interface on. Set this to 0 for auto channel selection.

If this profile is used in the station mode, this constrains the channel on which the network to connect should be present. Set this to 0 to allow the network to be found on any channel.

4.69.2.6 sec_channel_offset

```
uint8_t wlan_network::sec_channel_offset
```

The secondary channel offset

4.69.2.7 acs_band

```
uint16_t wlan_network::acs_band
```

The ACS band if set channel to 0.

4.69.2.8 rssi

```
int wlan_network::rssi
```

RSSI

4.69.2.9 ht_capab

```
unsigned short wlan_network::ht_capab
```

HT capabilities

4.69.2.10 vht_capab

```
unsigned int wlan_network::vht_capab
```

VHT capabilities

4.69.2.11 vht_oper_chwidth

```
unsigned char wlan_network::vht_oper_chwidth
```

VHT bandwidth

4.69.2.12 he_oper_chwidth

```
unsigned char wlan_network::he_oper_chwidth
```

HE bandwidth

4.69.2.13 type

```
enum wlan_bss_type wlan_network::type
```

BSS type

4.69.2.14 role

```
enum wlan_bss_role wlan_network::role
```

The network wireless mode enum wlan_bss_role. Set this to specify what type of wireless network mode to use. This can either be WLAN_BSS_ROLE_STA for use in the station mode, or it can be WLAN_BSS_ROLE_UAP for use in the micro-AP mode.

4.69.2.15 security

```
struct wlan_network_security wlan_network::security
```

The network security configuration specified by struct wlan_network_security for the network.

4.69.2.16 ip

`struct wlan_ip_config wlan_network::ip`

The network IP address configuration specified by struct wlan_ip_config that should be associated with this interface.

4.69.2.17 ssid_specific

`unsigned wlan_network::ssid_specific`

If set to 1, the ssid field contains the specific SSID for this network. The WLAN Connection Manager will only connect to networks whose SSID matches. If set to 0, the ssid field contents are not used when deciding whether to connect to a network, the BSSID field is used instead and any network whose BSSID matches is accepted.

This field will be set to 1 if the network is added with the SSID specified (not an empty string), otherwise it is set to 0.

4.69.2.18 trans_ssid_specific

`unsigned wlan_network::trans_ssid_specific`

If set to 1, the ssid field contains the transitional SSID for this network.

4.69.2.19 bssid_specific

`unsigned wlan_network::bssid_specific`

If set to 1, the bssid field contains the specific BSSID for this network. The WLAN Connection Manager will not connect to any other network with the same SSID unless the BSSID matches. If set to 0, the WLAN Connection Manager will connect to any network whose SSID matches.

This field will be set to 1 if the network is added with the BSSID specified (not set to all zeroes), otherwise it is set to 0.

4.69.2.20 channel_specific

`unsigned wlan_network::channel_specific`

If set to 1, the channel field contains the specific channel for this network. The WLAN Connection Manager will not look for this network on any other channel. If set to 0, the WLAN Connection Manager will look for this network on any available channel.

This field will be set to 1 if the network is added with the channel specified (not set to 0), otherwise it is set to 0.

4.69.2.21 security_specific

`unsigned wlan_network::security_specific`

If set to 0, any security that matches is used. This field is internally set when the security type parameter above is set to WLAN_SECURITY_WILDCARD.

4.69.2.22 dot11n

```
unsigned wlan_network::dot11n
```

The network supports 802.11N. (For internal use only)

4.69.2.23 dot11ac

```
unsigned wlan_network::dot11ac
```

The network supports 802.11AC. (For internal use only)

4.69.2.24 dot11ax

```
unsigned wlan_network::dot11ax
```

The network supports 802.11AX. (For internal use only)

4.69.2.25 mdid

```
uint16_t wlan_network::mdid
```

Mobility Domain ID

4.69.2.26 ft_1x

```
unsigned wlan_network::ft_1x
```

The network uses FT 802.1x security (For internal use only)

4.69.2.27 ft_psk

```
unsigned wlan_network::ft_psk
```

The network uses FT PSK security (For internal use only)

4.69.2.28 ft_sae

```
unsigned wlan_network::ft_sae
```

The network uses FT SAE security (For internal use only)

4.69.2.29 owe_trans_mode

```
unsigned int wlan_network::owe_trans_mode
```

OWE Transition mode

4.69.2.30 trans_ssid

```
char wlan_network::trans_ssid[IEEEtypes_SSID_SIZE+1]
```

The network transitional SSID, represented as a C string of up to 32 characters in length.

This field is used internally.

4.69.2.31 trans_ssid_len

```
unsigned int wlan_network::trans_ssid_len
```

Transitional SSID length

This field is used internally.

4.69.2.32 beacon_period

```
uint16_t wlan_network::beacon_period
```

Beacon period of associated BSS

4.69.2.33 dtim_period

```
uint8_t wlan_network::dtim_period
```

DTIM period of associated BSS

4.69.2.34 wlan_capa

```
uint8_t wlan_network::wlan_capa
```

Wireless capabilities of uAP network 802.11n, 802.11ac or/and 802.11ax

4.69.2.35 btm_mode

```
uint8_t wlan_network::btm_mode
```

BTM mode

4.69.2.36 bss_transition_supported

```
bool wlan_network::bss_transition_supported
```

bss transition support (For internal use only)

4.69.2.37 neighbor_report_supported

```
bool wlan_network::neighbor_report_supported
```

Neighbor report support (For internal use only)

The documentation for this struct was generated from the following file:

- wlan.h

4.70 wlan_network_security Struct Reference

Data Fields

- enum wlan_security_type type
- int key_mgmt
- struct wlan_cipher mcstCipher
- struct wlan_cipher ucstCipher
- unsigned pkc: 1
- int group_cipher
- int pairwise_cipher
- int group_mgmt_cipher
- bool is_pmf_required
- char psk [WLAN_PSK_MAX_LENGTH]
- uint8_t psk_len
- char password [WLAN_PASSWORD_MAX_LENGTH]
- size_t password_len
- char ∗ sae_groups
- uint8_t pwe_derivation
- uint8_t transition_disable
- char ∗ owe_groups
- char pmk [WLAN_PMK_LENGTH]
- bool pmk_valid
- bool mfpc
- bool mfpr
- unsigned wpa3_sb: 1
- unsigned wpa3_sb_192: 1
- unsigned eap_ver: 1
- unsigned peap_label: 1
- uint8_t eap_crypto_binding
- unsigned eap_result_ind: 1
- char identity [IDENTITY_MAX_LENGTH]
- char anonymous_identity [IDENTITY_MAX_LENGTH]
- char eap_password [PASSWORD_MAX_LENGTH]
- unsigned char ∗ ca_cert_data
- size_t ca_cert_len
- unsigned char ∗ client_cert_data
- size_t client_cert_len
- unsigned char ∗ client_key_data
- size_t client_key_len
- char client_key_passwd [PASSWORD_MAX_LENGTH]
- char ca_cert_hash [HASH_MAX_LENGTH]

- char domain_match [DOMAIN_MATCH_MAX_LENGTH]
- char domain_suffix_match [DOMAIN_MATCH_MAX_LENGTH]
- unsigned char ∗ ca_cert2_data
- size_t ca_cert2_len
- unsigned char ∗ client_cert2_data
- size_t client_cert2_len
- unsigned char ∗ client_key2_data
- size_t client_key2_len
- char client_key2_passwd [PASSWORD_MAX_LENGTH]
- unsigned char ∗ dh_data
- size_t dh_len
- unsigned char ∗ server_cert_data
- size_t server_cert_len
- unsigned char ∗ server_key_data
- size_t server_key_len
- char server_key_passwd [PASSWORD_MAX_LENGTH]
- size_t nusers
- char identities [MAX_USERS][IDENTITY_MAX_LENGTH]
- char passwords [MAX_USERS][PASSWORD_MAX_LENGTH]
- char pac_opaque_encr_key [PAC_OPAQUE_ENCR_KEY_MAX_LENGTH]
- char a_id [A_ID_MAX_LENGTH]
- uint8_t fast_prov

4.70.1 Detailed Description

Network security configuration

4.70.2 Field Documentation

4.70.2.1 type

```
enum wlan_security_type wlan_network_security::type
```

Type of network security to use specified by enum wlan_security_type.

4.70.2.2 key_mgmt

```
int wlan_network_security::key_mgmt
```

Key management type

4.70.2.3 mcstCipher

```
struct wlan_cipher wlan_network_security::mcstCipher
```

Type of network security Group Cipher suite used internally

4.70.2.4 ucstCipher

```
struct wlan_cipher wlan_network_security::ucstCipher
```

Type of network security Pairwise Cipher suite used internally

4.70.2.5 pkc

```
unsigned wlan_network_security::pkc
```

Proactive Key Caching

4.70.2.6 group_cipher

```
int wlan_network_security::group_cipher
```

Type of network security Group Cipher suite

4.70.2.7 pairwise_cipher

```
int wlan_network_security::pairwise_cipher
```

Type of network security Pairwise Cipher suite

4.70.2.8 group_mgmt_cipher

```
int wlan_network_security::group_mgmt_cipher
```

Type of network security Pairwise Cipher suite

4.70.2.9 is_pmf_required

```
bool wlan_network_security::is_pmf_required
```

Is PMF required

4.70.2.10 psk

```
char wlan_network_security::psk[WLAN_PSK_MAX_LENGTH]
```

Pre-shared key (network password). For WEP networks this is a hex byte sequence of length psk_len, for WPA and WPA2 networks this is an ASCII pass-phrase of length psk_len. This field is ignored for networks with no security.

4.70.2.11 psk_len

```
uint8_t wlan_network_security::psk_len
```

Length of the WEP key or WPA/WPA2 pass phrase, WLAN_PSK_MIN_LENGTH to WLAN_PSK_MAX_LENGTH. Ignored for networks with no security.

4.70.2.12 password

```
char wlan_network_security::password[WLAN_PASSWORD_MAX_LENGTH]
```

WPA3 SAE password, for WPA3 SAE networks this is an ASCII password of length password_len. This field is ignored for networks with no security.

4.70.2.13 password_len

```
size_t wlan_network_security::password_len
```

Length of the WPA3 SAE Password, WLAN_PASSWORD_MIN_LENGTH to WLAN_PASSWORD_MAX_LENGTH. Ignored for networks with no security.

4.70.2.14 sae_groups

```
char* wlan_network_security::sae_groups
```

SAE Groups

4.70.2.15 pwe_derivation

```
uint8_t wlan_network_security::pwe_derivation
```

PWE derivation

4.70.2.16 transition_disable

```
uint8_t wlan_network_security::transition_disable
```

transition disable

4.70.2.17 owe_groups

```
char* wlan_network_security::owe_groups
```

OWE Groups

4.70.2.18 pmk

```
char wlan_network_security::pmk[WLAN_PMK_LENGTH]
```

Pairwise Master Key. When pmk_valid is set, this is the PMK calculated from the PSK for WPA/PSK networks. If pmk_valid is not set, this field is not valid. When adding networks with wlan_add_network, users can initialize pmk and set pmk_valid in lieu of setting the psk. After successfully connecting to a WPA/PSK network, users can call wlan_get_current_network to inspect pmk_valid and pmk. Thus, the pmk value can be populated in subsequent calls to wlan_add_network. This saves the CPU time required to otherwise calculate the PMK.

4.70.2.19 pmk_valid

```
bool wlan_network_security::pmk_valid
```

Flag reporting whether pmk is valid or not.

4.70.2.20 mfpc

```
bool wlan_network_security::mfpc
```

Management Frame Protection Capable (MFPC)

4.70.2.21 mfpr

```
bool wlan_network_security::mfpr
```

Management Frame Protection Required (MFPR)

4.70.2.22 wpa3_sb

```
unsigned wlan_network_security::wpa3_sb
```

WPA3 Enterprise mode

4.70.2.23 wpa3_sb_192

```
unsigned wlan_network_security::wpa3_sb_192
```

WPA3 Enterprise Suite B 192 mode

4.70.2.24 eap_ver

```
unsigned wlan_network_security::eap_ver
```

PEAP version

4.70.2.25 peap_label

```
unsigned wlan_network_security::peap_label
```

PEAP label

4.70.2.26 eap_crypto_binding

```
uint8_t wlan_network_security::eap_crypto_binding
```

crypto_binding option can be used to control WLAN_SECURITY_EAP_PEAP_MSCHAPV2, WLAN_SECURITY_EAP_PEAP_TLS and WLAN_SECURITY_EAP_PEAP_GTC version 0 cryptobinding behavior: 0 = do not use cryptobinding (default) 1 = use cryptobinding if server supports it 2 = require cryptobinding

4.70.2.27 eap_result_ind

```
unsigned wlan_network_security::eap_result_ind
```

eap_result_ind=1 can be used to enable WLAN_SECURITY_EAP_SIM, WLAN_SECURITY_EAP_AKA and WLAN_SECURITY_EAP_AKA_PRIME to use protected result indication.

4.70.2.28 identity

```
char wlan_network_security::identity[IDENTITY_MAX_LENGTH]
```

Identity string for EAP

4.70.2.29 anonymous_identity

```
char wlan_network_security::anonymous_identity[IDENTITY_MAX_LENGTH]
```

Anonymous identity string for EAP

4.70.2.30 eap_password

```
char wlan_network_security::eap_password[PASSWORD_MAX_LENGTH]
```

Password string for EAP. This field can include either the plaintext password (using ASCII or hex string)

4.70.2.31 ca_cert_data

```
unsigned char* wlan_network_security::ca_cert_data
```

CA cert blob in PEM/DER format

4.70.2.32 ca_cert_len

`size_t wlan_network_security::ca_cert_len`

CA cert blob len

4.70.2.33 client_cert_data

`unsigned char* wlan_network_security::client_cert_data`

Client cert blob in PEM/DER format

4.70.2.34 client_cert_len

`size_t wlan_network_security::client_cert_len`

Client cert blob len

4.70.2.35 client_key_data

`unsigned char* wlan_network_security::client_key_data`

Client key blob

4.70.2.36 client_key_len

`size_t wlan_network_security::client_key_len`

Client key blob len

4.70.2.37 client_key_passwd

`char wlan_network_security::client_key_passwd[PASSWORD_MAX_LENGTH]`

Client key password

4.70.2.38 ca_cert_hash

`char wlan_network_security::ca_cert_hash[HASH_MAX_LENGTH]`

CA cert HASH

4.70.2.39 domain_match

`char wlan_network_security::domain_match[DOMAIN_MATCH_MAX_LENGTH]`

Domain

4.70.2.40 domain_suffix_match

```
char wlan_network_security::domain_suffix_match[DOMAIN_MATCH_MAX_LENGTH]
```

Domain Suffix

4.70.2.41 ca_cert2_data

```
unsigned char* wlan_network_security::ca_cert2_data
```

CA cert2 blob in PEM/DER format

4.70.2.42 ca_cert2_len

```
size_t wlan_network_security::ca_cert2_len
```

CA cert2 blob len

4.70.2.43 client_cert2_data

```
unsigned char* wlan_network_security::client_cert2_data
```

Client cert2 blob in PEM/DER format

4.70.2.44 client_cert2_len

```
size_t wlan_network_security::client_cert2_len
```

Client cert2 blob len

4.70.2.45 client_key2_data

```
unsigned char* wlan_network_security::client_key2_data
```

Client key2 blob

4.70.2.46 client_key2_len

```
size_t wlan_network_security::client_key2_len
```

Client key2 blob len

4.70.2.47 client_key2_passwd

```
char wlan_network_security::client_key2_passwd[PASSWORD_MAX_LENGTH]
```

Client key2 password

4.70.2.48 dh_data

`unsigned char* wlan_network_security::dh_data`

DH params blob

4.70.2.49 dh_len

`size_t wlan_network_security::dh_len`

DH params blob len

4.70.2.50 server_cert_data

`unsigned char* wlan_network_security::server_cert_data`

Server cert blob in PEM/DER format

4.70.2.51 server_cert_len

`size_t wlan_network_security::server_cert_len`

Server cert blob len

4.70.2.52 server_key_data

`unsigned char* wlan_network_security::server_key_data`

Server key blob

4.70.2.53 server_key_len

`size_t wlan_network_security::server_key_len`

Server key blob len

4.70.2.54 server_key_passwd

`char wlan_network_security::server_key_passwd[PASSWORD_MAX_LENGTH]`

Server key password

4.70.2.55 nusers

`size_t wlan_network_security::nusers`

Number of EAP users

4.70.2.56 identities

```
char wlan_network_security::identities[MAX_USERS][IDENTITY_MAX_LENGTH]
```

User Identities

4.70.2.57 passwords

```
char wlan_network_security::passwords[MAX_USERS][PASSWORD_MAX_LENGTH]
```

User Passwords

4.70.2.58 pac_opaque_encr_key

```
char wlan_network_security::pac_opaque_encr_key[PAC_OPAQUE_ENCR_KEY_MAX_LENGTH]
```

Encryption key for EAP-FAST PAC-Opaque values

4.70.2.59 a_id

```
char wlan_network_security::a_id[A_ID_MAX_LENGTH]
```

EAP-FAST authority identity (A-ID)

4.70.2.60 fast_prov

```
uint8_t wlan_network_security::fast_prov
```

EAP-FAST provisioning modes: 0 = provisioning disabled 1 = only anonymous provisioning allowed 2 = only authenticated provisioning allowed 3 = both provisioning modes allowed (default)

The documentation for this struct was generated from the following file:

- wlan.h

4.71 wlan_scan_result Struct Reference

Data Fields

- char ssid [33]
- unsigned int ssid_len
- char bssid [6]
- unsigned int channel
- enum wlan_bss_type type
- enum wlan_bss_role role
- unsigned dot11n: 1
- unsigned dot11ac: 1
- unsigned dot11ax: 1
- unsigned wmm: 1
- unsigned wps: 1
- unsigned int wps_session
- unsigned wep: 1
- unsigned wpa: 1
- unsigned wpa2: 1
- unsigned wpa2_sha256: 1
- unsigned owe: 1
- unsigned wpa3_sae: 1
- unsigned wpa2_entp: 1
- unsigned wpa2_entp_sha256: 1
- unsigned wpa3_1x_sha256: 1
- unsigned wpa3_1x_sha384: 1
- unsigned ft_1x: 1
- unsigned ft_1x_sha384: 1
- unsigned ft_psk: 1
- unsigned ft_sae: 1
- unsigned char rssi
- char trans_ssid [33]
- unsigned int trans_ssid_len
- char trans_bssid [6]
- uint16_t beacon_period
- uint8_t dtim_period
- t_u8 ap_mfpc
- t_u8 ap_mfpr
- bool neighbor_report_supported
- bool bss_transition_supported

4.71.1 Detailed Description

Scan Result

4.71.2 Field Documentation

4.71.2.1 ssid

```
char wlan_scan_result::ssid[33]
```

The network SSID, represented as a NULL-terminated C string of 0 to 32 characters. If the network has a hidden SSID, this will be the empty string.

4.71.2.2 ssid_len

```
unsigned int wlan_scan_result::ssid_len
```

SSID length

4.71.2.3 bssid

```
char wlan_scan_result::bssid[6]
```

The network BSSID, represented as a 6-byte array.

4.71.2.4 channel

```
unsigned int wlan_scan_result::channel
```

The network channel.

4.71.2.5 type

```
enum wlan_bss_type wlan_scan_result::type
```

The network wireless type.

4.71.2.6 role

```
enum wlan_bss_role wlan_scan_result::role
```

The network wireless mode.

4.71.2.7 dot11n

```
unsigned wlan_scan_result::dot11n
```

The network supports 802.11N. This is set to 0 if the network does not support 802.11N or if the system does not have 802.11N support enabled.

4.71.2.8 dot11ac

```
unsigned wlan_scan_result::dot11ac
```

The network supports 802.11AC. This is set to 0 if the network does not support 802.11AC or if the system does not have 802.11AC support enabled.

4.71.2.9 dot11ax

```
unsigned wlan_scan_result::dot11ax
```

The network supports 802.11AX. This is set to 0 if the network does not support 802.11AX or if the system does not have 802.11AX support enabled.

4.71.2.10 wmm

```
unsigned wlan_scan_result::wmm
```

The network supports WMM. This is set to 0 if the network does not support WMM or if the system does not have WMM support enabled.

4.71.2.11 wps

```
unsigned wlan_scan_result::wps
```

The network supports WPS. This is set to 0 if the network does not support WPS or if the system does not have WPS support enabled.

4.71.2.12 wps_session

```
unsigned int wlan_scan_result::wps_session
```

WPS Type PBC/PIN

4.71.2.13 wep

```
unsigned wlan_scan_result::wep
```

The network uses WEP security.

4.71.2.14 wpa

```
unsigned wlan_scan_result::wpa
```

The network uses WPA security.

4.71.2.15 wpa2

```
unsigned wlan_scan_result::wpa2
```

The network uses WPA2 security

4.71.2.16 wpa2_sha256

```
unsigned wlan_scan_result::wpa2_sha256
```

The network uses WPA2 SHA256 security

4.71.2.17 owe

```
unsigned wlan_scan_result::owe
```

The network uses OWE security

4.71.2.18 wpa3_sae

```
unsigned wlan_scan_result::wpa3_sae
```

The network uses WPA3 SAE security

4.71.2.19 wpa2_entp

```
unsigned wlan_scan_result::wpa2_entp
```

The network uses WPA2 Enterprise security

4.71.2.20 wpa2_entp_sha256

```
unsigned wlan_scan_result::wpa2_entp_sha256
```

The network uses WPA2 Enterprise SHA256 security

4.71.2.21 wpa3_1x_sha256

```
unsigned wlan_scan_result::wpa3_1x_sha256
```

The network uses WPA3 Enterprise SHA256 security

4.71.2.22 wpa3_1x_sha384

```
unsigned wlan_scan_result::wpa3_1x_sha384
```

The network uses WPA3 Enterprise SHA384 security

4.71.2.23 ft_1x

```
unsigned wlan_scan_result::ft_1x
```

The network uses FT 802.1x security (For internal use only)

4.71.2.24 ft_1x_sha384

```
unsigned wlan_scan_result::ft_1x_sha384
```

The network uses FT 892.1x SHA384 security

4.71.2.25 ft_psk

```
unsigned wlan_scan_result::ft_psk
```

The network uses FT PSK security (For internal use only)

4.71.2.26 ft_sae

```
unsigned wlan_scan_result::ft_sae
```

The network uses FT SAE security (For internal use only)

4.71.2.27 rssi

```
unsigned char wlan_scan_result::rssi
```

The signal strength of the beacon

4.71.2.28 trans_ssid

```
char wlan_scan_result::trans_ssid[33]
```

The network SSID, represented as a NULL-terminated C string of 0 to 32 characters. If the network has a hidden SSID, this will be the empty string.

4.71.2.29 trans_ssid_len

```
unsigned int wlan_scan_result::trans_ssid_len
```

SSID length

4.71.2.30 trans_bssid

```
char wlan_scan_result::trans_bssid[6]
```

The network BSSID, represented as a 6-byte array.

4.71.2.31 beacon_period

```
uint16_t wlan_scan_result::beacon_period
```

Beacon Period

4.71.2.32 dtim_period

```
uint8_t wlan_scan_result::dtim_period
```

DTIM Period

4.71.2.33 ap_mfpc

```
t_u8 wlan_scan_result::ap_mfpc
```

MFPC bit of AP

4.71.2.34 ap_mfpr

```
t_u8 wlan_scan_result::ap_mfpr
```

MFPR bit of AP

4.71.2.35 neighbor_report_supported

```
bool wlan_scan_result::neighbor_report_supported
```

Neigbort report support (For internal use only)

4.71.2.36 bss_transition_supported

```
bool wlan_scan_result::bss_transition_supported
```

bss transition support (For internal use only)

The documentation for this struct was generated from the following file:

- wlan.h

Chapter 5

File Documentation

5.1 cli.h File Reference

CLI module.

5.1.1 Detailed Description

5.1.2 Usage

The CLI module lets you register commands with the CLI interface. Modules that wish to register the commands should initialize the struct cli_command structure and pass this to cli_register_command(). These commands will then be available on the CLI.

5.1.3 Function Documentation

5.1.3.1 cli_register_command()

```
int cli_register_command (
            const struct cli_command * command )
```

Register a CLI command

This function registers a command with the command-line interface.

Parameters

in	*command*	The structure to register one CLI command

Returns

0 on success
1 on failure

5.1.3.2 cli_unregister_command()

```
int cli_unregister_command (
            const struct cli_command * command )
```

Unregister a CLI command

This function unregisters a command from the command-line interface.

Parameters

in	*command*	The structure to unregister one CLI command

Returns

0 on success
1 on failure

5.1.3.3 cli_init()

```
int cli_init (
            void )
```

Initialize the CLI module

Returns

WM_SUCCESS on success
error code otherwise.

5.1.3.4 cli_deinit()

```
int cli_deinit (
            void )
```

DeInitialize the CLI module

Returns

WM_SUCCESS on success
error code otherwise.

5.1.3.5 cli_stop()

```
int cli_stop (
            void )
```

Stop the CLI thread and carry out the cleanup

Returns

WM_SUCCESS on success
error code otherwise.

5.1.3.6 cli_register_commands()

```
int cli_register_commands (
            const struct cli_command * commands,
            int num_commands )
```

Register a batch of CLI commands

Often, a module will want to register several commands.

Parameters

in	*commands*	Pointer to an array of commands.
in	*num_commands*	Number of commands in the array.

Returns

0 on success
1 on failure

5.1.3.7 cli_unregister_commands()

```
int cli_unregister_commands (
            const struct cli_command * commands,
            int num_commands )
```

Unregister a batch of CLI commands

Parameters

in	*commands*	Pointer to an array of commands.
in	*num_commands*	Number of commands in the array.

Returns

0 on success
1 on failure

5.1.3.8 cli_get_cmd_buffer()

```
int cli_get_cmd_buffer (
            char ** buff )
```

Get a command buffer

If an external input task wants to use the CLI, it can use cli_get_cmd_buffer() to get a command buffer that it can then submit to the CLI later using cli_submit_cmd_buffer().

Parameters

buff	Pointer to a char ∗ to place the buffer pointer in.

Returns

WM_SUCCESS on success
error code otherwise.

5.1.3.9 cli_submit_cmd_buffer()

```
int cli_submit_cmd_buffer (
            char ** buff )
```

Submit a command buffer to the CLI

Sends the command buffer to the CLI for processing.

Parameters

buff	Pointer to a char ∗ buffer.

Returns

WM_SUCCESS on success
error code otherwise.

5.2 dhcp-server.h File Reference

DHCP server.

5.2.1 Detailed Description

The DHCP Server is required in the provisioning mode of the application to assign IP Address to Wireless Clients that connect to the WM.

5.2.2 Function Documentation

5.2.2.1 dhcpd_cli_init()

```
int dhcpd_cli_init (
            void  )
```

Register DHCP server commands

This function registers the CLI dhcp-stat for the DHCP server. dhcp-stat command displays ip to associated client mac mapping.

Returns

-WM_E_DHCPD_REGISTER_CMDS if cli init operation failed.
WM_SUCCESS if cli init operation success.

5.2.2.2 dhcpd_cli_deinit()

```
int dhcpd_cli_deinit (
            void  )
```

Unrgister DHCP server commands

This function unregisters the CLI dhcp-stat for the DHCP server. dhcp-stat command displays ip to associated client mac mapping.

Returns

-WM_E_DHCPD_REGISTER_CMDS if cli init operation failed.
WM_SUCCESS if cli init operation success.

5.2.2.3 dhcp_server_start()

```
int dhcp_server_start (
            void * intrfc_handle )
```

Start DHCP server

This starts the DHCP server on the interface specified. Typically DHCP server should be running on the micro-AP interface but it can also run on wifi direct interface if configured as group owner. Use net_get_uap_handle() to get micro-AP interface handle.

Parameters

in	*intrfc_handle*	The interface handle on which DHCP server will start

Returns

WM_SUCCESS on success or error code

5.2.2.4 dhcp_enable_dns_server()

```
void dhcp_enable_dns_server (
            char ** domain_names )
```

Start DNS server

This starts the DNS server on the interface specified for dhcp server. This function needs to be used before dhcp_server_start() function and can be invoked on receiving WLAN_REASON_INITIALIZED event in the application before starting micro-AP.

The application needs to define its own list of domain names with the last entry as NULL. The dns server handles dns queries and if domain name match is found then resolves it to device ip address. Currently the maximum length for each domain name is set to 32 bytes.

Eg. char *domain_names[] = {"nxpprov.net", "www.nxpprov.net", NULL};

dhcp_enable_dns_server(domain_names);

However, application can also start dns server without any domain names specified to solve following issue. Some of the client devices do not show WiFi signal strength symbol when connected to micro-AP in open mode, if dns queries are not resolved. With dns server support enabled, dns server responds with ERROR_REFUSED indicating that the DNS server refuses to provide whatever data client is asking for.

Parameters

in	*domain_names*	Pointer to the list of domain names or NULL.

5.2.2.5 dhcp_server_stop()

```
void dhcp_server_stop (
            void )
```

Stop DHCP server

5.2.2.6 dhcp_server_lease_timeout()

```
int dhcp_server_lease_timeout (
            uint32_t val )
```

Configure the DHCP dynamic IP lease time

This API configures the dynamic IP lease time, which should be invoked before DHCP server initialization

Parameters

in	*val*	Number of seconds, use (60U∗60U∗number of hours) for clarity. Max value is (60U∗60U∗24U∗49700U)

Returns

Error status code

5.2.2.7 dhcp_get_ip_from_mac()

```
int dhcp_get_ip_from_mac (
            uint8_t * client_mac,
            uint32_t * client_ip )
```

Get IP address corresponding to MAC address from dhcpd ip-mac mapping

This API returns IP address mapping to the MAC address present in cache. IP-MAC cache stores MAC to IP mapping of previously or currently connected clients.

Parameters

in	*client_mac*	Pointer to a six byte array containing the MAC address of the client
out	*client_ip*	Pointer to IP address of the client

Returns

WM_SUCCESS on success or -WM_FAIL.

5.2.2.8 dhcp_stat()

```
void dhcp_stat (
            void )
```

Print DHCP stats on the console

This API prints DHCP stats on the console

5.2.3 Enumeration Type Documentation

5.2.3.1 wm_dhcpd_errno

```
enum wm_dhcpd_errno
```

DHCPD Error Codes

Enumerator

WM_E_DHCPD_SERVER_RUNNING	Dhcp server is already running
WM_E_DHCPD_THREAD_CREATE	Failed to create dhcp thread
WM_E_DHCPD_MUTEX_CREATE	Failed to create dhcp mutex
WM_E_DHCPD_REGISTER_CMDS	Failed to register dhcp commands
WM_E_DHCPD_RESP_SEND	Failed to send dhcp response
WM_E_DHCPD_DNS_IGNORE	Ignore as msg is not a valid dns query
WM_E_DHCPD_BUFFER_FULL	Buffer overflow occurred
WM_E_DHCPD_INVALID_INPUT	The input message is NULL or has incorrect length
WM_E_DHCPD_INVALID_OPCODE	Invalid opcode in the dhcp message
WM_E_DHCPD_INCORRECT_HEADER	Invalid header type or incorrect header length
WM_E_DHCPD_SPOOF_NAME	Spoof length is either NULL or it exceeds max length
WM_E_DHCPD_BCAST_ADDR	Failed to get broadcast address
WM_E_DHCPD_IP_ADDR	Failed to look up requested IP address from the interface
WM_E_DHCPD_NETMASK	Failed to look up requested netmask from the interface
WM_E_DHCPD_SOCKET	Failed to create the socket
WM_E_DHCPD_ARP_SEND	Failed to send Gratuitous ARP
WM_E_DHCPD_IOCTL_CALL	Error in ioctl call
WM_E_DHCPD_INIT	Failed to init dhcp server

5.3 iperf.h File Reference

This file provides the support for network utility iperf.

5.3.1 Function Documentation

5.3.1.1 iperf_cli_init()

```
int iperf_cli_init ( )
```

Register the Network Utility CLI command iperf.

Note

This function can only be called by the application after wlan_init() called.

Returns

WM_SUCCESS if the CLI commands are registered
-WM_FAIL otherwise (for example if this function was called while the CLI commands were already registered)

5.3.1.2 iperf_cli_deinit()

```
int iperf_cli_deinit ( )
```

Unregister Network Utility CLI command iperf.

Returns

WM_SUCCESS if the CLI commands are unregistered
-WM_FAIL otherwise

5.4 wifi-decl.h File Reference

Wifi structure declarations.

