RM00282

RW61x Wi-Fi Driver API for SDK 2.13.3

Rev. 1 — 12 December 2023

Reference manual
CONFIDENTIAL

Document information

Information	Content
Keywords	Wi-Fi driver, data structures, files
Abstract	Describes the data structures and files for RW61x Wi-Fi C API for SDK 2.13.3.

1 Data structure index

1.1 Data structures

Here are the data structures with brief descriptions:

2 File index

2.1 File list

List of the documented files with a brief description for each:

cli.h: CLI module

cli_utils.h: CLI Utils

dhcp-server.h: DHCP server

iperf.h: This file provides the support for network utility iperf

wifi-decl.h: Wi-Fi structure declarations

wifi.h: This file contains interface to Wi-Fi driver

wifi_cal_data_ext.h : This file contains the calibration data

wifi_events.h: Wi-Fi events

wifi_nxp.h: This file provides Core Wi-Fi definition for WPA supplicant RTOS driver

wifi_nxp_wps.h: WPS - Wi-Fi Protected Setup

wifi_ping.h: This file provides the support for network utility ping

wlan.h: Wi-Fi Connection Manager

wlan_11d.h: Wi-Fi module 11d API

wlan_tests.h: Wi-Fi Connection Manager Tests

wm_net.h: Network Abstraction Layer

wm_os.h: OS Abstraction Layer

wm_utils.h: Utility functions

wmcrypto.h: Crypto Functions

wmerrno.h: Error Management

wmlog.h: This file contains macros to print logs

wmstats.h: Wireless Microcontroller statistics

wmtime.h: Time Management Subsystem

wmtypes.h: Consolidated Header for Data types

3 Data structure documentation

3.1 cli_command structure reference

3.1.1 Data fields

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

3.1.2 Detailed description

Structure for registering CLI commands

3.1.3 Field documentation

3.1.3.1 const char* cli_command::name

The name of the CLI command

3.1.3.2 const char* cli_command::help

The help text associated with the command

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

The function that should be invoked for this command.

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

- cli.h

3.2 datetime_t structure reference

Structure is used to hold the date and time.

3.2.1 Data fields

- uint16_t year
- uint8_t month
- uint8_t day
- uint8_t hour
- uint8_t minute
- uint8_t second

3.2.2 Field documentation

3.2.2.1 uint16_t datetime_t::year

Range from 1970 to 2099.

3.2.2.2 uint8_t datetime_t::month

Range from 1 to 12.

3.2.2.3 uint8_t datetime_t::day

Range from 1 to 31 (depending on month).

3.2.2.4 uint8_t datetime_t::hour

Range from 0 to 23.

3.2.2.5 uint8_t datetime_t::minute

Range from 0 to 59.

3.2.2.6 uint8_t datetime_t::second

Range from 0 to 59.

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

- wmtime.h

3.3 DH_PG_PARAMS structure reference

3.3.1 Data fields

- unsigned char * prime
- unsigned int primeLen
- unsigned char * generator
- unsigned int generatorLen

3.3.2 Detailed description

- Diffie-Hellman parameters.

3.3.3 Field documentation

3.3.3.1 unsigned char* DH_PG_PARAMS::prime

prime

3.3.3.2 unsigned int DH_PG_PARAMS::primeLen

length of prime

3.3.3.3 unsigned char* DH_PG_PARAMS::generator

generator

3.3.3.4 unsigned int DH_PG_PARAMS::generatorLen

length of generator

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

- wmcrypto.h

3.4 ipv4_config structure reference

3.4.1 Data fields

- enum address_typesaddr_type
- unsigned address
- unsigned gw
- unsigned netmask
- unsigned dns1
- unsigned dns2

3.4.2 Detailed description

This data structure represents an IPv4 address

3.4.3 Field documentation

3.4.3.1 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.

3.4.3.2 unsigned ipv4_config::address

The system's IP address in network order.

3.4.3.3 unsigned ipv4_config::gw

The system's default gateway in network order.

3.4.3.4 unsigned ipv4_config::netmask

The system's subnet mask in network order.

3.4.3.5 unsigned ipv4_config::dns1

The system's primary dns server in network order.

3.4.3.6 unsigned ipv4_config::dns2

The system's secondary dns server in network order.

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

- wlan.h

3.5 ipv6_config structure reference

3.5.1 Data fields

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

3.5.2 Detailed description

This data structure represents an IPv6 address

3.5.3 Field documentation

3.5.3.1 unsigned ipv6_config::address[4]

The system's IPv6 address in network order.

3.5.3.2 unsigned char ipv6_config::addr_type

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

3.5.3.3 unsigned char ipv6_config::addr_state

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

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

- wlan.h

3.6 net_ip_config structure reference

3.6.1 Data fields

- struct net_ipv4_configipv4

3.6.2 Detailed description

Network IP configuration.

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

3.6.3 Field documentation

3.6.3.1 struct net_ipv4_config net_ip_config::ipv4

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

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

- wm_net.h

3.7 net_ipv4_config structure reference

3.7.1 Data fields

- enum net_address_typesaddr_type
- unsigned address
- unsigned gw
- unsigned netmask
- unsigned dns1
- unsigned dns2

3.7.2 Detailed description

This data structure represents an IPv4 address

3.7.3 Field documentation

3.7.3.1 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.

3.7.3.2 unsigned net_ipv4_config::address

The system's IP address in network order.

3.7.3.3 unsigned net_ipv4_config::gw

The system's default gateway in network order.

3.7.3.4 unsigned net_ipv4_config::netmask

The system's subnet mask in network order.

3.7.3.5 unsigned net_ipv4_config::dns1

The system's primary dns server in network order.

3.7.3.6 unsigned net_ipv4_config::dns2

The system's secondary dns server in network order.

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

- wm_net.h

3.8 os_queue_pool_t structure reference

3.8.1 Data fields

- int size

3.8.2 Detailed description

Structure used for queue definition

3.8.3 Field documentation

3.8.3.1 int os_queue_pool_t::size

Size of the queue

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

- wm_os.h

3.9 os_thread_stack_t structure reference

3.9.1 Data fields

- size_t size

3.9.2 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.

3.9.3 Field documentation

3.9.3.1 size_t os_thread_stack_t::size

Total stack size

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

- wm_os.h

3.10 rx_pkt_he_rate_info structure reference

3.10.1 Data fields

- t_u32 hemcs_rxcnt [12]
- t_u32 hestbcrate_rxcnt [12]

3.10.2 Detailed description

RX histiogram he statistic parameters

3.10.3 Field documentation

3.10.3.1 t_u32 rx_pkt_he_rate_info::hemcs_rxcnt[12]

Rx packet counter of MCS0~MCS11

3.10.3.2 t_u32 rx_pkt_he_rate_info::hestbcrate_rxcnt[12]

Rx STBC packet counter of MCS0~MCS11

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

- wlan.h

3.11 rx_pkt_ht_rate_info structure reference

3.11.1 Data fields

- t_u32 htmcs_rxcnt [16]
- t_u32 htsgi_rxcnt [16]
- t_u32 htstbcrate_rxcnt [16]

3.11.2 Detailed description

RX histiogram ht statistic parameters

3.11.3 Field documentation

3.11.3.1 t_u32 rx_pkt_ht_rate_info::htmcs_rxcnt[16]

Rx packet counter of MCS0~MCS15

3.11.3.2 t_u32 rx_pkt_ht_rate_info::htsgi_rxcnt[16]

Rx packet's short GI counter of MCS0~MCS15

3.11.3.3 t_u32 rx_pkt_ht_rate_info::htstbcrate_rxcnt[16]

Rx STBC packet counter of MCS0~MCS15

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

- wlan.h

3.12 rx_pkt_rate_info structure reference

3.12.1 Data fields

- t_u32 nss_rxcnt [2]
- t_u32 nsts_rxcnt
- t_u32 bandwidth_rxcnt [3]
- t_u32 preamble_rxcnt [6]
- t_u32 ldpc_txbfcnt [2]
- t_s32 rssi_value [2]
- t_s32 rssi_chain0 [4]
- t_s32 rssi_chain1 [4]

3.12.2 Detailed description

RX histogram statistic parameters

3.12.3 Field documentation

3.12.3.1 t_u32 rx_pkt_rate_info::nss_rxcnt[2]

Rx packet counter of every NSS, NSS=1,2

3.12.3.2 t_u32 rx_pkt_rate_info::nsts_rxcnt

Received packet counter which using STBC

3.12.3.3 t_u32 rx_pkt_rate_info::bandwidth_rxcnt[3]

Rx packet counter of every bandwith

3.12.3.4 t_u32 rx_pkt_rate_info::preamble_rxcnt[6]

Different preamble Rx packet counter

3.12.3.5 t_u32 rx_pkt_rate_info::ldpc_txbfcnt[2]

VHT SIGA2 LDPC bit

3.12.3.6 t_s32 rx_pkt_rate_info::rssi_value[2]

Average RSSI

3.12.3.7 t_s32 rx_pkt_rate_info::rssi_chain0[4]

RSSI value of path A

3.12.3.8 t_s32 rx_pkt_rate_info::rssi_chain1[4]

RSSI value of path B

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

- wlan.h

3.13 rx_pkt_vht_rate_info structure reference

3.13.1 Data fields

- t_u32 vhtmcs_rxcnt [10]
- t_u32 vhtsgi_rxcnt [10]
- t_u32 vhtstbcrate_rxcnt [10]

3.13.2 Detailed description

RX histiogram vht statistic parameters

3.13.3 Field documentation

3.13.3.1 t_u32 rx_pkt_vht_rate_info::vhtmcs_rxcnt[10]

Rx packet counter of MCS0~MCS9

3.13.3.2 t_u32 rx_pkt_vht_rate_info::vhtsgi_rxcnt[10]

Rx packet's short GI counter of MCS0~MCS9

3.13.3.3 t_u32 rx_pkt_vht_rate_info::vhtstbcrate_rxcnt[10]

Rx STBC packet counter of MCS0~MCS9

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

- wlan.h

3.14 tx_ampdu_prot_mode_para structure reference

3.14.1 Data fields

- int mode

3.14.2 Detailed description

tx_ampdu_prot_mode parameters

3.14.3 Field documentation

3.14.3.1 int tx_ampdu_prot_mode_para::mode

set prot mode

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

- wlan.h

3.15 tx_pkt_he_rate_info structure reference

3.15.1 Data fields

- t_u32 hemcs_txcnt [12]
- t_u32 hestbcrate_txcnt [12]

3.15.2 Detailed description

TX histiogram he statistic parameters

3.15.3 Field documentation

3.15.3.1 t_u32 tx_pkt_he_rate_info::hemcs_txcnt[12]

tx packet counter of MCS0~MCS11

3.15.3.2 t_u32 tx_pkt_he_rate_info::hestbcrate_txcnt[12]

tx STBC packet counter of MCS0~MCS11

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

- wlan.h

3.16 tx_pkt_ht_rate_info structure reference

3.16.1 Data fields

- t_u32 htmcs_txcnt [16]
- t_u32 htsgi_txcnt [16]
- t_u32 htstbcrate_txcnt [16]

3.16.2 Detailed description

TX histiogram ht statistic parameters

3.16.3 Field documentation

3.16.3.1 t_u32 tx_pkt_ht_rate_info::htmcs_txcnt[16]

tx packet counter of MCS0~MCS15

3.16.3.2 t_u32 tx_pkt_ht_rate_info::htsgi_txcnt[16]

tx packet's short GI counter of MCS0~MCS15

3.16.3.3 t_u32 tx_pkt_ht_rate_info::htstbcrate_txcnt[16]

tx STBC packet counter of MCS0~MCS15

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

- wlan.h

3.17 tx_pkt_rate_info structure reference

3.17.1 Data fields

- t_u32 nss_txcnt [2]
- t_u32 bandwidth_txcnt [3]
- t_u32 preamble_txcnt [4]
- t_u32 ldpc_txcnt
- t_u32 rts_txcnt
- t_s32 ack_RSSI