5.4.1 Macro Documentation

5.4.1.1 MLAN_MAX_VER_STR_LEN

```
#define MLAN_MAX_VER_STR_LEN 128
```

Version string buffer length

5.4.1.2 OWE_TRANS_MODE_OPEN

```
#define OWE_TRANS_MODE_OPEN 1U
```

The open AP in OWE transmition Mode

5.4.1.3 OWE_TRANS_MODE_OWE

```
#define OWE_TRANS_MODE_OWE 2U
```

The security AP in OWE trsnsition Mode

5.4.1.4 BSS_TYPE_STA

```
#define BSS_TYPE_STA 0U
```

BSS type : STA

5.4.1.5 BSS_TYPE_UAP

```
#define BSS_TYPE_UAP 1U
```

BSS type : UAP

5.4.1.6 MLAN_MAX_SSID_LENGTH

```
#define MLAN_MAX_SSID_LENGTH (32U)
```

MLAN Maximum SSID Length

5.4.1.7 MLAN_MAX_PASS_LENGTH

```
#define MLAN_MAX_PASS_LENGTH (64)
```

MLAN Maximum PASSPHRASE Length

5.4.2 Enumeration Type Documentation

5.4.2.1 wifi_SubBand_t

```
enum wifi_SubBand_t
```

Wifi subband enum

Enumerator

SubBand_2_4_GHz	Subband 2.4 GHz
SubBand_5_GHz_0	Subband 5 GHz 0
SubBand_5_GHz_1	Subband 5 GHz 1
SubBand_5_GHz_2	Subband 5 GHz 2
SubBand_5_GHz_3	Subband 5 GHz 3

5.4.2.2 wifi_frame_type_t

```
enum wifi_frame_type_t
```

Wifi frame types

Enumerator

ASSOC_REQ_FRAME	Assoc request frame
ASSOC_RESP_FRAME	Assoc response frame
REASSOC_REQ_FRAME	ReAssoc request frame
REASSOC_RESP_FRAME	ReAssoc response frame
PROBE_REQ_FRAME	Probe request frame
PROBE_RESP_FRAME	Probe response frame
BEACON_FRAME	BEACON frame
DISASSOC_FRAME	Dis assoc frame
AUTH_FRAME	Auth frame
DEAUTH_FRAME	Deauth frame
ACTION_FRAME	Action frame
DATA_FRAME	Data frame
QOS_DATA_FRAME	QOS frame

5.5 wifi.h File Reference

This file contains interface to wifi driver.

5.5.1 Function Documentation

5.5.1.1 wifi_init()

```
int wifi_init (
            const uint8_t * fw_start_addr,
            const size_t size )
```

Initialize Wi-Fi driver module.

Performs SDIO init, downloads Wi-Fi Firmware, creates Wi-Fi Driver and command response processor thread.

Also creates mutex, and semaphores used in command and data synchronizations.

Parameters

in	*fw_start_addr*	address of stored Wi-Fi Firmware.
in	*size*	Size of Wi-Fi Firmware.

Returns

WM_SUCCESS on success or -WM_FAIL on error.

5.5.1.2 wifi_init_fcc()

```
int wifi_init_fcc (
            const uint8_t * fw_start_addr,
            const size_t size )
```

Initialize Wi-Fi driver module for FCC Certification.

Performs SDIO init, downloads Wi-Fi Firmware, creates Wi-Fi Driver and command response processor thread.

Also creates mutex, and semaphores used in command and data synchronizations.

Parameters

in	*fw_start_addr*	address of stored Manufacturing Wi-Fi Firmware.
in	*size*	Size of Manufacturing Wi-Fi Firmware.

Returns

WM_SUCCESS on success or -WM_FAIL on error.

5.5.1.3 wifi_deinit()

```
void wifi_deinit (
            void )
```

Deinitialize Wi-Fi driver module.

Performs SDIO deinit, send shutdown command to Wi-Fi Firmware, deletes Wi-Fi Driver and command processor thread.

Also deletes mutex and semaphores used in command and data synchronizations.

5.5.1.4 wifi_set_tx_status()

```
void wifi_set_tx_status (
            t_u8 status )
```

This API can be used to set wifi driver tx status.

Parameters

in	*status*	Status to set for TX

5.5.1.5 wifi_set_rx_status()

```
void wifi_set_rx_status (
            t_u8 status )
```

This API can be used to set wifi driver rx status.

Parameters

in	*status*	Status to set for RX

5.5.1.6 reset_ie_index()

```
void reset_ie_index ( )
```

This API can be used to reset mgmt_ie_index_bitmap.

5.5.1.7 wifi_register_data_input_callback()

```
int wifi_register_data_input_callback (
            void(*)(const uint8_t interface, const uint8_t *buffer, const uint16_t len) data←
_intput_callback )
```

Register Data callback function with Wi-Fi Driver to receive DATA from SDIO.

This callback function is used to send data received from Wi-Fi firmware to the networking stack.

Parameters

in	*data_intput_callback*	Function that needs to be called

Returns

WM_SUCCESS

5.5.1.8 wifi_deregister_data_input_callback()

```
void wifi_deregister_data_input_callback (
            void )
```

Deregister Data callback function from Wi-Fi Driver

5.5.1.9 wifi_register_amsdu_data_input_callback()

```
int wifi_register_amsdu_data_input_callback (
            void(*)(uint8_t interface, uint8_t *buffer, uint16_t len) amsdu_data_intput_←
callback )
```

Register Data callback function with Wi-Fi Driver to receive processed AMSDU DATA from Wi-Fi driver.

This callback function is used to send data received from Wi-Fi firmware to the networking stack.

Parameters

in	*amsdu_data_intput_callback*	Function that needs to be called

Returns

WM_SUCESS

5.5.1.10 wifi_deregister_amsdu_data_input_callback()

```
void wifi_deregister_amsdu_data_input_callback (
            void )
```

Deregister Data callback function from Wi-Fi Driver

5.5.1.11 wifi_low_level_output()

```
int wifi_low_level_output (
            const uint8_t interface,
            const uint8_t * buffer,
            const uint16_t len,
            uint8_t pkt_prio,
            uint8_t tid )
```

Wi-Fi Driver low level output function.

Data received from upper layer is passed to Wi-Fi Driver for transmission.

Parameters

in	*interface*	Interface on which DATA frame will be transmitted. 0 for Station interface, 1 for uAP interface and 2 for Wi-Fi Direct interface.
in	*buffer*	A pointer pointing to DATA frame.
in	*len*	Length of DATA frame.
in	*pkt_prio*	Priority for.sending packet.
in	*tid*	TID for tx.

Returns

WM_SUCCESS on success or -WM_E_NOMEM if memory is not available or -WM_E_BUSY if SDIO is busy.

5.5.1.12 wifi_set_packet_retry_count()

```
void wifi_set_packet_retry_count (
            const int count )
```

API to enable packet retries at wifi driver level.

This API sets retry count which will be used by wifi driver to retry packet transmission in case there was failure in earlier attempt. Failure may happen due to SDIO write port un-availability or other failures in SDIO write operation.

Note

Default value of retry count is zero.

Parameters

in	*count*	No of retry attempts.

5.5.1.13 wifi_sta_ampdu_tx_enable()

```
void wifi_sta_ampdu_tx_enable (
            void )
```

This API can be used to enable AMPDU support on the go when station is a transmitter.

5.5.1.14 wifi_sta_ampdu_tx_disable()

```
void wifi_sta_ampdu_tx_disable (
            void )
```

This API can be used to disable AMPDU support on the go when station is a transmitter.

5.5.1.15 wifi_sta_ampdu_tx_enable_per_tid()

```
void wifi_sta_ampdu_tx_enable_per_tid (
            t_u8 tid )
```

This API can be used to set tid to enable AMPDU support on the go when station is a transmitter.

Parameters

in	*tid*	tid value

5.5.1.16 wifi_sta_ampdu_tx_enable_per_tid_is_allowed()

```
t_u8 wifi_sta_ampdu_tx_enable_per_tid_is_allowed (
            t_u8 tid )
```

This API can be used to check if tid to enable AMPDU is allowed when station is a transmitter.

Parameters

in	*tid*	tid value

Returns

MTRUE or MFALSE

5.5.1.17 wifi_sta_ampdu_rx_enable()

```
void wifi_sta_ampdu_rx_enable (
            void )
```

This API can be used to enable AMPDU support on the go when station is a receiver.

5.5.1.18 wifi_sta_ampdu_rx_enable_per_tid()

```
void wifi_sta_ampdu_rx_enable_per_tid (
            t_u8 tid )
```

This API can be used to set tid to enable AMPDU support on the go when station is a receiver.

Parameters

in	*tid*	tid value

5.5.1.19 wifi_sta_ampdu_rx_enable_per_tid_is_allowed()

```
t_u8 wifi_sta_ampdu_rx_enable_per_tid_is_allowed (
            t_u8 tid )
```

This API can be used to check if tid to enable AMPDU is allowed when station is a receiver.

Parameters

in	*tid*	tid value

Returns

MTRUE or MFALSE

5.5.1.20 wifi_uap_ampdu_rx_enable()

```
void wifi_uap_ampdu_rx_enable (
            void )
```

This API can be used to enable AMPDU support on the go when uap is a receiver.

5.5.1.21 wifi_uap_ampdu_rx_enable_per_tid()

```
void wifi_uap_ampdu_rx_enable_per_tid (
            t_u8 tid )
```

This API can be used to set tid to enable AMPDU support on the go when uap is a receiver.

Parameters

in	*tid*	tid value

5.5.1.22 wifi_uap_ampdu_rx_enable_per_tid_is_allowed()

```
t_u8 wifi_uap_ampdu_rx_enable_per_tid_is_allowed (
            t_u8 tid )
```

This API can be used to check if tid to enable AMPDU is allowed when uap is a receiver.

Parameters

in	*tid*	tid value

Returns

MTRUE or MFALSE

5.5.1.23 wifi_uap_ampdu_rx_disable()

```
void wifi_uap_ampdu_rx_disable (
            void )
```

This API can be used to disable AMPDU support on the go when uap is a receiver.

5.5.1.24 wifi_uap_ampdu_tx_enable()

```
void wifi_uap_ampdu_tx_enable (
            void )
```

This API can be used to enable AMPDU support on the go when uap is a transmitter.

5.5.1.25 wifi_uap_ampdu_tx_enable_per_tid()

```
void wifi_uap_ampdu_tx_enable_per_tid (
            t_u8 tid )
```

This API can be used to set tid to enable AMPDU support on the go when uap is a transmitter.

Parameters

in	*tid*	tid value

5.5.1.26 wifi_uap_ampdu_tx_enable_per_tid_is_allowed()

```
t_u8 wifi_uap_ampdu_tx_enable_per_tid_is_allowed (
            t_u8 tid )
```

This API can be used to check if tid to enable AMPDU is allowed when uap is a transmitter.

Parameters

in	*tid*	tid value

Returns

MTRUE or MFALSE

5.5.1.27 wifi_uap_ampdu_tx_disable()

```
void wifi_uap_ampdu_tx_disable (
            void )
```

This API can be used to disable AMPDU support on the go when uap is a transmitter.

5.5.1.28 wifi_sta_ampdu_rx_disable()

```
void wifi_sta_ampdu_rx_disable (
            void )
```

This API can be used to disable AMPDU support on the go when station is a receiver.

5.5.1.29 wifi_get_device_mac_addr()

```
int wifi_get_device_mac_addr (
            wifi_mac_addr_t * mac_addr )
```

Get the device sta MAC address

Parameters

out	*mac_addr*	Mac address

Returns

WM_SUCESS

5.5.1.30 wifi_get_device_uap_mac_addr()

```
int wifi_get_device_uap_mac_addr (
            wifi_mac_addr_t * mac_addr_uap )
```

Get the device uap MAC address

Parameters

out	*mac_addr_uap*	Mac address

Returns

WM_SUCESS

5.5.1.31 wifi_get_device_firmware_version_ext()

```
int wifi_get_device_firmware_version_ext (
            wifi_fw_version_ext_t * fw_ver_ext )
```

Get the cached string representation of the wlan firmware extended version.

Parameters

in	*fw_ver_ext*	Firmware Version Extended

Returns

WM_SUCCESS

5.5.1.32 wifi_get_last_cmd_sent_ms()

```
unsigned wifi_get_last_cmd_sent_ms (
            void )
```

Get the timestamp of the last command sent to the firmware

Returns

Timestamp in millisec of the last command sent

5.5.1.33 wifi_update_last_cmd_sent_ms()

```
void wifi_update_last_cmd_sent_ms (
            void )
```

This will update the last command sent variable value to current time. This is used for power management.

5.5.1.34 wifi_register_event_queue()

```
int wifi_register_event_queue (
            os_queue_t * event_queue )
```

Register an event queue with the wifi driver to receive events

The list of events which can be received from the wifi driver are enumerated in the file wifi_events.h

Parameters

in	*event_queue*	The queue to which wifi driver will post events.

Note

Only one queue can be registered. If the registered queue needs to be changed unregister the earlier queue first.

Returns

Standard SDK return codes

5.5.1.35 wifi_unregister_event_queue()

```
int wifi_unregister_event_queue (
            os_queue_t * event_queue )
```

Unregister an event queue from the wifi driver.

Parameters

in	*event_queue*	The queue to which was registered earlier with the wifi driver.

Returns

Standard SDK return codes

5.5.1.36 wifi_get_scan_result()

```
int wifi_get_scan_result (
            unsigned int index,
            struct wifi_scan_result2 ** desc )
```

Get scan list

Parameters

in	*index*	Index
out	*desc*	Descriptor of type wifi_scan_result2

Returns

WM_SUCCESS on success or error code.

5.5.1.37 wifi_get_scan_result_count()

```
int wifi_get_scan_result_count (
            unsigned * count )
```

Get the count of elements in the scan list

Parameters

in,out	*count*	Pointer to a variable which will hold the count after this call returns

Warning

The count returned by this function is the current count of the elements. A scan command given to the driver or some other background event may change this count in the wifi driver. Thus when the API wifi_get_scan_result is used to get individual elements of the scan list, do not assume that it will return exactly 'count' number of elements. Your application should not consider such situations as a major event.

Returns

Standard SDK return codes.

5.5.1.38 wifi_uap_bss_sta_list()

```
int wifi_uap_bss_sta_list (
            wifi_sta_list_t ** list )
```

Returns the current STA list connected to our uAP

This function gets its information after querying the firmware. It will block till the response is received from firmware or a timeout.

Parameters

in,out	*list*	After this call returns this points to the structure wifi_sta_list_t allocated by the callee. This is variable length structure and depends on count variable inside it. **The caller needs to free this buffer after use.**. If this function is unable to get the sta list, the value of list parameter will be NULL

Note

The caller needs to explicitly free the buffer returned by this function.

Returns

void

5.5.1.39 wifi_set_cal_data()

```
void wifi_set_cal_data (
            const uint8_t * cdata,
            const unsigned int clen )
```

Set wifi calibration data in firmware.

This function may be used to set wifi calibration data in firmware.

Parameters

in	*cdata*	The calibration data
in	*clen*	Length of calibration data

5.5.1.40 wifi_set_mac_addr()

```
void wifi_set_mac_addr (
            uint8_t * mac )
```

Set wifi MAC address in firmware at load time.

This function may be used to set wifi MAC address in firmware.

Parameters

in	*mac*	The new MAC Address

5.5.1.41 _wifi_set_mac_addr()

```
void _wifi_set_mac_addr (
            const uint8_t * mac,
            mlan_bss_type bss_type )
```

Set wifi MAC address in firmware at run time.

This function may be used to set wifi MAC address in firmware as per passed bss type.

Parameters

in	*mac*	The new MAC Address
in	*bss_type*	BSS Type

5.5.1.42 wifi_add_mcast_filter()

```
int wifi_add_mcast_filter (
            uint8_t * mac_addr )
```

Add Multicast Filter by MAC Address

Multicast filters should be registered with the WiFi driver for IP-level multicast addresses to work. This API allows for registration of such filters with the WiFi driver.

If multicast-mapped MAC address is 00:12:23:34:45:56 then pass mac_addr as below: mac_add[0] = 0x00 mac_add[1] = 0x12 mac_add[2] = 0x23 mac_add[3] = 0x34 mac_add[4] = 0x45 mac_add[5] = 0x56

Parameters

in	*mac_addr*	multicast mapped MAC address

Returns

0 on Success or else Error

5.5.1.43 wifi_remove_mcast_filter()

```
int wifi_remove_mcast_filter (
            uint8_t * mac_addr )
```

Remove Multicast Filter by MAC Address

This function removes multicast filters for the given multicast-mapped MAC address. If multicast-mapped MAC address is 00:12:23:34:45:56 then pass mac_addr as below: mac_add[0] = 0x00 mac_add[1] = 0x12 mac_add[2] = 0x23 mac_add[3] = 0x34 mac_add[4] = 0x45 mac_add[5] = 0x56

Parameters

in	*mac_addr*	multicast mapped MAC address

Returns

0 on Success or else Error

5.5.1.44 wifi_get_ipv4_multicast_mac()

```
void wifi_get_ipv4_multicast_mac (
            uint32_t ipaddr,
            uint8_t * mac_addr )
```

Get Multicast Mapped Mac address from IPv4

This function will generate Multicast Mapped MAC address from IPv4 Multicast Mapped MAC address will be in following format: 1) Higher 24-bits filled with IANA Multicast OUI (01-00-5E) 2) 24th bit set as Zero 3) Lower 23-bits filled with IP address (ignoring higher 9bits).

Parameters

in	*ipaddr*	ipaddress(input)
in	*mac_addr*	multicast mapped MAC address(output)

5.5.1.45 wifi_get_ipv6_multicast_mac()

```
void wifi_get_ipv6_multicast_mac (
            uint32_t ipaddr,
            uint8_t * mac_addr )
```

Get Multicast Mapped Mac address from IPv6 address

This function will generate Multicast Mapped MAC address from IPv6 address. Multicast Mapped MAC address will be in following format: 1) Higher 16-bits filled with IANA Multicast OUI (33-33) 2) Lower 32-bits filled with last 4 bytes of IPv6 address

Parameters

in	*ipaddr*	last 4 bytes of IPv6 address
in	*mac_addr*	multicast mapped MAC address

5.5.1.46 wifi_get_region_code()

```
int wifi_get_region_code (
            t_u32 * region_code )
```

Get the wifi region code

This function will return one of the following values in the region_code variable.
0x10 : US FCC
0x20 : CANADA
0x30 : EU
0x32 : FRANCE
0x40 : JAPAN
0x41 : JAPAN
0x50 : China
0xfe : JAPAN
0xff : Special

Parameters

out	*region_code*	Region Code

Returns

Standard WMSDK return codes.

5.5.1.47 wifi_set_region_code()

```
int wifi_set_region_code (
            t_u32 region_code )
```

Set the wifi region code.

This function takes one of the values from the following array.
0x10 : US FCC
0x20 : CANADA
0x30 : EU
0x32 : FRANCE
0x40 : JAPAN
0x41 : JAPAN
0x50 : China
0xfe : JAPAN
0xff : Special

Parameters

in	*region_code*	Region Code

Returns

Standard WMSDK return codes.

5.5.1.48 wifi_set_country_code()

```
int wifi_set_country_code (
            const char * alpha2 )
```

Set/Get country code

Parameters

in	*alpha2*	country code in 3bytes string, 2bytes country code and 1byte 0 WW : World Wide Safe US : US FCC CA : IC Canada SG : Singapore EU : ETSI AU : Australia KR : Republic Of Korea FR : France JP : Japan CN : China

Returns

WM_SUCCESS if successful otherwise failure.

5.5.1.49 wifi_get_uap_channel()

```
int wifi_get_uap_channel (
            int * channel )
```

Get the uAP channel number

Parameters

in	*channel*	Pointer to channel number. Will be initialized by callee

Returns

Standard WMSDK return code

5.5.1.50 wifi_uap_pmf_getset()

```
int wifi_uap_pmf_getset (
            uint8_t action,
            uint8_t * mfpc,
            uint8_t * mfpr )
```

Get/Set the uAP mfpc and mfpr

Parameters

in	*action*	
in,out	*mfpc*	Management Frame Protection Capable (MFPC) 1: Management Frame Protection Capable 0: Management Frame Protection not Capable
in,out	*mfpr*	Management Frame Protection Required (MFPR) 1: Management Frame Protection Required 0: Management Frame Protection Optional

Returns

cmd response status

5.5.1.51 wifi_uap_enable_11d_support()

```
int wifi_uap_enable_11d_support ( )
```

enable/disable 80211d domain feature for the uAP.

Note

This API only set 80211d domain feature. The actual application will happen only during starting phase of uAP. So, if the uAP is already started then the configuration will not apply till uAP re-start.

Returns

WM_SUCCESS on success or error code.

5.5.1.52 wifi_uap_config_wifi_capa()

```
void wifi_uap_config_wifi_capa (
            uint8_t wlan_capa )
```

Set uAP capability

User can set uAP capability of 11ax/11ac/11n/legacy. Default is 11ax.

Parameters

in	*wlan_capa*	uAP capability bitmap. 1111 - 11AX 0111 - 11AC 0011 - 11N 0001 - legacy

5.5.1.53 wifi_set_11ax_cfg()

```
int wifi_set_11ax_cfg (
            wifi_11ax_config_t * ax_config )
```

Set 11ax config params

Parameters

in,out	*ax_config*	11AX config parameters to be sent to Firmware

Returns

WM_SUCCESS if successful otherwise failure.

5.5.1.54 wifi_set_btwt_cfg()

```
int wifi_set_btwt_cfg (
            const wifi_btwt_config_t * btwt_config )
```

Set btwt config params

Parameters

in	*btwt_config*	Broadcast TWT setup parameters to be sent to Firmware

Returns

WM_SUCCESS if successful otherwise failure.

5.5.1.55 wifi_set_twt_setup_cfg()

```
int wifi_set_twt_setup_cfg (
            const wifi_twt_setup_config_t * twt_setup )
```

Set twt setup config params

Parameters

in	*twt_setup*	TWT Setup parameters to be sent to Firmware

Returns

WM_SUCCESS if successful otherwise failure.

5.5.1.56 wifi_set_twt_teardown_cfg()

```
int wifi_set_twt_teardown_cfg (
            const wifi_twt_teardown_config_t * teardown_config )
```

Set twt teardown config params

Parameters

in	*teardown_config*	TWT Teardown parameters to be sent to Firmware

Returns

WM_SUCCESS if successful otherwise failure.

5.5.1.57 wifi_get_twt_report()

```
int wifi_get_twt_report (
            wifi_twt_report_t * twt_report )
```

Get twt report

Parameters

out	*twt_report*	TWT Report parameters to be sent to Firmware

Returns

WM_SUCCESS if successful otherwise failure.

5.5.1.58 wifi_register_fw_dump_cb()

```
void wifi_register_fw_dump_cb (
            int(*)() wifi_usb_mount_cb,
            int(*)(char *test_file_name) wifi_usb_file_open_cb,
            int(*)(uint8_t *data, size_t data_len) wifi_usb_file_write_cb,
            int(*)() wifi_usb_file_close_cb )
```

This function registers callbacks which are used to generate FW Dump on USB device.

Parameters

in	*wifi_usb_mount_cb*	Callback to mount usb device.
in	*wifi_usb_file_open_cb*	Callback to open file on usb device for FW dump.
in	*wifi_usb_file_write_cb*	Callback to write FW dump data to opened file.
in	*wifi_usb_file_close_cb*	Callback to close FW dump file.

5.5.1.59 wifi_show_os_mem_stat()

```
void wifi_show_os_mem_stat ( )
```

Show os mem alloc and free info.

5.5.1.60 wifi_inject_frame()

```
int wifi_inject_frame (
            const enum wlan_bss_type bss_type,
            const uint8_t * buff,
            const size_t len )
```

Frame Tx - Injecting Wireless frames from Host

This function is used to Inject Wireless frames from application directly.

Note

All injected frames will be sent on station interface. Application needs minimum of 2 KBytes stack for successful operation. Also application have to take care of allocating buffer for 802.11 Wireless frame (Header + Data) and freeing allocated buffer. Also this API may not work when Power Save is enabled on station interface.

Parameters

in	*bss_type*	The interface on which management frame needs to be send.
in	*buff*	Buffer holding 802.11 Wireless frame (Header + Data).
in	*len*	Length of the 802.11 Wireless frame.

Returns

WM_SUCCESS on success or error code.

5.5.1.61 wifi_csi_cfg()

```
int wifi_csi_cfg (
            wifi_csi_config_params_t * csi_params )
```

Send the csi config parameter to FW.

Parameters

in	*csi_params*	Csi config parameter

Returns

WM_SUCCESS if successful otherwise failure.

5.5.1.62 region_string_2_region_code()

```
t_u8 region_string_2_region_code (
            t_u8 * region_string )
```

Parameters

region_string	Region string

Returns

Region code

5.5.2 Macro Documentation

5.5.2.1 MBIT

```
#define MBIT(
              x ) (((t_u32)1) << (x))
```

BIT value

5.5.2.2 WIFI_MGMT_ACTION

```
#define WIFI_MGMT_ACTION MBIT(13)
```

BITMAP for Action frame

5.5.3 Enumeration Type Documentation

5.5.3.1 anonymous enum

```
anonymous enum
```

WiFi Error Code

Enumerator

WIFI_ERROR_FW_DNLD_FAILED	The Firmware download operation failed.
WIFI_ERROR_FW_NOT_READY	The Firmware ready register not set.
WIFI_ERROR_CARD_NOT_DETECTED	The WiFi card not found.
WIFI_ERROR_FW_NOT_DETECTED	The WiFi Firmware not found.

5.5.3.2 anonymous enum

```
anonymous enum
```

WiFi driver TX/RX data status

Enumerator

WIFI_DATA_RUNNING	Data in running status
WIFI_DATA_BLOCK	Data in block status

5.6 wifi_events.h File Reference

Wi-Fi events.

5.6.1 Enumeration Type Documentation

5.6.1.1 wifi_event

```
enum wifi_event
```

Wifi events

Enumerator

WIFI_EVENT_UAP_STARTED	uAP Started
WIFI_EVENT_UAP_CLIENT_ASSOC	uAP Client Assoc
WIFI_EVENT_UAP_CLIENT_CONN	uAP Client connected
WIFI_EVENT_UAP_CLIENT_DEAUTH	uAP Client De-authentication
WIFI_EVENT_UAP_NET_ADDR_CONFIG	uAP Network Address Configuration
WIFI_EVENT_UAP_STOPPED	uAP Stopped
WIFI_EVENT_UAP_LAST	uAP Last
WIFI_EVENT_SCAN_START	Scan start event when scan is started
WIFI_EVENT_SCAN_RESULT	Scan Result

Enumerator

WIFI_EVENT_SURVEY_RESULT_GET	Survey Result Get
WIFI_EVENT_GET_HW_SPEC	Get hardware spec
WIFI_EVENT_ASSOCIATION	Association
WIFI_EVENT_ASSOCIATION_NOTIFY	Association Notify
WIFI_EVENT_PMK	PMK
WIFI_EVENT_AUTHENTICATION	Authentication
WIFI_EVENT_DISASSOCIATION	Disassociation
WIFI_EVENT_DEAUTHENTICATION	De-authentication
WIFI_EVENT_LINK_LOSS	Link Loss
WIFI_EVENT_FW_HANG	Firmware Hang event
WIFI_EVENT_FW_RESET	Firmware Reset event
WIFI_EVENT_NET_STA_ADDR_CONFIG	Network station address configuration
WIFI_EVENT_NET_INTERFACE_CONFIG	Network interface configuration
WIFI_EVENT_WEP_CONFIG	WEP configuration
WIFI_EVENT_STA_MAC_ADDR_CONFIG	STA MAC address configuration
WIFI_EVENT_UAP_MAC_ADDR_CONFIG	UAP MAC address configuration
WIFI_EVENT_NET_DHCP_CONFIG	Network DHCP configuration
WIFI_EVENT_SUPPLICANT_PMK	Supplicant PMK
WIFI_EVENT_SLEEP	Sleep
WIFI_EVENT_IEEE_PS	IEEE PS
WIFI_EVENT_DEEP_SLEEP	Deep Sleep
WIFI_EVENT_WNM_PS	WNM ps
WIFI_EVENT_IEEE_DEEP_SLEEP	IEEE and Deep Sleep
WIFI_EVENT_WNM_DEEP_SLEEP	WNM and Deep Sleep
WIFI_EVENT_PS_INVALID	PS Invalid
WIFI_EVENT_ERR_MULTICAST	Error Multicast
WIFI_EVENT_ERR_UNICAST	error Unicast
WIFI_EVENT_NLIST_REPORT	802.11K/11V neighbor report
WIFI_EVENT_11N_ADDBA	802.11N add block ack
WIFI_EVENT_11N_BA_STREAM_TIMEOUT	802.11N block Ack stream timeout
WIFI_EVENT_11N_DELBA	802.11n Delete block add
WIFI_EVENT_11N_AGGR_CTRL	802.11n aggregation control
WIFI_EVENT_CHAN_SWITCH_ANN	Channel Switch Announcement
WIFI_EVENT_CHAN_SWITCH	Channel Switch
WIFI_EVENT_NET_IPV6_CONFIG	IPv6 address state change
WIFI_EVENT_SYNC_REGION_CODE	Event to sync region code with connected AP
WIFI_EVENT_LAST	Event to indicate end of Wi-Fi events

5.6.1.2 wifi_event_reason

```
enum wifi_event_reason
```

WiFi Event Reason

Enumerator

WIFI_EVENT_REASON_SUCCESS	Success
WIFI_EVENT_REASON_TIMEOUT	Timeout
WIFI_EVENT_REASON_FAILURE	Failure

5.6.1.3 wlan_bss_type

```
enum wlan_bss_type
```

Network wireless BSS Type

Enumerator

WLAN_BSS_TYPE_STA	Station
WLAN_BSS_TYPE_UAP	uAP
WLAN_BSS_TYPE_ANY	Any

5.6.1.4 wlan_bss_role

```
enum wlan_bss_role
```

Network wireless BSS Role

Enumerator

WLAN_BSS_ROLE_STA	Infrastructure network. The system will act as a station connected to an Access Point.
WLAN_BSS_ROLE_UAP	uAP (micro-AP) network. The system will act as an uAP node to which other Wireless clients can connect.
WLAN_BSS_ROLE_ANY	Either Infrastructure network or micro-AP network

5.6.1.5 wifi_wakeup_event_t

```
enum wifi_wakeup_event_t
```

This enum defines various wakeup events for which wakeup will occur

Enumerator

WIFI_WAKE_ON_ALL_BROADCAST	Wakeup on broadcast
WIFI_WAKE_ON_UNICAST	Wakeup on unicast

Enumerator

WIFI_WAKE_ON_MAC_EVENT	Wakeup on MAC event
WIFI_WAKE_ON_MULTICAST	Wakeup on multicast
WIFI_WAKE_ON_ARP_BROADCAST	Wakeup on ARP broadcast
WIFI_WAKE_ON_MGMT_FRAME	Wakeup on receiving a management frame

5.7 wifi_ping.h File Reference

This file provides the support for network utility ping.

5.7.1 Function Documentation

5.7.1.1 ping_cli_init()

```
int ping_cli_init (
            void  )
```

Register Network Utility CLI commands.

Register the Network Utility CLI commands. Currently, only ping command is supported.

Note

This function can only be called by the application after wlan_init() called.

Returns

WM_SUCCESS if the CLI commands are registered
-WM_FAIL otherwise (for example if this function was called while the CLI commands were already registered)

5.7.1.2 ping_cli_deinit()

```
int ping_cli_deinit (
            void  )
```

Unregister Network Utility CLI commands.

Unregister the Network Utility CLI commands.

Returns

WM_SUCCESS if the CLI commands are unregistered
-WM_FAIL otherwise

5.8 wlan.h File Reference

WLAN Connection Manager.

5.8.1 Detailed Description

The WLAN Connection Manager (WLCMGR) is one of the core components that provides WiFi-level functionality like scanning for networks, starting a network (Access Point) and associating / disassociating with other wireless networks. The WLCMGR manages two logical interfaces, the station interface and the micro-AP interface. Both these interfaces can be active at the same time.

5.8.2 Usage

The WLCMGR is initialized by calling wlan_init() and started by calling wlan_start(), one of the arguments of this function is a callback handler. Many of the WLCMGR tasks are asynchronous in nature, and the events are provided by invoking the callback handler. The various usage scenarios of the WLCMGR are outlined below:

- **Scanning:** A call to wlan_scan() initiates an asynchronous scan of the nearby wireless networks. The results are reported via the callback handler.
- **Network Profiles:** Starting / stopping wireless interfaces or associating / disassociating with other wireless networks is managed through network profiles. The network profiles record details about the wireless network like the SSID, type of security, security passphrase among other things. The network profiles can be managed by means of the wlan_add_network() and wlan_remove_network() calls.
- **Association:** The wlan_connect() and wlan_disconnect() calls can be used to manage connectivity with other wireless networks (Access Points). These calls manage the station interface of the system.
- **Starting a Wireless Network:** The wlan_start_network() and wlan_stop_network() calls can be used to start/stop our own (micro-AP) network. These calls manage the micro-AP interface of the system.