3.17.2 Detailed description

TX histogram statistic parameters

3.17.3 Field documentation

3.17.3.1 t_u32 tx_pkt_rate_info::nss_txcnt[2]

tx packet counter of every NSS, NSS=1,2

3.17.3.2 t_u32 tx_pkt_rate_info::bandwidth_txcnt[3]

tx packet counter of every bandwith

3.17.3.3 t_u32 tx_pkt_rate_info::preamble_txcnt[4]

different preamble tx packet counter

3.17.3.4 t_u32 tx_pkt_rate_info::ldpc_txcnt

tx packet counter of using LDPC coding

3.17.3.5 t_u32 tx_pkt_rate_info::rts_txcnt

transmitted RTS counter

3.17.3.6 t_s32 tx_pkt_rate_info::ack_RSSI

RSSI of ack

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

- wlan.h

3.18 tx_pkt_vht_rate_info structure reference

3.18.1 Data fields

- t_u32 vhtmcs_txcnt [10]
- t_u32 vhtsgi_txcnt [10]
- t_u32 vhtstbcrate_txcnt [10]

3.18.2 Detailed description

TX histiogram vht statistic parameters

3.18.3 Field documentation

3.18.3.1 t_u32 tx_pkt_vht_rate_info::vhtmcs_txcnt[10]

tx packet counter of MCS0~MCS9

3.18.3.2 t_u32 tx_pkt_vht_rate_info::vhtsgi_txcnt[10]

tx packet's short GI counter of MCS0~MCS9

3.18.3.3 t_u32 tx_pkt_vht_rate_info::vhtstbcrate_txcnt[10]

tx STBC packet counter of MCS0~MCS9

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

- wlan.h

3.19 wifi_antcfg_t structure reference

3.19.1 Data fields

- t_u32 * ant_mode
- t_u16 * evaluate_time
- t_u16 * current_antenna
- t_u8 ∗ evaluate_mode

3.19.2 Detailed description

Type definition of wifi_antcfg_t

3.19.3 Field documentation

3.19.3.1 t_u32* wifi_antcfg_t::ant_mode

Antenna Mode

3.19.3.2 t_u16* wifi_antcfg_t::evaluate_time

Evaluate Time

3.19.3.3 t_u16* wifi_antcfg_t::current_antenna

Current antenna

3.19.3.4 t_u8* wifi_antcfg_t::evaluate_mode

Evaluate mode

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

- wifi-decl.h

3.20 wifi_auto_reconnect_config_t structure reference

3.20.1 Data fields

- t_u8 reconnect_counter
- t_u8 reconnect_internal
- t_u16 config_tflags

3.20.2 Detailed description

Auto reconnect structure

3.20.3 Field documentation

3.20.3.1 t_u8 wifi_auto_reconnect_config_t::reconnect_counter

Reconnect counter

3.20.3.2 t_u8 wifi_auto_reconnect_config_t::reconnect_interval

Reconnect interval

3.20.3.3 t_u16 wifi_auto_reconnect_config_t::flags

Flags

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

- wifi-decl.h

3.21 wifi_bandcfg_t structure reference

3.21.1 Data fields

- t_u16 config_bands
- t_u16 fw_bands

3.21.2 Detailed description

Type definition of wifi_bandcfg_t

3.21.3 Field documentation

3.21.3.1 t_u16 wifi_bandcfg_t::config_bands

Infra band

3.21.3.2 t_u16 wifi_bandcfg_t::fw_bands

fw supported band

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

- wifi-decl.h

3.22 wifi_cal_data_t structure reference

3.22.1 Data fields

- t_u16 data_len
- t_u8 * data

3.22.2 Detailed description

Calibration Data

3.22.3 Field documentation

3.22.3.1 t_u16 wifi_cal_data_t::data_len

Calibration data length

3.22.3.2 t_u8* wifi_cal_data_t::data

Calibration data

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

- wifi-decl.h

3.23 wifi_chan_info_t structure reference

3.23.1 Data fields

- t_u8 chan_num
- t_u16 chan_freq
- bool passive_scan_or_radar_detect

3.23.2 Detailed description

Data structure for Channel attributes

3.23.3 Field documentation

3.23.3.1 t_u8 wifi_chan_info_t::chan_num

Channel Number

3.23.3.2 t_u16 wifi_chan_info_t::chan_freq

Channel frequency for this channel

3.23.3.3 bool wifi_chan_info_t::passive_scan_or_radar_detect

Passice Scan or RADAR Detect

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

- wifi-decl.h

3.24 wifi_chan_list_param_set_t structure reference

3.24.1 Data fields

- t_u8 no_of_channels
- wifi_chan_scan_param_set_tchan_scan_param [1]

3.24.2 Detailed description

Channel list parameter set

3.24.3 Field documentation

3.24.3.1 t_u8 wifi_chan_list_param_set_t::no_of_channels

number of channels

3.24.3.2 wifi_chan_scan_param_set_t wifi_chan_list_param_set_t::chan_scan_param[1]

channel scan array

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

- wifi-decl.h

3.25 wifi_chan_scan_param_set_t structure reference

3.25.1 Data fields

- t_u8 chan_number
- t_u16 min_scan_time
- t_u16 max_scan_time

3.25.2 Detailed description

Channel scan parameters

3.25.3 Field documentation

3.25.3.1 t_u8 wifi_chan_scan_param_set_t::chan_number

channel number

3.25.3.2 t_u16 wifi_chan_scan_param_set_t::min_scan_time

minimum scan time

3.25.3.3 t_u16 wifi_chan_scan_param_set_t::max_scan_time

maximum scan time

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

- wifi-decl.h

3.26 wifi_chanlist_t structure reference

3.26.1 Data fields

- t_u8 num_chans
- wifi_chan_info_tchan_info [54]

3.26.2 Detailed description

Data structure for Channel List Config

3.26.3 Field documentation

3.26.3.1 t_u8 wifi_chanlist_t::num_chans

Number of Channels

3.26.3.2 wifi_chan_info_t wifi_chanlist_t::chan_info[54]

Channel Info

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

- wifi-decl.h

3.27 wifi_channel_desc_t structure reference

3.27.1 Data fields

- t_u16 start_freq
- t_u8 chan_width
- t_u8 chan_num

3.27.2 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

3.27.3 Field documentation

3.27.3.1 t_u16 wifi_channel_desc_t::start_freq

Starting frequency of the band for this channel

3.27.3.2 t_u8 wifi_channel_desc_t::chan_width

Channel width

3.27.3.3 t_u8 wifi_channel_desc_t::chan_num

Channel Number

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

- wifi-decl.h

3.28 wifi_cw_mode_ctrl_t structure reference

3.28.1 Data fields

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

3.28.2 Detailed description

CW_MODE_CTRL structure

3.28.3 Field documentation

3.28.3.1 t_u8 wifi_cw_mode_ctrl_t::mode

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

3.28.3.2 t_u8 wifi_cw_mode_ctrl_t::channel

channel

3.28.3.3 t_u8 wifi_cw_mode_ctrl_t::chanInfo

channel info

3.28.3.4 t_u16 wifi_cw_mode_ctrl_t::txPower

Tx Power level in dBm

3.28.3.5 t_u16 wifi_cw_mode_ctrl_t::pktLength

Packet Length

3.28.3.6 t_u32 wifi_cw_mode_ctrl_t::rateInfo

bit rate info

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

- wifi-decl.h

3.29 wifi_data_rate_t structure reference

3.29.1 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
- t_u32 tx_mcs_index
- t_u32 rx_mcs_index
- mlan_rate_format tx_rate_format
- mlan_rate_format rx_rate_format

3.29.2 Detailed description

Data structure for cmd get data rate

3.29.3 Field documentation

3.29.3.1 t_u32 wifi_data_rate_t::tx_data_rate

Tx data rate

3.29.3.2 t_u32 wifi_data_rate_t::rx_data_rate

Rx data rate

3.29.3.3 t_u32 wifi_data_rate_t::tx_bw

Tx channel bandwidth

3.29.3.4 t_u32 wifi_data_rate_t::tx_gi

Tx guard interval

3.29.3.5 t_u32 wifi_data_rate_t::rx_bw

Rx channel bandwidth

3.29.3.6 t_u32 wifi_data_rate_t::rx_gi

Rx guard interval

3.29.3.7 t_u32 wifi_data_rate_t::tx_mcs_index

MCS index

3.29.3.8 t_u32 wifi_data_rate_t::rx_mcs_index

MCS index

3.29.3.9 mlan_rate_format wifi_data_rate_t::tx_rate_format

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

3.29.3.10 mlan_rate_format wifi_data_rate_t::rx_rate_format

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

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

- wifi-decl.h

3.30 wifi_ds_rate structure reference

3.30.1 Data fields

- enum wifi_ds_command_type sub_command
- union {
- wifi_rate_cfg_trate_cfg
- wifi_data_rate_tdata_rate
- } param

3.30.2 Detailed description

Type definition of wifi_ds_rate

3.30.3 Field documentation

3.30.3.1 enum wifi_ds_command_type wifi_ds_rate::sub_command

Sub-command

3.30.3.2 wifi_rate_cfg_t wifi_ds_rate::rate_cfg

Rate configuration for MLAN_OID_RATE_CFG

3.30.3.3 wifi_data_rate_t wifi_ds_rate::data_rate

Data rate for MLAN_OID_GET_DATA_RATE

3.30.3.4 union { ... } wifi_ds_rate::param

Rate configuration parameter

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

- wifi-decl.h

3.31 wifi_ed_mac_ctrl_t structure reference

3.31.1 Data fields

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

3.31.2 Detailed description

Type definition of wifi_ed_mac_ctrl_t

3.31.3 Field documentation

3.31.3.1 t_u16 wifi_ed_mac_ctrl_t::ed_ctrl_2g

ED CTRL 2G

3.31.3.2 t_s16 wifi_ed_mac_ctrl_t::ed_offset_2g

ED Offset 2G

3.31.3.3 t_u16 wifi_ed_mac_ctrl_t::ed_ctrl_5g

ED CTRL 5G

3.31.3.4 t_s16 wifi_ed_mac_ctrl_t::ed_offset_5g

ED Offset 5G

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

- wifi-decl.h

3.32 wifi_flt_cfg_t structure reference

3.32.1 Data fields

- t_u32 criteria
- t_u16 nentries
- wifi_mef_entry_tmef_entry [MAX_NUM_ENTRIES]

3.32.2 Detailed description

Wifi filter config struct

3.32.3 Field documentation

3.32.3.1 t_u32 wifi_flt_cfg_t::criteria

Filter Criteria

3.32.3.2 t_u16 wifi_flt_cfg_t::nentries

Number of entries

3.32.3.3 wifi_mef_entry_t wifi_flt_cfg_t::mef_entry[MAX_NUM_ENTRIES]

MEF entry

Refer to wifi_mef_entry_t structure reference .