5.8.3 Function Documentation

5.8.3.1 wlan_init()

```
int wlan_init (
            const uint8_t * fw_start_addr,
            const size_t size )
```

Initialize the SDIO driver and create the wifi driver thread.

Parameters

in	*fw_start_addr*	Start address of the WLAN firmware.
in	*size*	Size of the WLAN firmware.

Returns

WM_SUCCESS if the WLAN Connection Manager service has initialized successfully.
Negative value if initialization failed.

5.8.3.2 wlan_start()

```
int wlan_start (
            int(*)(enum wlan_event_reason reason, void *data) cb )
```

Start the WLAN Connection Manager service.

This function starts the WLAN Connection Manager.

Note

The status of the WLAN Connection Manager is notified asynchronously through the callback, *cb*, with a WLAN_REASON_INITIALIZED event (if initialization succeeded) or WLAN_REASON_INITIALIZATION_FAILED (if initialization failed).
If the WLAN Connection Manager fails to initialize, the caller should stop WLAN Connection Manager via wlan_stop() and try wlan_start() again.

Parameters

in	*cb*	A pointer to a callback function that handles WLAN events. All further WLCMGR events will be notified in this callback. Refer to enum wlan_event_reason for the various events for which this callback is called.

Returns

WM_SUCCESS if the WLAN Connection Manager service has started successfully.
-WM_E_INVAL if the *cb* pointer is NULL.
-WM_FAIL if an internal error occurred.
WLAN_ERROR_STATE if the WLAN Connection Manager is already running.

5.8.3.3 wlan_stop()

```
int wlan_stop (
            void )
```

Stop the WLAN Connection Manager service.

This function stops the WLAN Connection Manager, causing station interface to disconnect from the currently connected network and stop the micro-AP interface.

Returns

WM_SUCCESS if the WLAN Connection Manager service has been stopped successfully.
WLAN_ERROR_STATE if the WLAN Connection Manager was not running.

5.8.3.4 wlan_deinit()

```
void wlan_deinit (
            int action )
```

Deinitialize SDIO driver, send shutdown command to WLAN firmware and delete the wifi driver thread.

Parameters

action	Additional action to be taken with deinit WLAN_ACTIVE: no action to be taken

5.8.3.5 wlan_initialize_uap_network()

```
void wlan_initialize_uap_network (
            struct wlan_network * net )
```

WLAN initialize micro-AP network information

This API intializes a default micro-AP network. The network ssid, passphrase is initialized to NULL. Channel is set to auto. The IP Address of the micro-AP interface is 192.168.10.1/255.255.255.0. Network name is set to 'uap-network'.

Parameters

out	*net*	Pointer to the initialized micro-AP network

5.8.3.6 wlan_initialize_sta_network()

```
void wlan_initialize_sta_network (
            struct wlan_network * net )
```

WLAN initialize station network information

This API intializes a default station network. The network ssid, passphrase is initialized to NULL. Channel is set to auto.

Parameters

out	*net*	Pointer to the initialized micro-AP network

5.8.3.7 wlan_add_network()

```
int wlan_add_network (
            struct wlan_network * network )
```

Add a network profile to the list of known networks.

This function copies the contents of *network* to the list of known networks in the WLAN Connection Manager. The network's 'name' field must be unique and between WLAN_NETWORK_NAME_MIN_LENGTH and WLAN_NETWORK_NAME_MAX_LENGTH characters. The network must specify at least an SSID or BSSID. The WLAN Connection Manager may store up to WLAN_MAX_KNOWN_NETWORKS networks.

Note

Profiles for the station interface may be added only when the station interface is in the WLAN_DISCONNECTED or WLAN_CONNECTED state.
This API can be used to add profiles for station or micro-AP interfaces.

Parameters

in	*network*	A pointer to the wlan_network that will be copied to the list of known networks in the WLAN Connection Manager successfully.

Returns

WM_SUCCESS if the contents pointed to by *network* have been added to the WLAN Connection Manager.
-WM_E_INVAL if *network* is NULL or the network name is not unique or the network name length is not valid or network security is WLAN_SECURITY_WPA3_SAE but Management Frame Protection Capable is not enabled. in wlan_network_security field. if network security type is WLAN_SECURITY_WPA or WLAN_SECURITY_WPA2 or WLAN_SECURITY_WPA_WPA2_MIXED, but the passphrase length is less than 8 or greater than 63, or the psk length equal to 64 but not hexadecimal digits. if network security type is WLAN_SECURITY_WPA3_SAE, but the password length is less than 8 or greater than 255. if network security type is WLAN_SECURITY_WEP_OPEN or WLAN_SECURITY_WEP_SHARED.
-WM_E_NOMEM if there was no room to add the network.
WLAN_ERROR_STATE if the WLAN Connection Manager was running and not in the WLAN_DISCONNECTED, WLAN_ASSOCIATED or WLAN_CONNECTED state.

5.8.3.8 wlan_remove_network()

```
int wlan_remove_network (
            const char * name )
```

Remove a network profile from the list of known networks.

This function removes a network (identified by its name) from the WLAN Connection Manager, disconnecting from that network if connected.

Note

This function is asynchronous if it is called while the WLAN Connection Manager is running and connected to the network to be removed. In that case, the WLAN Connection Manager will disconnect from the network and generate an event with reason WLAN_REASON_USER_DISCONNECT. This function is synchronous otherwise.
This API can be used to remove profiles for station or micro-AP interfaces. Station network will not be removed if it is in WLAN_CONNECTED state and uAP network will not be removed if it is in WLAN_UAP_STARTED state.

Parameters

in	*name*	A pointer to the string representing the name of the network to remove.

Returns

WM_SUCCESS if the network named *name* was removed from the WLAN Connection Manager successfully. Otherwise, the network is not removed.
WLAN_ERROR_STATE if the WLAN Connection Manager was running and the station interface was not in the WLAN_DISCONNECTED state.
-WM_E_INVAL if *name* is NULL or the network was not found in the list of known networks.
-WM_FAIL if an internal error occurred while trying to disconnect from the network specified for removal.

5.8.3.9 wlan_connect()

```
int wlan_connect (
            char * name )
```

Connect to a wireless network (Access Point).

When this function is called, WLAN Connection Manager starts connection attempts to the network specified by *name*. The connection result will be notified asynchronously to the WLCMGR callback when the connection process has completed.

When connecting to a network, the event refers to the connection attempt to that network.

Calling this function when the station interface is in the WLAN_DISCONNECTED state will, if successful, cause the interface to transition into the WLAN_CONNECTING state. If the connection attempt succeeds, the station interface will transition to the WLAN_CONNECTED state, otherwise it will return to the WLAN_DISCONNECTED state. If this function is called while the station interface is in the WLAN_CONNECTING or WLAN_CONNECTED state, the WLAN Connection Manager will first cancel its connection attempt or disconnect from the network, respectively, and generate an event with reason WLAN_REASON_USER_DISCONNECT. This will be followed by a second event that reports the result of the new connection attempt.

If the connection attempt was successful the WLCMGR callback is notified with the event WLAN_REASON_SUCCESS, while if the connection attempt fails then either of the events, WLAN_REASON_NETWORK_NOT_FOUND, WLAN_REASON_NETWORK_AUTH_FAILED, WLAN_REASON_CONNECT_FAILED or WLAN_REASON_ADDRESS_FAILED are reported as appropriate.

Parameters

in	*name*	A pointer to a string representing the name of the network to connect to.

Returns

WM_SUCCESS if a connection attempt was started successfully
WLAN_ERROR_STATE if the WLAN Connection Manager was not running.
-WM_E_INVAL if there are no known networks to connect to or the network specified by *name* is not in the list of known networks or network *name* is NULL.
-WM_FAIL if an internal error has occurred.

5.8.3.10 wlan_connect_opt()

```
int wlan_connect_opt (
            char * name,
            bool skip_dfs )
```

Connect to a wireless network (Access Point) with options.

When this function is called, WLAN Connection Manager starts connection attempts to the network specified by *name*. The connection result will be notified asynchronously to the WLCMGR callback when the connection process has completed.

When connecting to a network, the event refers to the connection attempt to that network.

Calling this function when the station interface is in the WLAN_DISCONNECTED state will, if successful, cause the interface to transition into the WLAN_CONNECTING state. If the connection attempt succeeds, the station interface will transition to the WLAN_CONNECTED state, otherwise it will return to the WLAN_DISCONNECTED state. If this function is called while the station interface is in the WLAN_CONNECTING or WLAN_CONNECTED state, the WLAN Connection Manager will first cancel its connection attempt or disconnect from the network, respectively, and generate an event with reason WLAN_REASON_USER_DISCONNECT. This will be followed by a second event that reports the result of the new connection attempt.

If the connection attempt was successful the WLCMGR callback is notified with the event WLAN_REASON_SUCCESS, while if the connection attempt fails then either of the events, WLAN_REASON_NETWORK_NOT_FOUND, WLAN_REASON_NETWORK_AUTH_FAILED, WLAN_REASON_CONNECT_FAILED or WLAN_REASON_ADDRESS_FAILED are reported as appropriate.

Parameters

in	*name*	A pointer to a string representing the name of the network to connect to.
in	*skip_dfs*	Option to skip DFS channel when doing scan.

Returns

WM_SUCCESS if a connection attempt was started successfully
WLAN_ERROR_STATE if the WLAN Connection Manager was not running.
-WM_E_INVAL if there are no known networks to connect to or the network specified by *name* is not in the list of known networks or network *name* is NULL.
-WM_FAIL if an internal error has occurred.

5.8.3.11 wlan_reassociate()

```
int wlan_reassociate ( )
```

Reassociate to a wireless network (Access Point).

When this function is called, WLAN Connection Manager starts reassociation attempts using same SSID as currently connected network . The connection result will be notified asynchronously to the WLCMGR callback when the connection process has completed.

When connecting to a network, the event refers to the connection attempt to that network.

Calling this function when the station interface is in the WLAN_DISCONNECTED state will have no effect.

Calling this function when the station interface is in the WLAN_CONNECTED state will, if successful, cause the interface to reassociate to another network(AP).

If the connection attempt was successful the WLCMGR callback is notified with the event WLAN_REASON_SUCCESS, while if the connection attempt fails then either of the events, WLAN_REASON_NETWORK_AUTH_FAILED, WLAN_REASON_CONNECT_FAILED or WLAN_REASON_ADDRESS_FAILED are reported as appropriate.

Returns

WM_SUCCESS if a reassociation attempt was started successfully
WLAN_ERROR_STATE if the WLAN Connection Manager was not running. or WLAN Connection Manager was not in WLAN_CONNECTED state.
-WM_E_INVAL if there are no known networks to connect to
-WM_FAIL if an internal error has occurred.

5.8.3.12 wlan_disconnect()

```
int wlan_disconnect (
            void )
```

Disconnect from the current wireless network (Access Point).

When this function is called, the WLAN Connection Manager attempts to disconnect the station interface from its currently connected network (or cancel an in-progress connection attempt) and return to the WLAN_DISCONNECTED state. Calling this function has no effect if the station interface is already disconnected.

Note

This is an asynchronous function and successful disconnection will be notified using the WLAN_REASON_USER_DISCONNECT.

Returns

WM_SUCCESS if successful
WLAN_ERROR_STATE otherwise

5.8.3.13 wlan_start_network()

```
int wlan_start_network (
            const char * name )
```

Start a wireless network (Access Point).

When this function is called, the WLAN Connection Manager starts the network specified by *name*. The network with the specified *name* must be first added using wlan_add_network and must be a micro-AP network with a valid SSID.

Note

The WLCMGR callback is asynchronously notified of the status. On success, the event WLAN_REASON_UAP_SUCCESS is reported, while on failure, the event WLAN_REASON_UAP_START_FAILED is reported.

Parameters

in	*name*	A pointer to string representing the name of the network to connect to.

Returns

WM_SUCCESS if successful.
WLAN_ERROR_STATE if in power save state or uAP already running.
-WM_E_INVAL if *name* was NULL or the network *name* was not found or it not have a specified SSID.

5.8.3.14 wlan_stop_network()

```
int wlan_stop_network (
            const char * name )
```

Stop a wireless network (Access Point).

When this function is called, the WLAN Connection Manager stops the network specified by *name*. The specified network must be a valid micro-AP network that has already been started.

Note

The WLCMGR callback is asynchronously notified of the status. On success, the event WLAN_REASON_UAP_STOPPED is reported, while on failure, the event WLAN_REASON_UAP_STOP_FAILED is reported.

Parameters

in	*name*	A pointer to a string representing the name of the network to stop.

Returns

WM_SUCCESS if successful.
WLAN_ERROR_STATE if uAP is in power save state.
-WM_E_INVAL if *name* was NULL or the network *name* was not found or that the network *name* is not a micro-AP network or it is a micro-AP network but does not have a specified SSID.

5.8.3.15 wlan_get_mac_address()

```
int wlan_get_mac_address (
            uint8_t * dest )
```

Retrieve the wireless MAC address of station interface.

This function copies the MAC address of the station interface to sta mac address and uAP interface to uap mac address.

Parameters

out	*dest*	A pointer to a 6-byte array where the MAC address will be copied.

Returns

WM_SUCCESS if the MAC address was copied.
-WM_E_INVAL if *sta_mac* or uap_mac is NULL.

5.8.3.16 wlan_get_mac_address_uap()

```
int wlan_get_mac_address_uap (
            uint8_t * dest )
```

Retrieve the wireless MAC address of micro-AP interface.

This function copies the MAC address of the wireless interface to the 6-byte array pointed to by *dest*. In the event of an error, nothing is copied to *dest*.

Parameters

out	*dest*	A pointer to a 6-byte array where the MAC address will be copied.

Returns

WM_SUCCESS if the MAC address was copied.
-WM_E_INVAL if *dest* is NULL.

5.8.3.17 wlan_get_address()

```
int wlan_get_address (
            struct wlan_ip_config * addr )
```

Retrieve the IP address configuration of the station interface.

This function retrieves the IP address configuration of the station interface and copies it to the memory location pointed to by *addr*.

Note

This function may only be called when the station interface is in the WLAN_CONNECTED state.

Parameters

out	*addr*	A pointer to the wlan_ip_config.

Returns

WM_SUCCESS if successful.
-WM_E_INVAL if *addr* is NULL.
WLAN_ERROR_STATE if the WLAN Connection Manager was not running or was not in the WLAN_CONNECTED state.
-WM_FAIL if an internal error occurred when retrieving IP address information from the TCP stack.

5.8.3.18 wlan_get_uap_address()

```
int wlan_get_uap_address (
            struct wlan_ip_config * addr )
```

Retrieve the IP address of micro-AP interface.

This function retrieves the current IP address configuration of micro-AP and copies it to the memory location pointed to by *addr*.

Note

This function may only be called when the micro-AP interface is in the WLAN_UAP_STARTED state.

Parameters

out	*addr*	A pointer to the wlan_ip_config.

Returns

WM_SUCCESS if successful.
-WM_E_INVAL if *addr* is NULL.
WLAN_ERROR_STATE if the WLAN Connection Manager was not running or the micro-AP interface was not in the WLAN_UAP_STARTED state.
-WM_FAIL if an internal error occurred when retrieving IP address information from the TCP stack.

5.8.3.19 wlan_get_uap_channel()

```
int wlan_get_uap_channel (
            int * channel )
```

Retrieve the channel of micro-AP interface.

This function retrieves the channel number of micro-AP and copies it to the memory location pointed to by *channel*.

Note

This function may only be called when the micro-AP interface is in the WLAN_UAP_STARTED state.

Parameters

out	*channel*	A pointer to variable that stores channel number.

Returns

WM_SUCCESS if successful.
-WM_E_INVAL if *channel* is NULL.
-WM_FAIL if an internal error has occurred.

5.8.3.20 wlan_get_current_network()

```
int wlan_get_current_network (
            struct wlan_network * network )
```

Retrieve the current network configuration of station interface.

This function retrieves the current network configuration of station interface when the station interface is in the WLAN_CONNECTED state.

Parameters

out	*network*	A pointer to the wlan_network.

Returns

WM_SUCCESS if successful.
-WM_E_INVAL if *network* is NULL.
WLAN_ERROR_STATE if the WLAN Connection Manager was not running or not in the WLAN_CONNECTED state.

5.8.3.21 wlan_get_current_network_ssid()

```
int wlan_get_current_network_ssid (
            char * ssid )
```

Retrieve the current network ssid of station interface.

This function retrieves the current network ssid of station interface when the station interface is in the WLAN_CONNECTED state.

Parameters

out	*ssid*	A pointer to the ssid.

Returns

WM_SUCCESS if successful.
-WM_E_INVAL if *network* is NULL.
WLAN_ERROR_STATE if the WLAN Connection Manager was not running or not in the WLAN_CONNECTED state.

5.8.3.22 wlan_get_current_uap_network()

```
int wlan_get_current_uap_network (
            struct wlan_network * network )
```

Retrieve the current network configuration of micro-AP interface.

This function retrieves the current network configuration of micro-AP interface when the micro-AP interface is in the WLAN_UAP_STARTED state.

Parameters

out	*network*	A pointer to the wlan_network.

Returns

WM_SUCCESS if successful.
-WM_E_INVAL if *network* is NULL.
WLAN_ERROR_STATE if the WLAN Connection Manager was not running or not in the WLAN_UAP_STARTED state.

5.8.3.23 wlan_get_current_uap_network_ssid()

```
int wlan_get_current_uap_network_ssid (
            char * ssid )
```

Retrieve the current network ssid of micro-AP interface.

This function retrieves the current network ssid of micro-AP interface when the micro-AP interface is in the WLAN_UAP_STARTED state.

Parameters

out	*ssid*	A pointer to the ssid.

Returns

WM_SUCCESS if successful.
-WM_E_INVAL if *network* is NULL.
WLAN_ERROR_STATE if the WLAN Connection Manager was not running or not in the WLAN_UAP_STARTED state.

5.8.3.24 is_uap_started()

```
bool is_uap_started (
            void )
```

Retrieve the status information of the micro-AP interface.

Returns

TRUE if micro-AP interface is in WLAN_UAP_STARTED state.
FALSE otherwise.

5.8.3.25 is_sta_connected()

```
bool is_sta_connected (
            void )
```

Retrieve the status information of the station interface.

Returns

TRUE if station interface is in WLAN_CONNECTED state.
FALSE otherwise.

5.8.3.26 is_sta_ipv4_connected()

```
bool is_sta_ipv4_connected (
            void )
```

Retrieve the status information of the ipv4 network of station interface.

Returns

TRUE if ipv4 network of station interface is in WLAN_CONNECTED state.
FALSE otherwise.

5.8.3.27 is_sta_ipv6_connected()

```
bool is_sta_ipv6_connected (
            void )
```

Retrieve the status information of the ipv6 network of station interface.

Returns

TRUE if ipv6 network of station interface is in WLAN_CONNECTED state.
FALSE otherwise.

5.8.3.28 wlan_get_network()

```
int wlan_get_network (
            unsigned int index,
            struct wlan_network * network )
```

Retrieve the information about a known network using *index*.

This function retrieves the contents of a network at *index* in the list of known networks maintained by the WLAN Connection Manager and copies it to the location pointed to by *network*.

Note

wlan_get_network_count() may be used to retrieve the number of known networks. wlan_get_network() may be used to retrieve information about networks at *index* 0 to one minus the number of networks.
This function may be called regardless of whether the WLAN Connection Manager is running. Calls to this function are synchronous.

Parameters

in	*index*	The index of the network to retrieve.
out	*network*	A pointer to the wlan_network where the network configuration for the network at *index* will be copied.

Returns

WM_SUCCESS if successful.
-WM_E_INVAL if *network* is NULL or *index* is out of range.

5.8.3.29 wlan_get_network_byname()

```
int wlan_get_network_byname (
            char * name,
            struct wlan_network * network )
```

Retrieve information about a known network using *name*.

This function retrieves the contents of a named network in the list of known networks maintained by the WLAN Connection Manager and copies it to the location pointed to by *network*.

Note

This function may be called regardless of whether the WLAN Connection Manager is running. Calls to this function are synchronous.

Parameters

in	*name*	The name of the network to retrieve.
out	*network*	A pointer to the wlan_network where the network configuration for the network having name as *name* will be copied.

Returns

WM_SUCCESS if successful.
-WM_E_INVAL if *network* is NULL or *name* is NULL.

5.8.3.30 wlan_get_network_count()

```
int wlan_get_network_count (
            unsigned int * count )
```

Retrieve the number of networks known to the WLAN Connection Manager.

This function retrieves the number of known networks in the list maintained by the WLAN Connection Manager and copies it to *count*.

Note

This function may be called regardless of whether the WLAN Connection Manager is running. Calls to this function are synchronous.

Parameters

out	*count*	A pointer to the memory location where the number of networks will be copied.

Returns

WM_SUCCESS if successful.
-WM_E_INVAL if *count* is NULL.

5.8.3.31 wlan_get_connection_state()

```
int wlan_get_connection_state (
            enum wlan_connection_state * state )
```

Retrieve the connection state of station interface.

This function retrieves the connection state of station interface, which is one of WLAN_DISCONNECTED, WLAN_CONNECTING, WLAN_ASSOCIATED or WLAN_CONNECTED.

Parameters

out	*state*	A pointer to the wlan_connection_state where the current connection state will be copied.

Returns

WM_SUCCESS if successful.
-WM_E_INVAL if *state* is NULL
WLAN_ERROR_STATE if the WLAN Connection Manager was not running.

5.8.3.32 wlan_get_uap_connection_state()

```
int wlan_get_uap_connection_state (
            enum wlan_connection_state * state )
```

Retrieve the connection state of micro-AP interface.

This function retrieves the connection state of micro-AP interface, which is one of WLAN_UAP_STARTED, or WLAN_UAP_STOPPED.

Parameters

out	*state*	A pointer to the wlan_connection_state where the current connection state will be copied.

Returns

WM_SUCCESS if successful.
-WM_E_INVAL if *state* is NULL
WLAN_ERROR_STATE if the WLAN Connection Manager was not running.

5.8.3.33 wlan_scan()

```
int wlan_scan (
            int(*)(unsigned int count) cb )
```

Scan for wireless networks.

When this function is called, the WLAN Connection Manager starts scan for wireless networks. On completion of the scan the WLAN Connection Manager will call the specified callback function *cb*. The callback function can then retrieve the scan results by using the wlan_get_scan_result() function.

Note

This function may only be called when the station interface is in the WLAN_DISCONNECTED or WLAN_CONNECTED state. Scanning is disabled in the WLAN_CONNECTING state.
This function will block until it can issue a scan request if called while another scan is in progress.

Parameters

in	*cb*	A pointer to the function that will be called to handle scan results when they are available.

Returns

WM_SUCCESS if successful.
-WM_E_NOMEM if failed to allocated memory for wlan_scan_params_v2_t structure.
-WM_E_INVAL if *cb* scan result callack functio pointer is NULL.
WLAN_ERROR_STATE if the WLAN Connection Manager was not running or not in the WLAN_DISCONNECTED or WLAN_CONNECTED states.
-WM_FAIL if an internal error has occurred and the system is unable to scan.

5.8.3.34 wlan_scan_with_opt()

```
int wlan_scan_with_opt (
            wlan_scan_params_v2_t t_wlan_scan_param )
```

Scan for wireless networks using options provided.

When this function is called, the WLAN Connection Manager starts scan for wireless networks. On completion of the scan the WLAN Connection Manager will call the specified callback function *cb*. The callback function can then retrieve the scan results by using the wlan_get_scan_result() function.

Note

This function may only be called when the station interface is in the WLAN_DISCONNECTED or WLAN_CONNECTED state. Scanning is disabled in the WLAN_CONNECTING state.
This function will block until it can issue a scan request if called while another scan is in progress.

Parameters

in	*t_wlan_scan_param*	A wlan_scan_params_v2_t structure holding a pointer to function that will be called to handle scan results when they are available, SSID of a wireless network, BSSID of a wireless network, number of channels with scan type information and number of probes.

Returns

WM_SUCCESS if successful.
-WM_E_NOMEM if failed to allocated memory for wlan_scan_params_v2_t structure.
-WM_E_INVAL if *cb* scan result callack function pointer is NULL.
WLAN_ERROR_STATE if the WLAN Connection Manager was not running or not in the WLAN_DISCONNECTED or WLAN_CONNECTED states.

-WM_FAIL if an internal error has occurred and the system is unable to scan.

5.8.3.35 wlan_get_scan_result()

```
int wlan_get_scan_result (
            unsigned int index,
            struct wlan_scan_result * res )
```

Retrieve a scan result.

This function may be called to retrieve scan results when the WLAN Connection Manager has finished scanning. It must be called from within the scan result callback (see wlan_scan()) as scan results are valid only in that context. The callback argument 'count' provides the number of scan results that may be retrieved and wlan_get_scan_result() may be used to retrieve scan results at *index* 0 through that number.

Note

This function may only be called in the context of the scan results callback.
Calls to this function are synchronous.

Parameters

in	*index*	The scan result to retrieve.
out	*res*	A pointer to the wlan_scan_result where the scan result information will be copied.

Returns

WM_SUCCESS if successful.
-WM_E_INVAL if *res* is NULL
WLAN_ERROR_STATE if the WLAN Connection Manager was not running
-WM_FAIL if the scan result at *index* could not be retrieved (that is, *index* is out of range).

5.8.3.36 wlan_enable_low_pwr_mode()

```
int wlan_enable_low_pwr_mode ( )
```

Enable Low Power Mode in Wireless Firmware.

Note

When low power mode is enabled, the output power will be clipped at ∼+10dBm and the expected PA current is expected to be in the 80-90 mA range for b/g/n modes.

This function may be called to enable low power mode in firmware. This should be called before wlan_init() function.

Returns

WM_SUCCESS if the call was successful.
-WM_FAIL if failed.

5.8.3.37 wlan_set_ed_mac_mode()

```
int wlan_set_ed_mac_mode (
            wlan_ed_mac_ctrl_t wlan_ed_mac_ctrl )
```

Configure ED MAC mode for Station in Wireless Firmware.

Note

When ed mac mode is enabled, Wireless Firmware will behave following way:

when background noise had reached -70dB or above, WiFi chipset/module should hold data transmitting until condition is removed. It is applicable for both 5GHz and 2.4GHz bands.

Parameters

in	*wlan_ed_mac_ctrl*	Struct with following parameters ed_ctrl_2g 0 - disable EU adaptivity for 2.4GHz band 1 - enable EU adaptivity for 2.4GHz band

ed_offset_2g 0 - Default Energy Detect threshold (Default: 0x9) offset value range: 0x80 to 0x7F

Note

If 5GH enabled then add following parameters

```
ed_ctrl_5g          0  - disable EU adaptivity for 5GHz band
                    1  - enable EU adaptivity for 5GHz band

ed_offset_5g        0  - Default Energy Detect threshold(Default: 0xC)
                    offset value range: 0x80 to 0x7F
```

Returns

WM_SUCCESS if the call was successful.
-WM_FAIL if failed.

5.8.3.38 wlan_set_uap_ed_mac_mode()

```
int wlan_set_uap_ed_mac_mode (
            wlan_ed_mac_ctrl_t wlan_ed_mac_ctrl )
```

Configure ED MAC mode for Micro AP in Wireless Firmware.

Note

When ed mac mode is enabled, Wireless Firmware will behave following way:

when background noise had reached -70dB or above, WiFi chipset/module should hold data transmitting until condition is removed. It is applicable for both 5GHz and 2.4GHz bands.

Parameters

in	*wlan_ed_mac_ctrl*	Struct with following parameters ed_ctrl_2g 0 - disable EU adaptivity for 2.4GHz band 1 - enable EU adaptivity for 2.4GHz band

ed_offset_2g 0 - Default Energy Detect threshold (Default: 0x9) offset value range: 0x80 to 0x7F

Note

If 5GH enabled then add following parameters

```
ed_ctrl_5g          0  - disable EU adaptivity for 5GHz band
                    1  - enable EU adaptivity for 5GHz band

ed_offset_5g        0  - Default Energy Detect threshold(Default: 0xC)
                    offset value range: 0x80 to 0x7F
```

Returns

WM_SUCCESS if the call was successful.
-WM_FAIL if failed.

5.8.3.39 wlan_get_ed_mac_mode()

```
int wlan_get_ed_mac_mode (
            wlan_ed_mac_ctrl_t * wlan_ed_mac_ctrl )
```

This API can be used to get current ED MAC MODE configuration for Station.

Parameters

out	*wlan_ed_mac_ctrl*	A pointer to wlan_ed_mac_ctrl_t with parameters mentioned in above set API.

Returns

WM_SUCCESS if the call was successful.
-WM_FAIL if failed.

5.8.3.40 wlan_get_uap_ed_mac_mode()

```
int wlan_get_uap_ed_mac_mode (
            wlan_ed_mac_ctrl_t * wlan_ed_mac_ctrl )
```

This API can be used to get current ED MAC MODE configuration for Micro AP.

Parameters

out	*wlan_ed_mac_ctrl*	A pointer to wlan_ed_mac_ctrl_t with parameters mentioned in above set API.

Returns

WM_SUCCESS if the call was successful.
-WM_FAIL if failed.

5.8.3.41 wlan_set_cal_data()

```
void wlan_set_cal_data (
            const uint8_t * cal_data,
            const unsigned int cal_data_size )
```

Set wireless calibration data in WLAN firmware.

This function may be called to set wireless calibration data in firmware. This should be call before wlan_init() function.

Parameters

in	*cal_data*	The calibration data buffer
in	*cal_data_size*	Size of calibration data buffer.

5.8.3.42 wlan_set_mac_addr()

```
void wlan_set_mac_addr (
            uint8_t * mac )
```

Set wireless MAC Address in WLAN firmware.

This function may be called to set wireless MAC Address in firmware. This should be call before wlan_init() function. When called after wlan init done, the incoming mac is treated as the sta mac address directly. And mac[4] plus 1 the modifed mac as the UAP mac address.

Parameters

in	*mac*	The MAC Address in 6 byte array format like uint8_t mac[] = { 0x00, 0x50, 0x43, 0x21, 0x19, 0x6E};

5.8.3.43 wlan_set_roaming()

```
int wlan_set_roaming (
```

```
            const int enable,
            const uint8_t rssi_low_threshold )
```

Set soft roaming config.

This function may be called to enable/disable soft roaming by specifying the RSSI threshold.

Note

RSSI Threshold setting for soft roaming: The provided RSSI low threshold value is used to subscribe RSSI low event from firmware, on reception of this event background scan is started in firmware with same RSSI threshold to find out APs with better signal strength than RSSI threshold.

If AP is found then roam attempt is initiated, otherwise background scan started again till limit reaches to BG_SCAN_LIMIT.

If still AP is not found then WLAN connection manager sends WLAN_REASON_BGSCAN_NETWORK_NOT_FOUND event to application. In this case, if application again wants to use soft roaming then it can call this API again or use wlan_set_rssi_low_threshold API to set RSSI low threshold again.

Parameters

in	*enable*	Enable/disable roaming.
in	*rssi_low_threshold*	RSSI low threshold value

Returns

WM_SUCCESS if the call was successful.
-WM_FAIL if failed.

5.8.3.44 wlan_set_ieeeps_cfg()

```
int wlan_set_ieeeps_cfg (
            struct wlan_ieeeps_config * ps_cfg )
```

Set configuration parameters of IEEE power save mode.

Parameters

in	*ps_cfg*	: powersave configuratiuon includes multiple parameters.

Returns

WM_SUCCESS if the call was successful.
-WM_FAIL if failed.

5.8.3.45 wlan_configure_listen_interval()

```
void wlan_configure_listen_interval (
            int listen_interval )
```

Configure Listen interval of IEEE power save mode.

Note

Delivery Traffic Indication Message (DTIM): It is a concept in 802.11 It is a time duration after which AP will send out buffered BROADCAST / MULTICAST data and stations connected to the AP should wakeup to take this broadcast / multicast data.
Traffic Indication Map (TIM): It is a bitmap which the AP sends with each beacon. The bitmap has one bit each for a station connected to AP.
Each station is recognized by an Association Id (AID). If AID is say 1 bit number 1 is set in the bitmap if unicast data is present with AP in its buffer for station with AID = 1 Ideally AP does not buffer any unicast data it just sends unicast data to the station on every beacon when station is not sleeping.
When broadcast data / multicast data is to be send AP sets bit 0 of TIM indicating broadcast / multicast.
The occurrence of DTIM is defined by AP.
Each beacon has a number indicating period at which DTIM occurs.
The number is expressed in terms of number of beacons.
This period is called DTIM Period / DTIM interval.
For example:
If AP has DTIM period = 3 the stations connected to AP have to wake up (if they are sleeping) to receive broadcast /multicast data on every third beacon.
Generic:
When DTIM period is X AP buffers broadcast data / multicast data for X beacons. Then it transmits the data no matter whether station is awake or not.
Listen interval:
This is time interval on station side which indicates when station will be awake to listen i.e. accept data.
Long listen interval:
It comes into picture when station sleeps (IEEEPS) and it does not want to wake up on every DTIM So station is not worried about broadcast data/multicast data in this case.
This should be a design decision what should be chosen Firmware suggests values which are about 3 times DTIM at the max to gain optimal usage and reliability.
In the IEEEPS power save mode, the WiFi firmware goes to sleep and periodically wakes up to check if the AP has any pending packets for it. A longer listen interval implies that the WiFi card stays in power save for a longer duration at the cost of additional delays while receiving data. Please note that choosing incorrect value for listen interval will causes poor response from device during data transfer. Actual listen interval selected by firmware is equal to closest DTIM.
For e.g.:-
AP beacon period : 100 ms
AP DTIM period : 2
Application request value: 500ms
Actual listen interval = 400ms (This is the closest DTIM). Actual listen interval set will be a multiple of DTIM closest to but lower than the value provided by the application.

This API can be called before/after association. The configured listen interval will be used in subsequent association attempt.

Parameters

in	*listen_interval*	Listen interval as below 0 : Unchanged, -1 : Disable, 1-49: Value in beacon intervals, >= 50: Value in TUs

5.8.3.46 wlan_configure_null_pkt_interval()

```
void wlan_configure_null_pkt_interval (
            int time_in_secs )
```

Configure Null packet interval of IEEE power save mode.

Note

In IEEEPS station sends a NULL packet to AP to indicate that the station is alive and AP should not kick it off. If null packet is not send some APs may disconnect station which might lead to a loss of connectivity. The time is specified in seconds. Default value is 30 seconds.
This API should be called before configuring IEEEPS

Parameters

in	*time_in_secs*	: -1 Disables null packet transmission, 0 Null packet interval is unchanged, n Null packet interval in seconds.

5.8.3.47 wlan_set_antcfg()

```
int wlan_set_antcfg (
            uint32_t ant,
            uint16_t evaluate_time )
```

This API can be used to set the mode of Tx/Rx antenna. If SAD is enabled, this API can also used to set SAD antenna evaluate time interval(antenna mode must be antenna diversity when set SAD evaluate time interval).

Parameters

in	*ant*	Antenna valid values are 1, 2 and 65535 1 : Tx/Rx antenna 1 2 : Tx/Rx antenna 2 0xFFFF: Tx/Rx antenna diversity
in	*evaluate_time*	SAD evaluate time interval, default value is 6s(0x1770).

Returns

WM_SUCCESS if successful.
WLAN_ERROR_STATE if unsuccessful.

5.8.3.48 wlan_get_antcfg()

```
int wlan_get_antcfg (
            uint32_t * ant,
            uint16_t * evaluate_time,
            uint16_t * current_antenna )
```

This API can be used to get the mode of Tx/Rx antenna. If SAD is enabled, this API can also used to get SAD antenna evaluate time interval(antenna mode must be antenna diversity when set SAD evaluate time interval).

Parameters

out	*ant*	pointer to antenna variable.
out	*evaluate_time*	pointer to evaluate_time variable for SAD.
out	*current_antenna*	pointer to current antenna.

Returns

WM_SUCCESS if successful.
WLAN_ERROR_STATE if unsuccessful.

5.8.3.49 wlan_get_firmware_version_ext()

```
char* wlan_get_firmware_version_ext (
            void )
```

Get the wifi firmware version extension string.

Note

This API does not allocate memory for pointer. It just returns pointer of WLCMGR internal static buffer. So no need to free the pointer by caller.

Returns

wifi firmware version extension string pointer stored in WLCMGR

5.8.3.50 wlan_version_extended()

```
void wlan_version_extended (
            void  )
```

Use this API to print wlan driver and firmware extended version.

5.8.3.51 wlan_get_tsf()

```
int wlan_get_tsf (
            uint32_t * tsf_high,
            uint32_t * tsf_low )
```

Use this API to get the TSF from Wi-Fi firmware.

Parameters

in	*tsf_high*	Pointer to store TSF higher 32bits.
in	*tsf_low*	Pointer to store TSF lower 32bits.

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

5.8.3.52 wlan_ieeeps_on()

```
int wlan_ieeeps_on (
            unsigned int wakeup_conditions )
```

Enable IEEEPS with Host Sleep Configuration

When enabled, it opportunistically puts the wireless card into IEEEPS mode. Before putting the Wireless card in power save this also sets the hostsleep configuration on the card as specified. This makes the card generate a wakeup for the processor if any of the wakeup conditions are met.

Parameters

in	*wakeup_conditions*	conditions to wake the host. This should be a logical OR of the conditions in wlan_wakeup_event_t. Typically devices would want to wake up on WAKE_ON_ALL_BROADCAST, WAKE_ON_UNICAST, WAKE_ON_MAC_EVENT. WAKE_ON_MULTICAST, WAKE_ON_ARP_BROADCAST, WAKE_ON_MGMT_FRAME

Returns

WM_SUCCESS if the call was successful.
-WM_FAIL otherwise.

5.8.3.53 wlan_ieeeps_off()

```
int wlan_ieeeps_off (
            void )
```

Turn off IEEE Power Save mode.

Note

This call is asynchronous. The system will exit the power-save mode only when all requisite conditions are met.

Returns

WM_SUCCESS if the call was successful.
-WM_FAIL otherwise.

5.8.3.54 wlan_deepsleepps_on()

```
int wlan_deepsleepps_on (
            void )
```

Turn on Deep Sleep Power Save mode.

Note

This call is asynchronous. The system will enter the power-save mode only when all requisite conditions are met. For example, wlan should be disconnected for this to work.

Returns

WM_SUCCESS if the call was successful.
-WM_FAIL otherwise.

5.8.3.55 wlan_deepsleepps_off()

```
int wlan_deepsleepps_off (
            void )
```

Turn off Deep Sleep Power Save mode.

Note

This call is asynchronous. The system will exit the power-save mode only when all requisite conditions are met.

Returns

WM_SUCCESS if the call was successful.
-WM_FAIL otherwise.

5.8.3.56 wlan_tcp_keep_alive()

```
int wlan_tcp_keep_alive (
            wlan_tcp_keep_alive_t * keep_alive )
```

Use this API to configure the TCP Keep alive parameters in Wi-Fi firmware. wlan_tcp_keep_alive_t provides the parameters which are available for configuration.

Note

To reset current TCP Keep alive configuration just pass the reset with value 1, all other parameters are ignored in this case.
Please note that this API must be called after successful connection and before putting Wi-Fi card in IEEE power save mode.

Parameters

in	*keep_alive*	A pointer to wlan_tcp_keep_alive_t with following parameters. enable Enable keep alive reset Reset keep alive timeout Keep alive timeout interval Keep alive interval max_keep_alives Maximum keep alives dst_mac Destination MAC address dst_ip Destination IP dst_tcp_port Destination TCP port src_tcp_port Source TCP port seq_no Sequence number

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

5.8.3.57 wlan_get_beacon_period()

```
uint16_t wlan_get_beacon_period (
            void )
```

Use this API to get the beacon period of associated BSS.

Returns

beacon_period if operation is successful.
0 if command fails.

5.8.3.58 wlan_get_dtim_period()

```
uint8_t wlan_get_dtim_period (
            void )
```

Use this API to get the dtim period of associated BSS.

Returns

dtim_period if operation is successful.
0 if DTIM IE Is not found in AP's Probe response.

Note

This API should not be called from WLAN event handler registered by application during wlan_start.

5.8.3.59 wlan_get_data_rate()

```
int wlan_get_data_rate (
            wlan_ds_rate * ds_rate,
            mlan_bss_type bss_type )
```

Use this API to get the current tx and rx rates along with bandwidth and guard interval information if rate is 11N.

Parameters

in	*ds_rate*	A pointer to structure which will have tx, rx rate information along with bandwidth and guard interval information.
in	*bss_type*	0: STA, 1: uAP

Note

If rate is greater than 11 then it is 11N rate and from 12 MCS0 rate starts. The bandwidth mapping is like value 0 is for 20MHz, 1 is 40MHz, 2 is for 80MHz. The guard interval value zero means Long otherwise Short.

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

5.8.3.60 wlan_get_pmfcfg()

```
int wlan_get_pmfcfg (
            uint8_t * mfpc,
            uint8_t * mfpr )
```

Use this API to get the set management frame protection parameters for sta.

Parameters

out	*mfpc*	Management Frame Protection Capable (MFPC) 1: Management Frame Protection Capable 0: Management Frame Protection not Capable
out	*mfpr*	Management Frame Protection Required (MFPR) 1: Management Frame Protection Required 0: Management Frame Protection Optional

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

5.8.3.61 wlan_uap_get_pmfcfg()

```
int wlan_uap_get_pmfcfg (
            uint8_t * mfpc,
            uint8_t * mfpr )
```

Use this API to get the set management frame protection parameters for Uap.

Parameters

out	*mfpc*	Management Frame Protection Capable (MFPC) 1: Management Frame Protection Capable 0: Management Frame Protection not Capable
out	*mfpr*	Management Frame Protection Required (MFPR) 1: Management Frame Protection Required 0: Management Frame Protection Optional

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

5.8.3.62 wlan_set_packet_filters()

```
int wlan_set_packet_filters (
            wlan_flt_cfg_t * flt_cfg )
```

Use this API to set packet filters in Wi-Fi firmware.

Parameters

Parameters

in	*flt_cfg*	A pointer to structure which holds the the packet filters in same way as given below.
		MEF Configuration command
		mefcfg={
		Criteria: bit0-broadcast, bit1-unicast, bit3-multicast
		Criteria=2 Unicast frames are received during hostsleepmode
		NumEntries=1 Number of activated MEF entries
		mef_entry_0: example filters to match TCP destination port 80 send by 192.168.0.88 pkt or magic pkt.
		mef_entry_0={
		mode: bit0–hostsleep mode, bit1–non hostsleep mode
		mode=1 HostSleep mode
		action: 0–discard and not wake host, 1–discard and wake host 3–allow and wake host
		action=3 Allow and Wake host
		filter_num=3 Number of filter
		RPN only support "&&" and "||" operator,space can not be removed between operator.
		RPN=Filter_0 && Filter_1 || Filter_2
		Byte comparison filter's type is 0x41,Decimal comparison filter's type is 0x42,
		Bit comparison filter's type is 0x43
		Filter_0 is decimal comparison filter, it always with type=0x42
		Decimal filter always has type, pattern, offset, numbyte 4 field
		Filter_0 will match rx pkt with TCP destination port 80
		Filter_0={
		type=0x42 decimal comparison filter
		pattern=80 80 is the decimal constant to be compared
		offset=44 44 is the byte offset of the field in RX pkt to be compare
		numbyte=2 2 is the number of bytes of the field
		}
		Filter_1 is Byte comparison filter, it always with type=0x41
		Byte filter always has type, byte, repeat, offset 4 filed
		Filter_1 will match rx pkt send by IP address 192.168.0.88
		Filter_1={
		type=0x41 Byte comparison filter
		repeat=1 1 copies of 'c0:a8:00:58'
		byte=c0:a8:00:58 'c0:a8:00:58' is the byte sequence constant with each byte
		in hex format, with ':' as delimiter between two byte.
		offset=34 34 is the byte offset of the equal length field of rx'd pkt.
		}
		Filter_2 is Magic packet, it will looking for 16 contiguous copies of '00:50:43:20:01:02' from the rx pkt's offset 14
		Filter_2={
		type=0x41 Byte comparison filter
		repeat=16 16 copies of '00:50:43:20:01:02'
		byte=00:50:43:20:01:02 # '00:50:43:20:01:02' is the byte sequence constant
		offset=14 14 is the byte offset of the equal length field of rx'd pkt.
		}
		}
		}
		Above filters can be set by filling values in following way in wlan_flt_cfg_t structure.
		wlan_flt_cfg_t flt_cfg;
		uint8_t byte_seq1[] = {0xc0, 0xa8, 0x00, 0x58};
		uint8_t byte_seq2[] = {0x00, 0x50, 0x43, 0x20, 0x01, 0x02};
		
		memset(&flt_cfg, 0, sizeof(wlan_flt_cfg_t));
		
		flt_cfg.criteria = 2;
		flt_cfg.nentries = 1;
		
		flt_cfg.mef_entry.mode = 1;
		flt_cfg.mef_entry.action = 3;
		
		flt_cfg.mef_entry.filter_num = 3;

Parameters

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

5.8.3.63 wlan_set_auto_arp()

```
int wlan_set_auto_arp (
            void )
```

Use this API to enable ARP Offload in Wi-Fi firmware

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

5.8.3.64 wlan_wowlan_cfg_ptn_match()

```
int wlan_wowlan_cfg_ptn_match (
            wlan_wowlan_ptn_cfg_t * ptn_cfg )
```

Use this API to enable WOWLAN on magic pkt rx in Wi-Fi firmware

Parameters

in	*ptn_cfg*	A pointer to wlan_wowlan_ptn_cfg_t containing Wake on WLAN pattern configuration

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails

5.8.3.65 wlan_set_ipv6_ns_offload()

```
int wlan_set_ipv6_ns_offload ( )
```

Use this API to enable NS Offload in Wi-Fi firmware.

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

5.8.3.66 wlan_get_current_bssid()

```
int wlan_get_current_bssid (
            uint8_t * bssid )
```

Use this API to get the BSSID of associated BSS.

Parameters

in	*bssid*	A pointer to array to store the BSSID.

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

5.8.3.67 wlan_get_current_channel()

```
uint8_t wlan_get_current_channel (
            void )
```

Use this API to get the channel number of associated BSS.

Returns

channel number if operation is successful.
0 if command fails.

5.8.3.68 wlan_get_ps_mode()

```
int wlan_get_ps_mode (
            enum wlan_ps_mode * ps_mode )
```

Get station interface power save mode.

Parameters

out	*ps_mode*	A pointer to wlan_ps_mode where station interface power save mode will be stored.

Returns

WM_SUCCESS if successful.
-WM_E_INVAL if *ps_mode* was NULL.

5.8.3.69 wlan_wlcmgr_send_msg()

```
int wlan_wlcmgr_send_msg (
            enum wifi_event event,
            enum wifi_event_reason reason,
            void * data )
```

Send message to WLAN Connection Manager thread.

Parameters

in	*event*	An event from wifi_event.
in	*reason*	A reason code.
in	*data*	A pointer to data buffer associated with event.

Returns

WM_SUCCESS if successful.
-WM_FAIL if failed.

5.8.3.70 wlan_wfa_basic_cli_init()

```
int wlan_wfa_basic_cli_init (
            void )
```

Register WFA basic WLAN CLI commands

This function registers basic WLAN CLI commands like showing version information, MAC address

Note

This function can only be called by the application after wlan_init() called.

Returns

WLAN_ERROR_NONE if the CLI commands were registered or
WLAN_ERROR_ACTION if they were not registered (for example if this function was called while the CLI commands were already registered).

5.8.3.71 wlan_wfa_basic_cli_deinit()

```
int wlan_wfa_basic_cli_deinit (
            void )
```

Unregister WFA basic WLAN CLI commands

This function unregisters basic WLAN CLI commands like showing version information, MAC address

Note

This function can only be called by the application after wlan_init() called.

Returns

WLAN_ERROR_NONE if the CLI commands were unregistered or
WLAN_ERROR_ACTION if they were not unregistered

5.8.3.72 wlan_basic_cli_init()

```
int wlan_basic_cli_init (
            void )
```

Register basic WLAN CLI commands

This function registers basic WLAN CLI commands like showing version information, MAC address

Note

This function can only be called by the application after wlan_init() called.
This function gets called by wlan_cli_init(), hence only one function out of these two functions should be called in the application.

Returns

WLAN_ERROR_NONE if the CLI commands were registered or
WLAN_ERROR_ACTION if they were not registered (for example if this function was called while the CLI commands were already registered).

5.8.3.73 wlan_basic_cli_deinit()

```
int wlan_basic_cli_deinit (
            void )
```

Unregister basic WLAN CLI commands

This function unregisters basic WLAN CLI commands like showing version information, MAC address

Note

This function can only be called by the application after wlan_init() called.
This function gets called by wlan_cli_init(), hence only one function out of these two functions should be called in the application.

Returns

WLAN_ERROR_NONE if the CLI commands were unregistered or
WLAN_ERROR_ACTION if they were not unregistered (for example if this function was called while the CLI commands were already registered).

5.8.3.74 wlan_cli_init()

```
int wlan_cli_init (
            void )
```

Register WLAN CLI commands.

Try to register the WLAN CLI commands with the CLI subsystem. This function is available for the application for use.

Note

This function can only be called by the application after wlan_init() called.
This function internally calls wlan_basic_cli_init(), hence only one function out of these two functions should be called in the application.

Returns

WM_SUCCESS if the CLI commands were registered or
-WM_FAIL if they were not (for example if this function was called while the CLI commands were already registered).

5.8.3.75 wlan_cli_deinit()

```
int wlan_cli_deinit (
            void )
```

Unregister WLAN CLI commands.

Try to unregister the WLAN CLI commands with the CLI subsystem. This function is available for the application for use.

Note

This function can only be called by the application after wlan_init() called.
This function internally calls wlan_basic_cli_deinit(), hence only one function out of these two functions should be called in the application.

Returns

WM_SUCCESS if the CLI commands were unregistered or
-WM_FAIL if they were not (for example if this function was called while the CLI commands were already unregistered).

5.8.3.76 wlan_enhanced_cli_init()

```
int wlan_enhanced_cli_init (
            void )
```

Register WLAN enhanced CLI commands.

Register the WLAN enhanced CLI commands like set or get tx-power, tx-datarate, tx-modulation etc with the CLI subsystem.

Note

This function can only be called by the application after wlan_init() called.

Returns

WM_SUCCESS if the CLI commands were registered or
-WM_FAIL if they were not (for example if this function was called while the CLI commands were already registered).

5.8.3.77 wlan_enhanced_cli_deinit()

```
int wlan_enhanced_cli_deinit (
            void )
```

Unregister WLAN enhanced CLI commands.

Unregister the WLAN enhanced CLI commands like set or get tx-power, tx-datarate, tx-modulation etc with the CLI subsystem.

Note

This function can only be called by the application after wlan_init() called.

Returns

WM_SUCCESS if the CLI commands were unregistered or
-WM_FAIL if they were not unregistered.

5.8.3.78 wlan_test_mode_cli_init()

```
int wlan_test_mode_cli_init (
            void )
```

Register WLAN Test Mode CLI commands.

Register the WLAN Test Mode CLI commands like set or get channel, band, bandwidth, PER and more with the CLI subsystem.

Note

This function can only be called by the application after wlan_init() called.

Returns

WM_SUCCESS if the CLI commands were registered or
-WM_FAIL if they were not (for example if this function was called while the CLI commands were already registered).

5.8.3.79 wlan_test_mode_cli_deinit()

```
int wlan_test_mode_cli_deinit (
            void )
```

Unregister WLAN Test Mode CLI commands.

Unregister the WLAN Test Mode CLI commands like set or get channel, band, bandwidth, PER and more with the CLI subsystem.

Note

This function can only be called by the application after wlan_init() called.

Returns

WM_SUCCESS if the CLI commands were unregistered or
-WM_FAIL if they were not unregistered

5.8.3.80 wlan_get_uap_supported_max_clients()

```
unsigned int wlan_get_uap_supported_max_clients (
            void )
```

Get maximum number of WLAN firmware supported stations that will be allowed to connect to the uAP.

Returns

Maximum number of WLAN firmware supported stations.

Note

Get operation is allowed in any uAP state.

5.8.3.81 wlan_get_uap_max_clients()

```
int wlan_get_uap_max_clients (
            unsigned int * max_sta_num )
```

Get current maximum number of stations that will be allowed to connect to the uAP.

Parameters

out	*max_sta_num*	A pointer to variable where current maximum number of stations of uAP interface will be stored.

Returns

WM_SUCCESS if successful.
-WM_FAIL if unsuccessful.

Note

Get operation is allowed in any uAP state.

5.8.3.82 wlan_set_uap_max_clients()

```
int wlan_set_uap_max_clients (
            unsigned int max_sta_num )
```

Set maximum number of stations that will be allowed to connect to the uAP.

Parameters

in	*max_sta_num*	Number of maximum stations for uAP.

Returns

WM_SUCCESS if successful.
-WM_FAIL if unsuccessful.

Note

Set operation in not allowed in WLAN_UAP_STARTED state.

5.8.3.83 wlan_set_htcapinfo()

```
int wlan_set_htcapinfo (
            unsigned int htcapinfo )
```

This API can be used to configure some of parameters in HTCapInfo IE (such as Short GI, Channel BW, and Green field support)

Parameters

in	*htcapinfo*	This is a bitmap and should be used as following Bit 29: Green field enable/disable Bit 26: Rx STBC Support enable/disable. (As we support single spatial stream only 1 bit is used for Rx STBC) Bit 25: Tx STBC support enable/disable. Bit 24: Short GI in 40 Mhz enable/disable Bit 23: Short GI in 20 Mhz enable/disable Bit 22: Rx LDPC enable/disable Bit 17: 20/40 Mhz enable disable. Bit 8: Enable/disable 40Mhz Intolarent bit in ht capinfo. 0 will reset this bit and 1 will set this bit in htcapinfo attached in assoc request. All others are reserved and should be set to 0.

Returns

WM_SUCCESS if successful.
-WM_FAIL if unsuccessful.

5.8.3.84 wlan_set_httxcfg()

```
int wlan_set_httxcfg (
            unsigned short httxcfg )
```

This API can be used to configure various 11n specific configuration for transmit (such as Short GI, Channel BW and Green field support)

Parameters

in	*httxcfg*	This is a bitmap and should be used as following Bit 15-10: Reserved set to 0 Bit 9-8: Rx STBC set to 0x01 BIT9 BIT8 Description 0 0 No spatial streams 0 1 One spatial streams supported 1 0 Reserved 1 1 Reserved Bit 7: STBC enable/disable Bit 6: Short GI in 40 Mhz enable/disable Bit 5: Short GI in 20 Mhz enable/disable Bit 4: Green field enable/disable Bit 3-2: Reserved set to 1 Bit 1: 20/40 Mhz enable disable. Bit 0: LDPC enable/disable When Bit 1 is set then firmware could transmit in 20Mhz or 40Mhz based on rate adaptation. When this bit is reset then firmware will only transmit in 20Mhz.

Returns

WM_SUCCESS if successful.
-WM_FAIL if unsuccessful.

5.8.3.85 wlan_set_txratecfg()

```
int wlan_set_txratecfg (
            wlan_ds_rate ds_rate,
            mlan_bss_type bss_type )
```

This API can be used to set the transmit data rate.

Note

The data rate can be set only after association.

Parameters

in	*ds_rate*	struct contains following fields sub_command It should be WIFI_DS_RATE_CFG and rate_cfg should have following parameters. rate_format - This parameter specifies the data rate format used in this command 0: LG 1: HT 2: VHT 0xff: Auto index - This parameter specifies the rate or MCS index If rate_format is 0 (LG), 0 1 Mbps 1 2 Mbps 2 5.5 Mbps 3 11 Mbps 4 6 Mbps 5 9 Mbps 6 12 Mbps 7 18 Mbps 8 24 Mbps 9 36 Mbps 10 48 Mbps 11 54 Mbps If rate_format is 1 (HT), 0 MCS0 1 MCS1 2 MCS2 3 MCS3 4 MCS4 5 MCS5 6 MCS6 7 MCS7 If STREAM_2X2 8 MCS8 9 MCS9 10 MCS10 11 MCS11 12 MCS12 13 MCS13 14 MCS14 15 MCS15 If rate_format is 2 (VHT), 0 MCS0 1 MCS1 2 MCS2 3 MCS3 4 MCS4 5 MCS5 6 MCS6 7 MCS7 8 MCS8 9 MCS9 nss - This parameter specifies the NSS. It is valid only for VHT If rate_format is 2 (VHT), 1 NSS1 2 NSS2

Parameters

in	*bss_type*	0: STA, 1: uAP

Returns

WM_SUCCESS if successful.
-WM_FAIL if unsuccessful.

5.8.3.86 wlan_get_txratecfg()

```
int wlan_get_txratecfg (
            wlan_ds_rate * ds_rate,
            mlan_bss_type bss_type )
```

This API can be used to get the transmit data rate.

Parameters

in	*ds_rate*	A pointer to wlan_ds_rate where Tx Rate configuration will be stored.
in	*bss_type*	0: STA, 1: uAP

Returns

WM_SUCCESS if successful.
-WM_FAIL if unsuccessful.

5.8.3.87 wlan_get_sta_tx_power()

```
int wlan_get_sta_tx_power (
            t_u32 * power_level )
```

Get Station interface transmit power

Parameters

out	*power_level*	Transmit power level.

Returns

WM_SUCCESS if successful.
-WM_FAIL if unsuccessful.

5.8.3.88 wlan_set_sta_tx_power()

```
int wlan_set_sta_tx_power (
            t_u32 power_level )
```

Set Station interface transmit power

Parameters

in	*power_level*	Transmit power level.

Returns

WM_SUCCESS if successful.
-WM_FAIL if unsuccessful.

5.8.3.89 wlan_set_wwsm_txpwrlimit()

```
int wlan_set_wwsm_txpwrlimit (
            void )
```

Set World Wide Safe Mode Tx Power Limits

Returns

WM_SUCCESS if successful.
-WM_FAIL if unsuccessful.

5.8.3.90 wlan_get_wlan_region_code()

```
const char* wlan_get_wlan_region_code (
            void )
```

Get wlan region code from tx power config

Returns

wlan region code in string format.

5.8.3.91 wlan_get_mgmt_ie()

```
int wlan_get_mgmt_ie (
            enum wlan_bss_type bss_type,
            IEEEtypes_ElementId_t index,
            void * buf,
            unsigned int * buf_len )
```

Get Management IE for given BSS type (interface) and index.

Parameters

in	*bss_type*	0: STA, 1: uAP
in	*index*	IE index.
out	*buf*	Buffer to store requested IE data.
out	*buf_len*	To store length of IE data.

Returns

WM_SUCCESS if successful.
-WM_FAIL if unsuccessful.

5.8.3.92 wlan_set_mgmt_ie()

```
int wlan_set_mgmt_ie (
            enum wlan_bss_type bss_type,
            IEEEtypes_ElementId_t id,
            void * buf,
            unsigned int buf_len )
```

Set Management IE for given BSS type (interface) and index.

Parameters

in	*bss_type*	0: STA, 1: uAP
in	*id*	Type/ID of Management IE.
in	*buf*	Buffer containing IE data.
in	*buf_len*	Length of IE data.

Returns

IE index if successful.
-WM_FAIL if unsuccessful.

5.8.3.93 wlan_get_ext_coex_stats()

```
int wlan_get_ext_coex_stats (
            wlan_ext_coex_stats_t * ext_coex_stats )
```

Get External Radio Coex statistics.

Parameters

out	*ext_coex_stats*	A pointer to structure to get coex statistics.

Returns

WM_SUCCESS if successful.
-WM_FAIL if unsuccessful.

5.8.3.94 wlan_set_ext_coex_config()

```
int wlan_set_ext_coex_config (
            const wlan_ext_coex_config_t ext_coex_config )
```

Set External Radio Coex configuration.

Parameters

in	*ext_coex_config*	to apply coex configuration.

Returns

IE index if successful.
-WM_FAIL if unsuccessful.

5.8.3.95 wlan_clear_mgmt_ie()

```
int wlan_clear_mgmt_ie (
            enum wlan_bss_type bss_type,
            IEEEtypes_ElementId_t index,
            int mgmt_bitmap_index )
```

Clear Management IE for given BSS type (interface) and index.

Parameters

in	*bss_type*	0: STA, 1: uAP
in	*index*	IE index.
in	*mgmt_bitmap_index*	mgmt bitmap index.

Returns

WM_SUCCESS if successful.
-WM_FAIL if unsuccessful.

5.8.3.96 wlan_get_11d_enable_status()

```
bool wlan_get_11d_enable_status (
            void )
```

Get current status of 11d support.

Returns

true if 11d support is enabled by application.
false if not enabled.

5.8.3.97 wlan_get_current_signal_strength()

```
int wlan_get_current_signal_strength (
            short * rssi,
            int * snr )
```

Get current RSSI and Signal to Noise ratio from WLAN firmware.

Parameters

in	*rssi*	A pointer to variable to store current RSSI
in	*snr*	A pointer to variable to store current SNR.

Returns

WM_SUCCESS if successful.

5.8.3.98 wlan_get_average_signal_strength()

```
int wlan_get_average_signal_strength (
            short * rssi,
            int * snr )
```

Get average RSSI and Signal to Noise ratio from WLAN firmware.

Parameters

in	*rssi*	A pointer to variable to store current RSSI
in	*snr*	A pointer to variable to store current SNR.

Returns

WM_SUCCESS if successful.

5.8.3.99 wlan_remain_on_channel()

```
int wlan_remain_on_channel (
            const enum wlan_bss_type bss_type,
            const bool status,
            const uint8_t channel,
            const uint32_t duration )
```

This API is is used to set/cancel the remain on channel configuration.

Note

When status is false, channel and duration parameters are ignored.

Parameters

in	*bss_type*	The interface to set channel bss_type 0: STA, 1: uAP
in	*status*	false : Cancel the remain on channel configuration true : Set the remain on channel configuration
in	*channel*	The channel to configure
in	*duration*	The duration for which to remain on channel in milliseconds.

Returns

WM_SUCCESS on success or error code.

5.8.3.100 wlan_get_otp_user_data()

```
int wlan_get_otp_user_data (
            uint8_t * buf,
            uint16_t len )
```

Get User Data from OTP Memory

Parameters

in	*buf*	Pointer to buffer where data will be stored
in	*len*	Number of bytes to read

Returns

WM_SUCCESS if user data read operation is successful.
-WM_E_INVAL if buf is not valid or of insufficient size.
-WM_FAIL if user data field is not present or command fails.

5.8.3.101 wlan_get_cal_data()

```
int wlan_get_cal_data (
            wlan_cal_data_t * cal_data )
```

Get calibration data from WLAN firmware

Parameters

out	*cal_data*	Pointer to calibration data structure where calibration data and it's length will be stored.

Returns

WM_SUCCESS if cal data read operation is successful.
-WM_E_INVAL if cal_data is not valid.
-WM_FAIL if command fails.

Note

The user of this API should free the allocated buffer for calibration data.

5.8.3.102 wlan_set_region_power_cfg()

```
int wlan_set_region_power_cfg (
            const t_u8 * data,
            t_u16 len )
```

Set the compressed Tx PWR Limit configuration.

Parameters

in	*data*	A pointer to TX PWR Limit configuration.
in	*len*	Length of TX PWR Limit configuration.

Returns

WM_SUCCESS on success, error otherwise.

5.8.3.103 wlan_set_chanlist_and_txpwrlimit()

```
int wlan_set_chanlist_and_txpwrlimit (
            wlan_chanlist_t * chanlist,
            wlan_txpwrlimit_t * txpwrlimit )
```

Set the Channel List and TRPC channel configuration.

Parameters

in	*chanlist*	A poiner to wlan_chanlist_t Channel List configuration.
in	*txpwrlimit*	A pointer to wlan_txpwrlimit_t TX PWR Limit configuration.

Returns

WM_SUCCESS on success, error otherwise.

5.8.3.104 wlan_set_chanlist()

```
int wlan_set_chanlist (
            wlan_chanlist_t * chanlist )
```

Set the Channel List configuration.

Parameters

in	*chanlist*	A pointer to wlan_chanlist_t Channel List configuration.

Returns

WM_SUCCESS on success, error otherwise.

Note

If Region Enforcement Flag is enabled in the OTP then this API will not take effect.

5.8.3.105 wlan_get_chanlist()

```
int wlan_get_chanlist (
            wlan_chanlist_t * chanlist )
```

Get the Channel List configuration.

Parameters

out	*chanlist*	A pointer to wlan_chanlist_t Channel List configuration.

Returns

WM_SUCCESS on success, error otherwise.

Note

The wlan_chanlist_t struct allocates memory for a maximum of 54 channels.

5.8.3.106 wlan_set_txpwrlimit()

```
int wlan_set_txpwrlimit (
            wlan_txpwrlimit_t * txpwrlimit )
```

Set the TRPC channel configuration.

Parameters

in	*txpwrlimit*	A pointer to wlan_txpwrlimit_t TX PWR Limit configuration.

Returns

WM_SUCCESS on success, error otherwise.

5.8.3.107 wlan_get_txpwrlimit()

```
int wlan_get_txpwrlimit (
            wifi_SubBand_t subband,
            wifi_txpwrlimit_t * txpwrlimit )
```

Get the TRPC channel configuration.