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

- wifi-decl.h

3.33 wifi_fw_version_ext_t structure reference

3.33.1 Data fields

- uint8_t version_str_sel
- char version_str [MLAN_MAX_VER_STR_LEN]

3.33.2 Detailed description

Extended Firmware version

3.33.3 Field documentation

3.33.3.1 uint8_t wifi_fw_version_ext_t::version_str_sel

ID for extended version select

3.33.3.2 char wifi_fw_version_ext_t::version_str[MLAN_MAX_VER_STR_LEN]

Firmware version string

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

- wifi-decl.h

3.34 wifi_fw_version_t structure reference

3.34.1 Data fields

- char version_str [MLAN_MAX_VER_STR_LEN]

3.34.2 Detailed description

Firmware version

3.34.3 Field documentation

3.34.3.1 char wifi_fw_version_t::version_str[MLAN_MAX_VER_STR_LEN]

Firmware version string

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

- wifi-decl.h

3.35 wifi_mac_addr_t structure reference

3.35.1 Data fields

- char mac [MLAN_MAC_ADDR_LENGTH]

3.35.2 Detailed description

MAC address

3.35.3 Field documentation

3.35.3.1 char wifi_mac_addr_t::mac[MLAN_MAC_ADDR_LENGTH]

Mac address array

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

- wifi-decl.h

3.36 wifi_mef_entry_t structure reference

3.36.1 Data fields

- t_u8 mode
- t_u8 action
- t_u8 filter_num
- wifi_mef_filter_tfilter_item [MAX_NUM_FILTERS]
- t_u8 rpn [MAX_NUM_FILTERS]

3.36.2 Detailed description

MEF entry struct

3.36.3 Field documentation

3.36.3.1 t_u8 wifi_mef_entry_t::mode

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

3.36.3.2 t_u8 wifi_mef_entry_t::action

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

3.36.3.3 t_u8 wifi_mef_entry_t::filter_num

filter number

3.36.3.4 wifi_mef_filter_t wifi_mef_entry_t::filter_item[MAX_NUM_FILTERS]

filter array

3.36.3.5 t_u8 wifi_mef_entry_t::rpn[MAX_NUM_FILTERS]

rpn array

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

- wifi-decl.h

3.37 wifi_mef_filter_t structure reference

3.37.1 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]

3.37.2 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

3.37.3 Field documentation

3.37.3.1 t_u32 wifi_mef_filter_t::fill_flag

flag

3.37.3.2 t_u16 wifi_mef_filter_t::type

BYTE 0X41; Decimal 0X42; Bit 0x43

3.37.3.3 t_u32 wifi_mef_filter_t::pattern

value

3.37.3.4 t_u16 wifi_mef_filter_t::offset

offset

3.37.3.5 t_u16 wifi_mef_filter_t::num_bytes

number of bytes

3.37.3.6 t_u16 wifi_mef_filter_t::repeat

repeat

3.37.3.7 t_u8 wifi_mef_filter_t::num_byte_seq

byte number

3.37.3.8 t_u8 wifi_mef_filter_t::byte_seq[MAX_NUM_BYTE_SEQ]

array

3.37.3.9 t_u8 wifi_mef_filter_t::num_mask_seq

mask numbers

3.37.3.10 t_u8 wifi_mef_filter_t::mask_seq[MAX_NUM_MASK_SEQ]

array

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

- wifi-decl.h

3.38 wifi_mgmt_frame_t structure reference

3.38.1 Data fields

- t_u16 frm_len
- wifi_frame_type_tframe_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]

3.38.2 Detailed description

802_11_header packet

3.38.3 Field documentation

3.38.3.1 t_u16 wifi_mgmt_frame_t::frm_len

Packet Length

3.38.3.2 wifi_frame_type_t wifi_mgmt_frame_t::frame_type

Frame Type

3.38.3.3 t_u8 wifi_mgmt_frame_t::frame_ctrl_flags

Frame Control flags

3.38.3.4 t_u16 wifi_mgmt_frame_t::duration_id

Duration ID

3.38.3.5 t_u8 wifi_mgmt_frame_t::addr1[MLAN_MAC_ADDR_LENGTH]

Address 1

3.38.3.6 t_u8 wifi_mgmt_frame_t::addr2[MLAN_MAC_ADDR_LENGTH]

Address 2

3.38.3.7 t_u8 wifi_mgmt_frame_t::addr3[MLAN_MAC_ADDR_LENGTH]

Address 3

3.38.3.8 t_u16 wifi_mgmt_frame_t::seq_ctl

Sequence Control

3.38.3.9 t_u8 wifi_mgmt_frame_t::addr4[MLAN_MAC_ADDR_LENGTH]

Address 4

3.38.3.10 t_u8 wifi_mgmt_frame_t::payload[1]

Frame payload

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

- wifi-decl.h

3.39 wifi_nat_keep_alive_t structure reference

3.39.1 Data fields

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

3.39.2 Detailed description

TCP nat keep alive information

3.39.3 Field documentation

3.39.3.1 t_u16 wifi_nat_keep_alive_t::interval

Keep alive interval

3.39.3.2 t_u8 wifi_nat_keep_alive_t::dst_mac[MLAN_MAC_ADDR_LENGTH]

Destination MAC address

3.39.3.3 t_u32 wifi_nat_keep_alive_t::dst_ip

Destination IP

3.39.3.4 t_u16 wifi_nat_keep_alive_t::dst_port

Destination port

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

- wifi-decl.h

3.40 wifi_rate_cfg_t structure reference

3.40.1 Data fields

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

3.40.2 Detailed description

Data structure for cmd txratecfg

3.40.3 Field documentation

3.40.3.1 mlan_rate_format wifi_rate_cfg_t::rate_format

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

3.40.3.2 t_u32 wifi_rate_cfg_t::rate_index

Rate/MCS index (0xFF: auto)

3.40.3.3 t_u32 wifi_rate_cfg_t::rate

Rate rate

3.40.3.4 t_u16 wifi_rate_cfg_t::rate_setting

Rate Setting

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

- wifi-decl.h

3.41 wifi_remain_on_channel_t structure reference

3.41.1 Data fields

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

3.41.2 Detailed description

Remain on channel info structure

3.41.3 Field documentation

3.41.3.1 uint16_t wifi_remain_on_channel_t::remove

Remove

3.41.3.2 uint8_t wifi_remain_on_channel_t::status

Current status

3.41.3.3 uint8_t wifi_remain_on_channel_t::bandcfg

band configuration

3.41.3.4 uint8_t wifi_remain_on_channel_t::channel

Channel

3.41.3.5 uint32_t wifi_remain_on_channel_t::remain_period

Remain on channel period

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

- wifi-decl.h

3.42 wifi_rf_channel_t structure reference

3.42.1 Data fields

- uint16_t current_channel
- uint16_t rf_type

3.42.2 Detailed description

Rf channel

3.42.3 Field documentation

3.42.3.1 uint16_t wifi_rf_channel_t::current_channel

Current channel

3.42.3.2 uint16_t wifi_rf_channel_t::rf_type

RF Type

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

- wifi-decl.h

3.43 wifi_rssi_info_t structure reference

3.43.1 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

3.43.2 Detailed description

RSSI information

3.43.3 Field documentation

3.43.3.1 int16_t wifi_rssi_info_t::data_rssi_last

Data RSSI last

3.43.3.2 int16_t wifi_rssi_info_t::data_nf_last

Data nf last

3.43.3.3 int16_t wifi_rssi_info_t::data_rssi_avg

Data RSSI average

3.43.3.4 int16_t wifi_rssi_info_t::data_nf_avg

Data nf average

3.43.3.5 int16_t wifi_rssi_info_t::bcn_snr_last

BCN SNR

3.43.3.6 int16_t wifi_rssi_info_t::bcn_snr_avg

BCN SNR average

3.43.3.7 int16_t wifi_rssi_info_t::data_snr_last

Data SNR last

3.43.3.8 int16_t wifi_rssi_info_t::data_snr_avg

Data SNR average

3.43.3.9 int16_t wifi_rssi_info_t::bcn_rssi_last

BCN RSSI

3.43.3.10 int16_t wifi_rssi_info_t::bcn_nf_last

BCN nf

3.43.3.11 int16_t wifi_rssi_info_t::bcn_rssi_avg

BCN RSSI average

3.43.3.12 int16_t wifi_rssi_info_t::bcn_nf_avg

BCN nf average

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

- wifi-decl.h

3.44 wifi_scan_chan_list_t structure reference

3.44.1 Data fields

- uint8_t num_of_chan
- uint8_t chan_number [MLAN_MAX_CHANNEL]

3.44.2 Detailed description

Channel list structure

3.44.3 Field documentation

3.44.3.1 uint8_t wifi_scan_chan_list_t::num_of_chan

Number of channels

3.44.3.2 uint8_t wifi_scan_chan_list_t::chan_number[MLAN_MAX_CHANNEL]

Channel number

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

- wifi-decl.h

3.45 wifi_scan_channel_list_t structure reference

3.45.1 Data fields

- t_u8 chan_number
- mlan_scan_type scan_type
- t_u16 scan_time

3.45.2 Detailed description

Scan channel list

3.45.3 Field documentation

3.45.3.1 t_u8 wifi_scan_channel_list_t::chan_number

Channel numder

3.45.3.2 mlan_scan_type wifi_scan_channel_list_t::scan_type

Scan type Active = 1, Passive = 2

3.45.3.3 t_u16 wifi_scan_channel_list_t::scan_time

Scan time

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

- wifi-decl.h

3.46 wifi_scan_params_v2_t structure reference

3.46.1 Data fields

- 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_tchan_list [MAX_CHANNEL_LIST]
- t_u8 num_probes
- int(* cb)(unsigned int count)

3.46.2 Detailed description

V2 scan parameters

3.46.3 Field documentation

3.46.3.1 t_u8 wifi_scan_params_v2_t::bssid[MLAN_MAC_ADDR_LENGTH]

BSSID to scan

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

SSID to scan

3.46.3.3 t_u8 wifi_scan_params_v2_t::num_channels

Number of channels

3.46.3.4 wifi_scan_channel_list_t wifi_scan_params_v2_t::chan_list[MAX_CHANNEL_LIST]

Channel list with channel information

3.46.3.5 t_u8 wifi_scan_params_v2_t::num_probes

Number of probes

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

Callback to be called when scan is completed

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

- wifi-decl.h

3.47 wifi_scan_result2 structure reference

3.47.1 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 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

3.47.2 Detailed description

Scan result information

3.47.3 Field documentation

3.47.3.1 uint8_t wifi_scan_result2::bssid[MLAN_MAC_ADDR_LENGTH]

BSSID array

3.47.3.2 bool wifi_scan_result2::is_ibss_bit_set

Is bssid set?

3.47.3.3 uint8_t wifi_scan_result2::ssid[MLAN_MAX_SSID_LENGTH]

ssid array

3.47.3.4 int wifi_scan_result2::ssid_len

SSID length

3.47.3.5 uint8_t wifi_scan_result2::Channel

Channel associated to the BSSID

3.47.3.6 uint8_t wifi_scan_result2::RSSI

Received signal strength

3.47.3.7 uint16_t wifi_scan_result2::beacon_period

Beacon period

3.47.3.8 uint16_t wifi_scan_result2::dtim_period

DTIM period

3.47.3.9 _SecurityMode_t wifi_scan_result2::WPA_WPA2_WEP

Security mode info

3.47.3.10 _Cipher_t wifi_scan_result2::wpa_mcstCipher

WPA multicast cipher

3.47.3.11 _Cipher_t wifi_scan_result2::wpa_ucstCipher

WPA unicast cipher

3.47.3.12 _Cipher_t wifi_scan_result2::rsn_mcstCipher

No security multicast cipher

3.47.3.13 _Cipher_t wifi_scan_result2::rsn_ucstCipher

No security unicast cipher

3.47.3.14 bool wifi_scan_result2::is_pmf_required

Is pmf required flag

3.47.3.15 t_u8 wifi_scan_result2::ap_mfpc

MFPC bit of AP

3.47.3.16 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

3.47.3.17 bool wifi_scan_result2::phtcap_ie_present

PHT CAP IE present info

3.47.3.18 bool wifi_scan_result2::phtinfo_ie_present

PHT INFO IE present info

3.47.3.19 bool wifi_scan_result2::wmm_ie_present

WMM IE present info

3.47.3.20 uint16_t wifi_scan_result2::band

Band info

3.47.3.21 bool wifi_scan_result2::wps_IE_exist

WPS IE exist info

3.47.3.22 uint16_t wifi_scan_result2::wps_session

WPS session

3.47.3.23 bool wifi_scan_result2::wpa2_entp_IE_exist

WPA2 enterprise IE exist info

3.47.3.24 uint8_t wifi_scan_result2::trans_mode

Trans mode

3.47.3.25 uint8_t wifi_scan_result2::trans_bssid[MLAN_MAC_ADDR_LENGTH]

Trans bssid array

3.47.3.26 uint8_t wifi_scan_result2::trans_ssid[MLAN_MAX_SSID_LENGTH]

Trans ssid array

3.47.3.27 int wifi_scan_result2::trans_ssid_len

Trans bssid length

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

- wifi-decl.h

3.48 wifi_sta_info_t structure reference

3.48.1 Data fields

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

3.48.2 Detailed description

Station information structure

3.48.3 Field documentation

3.48.3.1 t_u8 wifi_sta_info_t::mac[MLAN_MAC_ADDR_LENGTH]

MAC address buffer

3.48.3.2 t_u8 wifi_sta_info_t::power_mgmt_status

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

3.48.3.3 t_s8 wifi_sta_info_t::rssi

RSSI: dBm

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

- wifi-decl.h

3.49 wifi_sta_list_t structure reference

3.49.1 Data fields

- int count

3.49.2 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.*

3.49.3 Field documentation

3.49.3.1 int wifi_sta_list_t::count

Count

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

- wifi-decl.h

3.50 wifi_sub_band_set_t structure reference

3.50.1 Data fields

- t_u8 first_chan
- t_u8 no_of_chan
- t_u8 max_tx_pwr

3.50.2 Detailed description

Data structure for subband set

For uAP 11d support

3.50.3 Field documentation

3.50.3.1 t_u8 wifi_sub_band_set_t::first_chan

First channel

3.50.3.2 t_u8 wifi_sub_band_set_t::no_of_chan

Number of channels

3.50.3.3 t_u8 wifi_sub_band_set_t::max_tx_pwr

Maximum Tx power in dBm

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

- wifi-decl.h

3.51 wifi_tbtt_offset_t structure reference

3.51.1 Data fields

- t_u32 min_tbtt_offset
- t_u32 max_tbtt_offset
- t_u32 avg_tbtt_offset

3.51.2 Detailed description

TBTT offset structure

3.51.3 Field documentation

3.51.3.1 t_u32 wifi_tbtt_offset_t::min_tbtt_offset

Min TBTT offset

3.51.3.2 t_u32 wifi_tbtt_offset_t::max_tbtt_offset

Max TBTT offset

3.51.3.3 t_u32 wifi_tbtt_offset_t::avg_tbtt_offset

AVG TBTT offset

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

- wifi-decl.h

3.52 wifi_tcp_keep_alive_t structure reference

3.52.1 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

3.52.2 Detailed description

TCP keep alive information

3.52.3 Field documentation

3.52.3.1 t_u8 wifi_tcp_keep_alive_t::enable

Enable keep alive

3.52.3.2 t_u8 wifi_tcp_keep_alive_t::reset

Reset

3.52.3.3 t_u32 wifi_tcp_keep_alive_t::timeout

Keep alive timeout

3.52.3.4 t_u16 wifi_tcp_keep_alive_t::interval

Keep alive interval

3.52.3.5 t_u16 wifi_tcp_keep_alive_t::max_keep_alives

Maximum keep alives

3.52.3.6 t_u8 wifi_tcp_keep_alive_t::dst_mac[MLAN_MAC_ADDR_LENGTH]

Destination MAC address

3.52.3.7 t_u32 wifi_tcp_keep_alive_t::dst_ip

Destination IP

3.52.3.8 t_u16 wifi_tcp_keep_alive_t::dst_tcp_port

Destination TCP port

3.52.3.9 t_u16 wifi_tcp_keep_alive_t::src_tcp_port

Source TCP port

3.52.3.10 t_u32 wifi_tcp_keep_alive_t::seq_no

Sequence number

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

- wifi-decl.h

3.53 wifi_tx_power_t structure reference

3.53.1 Data fields

- uint16_t current_level
- uint8_t max_power
- uint8_t min_power

3.53.2 Detailed description

Tx power levels

3.53.3 Field documentation

3.53.3.1 uint16_t wifi_tx_power_t::current_level

Current power level

3.53.3.2 uint8_t wifi_tx_power_t::max_power

Maximum power level

3.53.3.3 uint8_t wifi_tx_power_t::min_power

Minimum power level

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

- wifi-decl.h

3.54 wifi_txpwrlimit_config_t structure reference

3.54.1 Data fields

- t_u8 num_mod_grps
- wifi_channel_desc_tchan_desc
- wifi_txpwrlimit_entry_ttxpwrlimit_entry [10]

3.54.2 Detailed description

Data structure for TRPC config

For TRPC support

3.54.3 Field documentation

3.54.3.1 t_u8 wifi_txpwrlimit_config_t::num_mod_grps

Number of modulation groups

3.54.3.2 wifi_channel_desc_t wifi_txpwrlimit_config_t::chan_desc

Chnannel descriptor

3.54.3.3 wifi_txpwrlimit_entry_t wifi_txpwrlimit_config_t::txpwrlimit_entry[10]

Channel Modulation groups

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

- wifi-decl.h

3.55 wifi_txpwrlimit_entry_t structure reference

3.55.1 Data fields

- t_u8 mod_group
- t_u8 tx_power

3.55.2 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

3.55.3 Field documentation

3.55.3.1 t_u8 wifi_txpwrlimit_entry_t::mod_group

Modulation group

3.55.3.2 t_u8 wifi_txpwrlimit_entry_t::tx_power

Tx Power

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

- wifi-decl.h

3.56 wifi_txpwrlimit_t structure reference

3.56.1 Data fields

- wifi_SubBand_tsubband
- t_u8 num_chans
- wifi_txpwrlimit_config_ttxpwrlimit_config [40]

3.56.2 Detailed description

Data structure for Channel TRPC config

For TRPC support

3.56.3 Field documentation

3.56.3.1 wifi_SubBand_t wifi_txpwrlimit_t::subband

SubBand

3.56.3.2 t_u8 wifi_txpwrlimit_t::num_chans

Number of Channels

3.56.3.3 wifi_txpwrlimit_config_t wifi_txpwrlimit_t::txpwrlimit_config[40]

TRPC config

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

- wifi-decl.h

3.57 wifi_wowlan_ptn_cfg_t structure reference

3.57.1 Data fields

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

3.57.2 Detailed description

Wowlan Pattern config struct

3.57.3 Field documentation

3.57.3.1 t_u8 wifi_wowlan_ptn_cfg_t::enable

Enable user defined pattern

3.57.3.2 t_u8 wifi_wowlan_ptn_cfg_t::n_patterns

number of patterns

3.57.3.3 wifi_wowlan_pattern_t wifi_wowlan_ptn_cfg_t::patterns[MAX_NUM_FILTERS]

user define pattern

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

- wifi-decl.h

3.58 wlan_cipher structure reference

3.58.1 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

3.58.2 Detailed description

Wlan Cipher structure

3.58.3 Field documentation

3.58.3.1 uint16_t wlan_cipher::none

1 bit value can be set for none

3.58.3.2 uint16_t wlan_cipher::wep40

1 bit value can be set for wep40

3.58.3.3 uint16_t wlan_cipher::wep104

1 bit value can be set for wep104

3.58.3.4 uint16_t wlan_cipher::tkip

1 bit value can be set for tkip

3.58.3.5 uint16_t wlan_cipher::ccmp

1 bit valuecan be set for ccmp

3.58.3.6 uint16_t wlan_cipher::aes_128_cmac

1 bit valuecan be set for aes 128 cmac

3.58.3.7 uint16_t wlan_cipher::gcmp

1 bit value can be set for gcmp

3.58.3.8 uint16_t wlan_cipher::sms4

1 bit value can be set for sms4

3.58.3.9 uint16_t wlan_cipher::gcmp_256

1 bit value can be set for gcmp 256

3.58.3.10 uint16_t wlan_cipher::ccmp_256

1 bit valuecan be set for ccmp 256

3.58.3.11 uint16_t wlan_cipher::rsvd

1 bit is reserved

3.58.3.12 uint16_t wlan_cipher::bip_gmac_128

1 bit value can be set for bip gmac 128

3.58.3.13 uint16_t wlan_cipher::bip_gmac_256

1 bit value can be set for bip gmac 256

3.58.3.14 uint16_t wlan_cipher::bip_cmac_256

1 bit value can be set for bip cmac 256

3.58.3.15 uint16_t wlan_cipher::gtk_not_used

1 bit valuecan be set for gtk not used

3.58.3.16 uint16_t wlan_cipher::rsvd2

4 bits are reserved

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

- wlan.h

3.59 wlan_ip_config structure reference

3.59.1 Data fields

- struct ipv6_configipv6 [CONFIG_MAX_IPV6_ADDRESSES]
- struct ipv4_configipv4

3.59.2 Detailed description

Network IP configuration.

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

3.59.3 Field documentation

3.59.3.1 struct ipv6_config wlan_ip_config::ipv6[CONFIG_MAX_IPV6_ADDRESSES]

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

3.59.3.2 struct ipv4_config wlan_ip_config::ipv4

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

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

- wlan.h

3.60 wlan_network structure reference

3.60.1 Data fields

- 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
- short rssi_threshold
- enum wlan_bss_typetype
- enum wlan_bss_rolerole
- struct wlan_network_securitysecurity
- struct wlan_ip_configip
- char **identity** [IDENTITY_MAX_LENGTH]
- unsigned ssid_specific: 1
- unsigned bssid_specific: 1
- unsigned channel_specific: 1
- unsigned security_specific: 1
- unsigned dot11n: 1
- uint16_t beacon_period
- uint8_t dtim_period
- uint8_t wlan_capa
- bool neighbor_report_supported

3.60.2 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.

3.60.3 Field documentation

3.60.3.1 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.

3.60.3.2 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.

3.60.3.3 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.

3.60.3.4 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.

3.60.3.5 uint8_t wlan_network::sec_channel_offset

The secondary channel offset

3.60.3.6 uint16_t wlan_network::acs_band

The ACS band if set channel to 0.

3.60.3.7 int wlan_network::rssi

RSSI

3.60.3.8 short wlan_network::rssi_threshold

Rssi threshold

3.60.3.9 enum wlan_bss_type wlan_network::type

BSS type

3.60.3.10 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.

3.60.3.11 struct wlan_network_security wlan_network::security

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

3.60.3.12 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.

3.60.3.13 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.

3.60.3.14 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.

3.60.3.15 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.

3.60.3.16 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.

3.60.3.17 unsigned wlan_network::dot11n

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

3.60.3.18 uint16_t wlan_network::beacon_period

Beacon period of associated BSS

3.60.3.19 uint8_t wlan_network::dtim_period

DTIM period of associated BSS

3.60.3.20 uint8_t wlan_network::wlan_capa

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

3.60.3.21 bool wlan_network::neighbor_report_supported

Neighbor report support (For internal use only)

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

- wlan.h

3.61 wlan_network_security structure reference

3.61.1 Data fields

- enum wlan_security_typetype
- int key_mgmt
- struct wlan_ciphermcstCipher
- struct wlan_cipherucstCipher
- 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 pmk [WLAN_PMK_LENGTH]
- bool pmk_valid
- bool mfpc
- bool mfpr
- wm_mbedtls_cert_t tls_cert
- mbedtls_ssl_config * wlan_ctx
- mbedtls_ssl_context * wlan_ssl

3.61.2 Detailed description

Network security configuration

3.61.3 Field documentation

3.61.3.1 enum wlan_security_type wlan_network_security::type

Type of network security to use specified by enum wlan_security_type.

3.61.3.2 int wlan_network_security::key_mgmt

Key management type

3.61.3.3 struct wlan_cipher wlan_network_security::mcstCipher

Type of network security Group Cipher suite used internally

3.61.3.4 struct wlan_cipher wlan_network_security::ucstCipher

Type of network security Pairwise Cipher suite used internally

3.61.3.5 bool wlan_network_security::is_pmf_required

Is PMF required

3.61.3.6 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.

3.61.3.7 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.

3.61.3.8 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.

3.61.3.9 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.

3.61.3.10 char* wlan_network_security::sae_groups

SAE Groups

3.61.3.11 uint8_t wlan_network_security::pwe_derivation

PWE derivation

3.61.3.12 uint8_t wlan_network_security::transition_disable

transition disable

3.61.3.13 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.

3.61.3.14 bool wlan_network_security::pmk_valid

Flag reporting whether pmk is valid or not.