Parameters

in	*subband*	Where subband is: 0x00 2G subband (2.4G: channel 1-14) 0x10 5G subband0 (5G: channel 36,40,44,48, 52,56,60,64) 0x11 5G subband1 (5G: channel 100,104,108,112, 116,120,124,128, 132,136,140,144) 0x12 5G subband2 (5G: channel 149,153,157,161,165,172) 0x13 5G subband3 (5G: channel 183,184,185,187,188, 189, 192,196; 5G: channel 7,8,11,12,16,34)
out	*txpwrlimit*	A pointer to wlan_txpwrlimit_t TX PWR Limit configuration structure where Wi-Fi firmware configuration will get copied.

Returns

WM_SUCCESS on success, error otherwise.

Note

application can use print_txpwrlimit API to print the content of the txpwrlimit structure.

5.8.3.108 wlan_auto_reconnect_enable()

```
int wlan_auto_reconnect_enable (
            wlan_auto_reconnect_config_t auto_reconnect_config )
```

Enable Auto Reconnect feature in WLAN firmware.

Parameters

in	*auto_reconnect_config*	Auto Reconnect configuration structure holding following parameters: 1. reconnect counter(0x1-0xff) - The number of times the WLAN firmware retries connection attempt with AP. The value 0xff means retry forever. (default 0xff). 2. reconnect interval(0x0-0xff) - Time gap in seconds between each connection attempt (default 10). 3. flags - Bit 0: Set to 1: Firmware should report link-loss to host if AP rejects authentication/association while reconnecting. Set to 0: Default behaviour: Firmware does not report link-loss to host on AP rejection and continues internally. Bit 1-15: Reserved.

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

5.8.3.109 wlan_auto_reconnect_disable()

```
int wlan_auto_reconnect_disable (
            void )
```

Disable Auto Reconnect feature in WLAN firmware.

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

5.8.3.110 wlan_get_auto_reconnect_config()

```
int wlan_get_auto_reconnect_config (
            wlan_auto_reconnect_config_t * auto_reconnect_config )
```

Get Auto Reconnect configuration from WLAN firmware.

Parameters

out	*auto_reconnect_config*	Auto Reconnect configuration structure where response from WLAN firmware will get stored.

Returns

WM_SUCCESS if operation is successful.
-WM_E_INVAL if auto_reconnect_config is not valid.
-WM_FAIL if command fails.

5.8.3.111 wlan_set_reassoc_control()

```
void wlan_set_reassoc_control (
            bool reassoc_control )
```

Set Reassociation Control in WLAN Connection Manager

Note

Reassociation is enabled by default in the WLAN Connection Manager.

Parameters

in	*reassoc_control*	Reassociation enable/disable

5.8.3.112 wlan_uap_set_beacon_period()

```
void wlan_uap_set_beacon_period (
            const uint16_t beacon_period )
```

API to set the beacon period of uAP

Parameters

in	*beacon_period*	Beacon period in TU (1 TU = 1024 micro seconds)

Note

Please call this API before calling uAP start API.

5.8.3.113 wlan_uap_set_bandwidth()

```
int wlan_uap_set_bandwidth (
            const uint8_t bandwidth )
```

API to set the bandwidth of uAP

Parameters

in	*bandwidth*	Wi-Fi AP Bandwidth (20MHz/40MHz) 1: 20 MHz 2: 40 MHz

Returns

WM_SUCCESS if successful otherwise failure.
-WM_FAIL if command fails.

Note

Please call this API before calling uAP start API.
Default bandwidth setting is 40 MHz.

5.8.3.114 wlan_uap_set_hidden_ssid()

```
int wlan_uap_set_hidden_ssid (
            const t_u8 hidden_ssid )
```

API to control SSID broadcast capability of uAP

This API enables/disables the SSID broadcast feature (also known as the hidden SSID feature). When broadcast SSID is enabled, the AP responds to probe requests from client stations that contain null SSID. When broadcast SSID is disabled, the AP does not respond to probe requests that contain null SSID and generates beacons that contain null SSID.

Parameters

in	*hidden_ssid*	Hidden SSID control hidden_ssid=0: broadcast SSID in beacons. hidden_ssid=1: send empty SSID (length=0) in beacon. hidden_ssid=2: clear SSID (ACSII 0), but keep the original length

Returns

WM_SUCCESS if successful otherwise failure.
-WM_FAIL if command fails.

Note

Please call this API before calling uAP start API.

5.8.3.115 wlan_uap_ctrl_deauth()

```
void wlan_uap_ctrl_deauth (
            const bool enable )
```

API to control the deauth during uAP channel switch

Parameters

in	*enable*	0 – Wi-Fi firmware will use default behaviour. 1 – Wi-Fi firmware will not send deauth packet when uap move to another channel.

Note

Please call this API before calling uAP start API.

5.8.3.116 wlan_uap_set_ecsa()

```
void wlan_uap_set_ecsa (
            void )
```

API to enable channel switch announcement functionality on uAP.

Note

Please call this API before calling uAP start API. Also note that 11N should be enabled on uAP. The channel switch announcement IE is transmitted in 7 beacons before the channel switch, during a station connection attempt on a different channel with Ex-AP.

5.8.3.117 wlan_uap_set_htcapinfo()

```
void wlan_uap_set_htcapinfo (
            const uint16_t ht_cap_info )
```

API to set the HT Capability Information of uAP

Parameters

in	*ht_cap_info*	- This is a bitmap and should be used as following Bit 15: L Sig TxOP protection - reserved, set to 0 Bit 14: 40 MHz intolerant - reserved, set to 0 Bit 13: PSMP - reserved, set to 0 Bit 12: DSSS Cck40MHz mode Bit 11: Maximal AMSDU size - reserved, set to 0 Bit 10: Delayed BA - reserved, set to 0 Bits 9:8: Rx STBC - reserved, set to 0 Bit 7: Tx STBC - reserved, set to 0 Bit 6: Short GI 40 MHz Bit 5: Short GI 20 MHz Bit 4: GF preamble Bits 3:2: MIMO power save - reserved, set to 0 Bit 1: SuppChanWidth - set to 0 for 2.4 GHz band Bit 0: LDPC coding - reserved, set to 0

Note

Please call this API before calling uAP start API.

5.8.3.118 wlan_uap_set_httxcfg()

```
void wlan_uap_set_httxcfg (
            unsigned short httxcfg )
```

This API can be used to configure various 11n specific configuration for transmit (such as Short GI, Channel BW and Green field support) for uAP interface.

Parameters

in	*httxcfg*	This is a bitmap and should be used as following Bit 15-8: Reserved set to 0 Bit 7: STBC enable/disable Bit 6: Short GI in 40 Mhz enable/disable Bit 5: Short GI in 20 Mhz enable/disable Bit 4: Green field enable/disable Bit 3-2: Reserved set to 1 Bit 1: 20/40 Mhz enable disable. Bit 0: LDPC enable/disable When Bit 1 is set then firmware could transmit in 20Mhz or 40Mhz based on rate adaptation. When this bit is reset then firmware will only transmit in 20Mhz.

Note

Please call this API before calling uAP start API.

5.8.3.119 wlan_sta_ampdu_tx_enable()

```
void wlan_sta_ampdu_tx_enable (
            void   )
```

This API can be used to enable AMPDU support on the go when station is a transmitter.

Note

By default the station AMPDU TX support is on if configuration option is enabled in defconfig.

5.8.3.120 wlan_sta_ampdu_tx_disable()

```
void wlan_sta_ampdu_tx_disable (
            void   )
```

This API can be used to disable AMPDU support on the go when station is a transmitter.

Note

By default the station AMPDU RX support is on if configuration option is enabled in defconfig.

5.8.3.121 wlan_sta_ampdu_rx_enable()

```
void wlan_sta_ampdu_rx_enable (
            void )
```

This API can be used to enable AMPDU support on the go when station is a receiver.

5.8.3.122 wlan_sta_ampdu_rx_disable()

```
void wlan_sta_ampdu_rx_disable (
            void )
```

This API can be used to disable AMPDU support on the go when station is a receiver.

5.8.3.123 wlan_uap_ampdu_tx_enable()

```
void wlan_uap_ampdu_tx_enable (
            void )
```

This API can be used to enable AMPDU support on the go when uap is a transmitter.

Note

By default the uap AMPDU TX support is on if configuration option is enabled in defconfig.

5.8.3.124 wlan_uap_ampdu_tx_disable()

```
void wlan_uap_ampdu_tx_disable (
            void )
```

This API can be used to disable AMPDU support on the go when uap is a transmitter.

Note

By default the uap AMPDU RX support is on if configuration option is enabled in defconfig.

5.8.3.125 wlan_uap_ampdu_rx_enable()

```
void wlan_uap_ampdu_rx_enable (
            void )
```

This API can be used to enable AMPDU support on the go when uap is a receiver.

5.8.3.126 wlan_uap_ampdu_rx_disable()

```
void wlan_uap_ampdu_rx_disable (
            void )
```

This API can be used to disable AMPDU support on the go when uap is a receiver.

5.8.3.127 wlan_uap_set_scan_chan_list()

```
void wlan_uap_set_scan_chan_list (
            wifi_scan_chan_list_t scan_chan_list )
```

Set number of channels and channel number used during automatic channel selection of uAP.

Parameters

in	*scan_chan_list*	A structure holding the number of channels and channel numbers.

Note

Please call this API before uAP start API in order to set the user defined channels, otherwise it will have no effect. There is no need to call this API every time before uAP start, if once set same channel configuration will get used in all upcoming uAP start call. If user wish to change the channels at run time then it make sense to call this API before every uAP start API.

5.8.3.128 wlan_set_rts()

```
int wlan_set_rts (
            int rts )
```

Set the rts threshold of sta in WLAN firmware.

Parameters

in	*rts*	the value of rts threshold configuration.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.129 wlan_set_uap_rts()

```
int wlan_set_uap_rts (
            int rts )
```

Set the rts threshold of uap in WLAN firmware.

Parameters

in	*rts*	the value of rts threshold configuration.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.130 wlan_set_frag()

```
int wlan_set_frag (
            int frag )
```

Set the fragment threshold of sta in WLAN firmware.

Parameters

in	*frag*	the value of fragment threshold configuration.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.131 wlan_set_uap_frag()

```
int wlan_set_uap_frag (
            int frag )
```

Set the fragment threshold of uap in WLAN firmware.

Parameters

in	*frag*	the value of fragment threshold configuration.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.132 wlan_set_sta_mac_filter()

```
int wlan_set_sta_mac_filter (
            int filter_mode,
            int mac_count,
            unsigned char * mac_addr )
```

Set the sta mac filter in Wi-Fi firmware.

Parameters

in	*filter_mode*	channel filter mode (disable/white/black list)
in	*mac_count*	the count of mac list
in	*mac_addr*	the pointer to mac address list

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.133 wlan_set_rf_test_mode()

```
int wlan_set_rf_test_mode (
            void )
```

Set the RF Test Mode on in Wi-Fi firmware.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.134 wlan_unset_rf_test_mode()

```
int wlan_unset_rf_test_mode (
            void )
```

UnSet the RF Test Mode on in Wi-Fi firmware.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.135 wlan_set_rf_channel()

```
int wlan_set_rf_channel (
            const uint8_t channel )
```

Set the RF Channel in Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

in	*channel*	The channel number to be set in Wi-Fi firmware.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.136 wlan_set_rf_radio_mode()

```
int wlan_set_rf_radio_mode (
            const uint8_t mode )
```

Set the RF radio mode in Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

in	*mode*	The radio mode number to be set in Wi-Fi firmware.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.137 wlan_get_rf_channel()

```
int wlan_get_rf_channel (
            uint8_t * channel )
```

Get the RF Channel from Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

out	*channel*	A Pointer to a variable where channel number to get.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.138 wlan_get_rf_radio_mode()

```
int wlan_get_rf_radio_mode (
            uint8_t * mode )
```

Get the RF Radio mode from Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

out	*mode*	A Pointer to a variable where radio mode number to get.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.139 wlan_set_rf_band()

```
int wlan_set_rf_band (
            const uint8_t band )
```

Set the RF Band in Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

in	*band*	The bandwidth to be set in Wi-Fi firmware.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.140 wlan_get_rf_band()

```
int wlan_get_rf_band (
            uint8_t * band )
```

Get the RF Band from Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

out	*band*	A Pointer to a variable where RF Band is to be stored.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.141 wlan_set_rf_bandwidth()

```
int wlan_set_rf_bandwidth (
            const uint8_t bandwidth )
```

Set the RF Bandwidth in Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

in	*bandwidth*	The bandwidth to be set in Wi-Fi firmware.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.142 wlan_get_rf_bandwidth()

```
int wlan_get_rf_bandwidth (
            uint8_t * bandwidth )
```

Get the RF Bandwidth from Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

out	*bandwidth*	A Pointer to a variable where bandwidth to get.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.143 wlan_get_rf_per()

```
int wlan_get_rf_per (
            uint32_t * rx_tot_pkt_count,
            uint32_t * rx_mcast_bcast_count,
            uint32_t * rx_pkt_fcs_error )
```

Get the RF PER from Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

out	*rx_tot_pkt_count*	A Pointer to a variable where Rx Total packet count to get.
out	*rx_mcast_bcast_count*	A Pointer to a variable where Rx Total Multicast/Broadcast packet count to get.
out	*rx_pkt_fcs_error*	A Pointer to a variable where Rx Total packet count with FCS error to get.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.144 wlan_set_rf_tx_cont_mode()

```
int wlan_set_rf_tx_cont_mode (
            const uint32_t enable_tx,
            const uint32_t cw_mode,
            const uint32_t payload_pattern,
            const uint32_t cs_mode,
            const uint32_t act_sub_ch,
            const uint32_t tx_rate )
```

Set the RF Tx continuous mode in Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

in	*enable_tx*	Enable Tx.
in	*cw_mode*	Set CW Mode.
in	*payload_pattern*	Set Payload Pattern.
in	*cs_mode*	Set CS Mode.
in	*act_sub_ch*	Act Sub Ch
in	*tx_rate*	Set Tx Rate.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.145 wlan_cfg_rf_he_tb_tx()

```
int wlan_cfg_rf_he_tb_tx (
            uint16_t enable,
            uint16_t qnum,
            uint16_t aid,
            uint16_t axq_mu_timer,
            int16_t tx_power )
```

Set the RF HE TB TX in Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

in	*enable*	Enable/Disable trigger response mode
in	*qnum*	AXQ to be used for the trigger response frame
in	*aid*	AID of the peer to which response is to be generated
in	*axq_mu_timer*	MU timer for the AXQ on which response is sent
in	*tx_power*	TxPwr to be configured for the response

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.146 wlan_rf_trigger_frame_cfg()

```
int wlan_rf_trigger_frame_cfg (
            uint32_t Enable_tx,
            uint32_t Standalone_hetb,
            uint8_t FRAME_CTRL_TYPE,
            uint8_t FRAME_CTRL_SUBTYPE,
            uint16_t FRAME_DURATION,
            uint64_t TriggerType,
            uint64_t UlLen,
            uint64_t MoreTF,
            uint64_t CSRequired,
            uint64_t UlBw,
            uint64_t LTFType,
            uint64_t LTFMode,
            uint64_t LTFSymbol,
```

```
            uint64_t UlSTBC,
            uint64_t LdpcESS,
            uint64_t ApTxPwr,
            uint64_t PreFecPadFct,
            uint64_t PeDisambig,
            uint64_t SpatialReuse,
            uint64_t Doppler,
            uint64_t HeSig2,
            uint32_t AID12,
            uint32_t RUAllocReg,
            uint32_t RUAlloc,
            uint32_t UlCodingType,
            uint32_t UlMCS,
            uint32_t UlDCM,
            uint32_t SSAlloc,
            uint8_t UlTargetRSSI,
            uint8_t MPDU_MU_SF,
            uint8_t TID_AL,
            uint8_t AC_PL,
            uint8_t Pref_AC )
```

Set the RF Trigger Frame Config in Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

in	*Enable_tx*	Enable or Disable trigger frame transmission.
in	*Standalone_hetb*	Enable or Disable Standalone HE TB support.
in	*FRAME_CTRL_TYPE*	Frame control type.
in	*FRAME_CTRL_SUBTYPE*	Frame control subtype.
in	*FRAME_DURATION*	Max Duration time.
in	*TriggerType*	Identifies the Trigger frame variant and its encoding.
in	*UlLen*	Indicates the value of the L-SIG LENGTH field of the solicited HE TB PPDU.
in	*MoreTF*	Indicates whether a subsequent Trigger frame is scheduled for transmission.
in	*CSRequired*	Required to use ED to sense the medium and to consider the medium state and the NAV in determining whether to respond.
in	*UlBw*	Indicates the bandwidth in the HE-SIG-A field of the HE TB PPDU.
in	*LTFType*	Indicates the LTF type of the HE TB PPDU response.
in	*LTFMode*	Indicates the LTF mode for an HE TB PPDU.
in	*LTFSymbol*	Indicates the number of LTF symbols present in the HE TB PPDU.
in	*UlSTBC*	Indicates the status of STBC encoding for the solicited HE TB PPDUs.
in	*LdpcESS*	Indicates the status of the LDPC extra symbol segment.
in	*ApTxPwr*	Indicates the AP's combined transmit power at the transmit antenna connector of all the antennas used to transmit the triggering PPDU.
in	*PreFecPadFct*	Indicates the pre-FEC padding factor.
in	*PeDisambig*	Indicates PE disambiguity.
in	*SpatialReuse*	Carries the values to be included in the Spatial Reuse fields in the HE-SIG-A field of the solicited HE TB PPDUs.
in	*Doppler*	Indicate that a midamble is present in the HE TB PPDU.

Parameters

in	*HeSig2*	Carries the value to be included in the Reserved field in the HE-SIG-A2 subfield of the solicited HE TB PPDUs.
in	*AID12*	If set to 0 allocates one or more contiguous RA-RUs for associated STAs.
in	*RUAllocReg*	RUAllocReg.
in	*RUAlloc*	Identifies the size and the location of the RU.
in	*UlCodingType*	Indicates the code type of the solicited HE TB PPDU.
in	*UlMCS*	Indicates the HE-MCS of the solicited HE TB PPDU.
in	*UlDCM*	Indicates DCM of the solicited HE TB PPDU.
in	*SSAlloc*	Indicates the spatial streams of the solicited HE TB PPDU.
in	*UlTargetRSSI*	Indicates the expected receive signal power.
in	*MPDU_MU_SF*	Used for calculating the value by which the minimum MPDU start spacing is multiplied.
in	*TID_AL*	Indicates the MPDUs allowed in an A-MPDU carried in the HE TB PPDU and the maximum number of TIDs that can be aggregated by the STA in the A-MPDU.
in	*AC_PL*	Reserved.
in	*Pref_AC*	Indicates the lowest AC that is recommended for aggregation of MPDUs in the A-MPDU contained in the HE TB PPDU sent as a response to the Trigger frame.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.147 wlan_set_rf_tx_antenna()

```
int wlan_set_rf_tx_antenna (
            const uint8_t antenna )
```

Set the RF Tx Antenna in Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

in	*antenna*	The Tx antenna to be set in Wi-Fi firmware.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.148 wlan_get_rf_tx_antenna()

```
int wlan_get_rf_tx_antenna (
            uint8_t * antenna )
```

Get the RF Tx Antenna from Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

out	*antenna*	A Pointer to a variable where Tx antenna is to be stored.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.149 wlan_set_rf_rx_antenna()

```
int wlan_set_rf_rx_antenna (
            const uint8_t antenna )
```

Set the RF Rx Antenna in Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

in	*antenna*	The Rx antenna to be set in Wi-Fi firmware.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.150 wlan_get_rf_rx_antenna()

```
int wlan_get_rf_rx_antenna (
            uint8_t * antenna )
```

Get the RF Rx Antenna from Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

out	*antenna*	A Pointer to a variable where Rx antenna is to be stored.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.151 wlan_set_rf_tx_power()

```
int wlan_set_rf_tx_power (
            const uint32_t power,
            const uint8_t mod,
            const uint8_t path_id )
```

Set the RF Tx Power in Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

in	*power*	The RF Tx Power to be set in Wi-Fi firmware. For RW610, 20M bandwidth max linear output power is 20db per data sheet.
in	*mod*	The modulation to be set in Wi-Fi firmware.
in	*path_id*	The Path ID to be set in Wi-Fi firmware.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.152 wlan_set_rf_tx_frame()

```
int wlan_set_rf_tx_frame (
            const uint32_t enable,
            const uint32_t data_rate,
            const uint32_t frame_pattern,
            const uint32_t frame_length,
            const uint16_t adjust_burst_sifs,
            const uint32_t burst_sifs_in_us,
            const uint32_t short_preamble,
            const uint32_t act_sub_ch,
            const uint32_t short_gi,
```

```
            const uint32_t adv_coding,
            const uint32_t tx_bf,
            const uint32_t gf_mode,
            const uint32_t stbc,
            const uint8_t * bssid )
```

Set the RF Tx Frame in Wi-Fi firmware.

Note

Please call wlan_set_rf_test_mode API before using this API.

Parameters

in	*enable*	Enable/Disable RF Tx Frame
in	*data_rate*	Rate Index corresponding to legacy/HT/VHT rates
in	*frame_pattern*	Payload Pattern
in	*frame_length*	Payload Length
in	*adjust_burst_sifs*	Enabl/Disable Adjust Burst SIFS3 Gap
in	*burst_sifs_in_us*	Burst SIFS in us
in	*short_preamble*	Enable/Disable Short Preamble
in	*act_sub_ch*	Enable/Disable Active SubChannel
in	*short_gi*	Short Guard Interval
in	*adv_coding*	Enable/Disable Adv Coding
in	*tx_bf*	Enable/Disable Beamforming
in	*gf_mode*	Enable/Disable GreenField Mode
in	*stbc*	Enable/Disable STBC
in	*bssid*	BSSID

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.153 wlan_register_fw_dump_cb()

```
void wlan_register_fw_dump_cb (
            void(*)(void) wlan_usb_init_cb,
            int(*)() wlan_usb_mount_cb,
            int(*)(char *test_file_name) wlan_usb_file_open_cb,
            int(*)(uint8_t *data, size_t data_len) wlan_usb_file_write_cb,
            int(*)() wlan_usb_file_close_cb )
```

This function registers callbacks which are used to generate FW Dump on USB device.

Parameters

in	*wlan_usb_init_cb*	Callback to initialize usb device.
in	*wlan_usb_mount_cb*	Callback to mount usb device.
in	*wlan_usb_file_open_cb*	Callback to open file on usb device for FW dump.
in	*wlan_usb_file_write_cb*	Callback to write FW dump data to opened file.
in	*wlan_usb_file_close_cb*	Callback to close FW dump file.

5.8.3.154 wlan_set_crypto_RC4_encrypt()

```
int wlan_set_crypto_RC4_encrypt (
            const t_u8 * Key,
            const t_u16 KeyLength,
            const t_u8 * KeyIV,
            const t_u16 KeyIVLength,
            t_u8 * Data,
            t_u16 * DataLength )
```

Set Crypto RC4 algorithm encrypt command param.

Parameters

in	*Key*	key
in	*KeyLength*	The maximum key length is 32.
in	*KeyIV*	KeyIV
in	*KeyIVLength*	The maximum keyIV length is 32.
in	*Data*	Data
in	*DataLength*	The maximum Data length is 1300.

Returns

WM_SUCCESS if successful.
-WM_E_PERM if not supported.
-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by Data is overwritten by the encrypted data. The value of DataLength is updated to the encrypted data length. The length of the encrypted data is the same as the origin DataLength.

5.8.3.155 wlan_set_crypto_RC4_decrypt()

```
int wlan_set_crypto_RC4_decrypt (
            const t_u8 * Key,
            const t_u16 KeyLength,
            const t_u8 * KeyIV,
            const t_u16 KeyIVLength,
            t_u8 * Data,
            t_u16 * DataLength )
```

Set Crypto RC4 algorithm decrypt command param.

Parameters

in	*Key*	key
in	*KeyLength*	The maximum key length is 32.
in	*KeyIV*	KeyIV
in	*KeyIVLength*	The maximum keyIV length is 32.
in	*Data*	Data
in	*DataLength*	The maximum Data length is 1300.

Returns

WM_SUCCESS if successful.
-WM_E_PERM if not supported.
-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by Data is overwritten by the decrypted data. The value of DataLength is updated to the decrypted data length. The length of the decrypted data is the same as the origin DataLength.

5.8.3.156 wlan_set_crypto_AES_ECB_encrypt()

```
int wlan_set_crypto_AES_ECB_encrypt (
            const t_u8 * Key,
            const t_u16 KeyLength,
            const t_u8 * KeyIV,
            const t_u16 KeyIVLength,
            t_u8 * Data,
            t_u16 * DataLength )
```

Set Crypto AES_ECB algorithm encrypt command param.

Parameters

in	*Key*	key
in	*KeyLength*	The maximum key length is 32.
in	*KeyIV*	KeyIV
in	*KeyIVLength*	The maximum keyIV length is 32.
in	*Data*	Data
in	*DataLength*	The maximum Data length is 1300.

Returns

WM_SUCCESS if successful.
-WM_E_PERM if not supported.
-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by Data is overwritten by the encrypted data. The value of DataLength is updated to the encrypted data length. The length of the encrypted data is the same as the origin DataLength.

5.8.3.157 wlan_set_crypto_AES_ECB_decrypt()

```
int wlan_set_crypto_AES_ECB_decrypt (
             const t_u8 * Key,
             const t_u16 KeyLength,
             const t_u8 * KeyIV,
             const t_u16 KeyIVLength,
             t_u8 * Data,
             t_u16 * DataLength )
```

Set Crypto AES_ECB algorithm decrypt command param.

Parameters

in	*Key*	key
in	*KeyLength*	The maximum key length is 32.
in	*KeyIV*	KeyIV
in	*KeyIVLength*	The maximum keyIV length is 32.
in	*Data*	Data
in	*DataLength*	The maximum Data length is 1300.

Returns

WM_SUCCESS if successful.
-WM_E_PERM if not supported.
-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by Data is overwritten by the decrypted data. The value of DataLength is updated to the decrypted data length. The length of the decrypted data is the same as the origin DataLength.

5.8.3.158 wlan_set_crypto_AES_WRAP_encrypt()

```
int wlan_set_crypto_AES_WRAP_encrypt (
             const t_u8 * Key,
             const t_u16 KeyLength,
             const t_u8 * KeyIV,
             const t_u16 KeyIVLength,
             t_u8 * Data,
             t_u16 * DataLength )
```

Set Crypto AES_WRAP algorithm encrypt command param.

Parameters

in	*Key*	key
in	*KeyLength*	The maximum key length is 32.
in	*KeyIV*	KeyIV
in	*KeyIVLength*	The maximum keyIV length is 32.
in	*Data*	Data
in	*DataLength*	The maximum Data length is 1300.

Returns

WM_SUCCESS if successful.
-WM_E_PERM if not supported.
-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by Data is overwritten by the encrypted data. The value of DataLength is updated to the encrypted data length. The encrypted data is 8 bytes more than the original data. Therefore, the address pointed to by Data needs to reserve enough space.

5.8.3.159 wlan_set_crypto_AES_WRAP_decrypt()

```
int wlan_set_crypto_AES_WRAP_decrypt (
            const t_u8 * Key,
            const t_u16 KeyLength,
            const t_u8 * KeyIV,
            const t_u16 KeyIVLength,
            t_u8 * Data,
            t_u16 * DataLength )
```

Set Crypto AES_WRAP algorithm decrypt command param.

Parameters

in	*Key*	key
in	*KeyLength*	The maximum key length is 32.
in	*KeyIV*	KeyIV
in	*KeyIVLength*	The maximum keyIV length is 32.
in	*Data*	Data
in	*DataLength*	The maximum Data length is 1300.

Returns

WM_SUCCESS if successful.
-WM_E_PERM if not supported.
-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by Data is overwritten by the decrypted data. The value of DataLength is updated to the decrypted data length. The decrypted data is 8 bytes less than the original data.

5.8.3.160 wlan_set_crypto_AES_CCMP_encrypt()

```
int wlan_set_crypto_AES_CCMP_encrypt (
            const t_u8 * Key,
            const t_u16 KeyLength,
            const t_u8 * AAD,
            const t_u16 AADLength,
            const t_u8 * Nonce,
            const t_u16 NonceLength,
            t_u8 * Data,
            t_u16 * DataLength )
```

Set Crypto AES_CCMP algorithm encrypt command param.

Parameters

in	*Key*	key
in	*KeyLength*	The maximum key length is 32.
in	*AAD*	AAD
in	*AADLength*	The maximum AAD length is 32.
in	*Nonce*	Nonce
in	*NonceLength*	The maximum Nonce length is 14.
in	*Data*	Data
in	*DataLength*	The maximum Data length is 1300.

Returns

WM_SUCCESS if successful.
-WM_E_PERM if not supported.
-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by Data is overwritten by the encrypted data. The value of DataLength is updated to the encrypted data length. The encrypted data is 8 or 16 bytes more than the original data. Therefore, the address pointed to by Data needs to reserve enough space.

5.8.3.161 wlan_set_crypto_AES_CCMP_decrypt()

```
int wlan_set_crypto_AES_CCMP_decrypt (
            const t_u8 * Key,
```

```
              const t_u16 KeyLength,
              const t_u8 * AAD,
              const t_u16 AADLength,
              const t_u8 * Nonce,
              const t_u16 NonceLength,
              t_u8 * Data,
              t_u16 * DataLength )
```

Set Crypto AES_CCMP algorithm decrypt command param.

Parameters

in	*Key*	key
in	*KeyLength*	The maximum key length is 32.
in	*AAD*	AAD
in	*AADLength*	The maximum AAD length is 32.
in	*Nonce*	Nonce
in	*NonceLength*	The maximum Nonce length is 14.
in	*Data*	Data
in	*DataLength*	The maximum Data length is 1300.

Returns

WM_SUCCESS if successful.
-WM_E_PERM if not supported.
-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by Data is overwritten by the decrypted data. The value of DataLength is updated to the decrypted data length. The decrypted data is 8 or 16 bytes less than the original data.

5.8.3.162 wlan_set_crypto_AES_GCMP_encrypt()

```
int wlan_set_crypto_AES_GCMP_encrypt (
              const t_u8 * Key,
              const t_u16 KeyLength,
              const t_u8 * AAD,
              const t_u16 AADLength,
              const t_u8 * Nonce,
              const t_u16 NonceLength,
              t_u8 * Data,
              t_u16 * DataLength )
```

Set Crypto AES_GCMP algorithm encrypt command param.

Parameters

in	*Key*	key
in	*KeyLength*	The maximum key length is 32.

Parameters

in	*AAD*	AAD
in	*AADLength*	The maximum AAD length is 32.
in	*Nonce*	Nonce
in	*NonceLength*	The maximum Nonce length is 14.
in	*Data*	Data
in	*DataLength*	The maximum Data length is 1300.

Returns

WM_SUCCESS if successful.
-WM_E_PERM if not supported.
-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by Data is overwritten by the encrypted data. The value of DataLength is updated to the encrypted data length. The encrypted data is 16 bytes more than the original data. Therefore, the address pointed to by Data needs to reserve enough space.

5.8.3.163 wlan_set_crypto_AES_GCMP_decrypt()

```
int wlan_set_crypto_AES_GCMP_decrypt (
            const t_u8 * Key,
            const t_u16 KeyLength,
            const t_u8 * AAD,
            const t_u16 AADLength,
            const t_u8 * Nonce,
            const t_u16 NonceLength,
            t_u8 * Data,
            t_u16 * DataLength )
```

Set Crypto AES_CCMP algorithm decrypt command param.

Parameters

in	*Key*	key
in	*KeyLength*	The maximum key length is 32.
in	*AAD*	AAD
in	*AADLength*	The maximum AAD length is 32.
in	*Nonce*	Nonce
in	*NonceLength*	The maximum Nonce length is 14.
in	*Data*	Data
in	*DataLength*	The maximum Data length is 1300.

Returns

WM_SUCCESS if successful.
-WM_E_PERM if not supported.
-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by Data is overwritten by the decrypted data. The value of DataLength is updated to the decrypted data length. The decrypted data is 16 bytes less than the original data.

5.8.3.164 wlan_send_hostcmd()

```
int wlan_send_hostcmd (
            const void * cmd_buf,
            uint32_t cmd_buf_len,
            void * host_resp_buf,
            uint32_t resp_buf_len,
            uint32_t * reqd_resp_len )
```

This function sends the host command to f/w and copies back response to caller provided buffer in case of success Response from firmware is not parsed by this function but just copied back to the caller buffer.

Parameters

in	*cmd_buf*	Buffer containing the host command with header
in	*cmd_buf_len*	length of valid bytes in cmd_buf
out	*host_resp_buf*	Caller provided buffer, in case of success command response is copied to this buffer Can be same as cmd_buf
in	*resp_buf_len*	resp_buf's allocated length
out	*reqd_resp_len*	length of valid bytes in response buffer if successful otherwise invalid.