3.61.3.15 bool wlan_network_security::mfpc

Management Frame Protection Capable (MFPC)

3.61.3.16 bool wlan_network_security::mfpr

Management Frame Protection Required (MFPR)

3.61.3.17 wm_mbedtls_cert_t wlan_network_security::tls_cert

TLS client cert configuration

3.61.3.18 mbedtls_ssl_config* wlan_network_security::wlan_ctx

mbedtls_ssl_config handle

3.61.3.19 mbedtls_ssl_context* wlan_network_security::wlan_ssl

mbedtls_ssl_context handle

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

- wlan.h

3.62 wlan_scan_result structure reference

3.62.1 Data fields

- char ssid [33]
- unsigned int ssid_len
- char bssid [6]
- unsigned int channel
- enum wlan_bss_typetype
- enum wlan_bss_rolerole
- unsigned dot11n: 1
- unsigned wmm: 1
- unsigned wep: 1
- unsigned wpa: 1
- unsigned wpa2: 1
- unsigned wpa2_sha256: 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 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

3.62.2 Detailed description

Scan Result

3.62.3 Field documentation

3.62.3.1 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.

3.62.3.2 unsigned int wlan_scan_result::ssid_len

SSID length

3.62.3.3 char wlan_scan_result::bssid[6]

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

3.62.3.4 unsigned int wlan_scan_result::channel

The network channel.

3.62.3.5 enum wlan_bss_type wlan_scan_result::type

The network wireless type.

3.62.3.6 enum wlan_bss_role wlan_scan_result::role

The network wireless mode.

3.62.3.7 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.

3.62.3.8 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.

3.62.3.9 unsigned wlan_scan_result::wep

The network uses WEP security.

3.62.3.10 unsigned wlan_scan_result::wpa

The network uses WPA security.

3.62.3.11 unsigned wlan_scan_result::wpa2

The network uses WPA2 security

3.62.3.12 unsigned wlan_scan_result::wpa2_sha256

The network uses WPA2 SHA256 security

3.62.3.13 unsigned wlan_scan_result::wpa3_sae

The network uses WPA3 SAE security

3.62.3.14 unsigned wlan_scan_result::wpa2_entp

The network uses WPA2 Enterprise security

3.62.3.15 unsigned wlan_scan_result::wpa2_entp_sha256

The network uses WPA2 Enterprise SHA256 security

3.62.3.16 unsigned wlan_scan_result::wpa3_1x_sha256

The network uses WPA3 Enterprise SHA256 security

3.62.3.17 unsigned wlan_scan_result::wpa3_1x_sha384

The network uses WPA3 Enterprise SHA384 security

3.62.3.18 unsigned char wlan_scan_result::rssi

The signal strength of the beacon

3.62.3.19 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.

3.62.3.20 unsigned int wlan_scan_result::trans_ssid_len

SSID length

3.62.3.21 char wlan_scan_result::trans_bssid[6]

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

3.62.3.22 uint16_t wlan_scan_result::beacon_period

Beacon Period

3.62.3.23 uint8_t wlan_scan_result::dtim_period

DTIM Period

3.62.3.24 t_u8 wlan_scan_result::ap_mfpc

MFPC bit of AP

3.62.3.25 t_u8 wlan_scan_result::ap_mfpr

MFPR bit of AP

3.62.3.26 bool wlan_scan_result::neighbor_report_supported

Neigbort report support (For internal use only)

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

- wlan.h

3.63 wps_config structure reference

3.63.1 Data fields

- uint8_t role
- uint8_t pin_generator
- uint8_t version
- uint8_t version2
- uint8_t device_name [32]
- uint8_t manufacture [64]
- uint8_t model_name [32]
- uint8_t model_number [32]
- uint8_t serial_number [32]
- uint16_t config_methods
- uint16_t primary_dev_category
- uint16_t primary_dev_subcategory
- uint8_t rf_bands
- uint32_t os_version
- uint8_t wps_msg_max_retry
- uint32_t wps_msg_timeout
- uint16_t pin_len
- int(* wps_callback)(enum wps_event event, void *data, uint16_t len)
- uint8_t prov_session

3.63.2 Detailed description

This struct is passed to wps_start(). The user must initialize it with parameters as described inline.

3.63.3 Field documentation

3.63.3.1 uint8_t wps_config::role

Enrollee: 1, Registrar: 2, WiFi Direct mode:4

3.63.3.2 uint8_t wps_config::pin_generator

PIN Generator - Enrollee or Registrar

3.63.3.3 uint8_t wps_config::version

version

3.63.3.4 uint8_t wps_config::version2

version

3.63.3.5 uint8_t wps_config::device_name[32]

Device name

3.63.3.6 uint8_t wps_config::manufacture[64]

Manufacture

3.63.3.7 uint8_t wps_config::model_name[32]

Model name

3.63.3.8 uint8_t wps_config::model_number[32]

Model number

3.63.3.9 uint8_t wps_config::serial_number[32]

Serial number

3.63.3.10 uint16_t wps_config::config_methods

Config methods

3.63.3.11 uint16_t wps_config::primary_dev_category

Primary Device category

3.63.3.12 uint16_t wps_config::primary_dev_subcategory

Primary Device subcategory

3.63.3.13 uint8_t wps_config::rf_bands

RF bands

3.63.3.14 uint32_t wps_config::os_version

OS Version

3.63.3.15 uint8_t wps_config::wps_msg_max_retry

WPS message max retry

3.63.3.16 uint32_t wps_config::wps_msg_timeout

WPS message timeout

3.63.3.17 uint16_t wps_config::pin_len

PIN length

3.63.3.18 int(* wps_config::wps_callback) (enum wps_event event, void *data, uint16_t len)

WPS callback

3.63.3.19 uint8_t wps_config::prov_session

session attempt PROV_NON_SESSION_ATTEMPT/PROV_WPS_SESSION_ATTEMPT/PROV_ENTP_SESSION_ATTEMPT

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

- wifi_nxp_wps.h

4 File documentation

4.1 cli.h file reference

CLI module.

4.1.1 Detailed description

4.1.2 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.

4.1.2.1 Function documentation

4.1.2.1.1 int cli_register_command (const struct cli_command * command)

Register a CLI command

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

4.1.2.1.1.1 Parameters

in	*command*	The structure to register one CLI command

4.1.2.1.1.2 Returns

0 on success

1 on failure

4.1.2.1.2 int cli_unregister_command (const struct cli_command * command)

Unregister a CLI command

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

4.1.2.1.2.1 Parameters

in	*command*	The structure to unregister one CLI command

4.1.2.1.2.2 Returns

0 on success

1 on failure

4.1.2.1.3 int cli_init (void)

Initialize the CLI module

4.1.2.1.3.1 Returns

WM_SUCCESS on success

error code otherwise.

4.1.2.1.4 int cli_deinit (void)

DeInitialize the CLI module

4.1.2.1.4.1 Returns

WM_SUCCESS on success

error code otherwise.

4.1.2.1.5 int cli_stop (void)

Stop the CLI thread and carry out the cleanup

4.1.2.1.5.1 Returns

WM_SUCCESS on success

error code otherwise.

4.1.2.1.6 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.

4.1.2.1.6.1 Parameters

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

4.1.2.1.6.2 Returns

0 on success

1 on failure

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

Unregister a batch of CLI commands

4.1.2.1.7.1 Parameters

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

4.1.2.1.7.2 Returns

0 on success

1 on failure

4.1.2.1.8 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().

4.1.2.1.8.1 Parameters

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

4.1.2.1.8.2 Returns

WM_SUCCESS on success

error code otherwise.

4.1.2.1.9 int cli_submit_cmd_buffer (char ** buff)

Submit a command buffer to the CLI

Sends the command buffer to the CLI for processing.

4.1.2.1.9.1 Parameters

buff	Pointer to a char * buffer.

4.1.2.1.9.2 Returns

WM_SUCCESS on success

error code otherwise.

4.2 cli_utils.h file reference

CLI Utils.

4.2.1 Detailed description

SPDX-License-Identifier: BSD-3-Clause

4.3 dhcp-server.h file reference

DHCP server.

4.3.1 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.

4.3.2 Function documentation

4.3.2.1 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.

4.3.2.1.1 Returns

-WM_E_DHCPD_REGISTER_CMDS if cli init operation failed.

WM_SUCCESS if cli init operation success.

4.3.2.2 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.

4.3.2.2.1 Returns

-WM_E_DHCPD_REGISTER_CMDS if cli init operation failed.

WM_SUCCESS if cli init operation success.

4.3.2.3 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.

4.3.2.3.1 Parameters

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

4.3.2.3.2 Returns

WM_SUCCESS on success or error code

4.3.2.4 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.

4.3.2.4.1 Parameters

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

4.3.2.5 void dhcp_server_stop (void)

Stop DHCP server

4.3.2.6 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

4.3.2.6.1 Parameters

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

4.3.2.6.2 Returns

Error status code

4.3.2.7 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.

4.3.2.7.1 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

4.3.2.7.2 Returns

WM_SUCCESS on success or -WM_FAIL.

4.3.2.8 void dhcp_stat (void)

Print DHCP stats on the console

This API prints DHCP stats on the console

4.3.3 Enumeration type documentation

4.3.3.1 enum wm_dhcpd_errno

DHCPD Error Codes

4.3.3.1.1 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

4.4 iperf.h file reference

This file provides the support for network utility iperf.

4.4.1 Function documentation

4.4.1.1 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.

4.4.1.1.1 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)

4.4.1.2 int iperf_cli_deinit ()

Unregister Network Utility CLI command iperf.

4.4.1.2.1 Returns

WM_SUCCESS if the CLI commands are unregistered

-WM_FAIL otherwise

4.5 wifi-decl.h file reference

Wifi structure declarations.

4.5.1 Macro documentation

4.5.1.1 #define MLAN_MAX_VER_STR_LEN 128

Version string buffer length

4.5.1.2 #define BSS_TYPE_STA 0U

BSS type : STA

4.5.1.3 #define BSS_TYPE_UAP 1U

BSS type : UAP

4.5.1.4 #define MLAN_MAX_SSID_LENGTH (32U)

MLAN Maximum SSID Length

4.5.1.5 #define MLAN_MAX_PASS_LENGTH (64)

MLAN Maximum PASSPHRASE Length

4.5.2 Enumeration type documentation

4.5.2.1 enum wifi_SubBand_t

Wifi subband enum

4.5.2.1.1 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

4.5.2.2 enum wifi_frame_type_t

Wifi frame types

4.5.2.2.1 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

4.6 wifi.h file reference

This file contains interface to wifi driver.

4.6.1 Function documentation

4.6.1.1 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.

4.6.1.1.1 Parameters

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

4.6.1.1.2 Returns

WM_SUCCESS on success or -WM_FAIL on error.

4.6.1.2 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.

4.6.1.2.1 Parameters

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

4.6.1.2.2 Returns

WM_SUCCESS on success or -WM_FAIL on error.

4.6.1.3 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.

4.6.1.4 void wifi_set_tx_status (t_u8 status)

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

4.6.1.4.1 Parameters

in	*status*	Status to set for TX

4.6.1.5 void wifi_set_rx_status (t_u8 status)

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

4.6.1.5.1 Parameters

in	*status*	Status to set for RX

4.6.1.6 void reset_ie_index ()

This API can be used to reset mgmt_ie_index_bitmap.

4.6.1.7 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.

4.6.1.7.1 Parameters

in	*data_intput_callback*	Function that needs to be called

4.6.1.7.2 Returns

WM_SUCCESS

4.6.1.8 void wifi_deregister_data_input_callback (void)

Deregister Data callback function from Wi-Fi Driver

4.6.1.9 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.

4.6.1.9.1 Parameters

in	*amsdu_data_intput_ callback*	Function that needs to be called

4.6.1.9.2 Returns

WM_SUCESS

4.6.1.10 void wifi_deregister_amsdu_data_input_callback (void)

Deregister Data callback function from Wi-Fi Driver

4.6.1.11 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.

4.6.1.11.1 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.

4.6.1.11.2 Returns

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

4.6.1.12 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.

4.6.1.12.1 Parameters

in	*count*	No of retry attempts.

4.6.1.13 void wifi_sta_ampdu_tx_enable (void)

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

4.6.1.14 void wifi_sta_ampdu_tx_disable (void)

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

4.6.1.15 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.

4.6.1.15.1 Parameters

in	*tid*	tid value

4.6.1.16 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.

4.6.1.16.1 Parameters

in	*tid*	tid value

4.6.1.16.2 Returns

MTRUE or MFALSE

4.6.1.17 void wifi_sta_ampdu_rx_enable (void)

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

4.6.1.18 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.

4.6.1.18.1 Parameters

in	*tid*	tid value

4.6.1.19 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.

4.6.1.19.1 Parameters

in	*tid*	tid value

4.6.1.19.2 Returns

MTRUE or MFALSE

4.6.1.20 void wifi_uap_ampdu_rx_enable (void)

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

4.6.1.21 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.

4.6.1.21.1 Parameters

in	*tid*	tid value

4.6.1.22 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.

4.6.1.22.1 Parameters

in	*tid*	tid value

4.6.1.22.2 Returns

MTRUE or MFALSE

4.6.1.23 void wifi_uap_ampdu_rx_disable (void)

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

4.6.1.24 void wifi_uap_ampdu_tx_enable (void)

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

4.6.1.25 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.

4.6.1.25.1 Parameters

in	*tid*	tid value

4.6.1.26 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.

4.6.1.26.1 Parameters

in	*tid*	tid value

4.6.1.26.2 Returns

MTRUE or MFALSE

4.6.1.27 void wifi_uap_ampdu_tx_disable (void)

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

4.6.1.28 void wifi_sta_ampdu_rx_disable (void)

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

4.6.1.29 int wifi_get_device_mac_addr (wifi_mac_addr_t * mac_addr)

Get the device sta MAC address

4.6.1.29.1 Parameters

out	*mac_addr*	Mac address

4.6.1.29.2 Returns

WM_SUCESS

4.6.1.30 int wifi_get_device_uap_mac_addr (wifi_mac_addr_t * mac_addr_uap)

Get the device uap MAC address

4.6.1.30.1 Parameters

out	*mac_addr_uap*	Mac address

4.6.1.30.2 Returns

WM_SUCESS

4.6.1.31 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.

4.6.1.31.1 Parameters

in	*fw_ver_ext*	Firmware Version Extended

4.6.1.31.2 Returns

WM_SUCCESS

4.6.1.32 unsigned wifi_get_last_cmd_sent_ms (void)

Get the timestamp of the last command sent to the firmware

4.6.1.32.1 Returns

Timestamp in millisec of the last command sent

4.6.1.33 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.

4.6.1.34 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

4.6.1.34.1 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.

4.6.1.34.2 Returns

Standard SDK return codes

4.6.1.35 int wifi_unregister_event_queue (os_queue_t * event_queue)

Unregister an event queue from the wifi driver.

4.6.1.35.1 Parameters

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

4.6.1.35.2 Returns

Standard SDK return codes

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

Get scan list

4.6.1.36.1 Parameters

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

4.6.1.36.2 Returns

WM_SUCCESS on success or error code.

4.6.1.37 int wifi_get_scan_result_count (unsigned * count)

Get the count of elements in the scan list

4.6.1.37.1 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.

4.6.1.37.2 Returns

Standard SDK return codes.

4.6.1.38 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.

4.6.1.38.1 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.

4.6.1.38.2 Returns

void

4.6.1.39 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.

4.6.1.39.1 Parameters

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

4.6.1.40 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.

4.6.1.40.1 Parameters

in	*mac*	The new MAC Address

4.6.1.41 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.

4.6.1.41.1 Parameters

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

4.6.1.42 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

4.6.1.42.1 Parameters

in	*mac_addr*	multicast mapped MAC address

4.6.1.42.2 Returns

0 on Success or else Error

4.6.1.43 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

4.6.1.43.1 Parameters

in	*mac_addr*	multicast mapped MAC address

4.6.1.43.2 Returns

0 on Success or else Error

4.6.1.44 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).

4.6.1.44.1 Parameters

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

4.6.1.45 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

4.6.1.45.1 Parameters

out	*region_code*	Region Code

4.6.1.45.2 Returns

Standard WMSDK return codes.

4.6.1.46 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

4.6.1.46.1 Parameters

in	*region_code*	Region Code

4.6.1.46.2 Returns

Standard WMSDK return codes.

4.6.1.47 int wifi_set_country_code (const char * alpha2)

Set/Get country code

4.6.1.47.1 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

4.6.1.47.2 Returns

WM_SUCCESS if successful otherwise failure.

4.6.1.48 int wifi_get_uap_channel (int * channel)

Get the uAP channel number

4.6.1.48.1 Parameters

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

4.6.1.48.2 Returns

Standard WMSDK return code

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

Get/Set the uAP mfpc and mfpr

4.6.1.49.1 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

4.6.1.49.2 Returns

cmd response status

4.6.1.50 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.

4.6.1.50.1 Returns

WM_SUCCESS on success or error code.

4.6.1.51 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.

4.6.1.51.1 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.

4.6.1.51.2 Returns

WM_SUCCESS on success or error code.

4.6.1.52 t_u8 region_string_2_region_code (t_u8 * region_string)

4.6.1.52.1 Parameters

region_string	Region string

4.6.1.52.2 Returns

Region code

4.6.2 Macro documentation

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

BIT value

4.6.2.2 #define WIFI_MGMT_ACTION MBIT(13)

BITMAP for Action frame

4.6.3 Enumeration type documentation

4.6.3.1 anonymous enum

Wi-Fi Error Code

4.6.3.1.1 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.

4.6.3.2 anonymous enum

Wi-Fi driver TX/RX data status

4.6.3.2.1 Enumerator

WIFI_DATA_RUNNING	Data in running status
WIFI_DATA_BLOCK	Data in block status

4.7 wifi_cal_data_ext.h file reference

This file contains the cal data.

4.8 wifi_events.h file reference

Wi-Fi events.

4.8.1 Enumeration type documentation

4.8.1.1 enum wifi_event

Wi-Fi events

4.8.1.1.1 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
WIFI_EVENT_SURVEY_RESULT_GET	Survey Result Get
WIFI_EVENT_GET_HW_SPEC	Get hardware spec
WIFI_EVENT_ASSOCIATION	Association
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_AWAKE	Awake
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_HS_CONFIG	HS configuration
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_LAST	Event to indicate end of Wi-Fi events

4.8.1.2 enum wifi_event_reason

Wi-Fi Event Reason

4.8.1.2.1 Enumerator

WIFI_EVENT_REASON_SUCCESS	Success
WIFI_EVENT_REASON_TIMEOUT	Timeout
WIFI_EVENT_REASON_FAILURE	Failure

4.8.1.3 enum wlan_bss_type

Network wireless BSS Type

4.8.1.3.1 Enumerator

WLAN_BSS_TYPE_STA	Station
WLAN_BSS_TYPE_UAP	uAP
WLAN_BSS_TYPE_ANY	Any

4.8.1.4 enum wlan_bss_role

Network wireless BSS Role

4.8.1.4.1 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

4.8.1.5 enum wifi_wakeup_event_t

This enum defines various wakeup events for which wakeup will occur

4.8.1.5.1 Enumerator

WIFI_WAKE_ON_ALL_BROADCAST	Wakeup on broadcast
WIFI_WAKE_ON_UNICAST	Wakeup on unicast
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

4.9 wifi_nxp.h file reference

This file provides Core Wi-Fi definition for wpa supplicant RTOS driver.

4.9.1 Detailed description

SPDX-License-Identifier: BSD-3-Clause

4.10 wifi_nxp_wps.h file reference

WPS - WiFi Protected Setup.

4.10.1 Detailed description

Wi-Fi Protected Setup (WPS) is a standard for easy and secure wireless network set up and connections. Using this standard, wireless clients can associate with the WPS enabled access point by entering the same PIN at client and AP end or by pushing a pushbutton on both. This eliminates the need for manually entering the security configuration on the client side.

Two methods that are supported by SDK are:

- PIN Method: In this method the same PIN (Personal Identification Number) is entered on the wireless client and access point. The PIN can be static or dynamically generated on any of the AP or wireless client.
- PBC Method: in which the user simply has to push a button, either an actual or virtual one, on both the AP and wireless client.

4.10.2 Usage

In a typical implementation WPS thread is started by calling wps_start. A valid initialized wps_config structure needs to be passed to this function. This structure contains a callback handler which is invoked by the WPS thread on occurrence of various events. Once the WPS thread is started, commands can be sent to it using wps_connect function. Typically these commands can be start PIN session, or start a pushbutton session. This requires passing wps_session_command structure that contains a valid wlan_scan_result that contains network information for a WPS enabled wireless network. On successful WPS session, callback handler is invoked with WPS_SESSION_SUCCESSFUL event with wlan_network structure that contains all the security information for the network with which WPS session was attempted. After this wps_stop should be called to terminate the WPS thread.

Note:

WPS implementation lets application decide PIN policy. PIN could either be generated by the application, or by the access point or can be statically generated at device manufacturing time. wps_generate_pin and wps_validate_pin utility functions can be used by the application to generate and validate PIN.

WPS thread internally uses multiple dynamic allocations. Please do not reduce the heap size below 64KB if the application wishes to use WPS.

4.10.2.1 Function documentation

4.10.2.1.1 int wps_start (struct wps_config * wps_conf)

Starts WPS thread and enables commands delivery

4.10.2.1.1.1 Parameters

wps_conf	A pointer to WPS custom configuration

4.10.2.1.1.2 Returns

WM_SUCCESS if successful, -WM_FAIL otherwise

4.10.2.1.2 int wps_connect (enum wps_session_command pbc, uint32_t pin, struct wlan_scan_result * res)

Connect to WPS enabled AP

Connect to a WPS enabled AP. This function is typically called whenever the users pushes the wps button or enters wps pin.

4.10.2.1.2.1 Parameters

pbc	Set to 1 if Push-button session is desired
pin	Ignore if pbc is 1. If pbc is 0, this indicates the pin that should be used. The pin can be 4 or 8 digits.
res	The WPS enabled AP to connect to.

4.10.2.1.2.2 Returns

WM_SUCCESS if successful, -WM_FAIL otherwise

4.10.2.1.3 int wps_generate_pin (uint32_t * wps_pin)

Generate 8 digit WPS PIN value with random number generator

4.10.2.1.3.1 Parameters

wps_pin	Generated 8 digit WPS PIN value

4.10.2.1.3.2 Returns

WM_SUCCESS if successful, -WM_FAIL otherwise

4.10.2.1.4 int wps_validate_pin (uint32_t wps_pin)

Validate checksum of PIN

4.10.2.1.4.1 Parameters

wps_pin	WPS PIN value

4.10.2.1.4.2 Returns

WM_SUCCESS if successful, -WM_FAIL otherwise

4.10.2.1.5 int wps_stop ()

Deletes WPS thread and message queue

4.10.2.1.5.1 Returns

WM_SUCCESS if successful, -WM_FAIL otherwise

4.10.2.2 Macro documentation

4.10.2.2.1 #define MAC2STR(a) (a)[0], (a)[1], (a)[2], (a)[3], (a)[4], (a)[5]

MAC to string

4.10.2.2.2 #define MACSTR "%02x:%02x:%02x:%02x:%02x:%02x"

MAC string

4.10.2.3 Enumeration type documentation

4.10.2.3.1 enum wps_session_command

enum : WPS session commands

4.10.2.3.1.1 Enumerator

CMD_WPS_PI	Command to start WPS PIN session
CMD_WPS_PB	Command to start WPS PBC session

4.10.2.3.2 enum wps_event

enum : WPS events

4.10.2.3.2.1 Enumerator

WPS_STARTED	WPS thread started
WPS_SESSION_STARTED	WPS PBC/PIN Session started
WPS_SESSION_PIN_CHKSUM_FAILED	WPS PIN checksum failed
WPS_SESSION_ABORTED	WPS Session aborted
WPS_SESSION_TIMEOUT	WPS Session registration timeout
WPS_SESSION_SUCCESSFUL	WPS Session attempt successful
WPS_SESSION_FAILED	WPS Session failed
WPS_FINISHED	WPS thread stopped

4.10.2.3.3 enum wps_session_types

Enum that indicates type of WPS session, either a push button or a PIN based session is determined by value for this enum

4.10.2.3.3.1 Enumerator

WPS_SESSION_INACTIVE	WPS session is not active
WPS_SESSION_PBC	WPS Push Button session active
WPS_SESSION_PIN	WPS PIN session active

4.11 wifi_ping.h file reference

This file provides the support for network utility ping.