Returns

WM_SUCCESS in case of success.
WM_E_INBIG in case cmd_buf_len is bigger than the commands that can be handled by driver.
WM_E_INSMALL in case cmd_buf_len is smaller than the minimum length. Minimum length is atleast the length of command header. Please see Note for same.
WM_E_OUTBIG in case the resp_buf_len is not sufficient to copy response from firmware. reqd_resp_len is updated with the response size.
WM_E_INVAL in case cmd_buf_len and resp_buf_len have invalid values.
WM_E_NOMEM in case cmd_buf, resp_buf and reqd_resp_len are NULL

Note

Brief on the Command Header: Start 8 bytes of cmd_buf should have these values set. Firmware would update resp_buf with these 8 bytes at the start.
2 bytes : Command.
2 bytes : Size.

2 bytes : Sequence number.
2 bytes : Result.
Rest of buffer length is Command/Response Body.

5.8.3.165 wlan_send_debug_htc()

```
int wlan_send_debug_htc (
            const uint8_t count,
            const uint8_t vht,
            const uint8_t he,
            const uint8_t rxNss,
            const uint8_t channelWidth,
            const uint8_t ulMuDisable,
            const uint8_t txNSTS,
            const uint8_t erSuDisable,
            const uint8_t dlResoundRecomm,
            const uint8_t ulMuDataDisable )
```

This function is used to set HTC parameter.

Parameters

in	*count*	
in	*vht*	
in	*he*	
in	*rxNss*	
in	*channelWidth*	
in	*ulMuDisable*	
in	*txNSTS*	
in	*erSuDisable*	
in	*dlResoundRecomm*	
in	*ulMuDataDisable*	

Returns

WM_SUCCESS if operation is successful, otherwise failure

5.8.3.166 wlan_enable_disable_htc()

```
int wlan_enable_disable_htc (
            uint8_t option )
```

This function is used to enable/disable HTC.

Parameters

in	*option*	1 => Enable; 0 => Disable

Returns

WM_SUCCESS if operation is successful, otherwise failure

5.8.3.167 wlan_set_11ax_tx_omi()

```
int wlan_set_11ax_tx_omi (
            const t_u8 interface,
            const t_u16 tx_omi,
            const t_u8 tx_option,
            const t_u8 num_data_pkts )
```

Use this API to set the set 11AX Tx OMI.

Parameters

in	*interface*	Interface type STA or uAP.
in	*tx_omi*	value to be sent to Firmware
in	*tx_option*	value to be sent to Firmware 1: send OMI in QoS data.
in	*num_data_pkts*	value to be sent to Firmware num_data_pkts is applied only if OMI is sent in QoS data frame. It specifies the number of consecutive data frames containing the OMI. Minimum value is 1 Maximum value is 16

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

5.8.3.168 wlan_set_11ax_tol_time()

```
int wlan_set_11ax_tol_time (
            const t_u32 tol_time )
```

Set 802_11 AX OBSS Narrow Bandwidth RU Tolerance Time In uplink transmission, AP sends a trigger frame to all the stations that will be involved in the upcoming transmission, and then these stations transmit Trigger-based(TB) PPDU in response to the trigger frame. If STA connects to AP which channel is set to 100,STA doesn't support 26 tones RU. The API should be called when station is in disconnected state.

Parameters

in	*tol_time*	Valid range [1...3600] tolerance time is in unit of seconds. STA periodically check AP's beacon for ext cap bit79 (OBSS Narrow bandwidth RU in ofdma tolerance support) and set 20 tone RU tolerance time if ext cap bit79 is not set

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.169 wlan_set_11ax_rutxpowerlimit()

```
int wlan_set_11ax_rutxpowerlimit (
            const void * rutx_pwr_cfg,
            uint32_t rutx_pwr_cfg_len )
```

Use this API to set the RU tx power limit.

Parameters

in	*rutx_pwr_cfg*	11AX rutxpwr of sub-bands to be sent to Firmware.
in	*rutx_pwr_cfg_len*	Size of rutx_pwr_cfg buffer.

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

5.8.3.170 wlan_set_11ax_rutxpowerlimit_legacy()

```
int wlan_set_11ax_rutxpowerlimit_legacy (
            const wlan_rutxpwrlimit_t * ru_pwr_cfg )
```

Use this API to set the RU tx power limit by channel based approach.

Parameters

in	*ru_pwr_cfg*	11AX rutxpwr of channels to be sent to Firmware.

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

5.8.3.171 wlan_get_11ax_rutxpowerlimit_legacy()

```
int wlan_get_11ax_rutxpowerlimit_legacy (
            wlan_rutxpwrlimit_t * ru_pwr_cfg )
```

Use this API to get the RU tx power limit by channel based approach.

Parameters

in	*ru_pwr_cfg*	11AX rutxpwr of channels to be get from Firmware

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

5.8.3.172 wlan_set_11ax_cfg()

```
int wlan_set_11ax_cfg (
            wlan_11ax_config_t * ax_config )
```

Set 11ax config params

Parameters

in,out	*ax_config*	11AX config parameters to be sent to Firmware

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.173 wlan_get_11ax_cfg()

```
uint8_t* wlan_get_11ax_cfg ( )
```

Get default 11ax config params

Returns

11AX config parameters default array.

5.8.3.174 wlan_set_btwt_cfg()

```
int wlan_set_btwt_cfg (
            const wlan_btwt_config_t * btwt_config )
```

Set btwt config params

Parameters

in	*btwt_config*	Broadcast TWT Setup parameters to be sent to Firmware

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.175 wlan_get_btwt_cfg()

```
uint8_t* wlan_get_btwt_cfg ( )
```

Get btwt config params

Returns

Broadcast TWT Setup parameters default config array.

5.8.3.176 wlan_set_twt_setup_cfg()

```
int wlan_set_twt_setup_cfg (
            const wlan_twt_setup_config_t * twt_setup )
```

Set twt setup config params

Parameters

in	*twt_setup*	TWT Setup parameters to be sent to Firmware

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.177 wlan_get_twt_setup_cfg()

```
uint8_t* wlan_get_twt_setup_cfg ( )
```

Get twt setup config params

Returns

TWT Setup parameters default array.

5.8.3.178 wlan_set_twt_teardown_cfg()

```
int wlan_set_twt_teardown_cfg (
            const wlan_twt_teardown_config_t * teardown_config )
```

Set twt teardown config params

Parameters

in	*teardown_config*	TWT Teardown parameters sent to Firmware

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.179 wlan_get_twt_teardown_cfg()

```
uint8_t* wlan_get_twt_teardown_cfg ( )
```

Get twt teardown config params

Returns

TWT Teardown parameters default array

5.8.3.180 wlan_get_twt_report()

```
int wlan_get_twt_report (
            wlan_twt_report_t * twt_report )
```

Get twt report

Parameters

out	*twt_report*	TWT Report parameter.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.181 wlan_set_clocksync_cfg()

```
int wlan_set_clocksync_cfg (
            const wlan_clock_sync_gpio_tsf_t * tsf_latch )
```

Set Clock Sync GPIO based TSF

Parameters

in	*tsf_latch*	Clock Sync TSF latch parameters to be sent to Firmware

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.182 wlan_get_tsf_info()

```
int wlan_get_tsf_info (
            wlan_tsf_info_t * tsf_info )
```

Get TSF info from firmware using GPIO latch

Parameters

out	*tsf_info*	TSF info parameter received from Firmware

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.183 wlan_show_os_mem_stat()

```
void wlan_show_os_mem_stat ( )
```

Show os mem alloc and free info.

5.8.3.184 wlan_ft_roam()

```
int wlan_ft_roam (
            const t_u8 * bssid,
            const t_u8 channel )
```

Start FT roaming : This API is used to initiate fast BSS transition based roaming.

Parameters

in	*bssid*	BSSID of AP to roam
in	*channel*	Channel of AP to roam

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.185 wlan_rx_mgmt_indication()

```
int wlan_rx_mgmt_indication (
            const enum wlan_bss_type bss_type,
            const uint32_t mgmt_subtype_mask,
            int(*)(const enum wlan_bss_type bss_type, const wlan_mgmt_frame_t *frame, const
size_t len) rx_mgmt_callback )
```

This API can be used to start/stop the management frame forwards to host through datapath.

Parameters

in	*bss_type*	The interface from which management frame needs to be collected 0: STA, 1: uAP
in	*mgmt_subtype_mask*	Management Subtype Mask If Bit X is set in mask, it means that IEEE Management Frame SubTyoe X is to be filtered and passed through to host. Bit Description [31:14] Reserved [13] Action frame [12:9] Reserved [8] Beacon [7:6] Reserved [5] Probe response [4] Probe request [3] Reassociation response [2] Reassociation request [1] Association response [0] Association request Support multiple bits set. 0 = stop forward frame 1 = start forward frame
in	*rx_mgmt_callback*	The receive callback where the received management frames are passed.

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

Note

Pass Management Subtype Mask all zero to disable all the management frame forward to host.

5.8.3.186 wlan_set_scan_channel_gap()

```
void wlan_set_scan_channel_gap (
            unsigned scan_chan_gap )
```

Set scan channel gap.

Parameters

in	*scan_chan_gap*	Time gap to be used between two consecutive channels scan.

5.8.3.187 wlan_host_11k_cfg()

```
int wlan_host_11k_cfg (
            int enable_11k )
```

enable/disable host 11k feature

Parameters

in	*enable_11k*	the value of 11k configuration.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.188 wlan_host_11k_neighbor_req()

```
int wlan_host_11k_neighbor_req (
            t_u8 * ssid )
```

host send neighbor report request

Parameters

in	*ssid*	the SSID for neighbor report

Note

ssid parameter is optional

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.189 wlan_host_11v_bss_trans_query()

```
int wlan_host_11v_bss_trans_query (
            t_u8 query_reason )
```

host send bss transition management query

Parameters

in	*query_reason*	BTM request query reason code

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.190 wlan_mbo_peferch_cfg()

```
int wlan_mbo_peferch_cfg (
            const char * non_pref_chan )
```

Multi Band Operation (MBO) non-preferred channels

A space delimited list of non-preferred channels where each channel is a colon delimited list of values.

Format:

non_pref_chan=oper_class:chan:preference:reason Example:

non_pref_chan=81:5:10:2 81:1:0:2 81:9:0:2

Parameters

in	*non_pref_chan*	list of non-preferred channels.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.191 wlan_mbo_set_cell_capa()

```
int wlan_mbo_set_cell_capa (
            t_u8 cell_capa )
```

MBO set Cellular Data Capabilities

Parameters

in	*cell_capa*	1 = Cellular data connection available 2 = Cellular data connection not available 3 = Not cellular capable (default)

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.192 wlan_mbo_set_oce()

```
int wlan_mbo_set_oce (
            t_u8 oce )
```

Optimized Connectivity Experience (OCE)

Parameters

in	*oce*	Enable OCE features 1 = Enable OCE in non-AP STA mode (default; disabled if the driver does not indicate support for OCE in STA mode). 2 = Enable OCE in STA-CFON mode.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.193 wlan_set_okc()

```
int wlan_set_okc (
            t_u8 okc )
```

Opportunistic Key Caching (also known as Proactive Key Caching) default This parameter can be used to set the default behavior for the proactive_key_caching parameter. By default, OKC is disabled unless enabled with the global okc=1 parameter or with the per-network pkc(proactive_key_caching)=1 parameter. With okc=1, OKC is enabled by default, but can be disabled with per-network pkc(proactive_key_caching)=0 parameter.

Parameters

in	*okc*	Enable Opportunistic Key Caching

0 = Disable OKC (default) 1 = Enable OKC

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.194 wlan_pmksa_list()

```
int wlan_pmksa_list (
            char * buf,
            size_t buflen )
```

Dump text list of entries in PMKSA cache

Parameters

out	*buf*	Buffer to save PMKSA cache text list
in	*buflen*	length of the buffer

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.195 wlan_pmksa_flush()

```
int wlan_pmksa_flush ( )
```

Flush PTKSA cache entries

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.196 wlan_set_scan_interval()

```
int wlan_set_scan_interval (
            int scan_int )
```

Set wpa supplicant scan interval in seconds

Parameters

in	*scan_int*	Scan interval in seconds

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.197 wlan_tx_ampdu_prot_mode()

```
int wlan_tx_ampdu_prot_mode (
            tx_ampdu_prot_mode_para * prot_mode,
            t_u16 action )
```

Set/Get Tx ampdu prot mode.

Parameters

in,out	*prot_mode*	Tx ampdu prot mode
in	*action*	Command action

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.198 wlan_mef_set_auto_arp()

```
int wlan_mef_set_auto_arp (
            t_u8 mef_action )
```

This function set auto ARP configuration.

Parameters

in	*mef_action*	To be 0–discard and not wake host, 1–discard and wake host 3–allow and wake host.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.199 wlan_mef_set_auto_ping()

```
int wlan_mef_set_auto_ping (
            t_u8 mef_action )
```

This function set auto ping configuration.

Parameters

in	*mef_action*	To be 0–discard and not wake host, 1–discard and wake host 3–allow and wake host.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.200 wlan_config_mef()

```
int wlan_config_mef (
            int type,
            t_u8 mef_action )
```

This function set/delete mef entries configuration.

Parameters

in	*type*	MEF type: MEF_TYPE_DELETE, MEF_TYPE_AUTO_PING, MEF_TYPE_AUTO_ARP
in	*mef_action*	To be 0–discard and not wake host, 1–discard and wake host 3–allow and wake host.

Returns

WM_SUCCESS if the call was successful.
-WM_FAIL if failed.

5.8.3.201 wlan_set_ipv6_ns_mef()

```
int wlan_set_ipv6_ns_mef (
            t_u8 mef_action )
```

Use this API to enable IPv6 Neighbor Solicitation offload in Wi-Fi firmware

Parameters

in	*mef_action*	0–discard and not wake host, 1–discard and wake host 3–allow and wake host.

Returns

WM_SUCCESS if operation is successful.
-WM_FAIL if command fails.

5.8.3.202 wlan_csi_cfg()

```
int wlan_csi_cfg (
            wlan_csi_config_params_t * csi_params )
```

Send the csi config parameter to FW.

Parameters

in	*csi_params*	Csi config parameter

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.203 wlan_register_csi_user_callback()

```
int wlan_register_csi_user_callback (
            int(*)(void *buffer, size_t len) csi_data_recv_callback )
```

This function registers callback which are used to deliver CSI data to user.

Parameters

in	*csi_data_recv_callback*	Callback to deliver CSI data and max data length is 768 bytes. Pls save data as soon as possible in callback Type of callback return vale is int. Memory layout of buffer: size(byte) items 2 buffer len bit 0:12 2 CSI signature, 0xABCD fixed 4 User defined HeaderID 2 Packet info 2 Frame control field for the received packet 8 Timestamp when packet received 6 Received Packet Destination MAC Address 6 Received Packet Source MAC Address 1 RSSI for antenna A 1 RSSI for antenna B 1 Noise floor for antenna A 1 Noise floor for antenna B 1 Rx signal strength above noise floor 1 Channel 2 user defined Chip ID 4 Reserved 4 CSI data length in DWORDs CSI data

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.204 wlan_unregister_csi_user_callback()

```
int wlan_unregister_csi_user_callback (
            void )
```

This function unregisters callback which are used to deliver CSI data to user.

Returns

WM_SUCCESS if successful

5.8.3.205 wlan_set_rssi_low_threshold()

```
void wlan_set_rssi_low_threshold (
            uint8_t threshold )
```

Use this API to set the RSSI threshold value for low RSSI event subscription. When RSSI falls below this threshold firmware will generate the low RSSI event to driver. This low RSSI event is used when either of CONFIG_11R, CONFIG_11K, CONFIG_11V or CONFIG_ROAMING is enabled. NOTE: By default rssi low threshold is set at -70 dbm

Parameters

in	*threshold*	Threshold rssi value to be set

5.8.3.206 wlan_wps_generate_pin()

```
void wlan_wps_generate_pin (
            uint32_t * pin )
```

Generate valid PIN for WPS session.

This function generate PIN for WPS PIN session.

Parameters

in	*pin*	A pointer to WPS pin to be generated.

5.8.3.207 wlan_start_wps_pin()

```
int wlan_start_wps_pin (
            const char * pin )
```

Start WPS PIN session.

This function starts WPS PIN session.

Parameters

in	*pin*	Pin for WPS session.

Returns

WM_SUCCESS if the pin entered is valid.
-WM_FAIL if invalid pin entered.

5.8.3.208 wlan_start_wps_pbc()

```
int wlan_start_wps_pbc (
            void )
```

Start WPS PBC session.

This function starts WPS PBC session.

Returns

WM_SUCCESS if successful
-WM_FAIL if invalid pin entered.

5.8.3.209 wlan_wps_cancel()

```
int wlan_wps_cancel (
            void )
```

Cancel WPS session.

This function cancels ongoing WPS session.

Returns

WM_SUCCESS if successful
-WM_FAIL if invalid pin entered.

5.8.3.210 wlan_start_ap_wps_pin()

```
int wlan_start_ap_wps_pin (
            const char * pin )
```

Start WPS PIN session.

This function starts AP WPS PIN session.

Parameters

in	*pin*	Pin for WPS session.

Returns

WM_SUCCESS if the pin entered is valid.
-WM_FAIL if invalid pin entered.

5.8.3.211 wlan_start_ap_wps_pbc()

```
int wlan_start_ap_wps_pbc (
            void )
```

Start WPS PBC session.

This function starts AP WPS PBC session.

Returns

WM_SUCCESS if successful
-WM_FAIL if invalid pin entered.

5.8.3.212 wlan_wps_ap_cancel()

```
int wlan_wps_ap_cancel (
            void )
```

Cancel AP's WPS session.

This function cancels ongoing WPS session.

Returns

WM_SUCCESS if successful
-WM_FAIL if invalid pin entered.

5.8.3.213 wlan_set_entp_cert_files()

```
int wlan_set_entp_cert_files (
            int cert_type,
            t_u8 * data,
            t_u32 data_len )
```

This function specifies the enterprise certificate file This function must be used before adding network profile. It will store certificate data in "wlan" global structure. When adding new network profile, it will be get by wlan_get_entp_cert_files(), and put into profile security structure after mbedtls parse.

Parameters

in	*cert_type*	certificate file type: 1 – FILE_TYPE_ENTP_CA_CERT, 2 – FILE_TYPE_ENTP_CLIENT_CERT, 3 – FILE_TYPE_ENTP_CLIENT_KEY.
in	*data*	raw data
in	*data_len*	size of raw data

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.214 wlan_get_entp_cert_files()

```
t_u32 wlan_get_entp_cert_files (
            int cert_type,
            t_u8 ** data )
```

This function get enterprise certificate data from "wlan" global structure *

Parameters

in	*cert_type*	certificate file type: 1 – FILE_TYPE_ENTP_CA_CERT, 2 – FILE_TYPE_ENTP_CLIENT_CERT, 3 – FILE_TYPE_ENTP_CLIENT_KEY.
in	*data*	raw data

Returns

size of raw data

5.8.3.215 wlan_free_entp_cert_files()

```
void wlan_free_entp_cert_files (
            void )
```

This function free the temporary memory of enterprise certificate data After add new enterprise network profile, the certificate data has been parsed by mbedtls into another data, which can be freed.

5.8.3.216 wlan_check_11n_capa()

```
uint8_t wlan_check_11n_capa (
            unsigned int channel )
```

Check if 11n(2G or 5G) is supported by hardware or not.

Parameters

in	*channel*	Channel number.

Returns

true if 11n is supported or false if not.

5.8.3.217 wlan_check_11ac_capa()

```
uint8_t wlan_check_11ac_capa (
            unsigned int channel )
```

Check if 11ac(2G or 5G) is supported by hardware or not.

Parameters

in	*channel*	Channel number.

Returns

true if 11ac is supported or false if not.

5.8.3.218 wlan_check_11ax_capa()

```
uint8_t wlan_check_11ax_capa (
            unsigned int channel )
```

Check if 11ax(2G or 5G) is supported by hardware or not.

Parameters

in	*channel*	Channel number.

Returns

true if 11ax is supported or false if not.

5.8.3.219 wlan_get_signal_info()

```
int wlan_get_signal_info (
            wlan_rssi_info_t * signal )
```

Get rssi information.

Parameters

out	*signal*	rssi infomation get report buffer

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.220 wlan_set_rg_power_cfg()

```
int wlan_set_rg_power_cfg (
            t_u16 region_code )
```

set region power table

Parameters

in	*region_code*	region code

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.221 wlan_get_turbo_mode()

```
int wlan_get_turbo_mode (
            t_u8 * mode )
```

Get Turbo mode.

Parameters

out	*mode*	turbo mode 0: disable turbo mode 1: turbo mode 1 2: turbo mode 2 3: turbo mode 3

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.222 wlan_get_uap_turbo_mode()

```
int wlan_get_uap_turbo_mode (
            t_u8 * mode )
```

Get UAP Turbo mode.

Parameters

out	*mode*	turbo mode 0: disable turbo mode 1: turbo mode 1 2: turbo mode 2 3: turbo mode 3

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.223 wlan_set_turbo_mode()

```
int wlan_set_turbo_mode (
            t_u8 mode )
```

Set Turbo mode.

Parameters

in	*mode*	turbo mode 0: disable turbo mode 1: turbo mode 1 2: turbo mode 2 3: turbo mode 3

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.224 wlan_set_uap_turbo_mode()

```
int wlan_set_uap_turbo_mode (
            t_u8 mode )
```

Set UAP Turbo mode.

Parameters

in	*mode*	turbo mode 0: disable turbo mode 1: turbo mode 1 2: turbo mode 2 3: turbo mode 3

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.225 wlan_set_ps_cfg()

```
void wlan_set_ps_cfg (
            t_u16 multiple_dtims,
            t_u16 bcn_miss_timeout,
            t_u16 local_listen_interval,
            t_u16 adhoc_wake_period,
            t_u16 mode,
            t_u16 delay_to_ps )
```

set ps configuration. Currently only used to modify multiple dtim.

Parameters

in	*multiple_dtims*	num dtims, range [1,20]
in	*bcn_miss_timeout*	becaon miss interval
in	*local_listen_interval*	local listen interval
in	*adhoc_wake_period*	adhoc awake period
in	*mode*	mode - (0x01 - firmware to automatically choose PS_POLL or NULL mode, 0x02 - PS_POLL, 0x03 - NULL mode)
in	*delay_to_ps*	Delay to PS in milliseconds

5.8.3.226 wlan_save_cloud_keep_alive_params()

```
int wlan_save_cloud_keep_alive_params (
            wlan_cloud_keep_alive_t * cloud_keep_alive,
            t_u16 src_port,
            t_u16 dst_port,
            t_u32 seq_number,
            t_u32 ack_number,
            t_u8 enable )
```

Save start cloud keep alive parameters

Parameters

in	*cloud_keep_alive*	cloud keep alive information
in	*src_port*	Source port
in	*dst_port*	Destination port
in	*seq_number*	Sequence number
in	*ack_number*	Acknowledgement number
in	*enable*	Enable

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.227 wlan_cloud_keep_alive_enabled()

```
int wlan_cloud_keep_alive_enabled (
            t_u32 dst_ip,
            t_u16 dst_port )
```

Get cloud keep alive status for given destination ip and port

Parameters

in	*dst_ip*	Destination ip address
in	*dst_port*	Destination port

Returns

1 if enabled otherwise 0.

5.8.3.228 wlan_start_cloud_keep_alive()

```
int wlan_start_cloud_keep_alive (
            void )
```

Start cloud keep alive

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.229 wlan_stop_cloud_keep_alive()

```
int wlan_stop_cloud_keep_alive (
            wlan_cloud_keep_alive_t * cloud_keep_alive )
```

Stop cloud keep alive

Parameters

in	*cloud_keep_alive*	cloud keep alive information

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.230 wlan_set_country_code()

```
int wlan_set_country_code (
            const char * alpha2 )
```

Set country code

Note

This API should be called after WLAN is initialized but before starting uAP interface.

Parameters

in	*alpha2*	country code in 3 octets string, 2 octets country code and 1 octet environment 2 octets country code supported: WW : World Wide Safe US : US FCC CA : IC Canada SG : Singapore EU : ETSI AU : Australia KR : Republic Of Korea FR : France JP : Japan CN : China

For the third octet, STA is always 0. For uAP environment: All environments of the current frequency band and country (default) alpha2[2]=0x20 Outdoor environment only alpha2[2]=0x4f Indoor environment only alpha2[2]=0x49 Noncountry entity (country_code=XX) alpha[2]=0x58 IEEE 802.11 standard Annex E table indication: 0x01 .. 0x1f Annex E, Table E-4 (Global operating classes) alpha[2]=0x04

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.231 wlan_set_country_ie_ignore()

```
int wlan_set_country_ie_ignore (
            uint8_t * ignore )
```

Set ignore region code

Parameters

in	*ignore*	0: Don't ignore 1: ignore

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.232 wlan_set_region_code()

```
int wlan_set_region_code (
            unsigned int region_code )
```

Set region code

Parameters

in	*region_code*	

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.233 wlan_get_region_code()

```
int wlan_get_region_code (
            unsigned int * region_code )
```

Get region code

Parameters

out	*region_code*	pointer

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.234 wlan_set_11d_state()

```
int wlan_set_11d_state (
            int bss_type,
            int state )
```

Set STA/uAP 80211d feature enable/disable

Parameters

in	*bss_type*	0: STA, 1: uAP
in	*state*	0: disable, 1: enable

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.235 wlan_set_indrst_cfg()

```
int wlan_set_indrst_cfg (
            const wifi_indrst_cfg_t * indrst_cfg )
```

Set GPIO independent reset configuration

Parameters

in	*indrst_cfg*	GPIO independent reset config to be sent to Firmware

Returns

WM_SUCCESS if successful otherwise failure.

5.8.3.236 wlan_independent_reset()

```
int wlan_independent_reset ( )
```

Test Independent Firmware reset

This function will either send cmd that will cause timeout in firmware or send GPIO pulse that will cause out of band reset in firmware as per configuration int earlier wlan_set_indrst_cfg API.

Returns

WM_SUCCESS if successful otherwise failure.

5.8.4 Macro Documentation

5.8.4.1 ACTION_GET

```
#define ACTION_GET (0U)
```

Action GET

5.8.4.2 ACTION_SET

```
#define ACTION_SET (1)
```

Action SET

5.8.4.3 IEEEtypes_SSID_SIZE

```
#define IEEEtypes_SSID_SIZE 32U
```

Maximum SSID length

5.8.4.4 IEEEtypes_ADDRESS_SIZE

```
#define IEEEtypes_ADDRESS_SIZE 6
```

MAC Address length

5.8.4.5 WLAN_RESCAN_LIMIT

```
#define WLAN_RESCAN_LIMIT 30U
```

The number of times that the WLAN Connection Manager will look for a network before giving up.

5.8.4.6 WLAN_RECONNECT_LIMIT

```
#define WLAN_RECONNECT_LIMIT 5U
```

The number of times that the WLAN Connection Manager will attempt a reconnection with the network before giving up.

5.8.4.7 WLAN_NETWORK_NAME_MIN_LENGTH

```
#define WLAN_NETWORK_NAME_MIN_LENGTH 1U
```

The minimum length for network names, see wlan_network. This must be between 1 and WLAN_NETWORK_NAME_MAX_LENGTH

5.8.4.8 WLAN_NETWORK_NAME_MAX_LENGTH

```
#define WLAN_NETWORK_NAME_MAX_LENGTH 32U
```

The space reserved for storing network names, wlan_network

5.8.4.9 WLAN_PSK_MIN_LENGTH

```
#define WLAN_PSK_MIN_LENGTH 8U
```

The space reserved for storing PSK (password) phrases.

5.8.4.10 WLAN_PSK_MAX_LENGTH

```
#define WLAN_PSK_MAX_LENGTH 65U
```

Max WPA2 passphrase can be upto 63 ASCII chars or 64 hexadecimal digits

5.8.4.11 WLAN_PASSWORD_MIN_LENGTH

```
#define WLAN_PASSWORD_MIN_LENGTH 8U
```

Min WPA3 password can be upto 8 ASCII chars

5.8.4.12 WLAN_PASSWORD_MAX_LENGTH

```
#define WLAN_PASSWORD_MAX_LENGTH 255U
```

Max WPA3 password can be upto 255 ASCII chars

5.8.4.13 IDENTITY_MAX_LENGTH

```
#define IDENTITY_MAX_LENGTH 64U
```

Max WPA2 Enterprise identity can be upto 256 characters

5.8.4.14 PASSWORD_MAX_LENGTH

```
#define PASSWORD_MAX_LENGTH 128U
```

Max WPA2 Enterprise password can be upto 256 unicode characters

5.8.4.15 MAX_USERS

```
#define MAX_USERS 8U
```

Max identities for EAP server users

5.8.4.16 PAC_OPAQUE_ENCR_KEY_MAX_LENGTH

```
#define PAC_OPAQUE_ENCR_KEY_MAX_LENGTH 33U
```

Encryption key for EAP-FAST PAC-Opaque values. This key must be a secret, random value. It is configured as a 16-octet value in hex format.

5.8.4.17 A_ID_MAX_LENGTH

```
#define A_ID_MAX_LENGTH 33U
```

A-ID indicates the identity of the authority that issues PACs. The A-ID should be unique across all issuing servers. A-ID to be 16 octets in length

5.8.4.18 HASH_MAX_LENGTH

```
#define HASH_MAX_LENGTH 40U
```

MAX CA Cert hash len

5.8.4.19 DOMAIN_MATCH_MAX_LENGTH

```
#define DOMAIN_MATCH_MAX_LENGTH 64U
```

MAX domain len

5.8.4.20 WLAN_MAX_KNOWN_NETWORKS

```
#define WLAN_MAX_KNOWN_NETWORKS CONFIG_WLAN_KNOWN_NETWORKS
```

The size of the list of known networks maintained by the WLAN Connection Manager

5.8.4.21 WLAN_PMK_LENGTH

```
#define WLAN_PMK_LENGTH 32
```

Length of a pairwise master key (PMK). It's always 256 bits (32 Bytes)

5.8.4.22 WLAN_ERROR_NONE

```
#define WLAN_ERROR_NONE 0
```

The operation was successful.

5.8.4.23 WLAN_ERROR_PARAM

```
#define WLAN_ERROR_PARAM 1
```

The operation failed due to an error with one or more parameters.

5.8.4.24 WLAN_ERROR_NOMEM

```
#define WLAN_ERROR_NOMEM 2
```

The operation could not be performed because there is not enough memory.

5.8.4.25 WLAN_ERROR_STATE

```
#define WLAN_ERROR_STATE 3
```

The operation could not be performed in the current system state.

5.8.4.26 WLAN_ERROR_ACTION

```
#define WLAN_ERROR_ACTION 4
```

The operation failed due to an internal error.