4.11.1 Function documentation

4.11.1.1 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.

4.11.1.1.1 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)

4.11.1.2 int ping_cli_deinit (void)

Unregister Network Utility CLI commands.

Unregister the Network Utility CLI commands.

4.11.1.2.1 Returns

WM_SUCCESS if the CLI commands are unregistered

-WM_FAIL otherwise

4.12 wlan.h file reference

WLAN Connection Manager.

4.12.1 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.

4.12.2 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.

4.12.2.1 Function documentation

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

Initialize the SDIO driver and create the wifi driver thread.

4.12.2.1.1.1 Parameters

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

4.12.2.1.1.2 Returns

WM_SUCCESS if the WLAN Connection Manager service has initialized successfully.

Negative value if initialization failed.

4.12.2.1.2 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.

4.12.2.1.2.1 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.

4.12.2.1.2.2 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.

4.12.2.1.3 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.

4.12.2.1.3.1 Returns

WM_SUCCESS if the WLAN Connection Manager service has been stopped successfully.

WLAN_ERROR_STATE if the WLAN Connection Manager was not running.

4.12.2.1.4 void wlan_deinit (int action)

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

4.12.2.1.4.1 Parameters

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

4.12.2.1.5 int wlan_set_get_rx_abort_cfg (struct wlan_rx_abort_cfg * cfg, t_u16 action)

Set/Get RX abort configure to/from Fw.

4.12.2.1.5.1 Parameters

in,out	*cfg*	A pointer to information buffer
in	*action*	Command action: GET or SET

4.12.2.1.5.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.6 int wlan_set_rx_abort_cfg_ext (const struct wlan_rx_abort_cfg_ext * cfg)

Set Dynamic RX abort config to Fw.

4.12.2.1.6.1 Parameters

in	*cfg*	A pointer to information buffer

4.12.2.1.6.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.7 int wlan_get_rx_abort_cfg_ext (struct wlan_rx_abort_cfg_ext * cfg)

Get Dynamic RX abort config from Fw.

4.12.2.1.7.1 Parameters

in,out	*cfg*	A pointer to information buffer

4.12.2.1.7.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.8 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'.

4.12.2.1.8.1 Parameters

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

4.12.2.1.9 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.

4.12.2.1.9.1 Parameters

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

4.12.2.1.10 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.

4.12.2.1.10.1 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.

4.12.2.1.10.2 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.

4.12.2.1.11 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.

4.12.2.1.11.1 Parameters

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

4.12.2.1.11.2 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.

4.12.2.1.12 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.

4.12.2.1.12.1 Parameters

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

4.12.2.1.12.2 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.

4.12.2.1.13 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.

4.12.2.1.13.1 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.

4.12.2.1.13.2 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.

4.12.2.1.14 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.

4.12.2.1.14.1 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.

4.12.2.1.15 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.

4.12.2.1.15.1 Returns

WM_SUCCESS if successful

WLAN_ERROR_STATE otherwise

4.12.2.1.16 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.

4.12.2.1.16.1 Parameters

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

4.12.2.1.16.2 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.

4.12.2.1.17 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.

4.12.2.1.17.1 Parameters

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

4.12.2.1.17.2 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.

4.12.2.1.18 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.

4.12.2.1.18.1 Parameters

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

4.12.2.1.18.2 Returns

WM_SUCCESS if the MAC address was copied.

-WM_E_INVAL if *sta_mac* or uap_mac is NULL.

4.12.2.1.19 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* .

4.12.2.1.19.1 Parameters

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

4.12.2.1.19.2 Returns

WM_SUCCESS if the MAC address was copied.

-WM_E_INVAL if *dest* is NULL.

4.12.2.1.20 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.

4.12.2.1.20.1 Parameters

out	*addr*	A pointer to the wlan_ip_config.

4.12.2.1.20.2 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.

4.12.2.1.21 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.

4.12.2.1.21.1 Parameters

out	*addr*	A pointer to the wlan_ip_config.

4.12.2.1.21.2 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.

4.12.2.1.22 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.

4.12.2.1.22.1 Parameters

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

4.12.2.1.22.2 Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *channel* is NULL.

-WM_FAIL if an internal error has occurred.

4.12.2.1.23 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.

4.12.2.1.23.1 Parameters

out	*network*	A pointer to the wlan_network.

4.12.2.1.23.2 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.

4.12.2.1.24 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.

4.12.2.1.24.1 Parameters

out	*network*	A pointer to the wlan_network.

4.12.2.1.24.2 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.

4.12.2.1.25 bool is_uap_started (void)

Retrieve the status information of the micro-AP interface.

4.12.2.1.25.1 Returns

TRUE if micro-AP interface is in WLAN_UAP_STARTED state.

FALSE otherwise.

4.12.2.1.26 bool is_sta_connected (void)

Retrieve the status information of the station interface.

4.12.2.1.26.1 Returns

TRUE if station interface is in WLAN_CONNECTED state.

FALSE otherwise.

4.12.2.1.27 bool is_sta_ipv4_connected (void)

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

4.12.2.1.27.1 Returns

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

FALSE otherwise.

4.12.2.1.28 bool is_sta_ipv6_connected (void)

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

4.12.2.1.28.1 Returns

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

FALSE otherwise.

4.12.2.1.29 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.

4.12.2.1.29.1 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.

4.12.2.1.29.2 Returns

WM_SUCCESS if successful.

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

4.12.2.1.30 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.

4.12.2.1.30.1 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.

4.12.2.1.30.2 Returns

WM_SUCCESS if successful.

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

4.12.2.1.31 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.

4.12.2.1.31.1 Parameters

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

4.12.2.1.31.2 Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *count* is NULL.

4.12.2.1.32 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.

4.12.2.1.32.1 Parameters

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

4.12.2.1.32.2 Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *state* is NULL

WLAN_ERROR_STATE if the WLAN Connection Manager was not running.

4.12.2.1.33 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.

4.12.2.1.33.1 Parameters

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

4.12.2.1.33.2 Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *state* is NULL

WLAN_ERROR_STATE if the WLAN Connection Manager was not running.

4.12.2.1.34 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.

4.12.2.1.34.1 Parameters

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

4.12.2.1.34.2 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.

4.12.2.1.35 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.

4.12.2.1.35.1 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.

4.12.2.1.35.2 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.

4.12.2.1.36 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.

4.12.2.1.36.1 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.

4.12.2.1.36.2 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).

4.12.2.1.37 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.

4.12.2.1.37.1 Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

4.12.2.1.38 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.

4.12.2.1.38.1 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
```

4.12.2.1.38.2 Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

4.12.2.1.39 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.

4.12.2.1.39.1 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
```

4.12.2.1.39.2 Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

4.12.2.1.40 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.

4.12.2.1.40.1 Parameters

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

4.12.2.1.40.2 Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

4.12.2.1.41 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.

4.12.2.1.41.1 Parameters

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

4.12.2.1.41.2 Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

4.12.2.1.42 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.

4.12.2.1.42.1 Parameters

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

4.12.2.1.43 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.

4.12.2.1.43.1 Parameters

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

4.12.2.1.44 void wlan_set_txrx_histogram (struct wlan_txrx_histogram_info * txrx_histogram, t_u8 * data)

Set Tx Rx histogram config. This function may be called to set Tx Rx histogram config.

4.12.2.1.44.1 Parameters

in	*txrx_histogram*	User configured parameters of Tx Rx histogram including enable and action.
out	*data*	Tx Rx histogram data from fw.

4.12.2.1.45 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.

4.12.2.1.45.1 Parameters

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

4.12.2.1.45.2 Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

4.12.2.1.46 int wlan_wowlan_config (uint8_t is_mef, t_u32 wake_up_conds)

Wowlan configure. This function may be called to config host sleep in firmware.

4.12.2.1.46.1 Parameters

in	*is_mef*	Flag to indicate use MEF condition or not.
in	*wake_up_conds*	Bit map of default condition.

4.12.2.1.46.2 Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

4.12.2.1.47 void wlan_config_host_sleep (bool is_manual, t_u8 is_periodic)

Host sleep configure. This function may be called to config host sleep in firmware.

4.12.2.1.47.1 Parameters

in	*is_manual*	Flag to indicate host enter low power mode with power manager or by command.
in	*is_periodic*	Flag to indicate host enter low power periodically or once with power manager.

4.12.2.1.48 void wlan_cancel_host_sleep ()

Cancel host sleep. This function may be called to cancel host sleep in firmware.

4.12.2.1.49 void wlan_clear_host_sleep_config ()

Clear host sleep configurations in driver. This function clears all the host sleep related configures in driver.

4.12.2.1.50 int wlan_set_multicast (t_u8 mef_action)

This function set multicast MEF entry

4.12.2.1.50.1 Parameters

in	*mef_action*	To be 0–discard and not wake host, 1–discard and wake host 3–allow and wake host.

4.12.2.1.51 int wlan_set_ieeeps_cfg (struct wlan_ieeeps_config * ps_cfg)

Set configuration parameters of IEEE power save mode.

4.12.2.1.51.1 Parameters

in	*ps_cfg*	: powersave configuratiuon includes multiple parameters.

4.12.2.1.51.2 Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

4.12.2.1.52 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.

4.12.2.1.52.1 Parameters

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

4.12.2.1.53 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

4.12.2.1.53.1 Parameters

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

4.12.2.1.54 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).

4.12.2.1.54.1 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).

4.12.2.1.54.2 Returns

WM_SUCCESS if successful.

WLAN_ERROR_STATE if unsuccessful.

4.12.2.1.55 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).

4.12.2.1.55.1 Parameters

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

4.12.2.1.55.2 Returns

WM_SUCCESS if successful.

WLAN_ERROR_STATE if unsuccessful.

4.12.2.1.56 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.

4.12.2.1.56.1 Returns

wifi firmware version extension string pointer stored in WLCMGR

4.12.2.1.57 void wlan_version_extended (void)

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

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

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

4.12.2.1.58.1 Parameters

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

4.12.2.1.58.2 Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

4.12.2.1.59 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.

4.12.2.1.59.1 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

4.12.2.1.59.2 Returns

WM_SUCCESS if the call was successful.

-WM_FAIL otherwise.

4.12.2.1.60 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.

4.12.2.1.60.1 Returns

WM_SUCCESS if the call was successful.

-WM_FAIL otherwise.

4.12.2.1.61 int wlan_wnmps_on (unsigned int wakeup_conditions, t_u16 wnm_sleep_time)

Enable WNM 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.

4.12.2.1.61.1 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
in	*wnm_sleep_time*	wnm sleep interval.(number of dtims)

4.12.2.1.61.2 Returns

WM_SUCCESS if the call was successful.

-WM_FAIL otherwise.

4.12.2.1.62 int wlan_wnmps_off (void)

Turn off WNM Power Save mode.

Note:

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

4.12.2.1.62.1 Returns

WM_SUCCESS if the call was successful.

-WM_FAIL otherwise.

4.12.2.1.63 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.

4.12.2.1.63.1 Returns

WM_SUCCESS if the call was successful.

-WM_FAIL otherwise.

4.12.2.1.64 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.

4.12.2.1.64.1 Returns

WM_SUCCESS if the call was successful.

-WM_FAIL otherwise.

4.12.2.1.65 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.

4.12.2.1.65.1 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

4.12.2.1.65.2 Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

4.12.2.1.66 uint16_t wlan_get_beacon_period (void)

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

4.12.2.1.66.1 Returns

beacon_period if operation is successful.