5.8.4.27 WLAN_ERROR_PS_ACTION

```
#define WLAN_ERROR_PS_ACTION 5
```

The operation to change power state could not be performed

5.8.4.28 WLAN_ERROR_NOT_SUPPORTED

```
#define WLAN_ERROR_NOT_SUPPORTED 6
```

The requested feature is not supported

5.8.4.29 WLAN_MGMT_ACTION

```
#define WLAN_MGMT_ACTION MBIT(13)
```

BITMAP for Action frame

5.8.4.30 WLAN_KEY_MGMT_FT

```
#define WLAN_KEY_MGMT_FT
```

Value:

```
(WLAN_KEY_MGMT_FT_PSK | WLAN_KEY_MGMT_FT_IEEE8021X | WLAN_KEY_MGMT_FT_IEEE8021X_SHA384 |
      WLAN_KEY_MGMT_FT_SAE | \
     WLAN_KEY_MGMT_FT_FILS_SHA256 | WLAN_KEY_MGMT_FT_FILS_SHA384)
```

5.8.5 Typedef Documentation

5.8.5.1 wlan_scan_channel_list_t

```
typedef wifi_scan_channel_list_t wlan_scan_channel_list_t
```

Configuration for Wireless scan channel list from wifi_scan_channel_list_t

5.8.5.2 wlan_scan_params_v2_t

```
typedef wifi_scan_params_v2_t wlan_scan_params_v2_t
```

Configuration for wireless scanning parameters v2 from wifi_scan_params_v2_t

5.8.5.3 wlan_cal_data_t

```
typedef wifi_cal_data_t wlan_cal_data_t
```

Configuration for Wireless Calibration data from wifi_cal_data_t

5.8.5.4 wlan_auto_reconnect_config_t

```
typedef wifi_auto_reconnect_config_t wlan_auto_reconnect_config_t
```

Configuration for Auto reconnect configuration from wifi_auto_reconnect_config_t

5.8.5.5 wlan_flt_cfg_t

```
typedef wifi_flt_cfg_t wlan_flt_cfg_t
```

Configuration for Memory Efficient Filters in Wi-Fi firmware from wifi_flt_cfg_t

5.8.5.6 wlan_wowlan_ptn_cfg_t

```
typedef wifi_wowlan_ptn_cfg_t wlan_wowlan_ptn_cfg_t
```

Configuration for wowlan pattern parameters from wifi_wowlan_ptn_cfg_t

5.8.5.7 wlan_tcp_keep_alive_t

```
typedef wifi_tcp_keep_alive_t wlan_tcp_keep_alive_t
```

Configuration for TCP Keep alive parameters from wifi_tcp_keep_alive_t

5.8.5.8 wlan_cloud_keep_alive_t

```
typedef wifi_cloud_keep_alive_t wlan_cloud_keep_alive_t
```

Configuration for Cloud Keep alive parameters from wifi_cloud_keep_alive_t

5.8.5.9 wlan_ds_rate

```
typedef wifi_ds_rate wlan_ds_rate
```

Configuration for TX Rate and Get data rate from wifi_ds_rate

5.8.5.10 wlan_ed_mac_ctrl_t

```
typedef wifi_ed_mac_ctrl_t wlan_ed_mac_ctrl_t
```

Configuration for ED MAC Control parameters from wifi_ed_mac_ctrl_t

5.8.5.11 wlan_bandcfg_t

```
typedef wifi_bandcfg_t wlan_bandcfg_t
```

Configuration for Band from wifi_bandcfg_t

5.8.5.12 wlan_cw_mode_ctrl_t

```
typedef wifi_cw_mode_ctrl_t wlan_cw_mode_ctrl_t
```

Configuration for CW Mode parameters from wifi_cw_mode_ctrl_t

5.8.5.13 wlan_chanlist_t

```
typedef wifi_chanlist_t wlan_chanlist_t
```

Configuration for Channel list from wifi_chanlist_t

5.8.5.14 wlan_txpwrlimit_t

```
typedef wifi_txpwrlimit_t wlan_txpwrlimit_t
```

Configuration for TX Pwr Limit from wifi_txpwrlimit_t

5.8.5.15 wlan_ext_coex_stats_t

```
typedef wifi_ext_coex_stats_t wlan_ext_coex_stats_t
```

Statistic of External Coex from wifi_ext_coex_config_t

5.8.5.16 wlan_ext_coex_config_t

```
typedef wifi_ext_coex_config_t wlan_ext_coex_config_t
```

Configuration for External Coex from wifi_ext_coex_config_t

5.8.5.17 wlan_rutxpwrlimit_t

```
typedef wifi_rutxpwrlimit_t wlan_rutxpwrlimit_t
```

Configuration for RU TX Pwr Limit from wifi_rutxpwrlimit_t

5.8.5.18 wlan_11ax_config_t

```
typedef wifi_11ax_config_t wlan_11ax_config_t
```

Configuration for 11AX capabilities wifi_11ax_config_t

5.8.5.19 wlan_twt_setup_config_t

```
typedef wifi_twt_setup_config_t wlan_twt_setup_config_t
```

Configuration for TWT Setup wifi_twt_setup_config_t

5.8.5.20 wlan_twt_teardown_config_t

```
typedef wifi_twt_teardown_config_t wlan_twt_teardown_config_t
```

Configuration for TWT Teardown wifi_twt_teardown_config_t

5.8.5.21 wlan_btwt_config_t

```
typedef wifi_btwt_config_t wlan_btwt_config_t
```

Configuration for Broadcast TWT Setup wifi_btwt_config_t

5.8.5.22 wlan_twt_report_t

```
typedef wifi_twt_report_t wlan_twt_report_t
```

Configuration for TWT Report wifi_twt_report_t

5.8.5.23 wlan_clock_sync_gpio_tsf_t

```
typedef wifi_clock_sync_gpio_tsf_t wlan_clock_sync_gpio_tsf_t
```

Configuration for Clock Sync GPIO TSF latch wifi_clock_sync_gpio_tsf_t

5.8.5.24 wlan_tsf_info_t

```
typedef wifi_tsf_info_t wlan_tsf_info_t
```

Configuration for TSF info wifi_tsf_info_t

5.8.5.25 wlan_csi_config_params_t

```
typedef wifi_csi_config_params_t wlan_csi_config_params_t
```

Configuration for Csi Config Params from wifi_csi_config_params_t

5.8.5.26 wlan_indrst_cfg_t

```
typedef wifi_indrst_cfg_t wlan_indrst_cfg_t
```

Configuration for GPIO independent reset wifi_indrst_cfg_t

5.8.5.27 wlan_txrate_setting

```
typedef txrate_setting wlan_txrate_setting
```

Configuration for TX Rate Setting from txrate_setting

5.8.5.28 wlan_rssi_info_t

```
typedef wifi_rssi_info_t wlan_rssi_info_t
```

Configuration for RSSI information wifi_rssi_info_t

5.8.6 Enumeration Type Documentation

5.8.6.1 wm_wlan_errno

```
enum wm_wlan_errno
```

Enum for wlan errors

Enumerator

WLAN_ERROR_FW_DNLD_FAILED	The Firmware download operation failed.
WLAN_ERROR_FW_NOT_READY	The Firmware ready register not set.
WLAN_ERROR_CARD_NOT_DETECTED	The WiFi card not found.
WLAN_ERROR_FW_NOT_DETECTED	The WiFi Firmware not found.
WLAN_BSSID_NOT_FOUND_IN_SCAN_LIST	BSSID not found in scan list

5.8.6.2 wlan_event_reason

```
enum wlan_event_reason
```

WLAN Connection Manager event reason

Enumerator

WLAN_REASON_SUCCESS	The WLAN Connection Manager has successfully connected to a network and is now in the WLAN_CONNECTED state.
WLAN_REASON_AUTH_SUCCESS	The WLAN Connection Manager has successfully authenticated to a network and is now in the WLAN_ASSOCIATED state.
WLAN_REASON_CONNECT_FAILED	The WLAN Connection Manager failed to connect before actual connection attempt with AP due to incorrect wlan network profile. or The WLAN Connection Manager failed to reconnect to previously connected network and it is now in the WLAN_DISCONNECTED state.
WLAN_REASON_NETWORK_NOT_FOUND	The WLAN Connection Manager could not find the network that it was connecting to and it is now in the WLAN_DISCONNECTED state.

Enumerator

WLAN_REASON_BGSCAN_NETWORK_NOT_FOUND	The WLAN Connection Manager could not find the network in bg scan during roam attempt that it was connecting to and it is now in the WLAN_CONNECTED state with previous AP.
WLAN_REASON_NETWORK_AUTH_FAILED	The WLAN Connection Manager failed to authenticate with the network and is now in the WLAN_DISCONNECTED state.
WLAN_REASON_ADDRESS_SUCCESS	DHCP lease has been renewed.
WLAN_REASON_ADDRESS_FAILED	The WLAN Connection Manager failed to obtain an IP address or TCP stack configuration has failed or the IP address configuration was lost due to a DHCP error. The system is now in the WLAN_DISCONNECTED state.
WLAN_REASON_LINK_LOST	The WLAN Connection Manager has lost the link to the current network.
WLAN_REASON_CHAN_SWITCH	The WLAN Connection Manager has received the channel switch announcement from the current network.
WLAN_REASON_WPS_DISCONNECT	The WLAN Connection Manager has disconnected from the WPS network (or has canceled a connection attempt) by request and is now in the WLAN_DISCONNECTED state.
WLAN_REASON_USER_DISCONNECT	The WLAN Connection Manager has disconnected from the current network (or has canceled a connection attempt) by request and is now in the WLAN_DISCONNECTED state.
WLAN_REASON_INITIALIZED	The WLAN Connection Manager is initialized and is ready for use. That is, it's now possible to scan or to connect to a network.
WLAN_REASON_INITIALIZATION_FAILED	The WLAN Connection Manager has failed to initialize and is therefore not running. It is not possible to scan or to connect to a network. The WLAN Connection Manager should be stopped and started again via wlan_stop() and wlan_start() respectively.
WLAN_REASON_FW_HANG	The WLAN Connection Manager has entered in hang mode.
WLAN_REASON_FW_RESET	The WLAN Connection Manager has reset fw successfully.
WLAN_REASON_PS_ENTER	The WLAN Connection Manager has entered power save mode.
WLAN_REASON_PS_EXIT	The WLAN Connection Manager has exited from power save mode.
WLAN_REASON_UAP_SUCCESS	The WLAN Connection Manager has started uAP
WLAN_REASON_UAP_CLIENT_ASSOC	A wireless client has joined uAP's BSS network
WLAN_REASON_UAP_CLIENT_CONN	A wireless client has auhtenticated and connected to uAP's BSS network
WLAN_REASON_UAP_CLIENT_DISSOC	A wireless client has left uAP's BSS network
WLAN_REASON_UAP_START_FAILED	The WLAN Connection Manager has failed to start uAP
WLAN_REASON_UAP_STOP_FAILED	The WLAN Connection Manager has failed to stop uAP
WLAN_REASON_UAP_STOPPED	The WLAN Connection Manager has stopped uAP

Enumerator

WLAN_REASON_RSSI_LOW	The WLAN Connection Manager has received subscribed RSSI low event on station interface as per configured threshold and frequency. If CONFIG_11K, CONFIG_11V, CONFIG_11R or CONFIG_ROAMING enabled then RSSI low event is processed internally.

5.8.6.3 wlan_wakeup_event_t

```
enum wlan_wakeup_event_t
```

Wakeup events for which wakeup will occur

Enumerator

WAKE_ON_ALL_BROADCAST	Wakeup on broadcast
WAKE_ON_UNICAST	Wakeup on unicast
WAKE_ON_MAC_EVENT	Wakeup on MAC event
WAKE_ON_MULTICAST	Wakeup on multicast
WAKE_ON_ARP_BROADCAST	Wakeup on ARP broadcast
WAKE_ON_MGMT_FRAME	Wakeup on receiving a management frame

5.8.6.4 wlan_connection_state

```
enum wlan_connection_state
```

WLAN station/micro-AP/Wi-Fi Direct Connection/Status state

Enumerator

WLAN_DISCONNECTED	The WLAN Connection Manager is not connected and no connection attempt is in progress. It is possible to connect to a network or scan.
WLAN_CONNECTING	The WLAN Connection Manager is not connected but it is currently attempting to connect to a network. It is not possible to scan at this time. It is possible to connect to a different network.
WLAN_ASSOCIATED	The WLAN Connection Manager is not connected but associated.
WLAN_CONNECTED	The WLAN Connection Manager is connected. It is possible to scan and connect to another network at this time. Information about the current network configuration is available.
WLAN_UAP_STARTED	The WLAN Connection Manager has started uAP
WLAN_UAP_STOPPED	The WLAN Connection Manager has stopped uAP
WLAN_SCANNING	The WLAN Connection Manager is not connected and network scan is in progress.
WLAN_ASSOCIATING	The WLAN Connection Manager is not connected and network association is in progress.

5.8.6.5 wlan_ps_mode

```
enum wlan_ps_mode
```

Station Power save mode

Enumerator

WLAN_ACTIVE	Active mode
WLAN_IEEE	IEEE power save mode
WLAN_DEEP_SLEEP	Deep sleep power save mode
WLAN_IEEE_DEEP_SLEEP	IEEE and Deep sleep power save mode

5.8.6.6 wlan_security_type

```
enum wlan_security_type
```

Network security types

Enumerator

WLAN_SECURITY_NONE	The network does not use security.
WLAN_SECURITY_WEP_OPEN	The network uses WEP security with open key.
WLAN_SECURITY_WEP_SHARED	The network uses WEP security with shared key.
WLAN_SECURITY_WPA	The network uses WPA security with PSK.
WLAN_SECURITY_WPA2	The network uses WPA2 security with PSK.
WLAN_SECURITY_WPA_WPA2_MIXED	The network uses WPA/WPA2 mixed security with PSK
WLAN_SECURITY_WPA2_FT	The network uses WPA2 security with PSK FT.
WLAN_SECURITY_WPA3_SAE	The network uses WPA3 security with SAE.
WLAN_SECURITY_WPA3_FT_SAE	The network uses WPA3 security with SAE FT.
WLAN_SECURITY_WPA2_WPA3_SAE_MIXED	The network uses WPA2/WPA3 SAE mixed security with PSK. This security mode is specific to uAP or SoftAP only
WLAN_SECURITY_OWE_ONLY	The network uses OWE only security without Transition mode support.
WLAN_SECURITY_EAP_TLS	The network uses WPA2 Enterprise EAP-TLS security The identity field in wlan_network structure is used
WLAN_SECURITY_EAP_TLS_SHA256	The network uses WPA2 Enterprise EAP-TLS SHA256 security The identity field in wlan_network structure is used
WLAN_SECURITY_EAP_TLS_FT	The network uses WPA2 Enterprise EAP-TLS FT security The identity field in wlan_network structure is used
WLAN_SECURITY_EAP_TLS_FT_SHA384	The network uses WPA2 Enterprise EAP-TLS FT SHA384 security The identity field in wlan_network structure is used
WLAN_SECURITY_EAP_TTLS	The network uses WPA2 Enterprise EAP-TTLS security The identity field in wlan_network structure is used

Enumerator

WLAN_SECURITY_EAP_TTLS_MSCHAPV2	The network uses WPA2 Enterprise EAP-TTLS-MSCHAPV2 security The anonymous identity, identity and password fields in wlan_network structure are used
WLAN_SECURITY_EAP_PEAP_MSCHAPV2	The network uses WPA2 Enterprise EAP-PEAP-MSCHAPV2 security The anonymous identity, identity and password fields in wlan_network structure are used
WLAN_SECURITY_EAP_PEAP_TLS	The network uses WPA2 Enterprise EAP-PEAP-TLS security The anonymous identity, identity and password fields in wlan_network structure are used
WLAN_SECURITY_EAP_PEAP_GTC	The network uses WPA2 Enterprise EAP-PEAP-GTC security The anonymous identity, identity and password fields in wlan_network structure are used
WLAN_SECURITY_EAP_FAST_MSCHAPV2	The network uses WPA2 Enterprise EAP-FAST-MSCHAPV2 security The anonymous identity, identity and password fields in wlan_network structure are used
WLAN_SECURITY_EAP_FAST_GTC	The network uses WPA2 Enterprise EAP-FAST-GTC security The anonymous identity, identity and password fields in wlan_network structure are used
WLAN_SECURITY_EAP_SIM	The network uses WPA2 Enterprise EAP-SIM security The identity and password fields in wlan_network structure are used
WLAN_SECURITY_EAP_AKA	The network uses WPA2 Enterprise EAP-AKA security The identity and password fields in wlan_network structure are used
WLAN_SECURITY_EAP_AKA_PRIME	The network uses WPA2 Enterprise EAP-AKA-PRIME security The identity and password fields in wlan_network structure are used
WLAN_SECURITY_WILDCARD	The network can use any security method. This is often used when the user only knows the name and passphrase but not the security type.

5.8.6.7 address_types

```
enum address_types
```

Address types to be used by the element wlan_ip_config.addr_type below

Enumerator

ADDR_TYPE_STATIC	static IP address
ADDR_TYPE_DHCP	Dynamic IP address
ADDR_TYPE_LLA	Link level address

5.9 wlan_11d.h File Reference

WLAN module 11d API.

5.9.1 Function Documentation

5.9.1.1 wlan_enable_11d()

```
static int wlan_enable_11d (
            int state )  [inline], [static]
```

Enable 11D support in WLAN Driver.

Note

This API should be called after WLAN is initialized but before starting uAP or making any connection attempts on station interface.

Parameters

in	*state*	1: enable, 0: disable

Returns

-WM_FAIL if operation was failed.
WM_SUCCESS if operation was successful.

5.9.1.2 wlan_enable_uap_11d()

```
static int wlan_enable_uap_11d (
            int state )  [inline], [static]
```

Enable 11D support in WLAN Driver for uap interface.

Note

This API should be called after WLAN is initialized but before starting uAP or making any connection attempts on station interface.

Parameters

in	*state*	1: enable, 0: disable

Returns

-WM_FAIL if operation was failed.
WM_SUCCESS if operation was successful.

5.10 wm_net.h File Reference

Network Abstraction Layer.

5.10.1 Detailed Description

This provides the calls related to the network layer. The SDK uses lwIP as the network stack.

Here we document the network utility functions provided by the SDK. The detailed lwIP API documentation can be found at: http://lwip.wikia.com/wiki/Application_API_layers

5.10.2 Function Documentation

5.10.2.1 net_dhcp_hostname_set()

```
int net_dhcp_hostname_set (
            char * hostname )
```

Set hostname for network interface

Parameters

in	*hostname*	Hostname to be set.

Note

NULL is a valid value for hostname.

Returns

WM_SUCESS

5.10.2.2 net_stop_dhcp_timer()

```
void net_stop_dhcp_timer (
            void )
```

Deactivate the dhcp timer

5.10.2.3 net_socket_blocking()

```
static int net_socket_blocking (
            int sock,
            int state ) [inline], [static]
```

Set socket blocking option as on or off

Parameters

in	*sock*	socket number to be set for blocking option.
in	*state*	set blocking on or off

Returns

WM_SUCESS otherwise standard LWIP error codes.

5.10.2.4 net_get_sock_error()

```
static int net_get_sock_error (
            int sock ) [inline], [static]
```

Get error number from provided socket

Parameters

in	*sock*	socket number to get error number.

Returns

error number.

5.10.2.5 net_inet_aton()

```
static uint32_t net_inet_aton (
            const char * cp ) [inline], [static]
```

Converts Internet host address from the IPv4 dotted-decimal notation into binary form (in network byte order)

Parameters

in	*cp*	IPv4 host address in dotted-decimal notation.

Returns

IPv4 address in binary form

5.10.2.6 net_wlan_set_mac_address()

```
void net_wlan_set_mac_address (
            unsigned char * stamac,
            unsigned char * uapmac )
```

set MAC hardware address to lwip network interface

Parameters

in	*stamac*	sta MAC address.
in	*uapmac*	uap MAC address.

5.10.2.7 net_stack_buffer_skip()

```
static uint8_t* net_stack_buffer_skip (
            void * buf,
            uint16_t in_offset ) [inline], [static]
```

Skip a number of bytes at the start of a stack buffer

Parameters

in	*buf*	input stack buffer.
in	*in_offset*	offset to skip.

Returns

the payload pointer after skip a number of bytes

5.10.2.8 net_stack_buffer_free()

```
static void net_stack_buffer_free (
            void * buf ) [inline], [static]
```

Free a buffer allocated from stack memory

Parameters

in	*buf*	stack buffer pointer.

5.10.2.9 net_stack_buffer_copy_partial()

```
static int net_stack_buffer_copy_partial (
            void * stack_buffer,
            void * dst,
            uint16_t len,
            uint16_t offset )  [inline], [static]
```

Copy (part of) the contents of a packet buffer to an application supplied buffer

Parameters

in	*stack_buffer*	the stack buffer from which to copy data.
in	*dst*	the destination buffer.
in	*len*	length of data to copy.
in	*offset*	offset into the stack buffer from where to begin copying

Returns

copy status based on stack definition.

5.10.2.10 net_stack_buffer_get_payload()

```
static void* net_stack_buffer_get_payload (
            void * buf )  [inline], [static]
```

Get the data payload inside the stack buffer.

Parameters

in	*buf*	input stack buffer.

Returns

the payload pointer of the stack buffer.

5.10.2.11 net_gethostbyname()

```
static int net_gethostbyname (
            const char * cp,
            struct hostent ** hentry )  [inline], [static]
```

Get network host entry

Parameters

in	*cp*	Hostname or an IPv4 address in the standard dot notation.
in	*hentry*	Pointer to pointer of host entry structure.

Note

This function is not thread safe. If thread safety is required please use lwip_getaddrinfo() - lwip_freeaddrinfo() combination.

Returns

WM_SUCESS if operation successful.
-WM_FAIL if operation fails.

5.10.2.12 net_inet_ntoa()

```
static void net_inet_ntoa (
            unsigned long addr,
            char * cp )  [inline], [static]
```

Converts Internet host address in network byte order to a string in IPv4 dotted-decimal notation

Parameters

in	*addr*	IP address in network byte order.
out	*cp*	buffer in which IPv4 dotted-decimal string is returned.

5.10.2.13 net_is_ip_or_ipv6()

```
static bool net_is_ip_or_ipv6 (
            const uint8_t * buffer )  [inline], [static]
```

Check whether buffer is IPv4 or IPV6 packet type

Parameters

in	*buffer*	pointer to buffer where packet to be checked located.

Returns

true if buffer packet type matches with IPv4 or IPv6, false otherwise.

5.10.2.14 net_sock_to_interface()

```
void* net_sock_to_interface (
            int sock )
```

Get interface handle from socket descriptor

Given a socket descriptor this API returns which interface it is bound with.

Parameters

in	*sock*	socket descriptor

Returns

[out] interface handle

5.10.2.15 net_wlan_init()

```
int net_wlan_init (
            void )
```

Initialize TCP/IP networking stack

Returns

WM_SUCCESS on success
-WM_FAIL otherwise

5.10.2.16 net_wlan_deinit()

```
int net_wlan_deinit (
            void )
```

DiInitialize TCP/IP networking stack

Returns

WM_SUCCESS on success
-WM_FAIL otherwise

5.10.2.17 net_get_sta_interface()

```
struct netif* net_get_sta_interface (
            void )
```

Get STA interface netif structure pointer

Returns

A pointer to STA interface netif structure

5.10.2.18 net_get_uap_interface()

```
struct netif* net_get_uap_interface (
            void )
```

Get uAP interface netif structure pointer

Returns

A pointer to uAP interface netif structure

5.10.2.19 net_get_if_name_netif()

```
int net_get_if_name_netif (
            char * pif_name,
            struct netif * iface )
```

Get interface name for given netif

Parameters

out	*pif_name*	Buffer to store interface name
in	*iface*	Interface to get the name

Returns

WM_SUCCESS on success
-WM_FAIL otherwise

5.10.2.20 net_alloc_client_data_id()

```
int net_alloc_client_data_id ( )
```

Get client data index for storing private data in * netif.

Returns

allocated client data index, -1 if error or not supported.

5.10.2.21 net_get_sta_handle()

```
void* net_get_sta_handle (
            void )
```

Get station interface handle

Some APIs require the interface handle to be passed to them. The handle can be retrieved using this API.

Returns

station interface handle

5.10.2.22 net_get_uap_handle()

```
void* net_get_uap_handle (
            void )
```

Get micro-AP interface handle

Some APIs require the interface handle to be passed to them. The handle can be retrieved using this API.

Returns

micro-AP interface handle

5.10.2.23 net_interface_up()

```
void net_interface_up (
            void * intrfc_handle )
```

Take interface up

Change interface state to up. Use net_get_sta_handle(), net_get_uap_handle() to get interface handle.

Parameters

in	*intrfc_handle*	interface handle

5.10.2.24 net_interface_down()

```
void net_interface_down (
            void * intrfc_handle )
```

Take interface down

Change interface state to down. Use net_get_sta_handle(), net_get_uap_handle() to get interface handle.

Parameters

in	*intrfc_handle*	interface handle

5.10.2.25 net_interface_dhcp_stop()

```
void net_interface_dhcp_stop (
            void * intrfc_handle )
```

Stop DHCP client on given interface

Stop the DHCP client on given interface state. Use net_get_sta_handle(), net_get_uap_handle() to get interface handle.

Parameters

in	*intrfc_handle*	interface handle

5.10.2.26 net_interface_dhcp_cleanup()

```
void net_interface_dhcp_cleanup (
            void * intrfc_handle )
```

Cleanup DHCP client on given interface

Cleanup the DHCP client on given interface state. Use net_get_sta_handle(), net_get_uap_handle() to get interface handle.

Parameters

in	*intrfc_handle*	interface handle

5.10.2.27 net_configure_address()

```
int net_configure_address (
            struct net_ip_config * addr,
            void * intrfc_handle )
```

Configure IP address for interface

Parameters

in	*addr*	Address that needs to be configured.
in	*intrfc_handle*	Handle for network interface to be configured.

Returns

WM_SUCCESS on success or an error code.

5.10.2.28 net_configure_dns()

```
void net_configure_dns (
            struct net_ip_config * ip,
            unsigned int role )
```

Configure DNS server address

Parameters

in	*ip*	IP address of the DNS server to set
in	*role*	Network wireless BSS Role

5.10.2.29 net_get_if_addr()

```
int net_get_if_addr (
            struct net_ip_config * addr,
            void * intrfc_handle )
```

Get interface IP Address in net_ip_config

This function will get the IP address of a given interface. Use net_get_sta_handle(), net_get_uap_handle() to get interface handle.

Parameters

out	*addr*	net_ip_config
in	*intrfc_handle*	interface handle

Returns

WM_SUCCESS on success or error code.

5.10.2.30 net_get_if_ipv6_addr()

```
int net_get_if_ipv6_addr (
            struct net_ip_config * addr,
            void * intrfc_handle )
```

Get interface IPv6 Addresses & their states in net_ip_config

This function will get the IPv6 addresses & address states of a given interface. Use net_get_sta_handle() to get interface handle.

Parameters

out	*addr*	net_ip_config
in	*intrfc_handle*	interface handle

Returns

WM_SUCCESS on success or error code.

5.10.2.31 net_get_if_ipv6_pref_addr()

```
int net_get_if_ipv6_pref_addr (
            struct net_ip_config * addr,
            void * intrfc_handle )
```

Get list of preferred IPv6 Addresses of a given interface in net_ip_config

This function will get the list of IPv6 addresses whose address state is Preferred. Use net_get_sta_handle() to get interface handle.

Parameters

out	*addr*	net_ip_config
in	*intrfc_handle*	interface handle

Returns

Number of IPv6 addresses whose address state is Preferred

5.10.2.32 ipv6_addr_state_to_desc()

```
char* ipv6_addr_state_to_desc (
            unsigned char addr_state )
```

Get the description of IPv6 address state

This function will get the IPv6 address state description like - Invalid, Preferred, Deprecated

Parameters

in	*addr_state*	Address state

Returns

IPv6 address state description

5.10.2.33 ipv6_addr_addr_to_desc()

```
char* ipv6_addr_addr_to_desc (
            struct net_ipv6_config * ipv6_conf )
```

Get the description of IPv6 address

This function will get the IPv6 address type description like - Linklocal, Global, Sitelocal, Uniquelocal

Parameters

in	*ipv6_conf*	Pointer to IPv6 configuration of type net_ipv6_config

Returns

IPv6 address description

5.10.2.34 ipv6_addr_type_to_desc()

```
char* ipv6_addr_type_to_desc (
            struct net_ipv6_config * ipv6_conf )
```

Get the description of IPv6 address type

This function will get the IPv6 address type description like - Linklocal, Global, Sitelocal, Uniquelocal

Parameters

in	*ipv6_conf*	Pointer to IPv6 configuration of type net_ipv6_config

Returns

IPv6 address type description

5.10.2.35 net_get_if_name()

```
int net_get_if_name (
            char * if_name,
            void * intrfc_handle )
```

Get interface Name string containing name and number

This function will get the string containing name and number for given interface. Use net_get_sta_handle(), net_get_uap_handle() to get interface handle.

Parameters

out	*if_name*	interface name pointer
in	*intrfc_handle*	interface handle

Returns

WM_SUCCESS on success or error code.

5.10.2.36 net_get_if_ip_addr()

```
int net_get_if_ip_addr (
            uint32_t * ip,
            void * intrfc_handle )
```

Get interface IP Address

This function will get the IP Address of a given interface. Use net_get_sta_handle(), net_get_uap_handle() to get interface handle.

Parameters

out	*ip*	ip address pointer
in	*intrfc_handle*	interface handle

Returns

WM_SUCCESS on success or error code.

5.10.2.37 net_get_if_ip_mask()

```
int net_get_if_ip_mask (
            uint32_t * nm,
            void * intrfc_handle )
```

Get interface IP Subnet-Mask

This function will get the Subnet-Mask of a given interface. Use net_get_sta_handle(), net_get_uap_handle() to get interface handle.

Parameters

in	*nm*	Subnet Mask pointer
in	*intrfc_handle*	interface

Returns

WM_SUCCESS on success or error code.

5.10.2.38 net_ipv4stack_init()

```
void net_ipv4stack_init (
            void )
```

Initialize the network stack

This function initializes the network stack. This function is called by wlan_start().

Applications may optionally call this function directly: if they wish to use the networking stack (loopback interface) without the wlan functionality. if they wish to initialize the networking stack even before wlan comes up.

Note

This function may safely be called multiple times.

5.10.2.39 net_ipv6stack_init()

```
void net_ipv6stack_init (
            struct netif * netif )
```

Initialize the IPv6 network stack

Parameters

in	*netif*	network interface on which ipv6 stack is initialized.

5.10.2.40 net_stat()

```
void net_stat (
            void )
```

Display network statistics

5.10.3 Enumeration Type Documentation

5.10.3.1 net_address_types

```
enum net_address_types
```

Address types to be used by the element net_ip_config.addr_type below

Enumerator

NET_ADDR_TYPE_STATIC	static IP address
NET_ADDR_TYPE_DHCP	Dynamic IP address
NET_ADDR_TYPE_LLA	Link level address

5.11 wm_os.h File Reference

OS Abstraction Layer.

5.11.1 Detailed Description

The OS abstraction layer provides wrapper APIs over some of the commonly used OS primitives. Since the behaviour and semantics of the various OSes differs widely, some abstraction APIs require a specific handling as listed below.

5.11.2 Usage

The OS abstraction layer provides the following types of primitives:

- Thread: Create or delete a thread using os_thread_create() or os_thread_delete(). Block a thread using os_thread_sleep(). Complete a thread's execution using os_thread_self_complete().
- Message Queue: Create or delete a message queue using os_queue_create() or os_queue_delete(). Send a message using os_queue_send() and received a message using os_queue_recv().
- Mutex: Create or delete a mutex using os_mutex_create() or os_mutex_delete(). Acquire a mutex using os_mutex_get() and release it using os_mutex_put().
- Semaphores: Create or delete a semaphore using os_semaphore_create() / os_semaphore_create_↵ counting() or os_semaphore_delete. Acquire a semaphore using os_semaphore_get() and release it using os_semaphore_put().
- Timers: Create or delete a timer using os_timer_create() or os_timer_delete(). Change the timer using os↵ _timer_change(). Activate or de-activate the timer using os_timer_activate() or os_timer_deactivate(). Reset a timer using os_timer_reset().
- Dynamic Memory Allocation: Dynamically allocate memory using os_mem_alloc(), os_mem_calloc() and free it using os_mem_free().

5.11.3 Function Documentation

5.11.3.1 os_ticks_get()

```
unsigned os_ticks_get (
            void )
```

Get current OS tick counter value

Returns

32 bit value of ticks since boot-up

5.11.3.2 os_get_timestamp()

```
unsigned int os_get_timestamp (
            void )
```

Returns time in micro-secs since bootup

Note

The value returned will wrap around after sometime and caller is expected to guard itself against this.

Returns

Time in micro-secs since bootup

5.11.3.3 os_msec_to_ticks()

```
uint32_t os_msec_to_ticks (
            uint32_t msecs )
```

Convert milliseconds to OS ticks

This function converts the given millisecond value to the number of OS ticks.

This is useful as functions like os_thread_sleep() accept only ticks as input.

Parameters

in	*msecs*	Milliseconds

Returns

Number of OS ticks corresponding to msecs

5.11.3.4 os_ticks_to_msec()

```
unsigned long os_ticks_to_msec (
            unsigned long ticks )
```

Convert ticks to milliseconds

This function converts the given ticks value to milliseconds. This is useful as some functions, like os_ticks_get(), return values in units of OS ticks.

Parameters

in	*ticks*	OS ticks

Returns

Number of milliseconds corresponding to ticks

5.11.3.5 os_thread_create()

```
int os_thread_create (
            os_thread_t * thandle,
            const char * name,
            void(*)(os_thread_arg_t arg) main_func,
            void * arg,
            os_thread_stack_t * stack,
            int prio )
```

Create new thread

This function starts a new thread. The new thread starts execution by invoking main_func(). The parameter arg is passed as the sole argument of main_func().

After finishing execution, the new thread should either call:

- os_thread_self_complete() to suspend itself OR
- os_thread_delete() to delete itself

Failing to do this and just returning from main_func() will result in undefined behavior.

Parameters

out	*thandle*	Pointer to a thread handle
in	*name*	Name of the new thread. A copy of this string will be made by the OS for itself. The maximum name length is defined by the macro configMAX_TASK_NAME_LEN in FreeRTOS header file . Any name length above it will be truncated.
in	*main_func*	Function pointer to new thread function
in	*arg*	The sole argument passed to main_func()
in	*stack*	A pointer to initialized object of type os_thread_stack_t. The object should be created and initialized using os_thread_stack_define().
in	*prio*	The priority of the new thread. One value among OS_PRIO_0, OS_PRIO_1, OS_PRIO_2, OS_PRIO_3 and OS_PRIO_4 should be passed. OS_PRIO_0 represents the highest priority and OS_PRIO_4 represents the lowest priority.

Returns

WM_SUCCESS if thread was created successfully
-WM_FAIL if thread creation failed

5.11.3.6 os_thread_delete()

```
int os_thread_delete (
            os_thread_t * thandle )
```

Terminate a thread

This function deletes a thread. The task being deleted will be removed from all ready, blocked, suspended and event lists.

Parameters

in	*thandle*	Pointer to the thread handle of the thread to be deleted. If self deletion is required NULL should be passed.

Returns

WM_SUCCESS if operation success
-WM_FAIL if operation fails

5.11.3.7 os_thread_sleep()

```
void os_thread_sleep (
            uint32_t ticks )
```

Sleep for specified number of OS ticks

This function causes the calling thread to sleep and block for the given number of OS ticks. The actual time that the task remains blocked depends on the tick rate. The function os_msec_to_ticks() is provided to convert from real-time to ticks.

Any other thread can wake up this task specifically using the API os_thread_wait_abort()

Parameters

in	*ticks*	Number of ticks to sleep

5.11.3.8 os_thread_self_complete()

```
void os_thread_self_complete (
            os_thread_t * thandle )
```

Suspend the given thread

- The function os_thread_self_complete() will **permanently** suspend the given thread. Passing NULL will suspend the current thread. This function never returns.
- The thread continues to consume system resources. To delete the thread the function os_thread_delete() needs to be called separately.

Parameters

in	*thandle*	Pointer to thread handle

5.11.3.9 os_queue_create()

```
int os_queue_create (
            os_queue_t * qhandle,
            const char * name,
            int msgsize,
            os_queue_pool_t * poolname )
```

Create an OS queue

This function creates a new queue instance. This allocates the storage required by the new queue and returns a handle for the queue.

Parameters

out	*qhandle*	Pointer to the handle of the newly created queue
in	*name*	String specifying the name of the queue
in	*msgsize*	The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.
in	*poolname*	The object of the type os_queue_pool_t. The helper macro os_queue_pool_define() helps to define this object.

Returns

WM_SUCCESS if queue creation was successful
-WM_FAIL if queue creation failed

5.11.3.10 os_queue_send()

```
int os_queue_send (
            os_queue_t * qhandle,
            const void * msg,
            unsigned long wait )
```

Post an item to the back of the queue.

This function posts an item to the back of a queue. The item is queued by copy, not by reference. This function can also be called from an interrupt service routine.

Parameters

in	*qhandle*	Pointer to the handle of the queue
in	*msg*	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from msg into the queue storage area.
in	*wait*	The maximum amount of time, in OS ticks, the task should block waiting for space to become available on the queue, should it already be full. The function os_msec_to_ticks() can be used to convert from real-time to OS ticks. The special values OS_WAIT_FOREVER and OS_NO_WAIT are provided to respectively wait infinitely or return immediately.

Returns

WM_SUCCESS if send operation was successful
-WM_E_INVAL if invalid parameters are passed
-WM_FAIL if send operation failed

5.11.3.11 os_queue_recv()

```
int os_queue_recv (
            os_queue_t * qhandle,
            void * msg,
            unsigned long wait )
```

Receive an item from queue

This function receives an item from a queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Parameters

in	*qhandle*	Pointer to handle of the queue
out	*msg*	Pointer to the buffer into which the received item will be copied. The size of the items in the queue was defined when the queue was created. This pointer should point to a buffer as many bytes in size.
in	*wait*	The maximum amount of time, in OS ticks, the task should block waiting for messages to arrive on the queue, should it already be empty. The function os_msec_to_ticks() can be used to convert from real-time to OS ticks. The special values OS_WAIT_FOREVER and OS_NO_WAIT are provided to respectively wait infinitely or return immediately.

Returns

WM_SUCCESS if receive operation was successful
-WM_E_INVAL if invalid parameters are passed
-WM_FAIL if receive operation failed

Note

This function must not be used in an interrupt service routine.

5.11.3.12 os_queue_delete()

```
int os_queue_delete (
            os_queue_t * qhandle )
```

Delete queue

This function deletes a queue. It frees all the memory allocated for storing of items placed on the queue.

Parameters

in	*qhandle*	Pointer to handle of the queue to be deleted.

Returns

Currently always returns WM_SUCCESS

5.11.3.13 os_queue_get_msgs_waiting()

```
int os_queue_get_msgs_waiting (
            os_queue_t * qhandle )
```

Return the number of messages stored in queue.

Parameters

in	*qhandle*	Pointer to handle of the queue to be queried.

Returns

Number of items in the queue
-WM_E_INVAL if invalid parameters are passed

5.11.3.14 os_setup_idle_function()

```
int os_setup_idle_function (
            void(*)(void) func )
```

Setup idle function

This function sets up a callback function which will be called whenever the system enters the idle thread context.

Parameters

in	*func*	The callback function

Returns

WM_SUCCESS on success
-WM_FAIL on error

5.11.3.15 os_setup_tick_function()

```
int os_setup_tick_function (
            void(*)(void) func )
```

Setup tick function

This function sets up a callback function which will be called on every SysTick interrupt.

Parameters

in	*func*	The callback function

Returns

WM_SUCCESS on success
-WM_FAIL on error

5.11.3.16 os_remove_idle_function()

```
int os_remove_idle_function (
            void(*)(void) func )
```

Remove idle function

This function removes an idle callback function that was registered previously using os_setup_idle_function().

Parameters

in	*func*	The callback function

Returns

WM_SUCCESS on success
-WM_FAIL on error

5.11.3.17 os_remove_tick_function()

```
int os_remove_tick_function (
            void(*)(void) func )
```

Remove tick function

This function removes a tick callback function that was registered previously using os_setup_tick_function().

Parameters

in	*func*	Callback function

Returns

WM_SUCCESS on success
-WM_FAIL on error

5.11.3.18 os_mutex_create()

```
int os_mutex_create (
            os_mutex_t * mhandle,
            const char * name,
            int flags )
```

Create mutex

This function creates a mutex.

Parameters

out	*mhandle*	Pointer to a mutex handle
in	*name*	Name of the mutex
in	*flags*	Priority inheritance selection. Valid options are OS_MUTEX_INHERIT or OS_MUTEX_NO_INHERIT.

Note

Currently non-inheritance in mutex is not supported.

Returns

WM_SUCCESS on success
-WM_FAIL on error

5.11.3.19 os_mutex_get()

```
int os_mutex_get (
            os_mutex_t * mhandle,
            unsigned long wait )
```

Acquire mutex

This function acquires a mutex. Only one thread can acquire a mutex at any given time. If already acquired the callers will be blocked for the specified time duration.

Parameters

in	*mhandle*	Pointer to mutex handle
in	*wait*	The maximum amount of time, in OS ticks, the task should block waiting for the mutex to be acquired. The function os_msec_to_ticks() can be used to convert from real-time to OS ticks. The special values OS_WAIT_FOREVER and OS_NO_WAIT are provided to respectively wait infinitely or return immediately.

Returns

WM_SUCCESS when mutex is acquired
-WM_E_INVAL if invalid parameters are passed
-WM_FAIL on failure

5.11.3.20 os_mutex_put()

```
int os_mutex_put (
            os_mutex_t * mhandle )
```

Release mutex

This function releases a mutex previously acquired using os_mutex_get().

Note

The mutex should be released from the same thread context from which it was acquired. If you wish to acquire and release in different contexts, please use os_semaphore_get() and os_semaphore_put() variants.

Parameters

in	*mhandle*	Pointer to the mutex handle

Returns

WM_SUCCESS when mutex is released
-WM_E_INVAL if invalid parameters are passed
-WM_FAIL on failure

5.11.3.21 os_recursive_mutex_create()

```
int os_recursive_mutex_create (
            os_mutex_t * mhandle,
            const char * name )
```

Create recursive mutex

This function creates a recursive mutex. A mutex used recursively can be 'get' repeatedly by the owner. The mutex doesn't become available again until the owner has called os_recursive_mutex_put() for each successful 'get' request.

Note

This type of mutex uses a priority inheritance mechanism so a task 'get'ing a mutex MUST ALWAYS 'put' the mutex back once no longer required.

Parameters

out	*mhandle*	Pointer to a mutex handle
in	*name*	Name of the mutex as NULL terminated string

Returns

WM_SUCCESS on success
-WM_E_INVAL on invalid parameter.
-WM_FAIL on error

5.11.3.22 os_recursive_mutex_get()

```
int os_recursive_mutex_get (
            os_mutex_t * mhandle,
            unsigned long wait )
```

Get recursive mutex

This function recursively obtains, or 'get's, a mutex. The mutex must have previously been created using a call to os_recursive_mutex_create().

Parameters

in	*mhandle*	Pointer to mutex handle obtained from os_recursive_mutex_create().
in	*wait*	The maximum amount of time, in OS ticks, the task should block waiting for the mutex to be acquired. The function os_msec_to_ticks() can be used to convert from real-time to OS ticks. The special values OS_WAIT_FOREVER and OS_NO_WAIT are provided to respectively wait for portMAX_DELAY (0xffffffff) or return immediately.

Returns

WM_SUCCESS when recursive mutex is acquired
-WM_FAIL on failure

5.11.3.23 os_recursive_mutex_put()

```
int os_recursive_mutex_put (
            os_mutex_t * mhandle )
```

Put recursive mutex

This function recursively releases, or 'give's, a mutex. The mutex must have previously been created using a call to os_recursive_mutex_create()

Parameters

in	*mhandle*	Pointer to the mutex handle

Returns

WM_SUCCESS when mutex is released
-WM_FAIL on failure

5.11.3.24 os_mutex_delete()

```
int os_mutex_delete (
            os_mutex_t * mhandle )
```

Delete mutex

This function deletes a mutex.

Parameters

in	*mhandle*	Pointer to the mutex handle

Note

A mutex should not be deleted if other tasks are blocked on it.

Returns

WM_SUCCESS on success

5.11.3.25 os_event_notify_get()

```
int os_event_notify_get (
            unsigned long wait_time )
```

Wait for task notification

This function waits for task notification from other task or interrupt context. This is similar to binary semaphore, but uses less RAM and much faster than semaphore mechanism

Parameters

in	*wait_time*	Timeout specified in no. of OS ticks

Returns

WM_SUCCESS when notification is successful
-WM_FAIL on failure or timeout

5.11.3.26 os_event_notify_put()

```
int os_event_notify_put (
            os_thread_t task )
```

Give task notification

This function gives task notification so that waiting task can be unblocked. This is similar to binary semaphore, but uses less RAM and much faster than semaphore mechanism

Parameters

in	*task*	Task handle to be notified

Returns

WM_SUCCESS when notification is successful
-WM_FAIL on failure or timeout

5.11.3.27 os_semaphore_create()

```
int os_semaphore_create (
            os_semaphore_t * mhandle,
            const char * name )
```

Create binary semaphore

This function creates a binary semaphore. A binary semaphore can be acquired by only one entity at a given time.

Parameters

out	*mhandle*	Pointer to a semaphore handle
in	*name*	Name of the semaphore

Returns

WM_SUCCESS on success
-WM_FAIL on error

5.11.3.28 os_semaphore_create_counting()

```
int os_semaphore_create_counting (
            os_semaphore_t * mhandle,
            const char * name,
            unsigned long maxcount,
            unsigned long initcount )
```

Create counting semaphore

This function creates a counting semaphore. A counting semaphore can be acquired 'count' number of times at a given time.

Parameters

out	*mhandle*	Pointer to a semaphore handle
in	*name*	Name of the semaphore
in	*maxcount*	The maximum count value that can be reached. When the semaphore reaches this value it can no longer be 'put'
in	*initcount*	The count value assigned to the semaphore when it is created. For e.g. If '0' is passed, then os_semaphore_get() will block until some other thread does an os_semaphore_put().

Returns

WM_SUCCESS on success
-WM_FAIL on error

5.11.3.29 os_semaphore_get()

```
int os_semaphore_get (
            os_semaphore_t * mhandle,
            unsigned long wait )
```

Acquire semaphore

This function acquires a semaphore. At a given time, a binary semaphore can be acquired only once, while a counting semaphore can be acquired as many as 'count' number of times. Once this condition is reached, the other callers of this function will be blocked for the specified time duration.

Parameters

in	*mhandle*	Pointer to a semaphore handle
in	*wait*	The maximum amount of time, in OS ticks, the task should block waiting for the semaphore to be acquired. The function os_msec_to_ticks() can be used to convert from real-time to OS ticks. The special values OS_WAIT_FOREVER and OS_NO_WAIT are provided to respectively wait infinitely or return immediately.

Returns

WM_SUCCESS when semaphore is acquired
-WM_E_INVAL if invalid parameters are passed
-WM_FAIL on failure

5.11.3.30 os_semaphore_put()

```
int os_semaphore_put (
            os_semaphore_t * mhandle )
```

Release semaphore

This function releases a semaphore previously acquired using os_semaphore_get().

Note

This function can also be called from interrupt-context.

Parameters

in	*mhandle*	Pointer to a semaphore handle

Returns

WM_SUCCESS when semaphore is released
-WM_E_INVAL if invalid parameters are passed
-WM_FAIL on failure

5.11.3.31 os_semaphore_getcount()

```
int os_semaphore_getcount (
            os_semaphore_t * mhandle )
```

Get semaphore count

This function returns the current value of a semaphore.

Parameters

in	*mhandle*	Pointer to a semaphore handle

Returns

current value of the semaphore

5.11.3.32 os_semaphore_delete()

```
int os_semaphore_delete (
            os_semaphore_t * mhandle )
```

Delete a semaphore

This function deletes the semaphore.

Parameters

in	*mhandle*	Pointer to a semaphore handle

Note

Do not delete a semaphore that has tasks blocked on it (tasks that are in the Blocked state waiting for the semaphore to become available)

Returns

WM_SUCCESS on success

5.11.3.33 os_rwlock_create()

```
int os_rwlock_create (
            os_rw_lock_t * plock,
            const char * mutex_name,
            const char * lock_name )
```

Create reader-writer lock

This function creates a reader-writer lock.

Parameters

in	*plock*	Pointer to a reader-writer lock handle
in	*mutex_name*	Name of the mutex
in	*lock_name*	Name of the lock

Returns

WM_SUCCESS on success
-WM_FAIL on error

5.11.3.34 os_rwlock_delete()

```
void os_rwlock_delete (
            os_rw_lock_t * lock )
```

Delete a reader-write lock

This function deletes a reader-writer lock.

Parameters

in	*lock*	Pointer to the reader-writer lock handle

5.11.3.35 os_rwlock_write_lock()

```
int os_rwlock_write_lock (
            os_rw_lock_t * lock,
            unsigned int wait_time )
```

Acquire writer lock

This function acquires a writer lock. While readers can acquire the lock on a sharing basis, writers acquire the lock in an exclusive manner.

Parameters

in	*lock*	Pointer to the reader-writer lock handle
in	*wait_time*	The maximum amount of time, in OS ticks, the task should block waiting for the lock to be acquired. The function os_msec_to_ticks() can be used to convert from real-time to OS ticks. The special values OS_WAIT_FOREVER and OS_NO_WAIT are provided to respectively wait infinitely or return immediately.

Returns

WM_SUCCESS on success
-WM_FAIL on error

5.11.3.36 os_rwlock_write_unlock()

```
void os_rwlock_write_unlock (
            os_rw_lock_t * lock )
```

Release writer lock

This function releases a writer lock previously acquired using os_rwlock_write_lock().

Parameters

in	*lock*	Pointer to the reader-writer lock handle

5.11.3.37 os_rwlock_read_lock()

```
int os_rwlock_read_lock (
            os_rw_lock_t * lock,
            unsigned int wait_time )
```

Acquire reader lock

This function acquires a reader lock. While readers can acquire the lock on a sharing basis, writers acquire the lock in an exclusive manner.

Parameters

in	*lock*	pointer to the reader-writer lock handle
in	*wait_time*	The maximum amount of time, in OS ticks, the task should block waiting for the lock to be acquired. The function os_msec_to_ticks() can be used to convert from real-time to OS ticks. The special values OS_WAIT_FOREVER and OS_NO_WAIT are provided to respectively wait infinitely or return immediately.

Returns

WM_SUCCESS on success
-WM_FAIL on error

5.11.3.38 os_rwlock_read_unlock()

```
int os_rwlock_read_unlock (
            os_rw_lock_t * lock )
```

Release reader lock

This function releases a reader lock previously acquired using os_rwlock_read_lock().

Parameters

in	*lock*	pointer to the reader-writer lock handle

Returns

WM_SUCCESS if unlock operation successful.
-WM_FAIL if unlock operation failed.

5.11.3.39 os_timer_create()

```
int os_timer_create (
            os_timer_t * timer_t,
            const char * name,
            os_timer_tick ticks,
            void(*)(os_timer_arg_t) call_back,
            void * cb_arg,
            os_timer_reload_t reload,
            os_timer_activate_t activate )
```

Create timer

This function creates a timer.

Parameters

out	*timer_t*	Pointer to the timer handle
in	*name*	Name of the timer
in	*ticks*	Period in ticks
in	*call_back*	Timer expire callback function
in	*cb_arg*	Timer callback data
in	*reload*	Reload Options, valid values include OS_TIMER_ONE_SHOT or OS_TIMER_PERIODIC.
in	*activate*	Activate Options, valid values include OS_TIMER_AUTO_ACTIVATE or OS_TIMER_NO_ACTIVATE

Returns

WM_SUCCESS if timer created successfully
-WM_FAIL if timer creation fails

5.11.3.40 os_timer_activate()

```
int os_timer_activate (
            os_timer_t * timer_t )
```

Activate timer

This function activates (or starts) a timer that was previously created using os_timer_create(). If the timer had already started and was already in the active state, then this call is equivalent to os_timer_reset().

Parameters

in	*timer_t*	Pointer to a timer handle

Returns

WM_SUCCESS if timer activated successfully
-WM_E_INVAL if invalid parameters are passed
-WM_FAIL if timer fails to activate

5.11.3.41 os_timer_change()

```
int os_timer_change (
            os_timer_t * timer_t,
            os_timer_tick ntime,
            os_timer_tick block_time )
```

Change timer period

This function changes the period of a timer that was previously created using os_time_create(). This function changes the period of an active or dormant state timer.

Parameters

in	*timer_t*	Pointer to a timer handle
in	*ntime*	Time in ticks after which the timer will expire
in	*block_time*	This option is currently not supported

Returns

WM_SUCCESS on success
-WM_E_INVAL if invalid parameters are passed
-WM_FAIL on failure

5.11.3.42 os_timer_is_running()

```
bool os_timer_is_running (
            os_timer_t * timer_t )
```

Check the timer active state

This function checks if the timer is in the active or dormant state. A timer is in the dormant state if (a) it has been created but not started, or (b) it has expired and a one-shot timer.

Parameters

in	*timer_t*	Pointer to a timer handle

Returns

true if timer is active
false if time is not active

5.11.3.43 os_timer_get_context()

```
void* os_timer_get_context (
            os_timer_t * timer_t )
```

Get the timer context

This function helps to retrieve the timer context i.e. 'cb_arg' passed to os_timer_create().

Parameters

in	*timer_t*	Pointer to timer handle. The timer handle is received in the timer callback.

Returns

The timer context i.e. the callback argument passed to os_timer_create().

5.11.3.44 os_timer_reset()

```
int os_timer_reset (
            os_timer_t * timer_t )
```

Reset timer

This function resets a timer that was previously created using using os_timer_create(). If the timer had already been started and was already in the active state, then this call will cause the timer to re-evaluate its expiry time so that it is relative to when os_timer_reset() was called. If the timer was in the dormant state then this call behaves in the same way as os_timer_activate().

Parameters

in	*timer_t*	Pointer to a timer handle

Returns

WM_SUCCESS on success
-WM_E_INVAL if invalid parameters are passed
-WM_FAIL on failure

5.11.3.45 os_timer_deactivate()

```
int os_timer_deactivate (
            os_timer_t * timer_t )
```

Deactivate timer

This function deactivates (or stops) a timer that was previously started.

Parameters

in	*timer_t*	handle populated by os_timer_create()

Returns

WM_SUCCESS on success
-WM_E_INVAL if invalid parameters are passed
-WM_FAIL on failure

5.11.3.46 os_timer_delete()

```
int os_timer_delete (
            os_timer_t * timer_t )
```

Delete timer

This function deletes a timer.

Parameters

in	*timer_t*	Pointer to a timer handle

Returns

WM_SUCCESS on success
-WM_E_INVAL if invalid parameters are passed
-WM_FAIL on failure

5.11.3.47 os_mem_alloc()

```
void* os_mem_alloc (
            size_t size )
```

Allocate memory

This function allocates memory dynamically.

Parameters

in	*size*	Size of the memory to be allocated

Returns

Pointer to the allocated memory
NULL if allocation fails

5.11.3.48 os_mem_calloc()

```
void* os_mem_calloc (
            size_t size )
```

Allocate memory and zero it

This function allocates memory dynamically and sets the memory contents to zero.

Parameters

in	*size*	Size of the memory to be allocated

Returns

Pointer to the allocated memory
NULL if allocation fails

5.11.3.49 os_mem_free()

```
void os_mem_free (
            void * ptr )
```

Free Memory

This function frees dynamically allocated memory using any of the dynamic allocation primitives.

Parameters

in	*ptr*	Pointer to the memory to be freed

5.11.3.50 os_dump_mem_stats()

```
void os_dump_mem_stats (
            void )
```

This function dumps complete statistics of the heap memory.

5.11.3.51 os_disable_all_interrupts()

```
void os_disable_all_interrupts (
            void )
```

Disables all interrupts at NVIC level

5.11.3.52 os_enable_all_interrupts()

```
void os_enable_all_interrupts (
            void )
```

Enable all interrupts at NVIC lebel

5.11.3.53 os_lock_schedule()

```
static void os_lock_schedule (
            void ) [inline], [static]
```

Disable all tasks schedule

5.11.3.54 os_unlock_schedule()

```
static void os_unlock_schedule (
            void ) [inline], [static]
```

Enable all tasks schedule

5.11.3.55 os_srand()

```
static void os_srand (
            uint32_t seed ) [inline], [static]
```

This function initialize the seed for rand generator

Parameters

in	*seed*	Seed for random number generator

5.11.3.56 os_rand()

```
static uint32_t os_rand ( ) [inline], [static]
```

This function generate a random number

Returns

a uint32_t random numer

5.11.3.57 os_rand_range()

```
static uint32_t os_rand_range (
            uint32_t low,
            uint32_t high ) [inline], [static]
```

This function generate a random number in a range

Parameters

in	*low*	range low
in	*high*	range high

Returns

a uint32_t random numer

5.11.4 Macro Documentation

5.11.4.1 os_thread_relinquish

```
#define os_thread_relinquish( ) taskYIELD()
```

Get the current value of free running microsecond counter

Note

This will wraparound after CNTMAX and the caller is expected to take care of this.

Returns

The current value of microsecond counter.Force a context switch

5.11.4.2 os_ticks_to_unblock

```
#define os_ticks_to_unblock( ) xTaskGetUnblockTime()
```

Get ticks to next thread wakeup

5.11.4.3 os_thread_stack_define

```
#define os_thread_stack_define(
              stackname,
              stacksize ) os_thread_stack_t stackname = {(stacksize) / (sizeof(portSTACK_TY←
PE))}
```

Helper macro to define the stack size (in bytes) before a new thread is created using the function os_thread_create().

5.11.4.4 os_queue_pool_define

```
#define os_queue_pool_define(
              poolname,
              poolsize ) os_queue_pool_t poolname = {poolsize};
```

Define OS Queue pool

This macro helps define the name and size of the queue to be created using the function os_queue_create().

5.11.4.5 OS_WAIT_FOREVER

```
#define OS_WAIT_FOREVER portMAX_DELAY
```

Wait Forever

5.11.4.6 OS_NO_WAIT

```
#define OS_NO_WAIT 0
```

Do Not Wait

5.11.4.7 OS_MUTEX_INHERIT

```
#define OS_MUTEX_INHERIT 1
```

Priority Inheritance Enabled

5.11.4.8 OS_MUTEX_NO_INHERIT

```
#define OS_MUTEX_NO_INHERIT 0
```

Priority Inheritance Disabled

5.11.4.9 os_get_runtime_stats

```
#define os_get_runtime_stats(
              __buff__ ) vTaskGetRunTimeStats(__buff__)
```

Get ASCII formatted run time statistics

Please ensure that your buffer is big enough for the formatted data to fit. Failing to do this may cause memory data corruption.

5.11.4.10 os_get_task_list

```
#define os_get_task_list(
              __buff__ ) vTaskList(__buff__)
```

Get ASCII formatted task list

Please ensure that your buffer is big enough for the formatted data to fit. Failing to do this may cause memory data corruption.

5.11.5 Typedef Documentation

5.11.5.1 cb_fn

```
typedef int(* cb_fn) (os_rw_lock_t *plock, unsigned int wait_time)
```

This is prototype of reader callback

5.11.6 Enumeration Type Documentation

5.11.6.1 os_timer_reload_t

```
enum os_timer_reload_t
```

OS Timer reload Options

Enumerator

OS_TIMER_ONE_SHOT	Create one shot timer. Timer will be in the dormant state after it expires.
OS_TIMER_PERIODIC	Create a periodic timer. Timer will auto-reload after it expires.

5.11.6.2 os_timer_activate_t

```
enum os_timer_activate_t
```

OS Timer Activate Options

Enumerator

OS_TIMER_AUTO_ACTIVATE	Start the timer on creation.
OS_TIMER_NO_ACTIVATE	Do not start the timer on creation.

5.12 wm_utils.h File Reference

Utility functions.

5.12.1 Detailed Description

Collection of some common helper functions

5.12.2 Function Documentation

5.12.2.1 wm_hex2bin()

```
static unsigned int wm_hex2bin (
            const uint8_t * ibuf,
            uint8_t * obuf,
            unsigned max_olen )  [inline], [static]
```

Convert a given hex string to a equivalent binary representation.

E.g. If your input string of 4 bytes is {'F', 'F', 'F', 'F'} the output string will be of 2 bytes {255, 255} or to put the same in other way {0xFF, 0xFF}

Note that hex2bin is not the same as strtoul as the latter will properly return the integer in the correct machine binary format viz. little endian. hex2bin however does only in-place like replacement of two ASCII characters to one binary number taking 1 byte in memory.

Parameters

in	*ibuf*	input buffer
out	*obuf*	output buffer
in	*max_olen*	Maximum output buffer length

Returns

length of the binary string

5.12.2.2 bin2hex()

```
void bin2hex (
            uint8_t * src,
            char * dest,
            unsigned int src_len,
            unsigned int dest_len )
```

Convert given binary array to equivalent hex representation.

Parameters

in	*src*	Input buffer
out	*dest*	Output buffer
in	*src_len*	Length of the input buffer
in	*dest_len*	Length of the output buffer

5.12.2.3 random_register_handler()

```
int random_register_handler (
            random_hdlr_t func )
```

Register a random entropy generator handler

This API allows applications to register their own random entropy generator handlers that will be internally used by get_random_sequence() to add even more randomization to the byte stream generated by it.

Parameters

in	*func*	Function pointer of type random_hdlr_t

Returns

WM_SUCCESS if successful
-WM_E_NOSPC if there is no space available for additional handlers

5.12.2.4 random_unregister_handler()

```
int random_unregister_handler (
            random_hdlr_t func )
```

Un-register a random entropy generator handler

This API can be used to un-register a handler registered using random_register_handler()

Parameters

in	*func*	Function pointer of type random_hdlr_t used during registering

Returns

WM_SUCCESS if successful
-WM_E_INVAL if the passed pointer is invalid

5.12.2.5 random_register_seed_handler()

```
int random_register_seed_handler (
            random_hdlr_t func )
```

Register a random seed generator handler

For getting better random numbers, the initial seed (ideally required only once on every boot) should also be random. This API allows applications to register their own seed generators. Applications can use any logic such that a different seed is generated every time. A sample seed generator which uses a combination of DAC (generating random noise) and ADC (that internally samples the random noise) along with the flash id has already been provided. Please have a look at sample_initialise_random_seed().

The seed generator handler is called only once by the get_random_sequence() function. Applications can also explicitly initialize the seed by calling random_initialize_seed() after registering a handler.

Parameters

in	*func*	Function pointer of type random_hdlr_t

Returns

WM_SUCCESS if successful
-WM_E_NOSPC if there is no space available for additional handlers

5.12.2.6 random_unregister_seed_handler()

```
int random_unregister_seed_handler (
            random_hdlr_t func )
```

Un-register a random seed generator handler

This API can be used to un-register a handler registered using random_register_seed_handler()

Parameters

in	*func*	Function pointer of type random_hdlr_t used during registering

Returns

WM_SUCCESS if successful
-WM_E_INVAL if the passed pointer is invalid

5.12.2.7 random_initialize_seed()

```
void random_initialize_seed (
            void )
```

Initialize the random number generator's seed

The get_random_sequence() uses a random number generator that is initialized with a seed when get_random_sequence() is called for the first time. The handlers registered using random_register_seed_handler() are used to generate the seed. If an application wants to explicitly initialize the seed, this API can be used. The seed will then not be re-initialized in get_random_sequence().

5.12.2.8 sample_initialise_random_seed()

```
uint32_t sample_initialise_random_seed (
            void )
```

Sample random seed generator

This is a sample random seed generator handler that can be registered using random_register_seed_handler() to generate a random seed. This uses a combination of DAC (generating random noise) and ADC (that internally samples the random noise) along with the flash id to generate a seed. It is recommended to register this handler and immediately call random_initialize_seed() before executing any other application code, especially if the application is going to use ADC/DAC for its own purpose.

Returns

Random seed

5.12.2.9 get_random_sequence()

```
void get_random_sequence (
            void * buf,
            unsigned int size )
```

Generate random sequence of bytes

This function generates random sequence of bytes in the user provided buffer.

Parameters

out	*buf*	The buffer to be populated with random data
in	*size*	The number of bytes of the random sequence required

5.12.2.10 strdup()

```
char* strdup (
            const char * s )
```

Returns a pointer to a new string which is a duplicate of the input string s. Memory for the new string is obtained allocated by the function.

It is caller's responsibility to free the memory after its use.

Parameters

in	*s*	Pointer to string to be duplicated

Returns

Pointer to newly allocated string which is duplicate of input string
NULL on error

5.12.2.11 soft_crc32()

```
uint32_t soft_crc32 (
            const void * data__,
            int data_size,
            uint32_t crc )
```

Calculate CRC32 using software algorithm

Precondition

soft_crc32_init()

soft_crc32() allows the user to calculate CRC32 values of arbitrary sized buffers across multiple calls.

Parameters

in	*data__*	Input buffer over which CRC32 is calculated.
in	*data_size*	Length of the input buffer.
in	*crc*	Previous CRC32 value used as starting point for given buffer calculation.

Returns

Calculated CRC32 value

5.12.2.12 fill_sequential_pattern()

```
void fill_sequential_pattern (
            void * buffer,
            int size,
            uint8_t first_byte )
```

Fill the given buffer with a sequential pattern starting from given byte.

For example, if the 'first_byte' is 0x45 and buffer size of 5 then buffer will be set to {0x45, 0x46, 0x47, 0x48, 0x49}

Parameters

in	*buffer*	The pattern will be set to this buffer.
in	*size*	Number of pattern bytes to the be written to the buffer.
in	*first_byte*	This is the value of first byte in the sequential pattern.

5.12.2.13 verify_sequential_pattern()

```
bool verify_sequential_pattern (
            const void * buffer,
            int size,
            uint8_t first_byte )
```

Verify if the the given buffer has a sequential pattern starting from given byte.

For example, if the 'first_byte' is 0x45 and buffer size of 5 then buffer will be verified for presence of {0x45, 0x46, 0x47, 0x48, 0x49}

Parameters

in	*buffer*	The pattern will be verified from this buffer.
in	*size*	Number of pattern bytes to the be verified from the buffer.
in	*first_byte*	This is the value of first byte in the sequential pattern.

Returns

'true' If verification successful.
'false' If verification fails.

5.12.3 Macro Documentation

5.12.3.1 dump_hex

```
#define dump_hex(
              ... )
```

Value:

```
do                  \
    {                   \
    } while (0)
```

5.12.3.2 dump_hex_ascii

```
#define dump_hex_ascii(
              ... )
```

Value:

```
do                        \
    {                         \
    } while (0)
```

5.12.3.3 dump_ascii

```
#define dump_ascii(
                ... )
```

Value:

```
do                    \
    {                    \
    } while (0)
```

5.12.3.4 print_ascii

```
#define print_ascii(
                ... )
```

Value:

```
do                     \
    {                     \
    } while (0)
```

5.12.3.5 dump_json

```
#define dump_json(
                ... )
```

Value:

```
do                   \
    {                   \
    } while (0)
```

5.12.4 Typedef Documentation

5.12.4.1 random_hdlr_t

```
typedef uint32_t(* random_hdlr_t) (void)
```

Function prototype for a random entropy/seed generator

Returns

a 32bit random number

Index