0 if command fails.

4.12.2.1.67 uint8_t wlan_get_dtim_period (void)

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

4.12.2.1.67.1 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.

4.12.2.1.68 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.

4.12.2.1.68.1 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.

4.12.2.1.68.2 Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

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

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

4.12.2.1.69.1 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

4.12.2.1.69.2 Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

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

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

4.12.2.1.70.1 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

4.12.2.1.70.2 Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

4.12.2.1.71 int wlan_set_packet_filters (wlan_flt_cfg_t * flt_cfg)

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

4.12.2.1.71.1 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
		}
		– – – – *Continues on next page* – – – –

in
(continues)
	fit_cfg
	(continues)
		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;
		flt_cfg.mef_entry.filter_item[0].type = TYPE_DNUM_EQ;
		flt_cfg.mef_entry.filter_item[0].pattern = 80;
		flt_cfg.mef_entry.filter_item[0].offset = 44;
		flt_cfg.mef_entry.filter_item[0].num_bytes = 2;
		flt_cfg.mef_entry.filter_item[1].type = TYPE_BYTE_EQ;
		flt_cfg.mef_entry.filter_item[1].repeat = 1;
		flt_cfg.mef_entry.filter_item[1].offset = 34;
		flt_cfg.mef_entry.filter_item[1].num_byte_seq = 4;
		memcpy(flt_cfg.mef_entry.filter_item[1].byte_seq, byte_seq1, 4);
		flt_cfg.mef_entry.rpn[1] = RPN_TYPE_AND;
		flt_cfg.mef_entry.filter_item[2].type = TYPE_BYTE_EQ;
		flt_cfg.mef_entry.filter_item[2].repeat = 16;
		flt_cfg.mef_entry.filter_item[2].offset = 14;
		flt_cfg.mef_entry.filter_item[2].num_byte_seq = 6;
		memcpy(flt_cfg.mef_entry.filter_item[2].byte_seq, byte_seq2, 6);
		flt_cfg.mef_entry.rpn[2] = RPN_TYPE_OR;

4.12.2.1.71.2 Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

4.12.2.1.72 int wlan_set_auto_arp (void)

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

4.12.2.1.72.1 Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

4.12.2.1.73 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

4.12.2.1.73.1 Parameters

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

4.12.2.1.73.2 Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails

4.12.2.1.74 int wlan_set_ipv6_ns_offload ()

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

4.12.2.1.74.1 Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

4.12.2.1.75 int wlan_send_host_sleep (uint32_t wakeup_condition)

Use this API to configure host sleep params in Wi-Fi firmware.

4.12.2.1.75.1 Parameters

in	*wakeup_condition*	bit 0: WAKE_ON_ALL_BROADCAST bit 1: WAKE_ON_UNICAST bit 2: WAKE_ON_MAC_EVENT bit 3: WAKE_ON_MULTICAST bit 4: WAKE_ON_ARP_BROADCAST bit 6: WAKE_ON_MGMT_FRAME All bit 0 discard and not wakeup host

4.12.2.1.75.2 Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

4.12.2.1.76 int wlan_get_current_bssid (uint8_t * bssid)

Use this API to get the BSSID of associated BSS.

4.12.2.1.76.1 Parameters

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

4.12.2.1.76.2 Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

4.12.2.1.77 uint8_t wlan_get_current_channel (void)

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

4.12.2.1.77.1 Returns

channel number if operation is successful.

0 if command fails.

4.12.2.1.78 int wlan_get_log (wlan_pkt_stats_t * stats)

Use this API to get the various statistics of sta from Wi-Fi firmware like number of beacons received, missed and so on.

4.12.2.1.78.1 Parameters

in	*stats*	A pointer to structure where stats collected from Wi-Fi firmware will be copied.

Note:

Please explore the elements of the wlan_pkt_stats_t strucutre for more information on stats.

4.12.2.1.78.2 Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

4.12.2.1.79 int wlan_uap_get_log (wlan_pkt_stats_t * stats)

Use this API to get the various statistics of uap from Wi-Fi firmware like number of beacons received, missed and so on.

4.12.2.1.79.1 Parameters

in	*stats*	A pointer to structure where stats collected from Wi-Fi firmware will be copied.

Note:

Please explore the elements of the wlan_pkt_stats_t strucutre for more information on stats.

4.12.2.1.79.2 Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

4.12.2.1.80 int wlan_get_ps_mode (enum wlan_ps_mode * ps_mode)

Get station interface power save mode.

4.12.2.1.80.1 Parameters

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

4.12.2.1.80.2 Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *ps_mode* was NULL.

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

Send message to WLAN Connection Manager thread.

4.12.2.1.81.1 Parameters

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

4.12.2.1.81.2 Returns

WM_SUCCESS if successful.

-WM_FAIL if failed.

4.12.2.1.82 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.

4.12.2.1.82.1 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).

4.12.2.1.83 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.

4.12.2.1.83.1 Returns

WLAN_ERROR_NONE if the CLI commands were unregistered or

WLAN_ERROR_ACTION if they were not unregistered

4.12.2.1.84 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.

4.12.2.1.84.1 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).

4.12.2.1.85 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.

4.12.2.1.85.1 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).

4.12.2.1.86 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.

4.12.2.1.86.1 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).

4.12.2.1.87 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.

4.12.2.1.87.1 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).

4.12.2.1.88 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.

4.12.2.1.88.1 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).

4.12.2.1.89 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.

4.12.2.1.89.1 Returns

WM_SUCCESS if the CLI commands were unregistered or

-WM_FAIL if they were not unregistered.

4.12.2.1.90 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.

4.12.2.1.90.1 Returns

Maximum number of WLAN firmware supported stations.

Note:

Get operation is allowed in any uAP state.

4.12.2.1.91 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.

4.12.2.1.91.1 Parameters

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

4.12.2.1.91.2 Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

Note:

Get operation is allowed in any uAP state.

4.12.2.1.92 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.

4.12.2.1.92.1 Parameters

in	*max_sta_num*	Number of maximum stations for uAP.

4.12.2.1.92.2 Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

Note:

Set operation in not allowed in WLAN_UAP_STARTED state.

4.12.2.1.93 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)

4.12.2.1.93.1 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.

4.12.2.1.93.2 Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

4.12.2.1.94 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)

4.12.2.1.94.1 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.

4.12.2.1.94.2 Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

4.12.2.1.95 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.

4.12.2.1.95.1 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 – – – – *Continues on next page* – – – –

in (continues)	*ds_rate* (continues)	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
in	*bss_type*	0: STA, 1: uAP

4.12.2.1.95.2 Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

4.12.2.1.96 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.

4.12.2.1.96.1 Parameters

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

4.12.2.1.96.2 Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

4.12.2.1.97 int wlan_get_sta_tx_power (t_u32 * power_level)

Get Station interface transmit power

4.12.2.1.97.1 Parameters

out	*power_level*	Transmit power level.

4.12.2.1.97.2 Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

4.12.2.1.98 int wlan_set_sta_tx_power (t_u32 power_level)

Set Station interface transmit power

4.12.2.1.98.1 Parameters

in	*power_level*	Transmit power level.

4.12.2.1.98.2 Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

4.12.2.1.99 int wlan_set_wwsm_txpwrlimit (void)

Set World Wide Safe Mode Tx Power Limits

4.12.2.1.99.1 Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

4.12.2.1.100 const char* wlan_get_wlan_region_code (void)

Get wlan region code from tx power config

4.12.2.1.100.1 Returns

wlan region code in string format.

4.12.2.1.101 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.

4.12.2.1.101.1 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.

4.12.2.1.101.2 Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

4.12.2.1.102 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.

4.12.2.1.102.1 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.

4.12.2.1.102.2 Returns

IE index if successful.

-WM_FAIL if unsuccessful.

4.12.2.1.103 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.

4.12.2.1.103.1 Parameters

in	*bss_type*	0: STA, 1: uAP
in	*index*	IE index.
in	*mgmt_bitmap_index*	mgmt bitmap index.

4.12.2.1.103.2 Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

4.12.2.1.104 bool wlan_get_11d_enable_status (void)

Get current status of 11d support.

4.12.2.1.104.1 Returns

true if 11d support is enabled by application.

false if not enabled.

4.12.2.1.105 int wlan_get_current_signal_strength (short * rssi, int * snr)

Get current RSSI and Signal to Noise ratio from WLAN firmware.

4.12.2.1.105.1 Parameters

in	*rssi*	A pointer to variable to store current RSSI
in	*snr*	A pointer to variable to store current SNR.

4.12.2.1.105.2 Returns

WM_SUCCESS if successful.

4.12.2.1.106 int wlan_get_average_signal_strength (short * rssi, int * snr)

Get average RSSI and Signal to Noise ratio from WLAN firmware.

4.12.2.1.106.1 Parameters

in	*rssi*	A pointer to variable to store current RSSI
in	*snr*	A pointer to variable to store current SNR.

4.12.2.1.106.2 Returns

WM_SUCCESS if successful.

4.12.2.1.107 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.

4.12.2.1.107.1 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.

4.12.2.1.107.2 Returns

WM_SUCCESS on success or error code.

4.12.2.1.108 int wlan_get_otp_user_data (uint8_t * buf, uint16_t len)

Get User Data from OTP Memory

4.12.2.1.108.1 Parameters

in	*buf*	Pointer to buffer where data will be stored

in	*len*	Number of bytes to read

4.12.2.1.108.2 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.

4.12.2.1.109 int wlan_get_cal_data (wlan_cal_data_t * cal_data)

Get calibration data from WLAN firmware

4.12.2.1.109.1 Parameters

out	*cal_data*	Pointer to calibration data structure where calibration data and it's length will be stored.

4.12.2.1.109.2 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.

4.12.2.1.110 int wlan_set_region_power_cfg (const t_u8 * data, t_u16 len)

Set the compressed Tx PWR Limit configuration.

4.12.2.1.110.1 Parameters

in	*data*	A pointer to TX PWR Limit configuration.
in	*len*	Length of TX PWR Limit configuration.

4.12.2.1.110.2 Returns

WM_SUCCESS on success, error otherwise.

4.12.2.1.111 int wlan_set_chanlist_and_txpwrlimit (wlan_chanlist_t * chanlist, wlan_txpwrlimit_t * txpwrlimit)

Set the Channel List and TRPC channel configuration.

4.12.2.1.111.1 Parameters

in	*chanlist*	A poiner to wlan_chanlist_t Channel List configuration.
in	*txpwrlimit*	A pointer to wlan_txpwrlimit_t TX PWR Limit configuration.

4.12.2.1.111.2 Returns

WM_SUCCESS on success, error otherwise.

4.12.2.1.112 int wlan_set_chanlist (wlan_chanlist_t * chanlist)

Set the Channel List configuration.

4.12.2.1.112.1 Parameters

in	*chanlist*	A pointer to wlan_chanlist_t Channel List configuration.

4.12.2.1.112.2 Returns

WM_SUCCESS on success, error otherwise.

Note:

If Region Enforcement Flag is enabled in the OTP then this API will not take effect.

4.12.2.1.113 int wlan_get_chanlist (wlan_chanlist_t * chanlist)

Get the Channel List configuration.

4.12.2.1.113.1 Parameters

out	*chanlist*	A pointer to wlan_chanlist_t Channel List configuration.

4.12.2.1.113.2 Returns

WM_SUCCESS on success, error otherwise.

Note:

The wlan_chanlist_t struct allocates memory for a maximum of 54 channels.

4.12.2.1.114 int wlan_set_txpwrlimit (wlan_txpwrlimit_t * txpwrlimit)

Set the TRPC channel configuration.

4.12.2.1.114.1 Parameters

in	*txpwrlimit*	A pointer to wlan_txpwrlimit_t TX PWR Limit configuration.

4.12.2.1.114.2 Returns

WM_SUCCESS on success, error otherwise.

4.12.2.1.115 int wlan_get_txpwrlimit (wifi_SubBand_t subband, wifi_txpwrlimit_t * txpwrlimit)

Get the TRPC channel configuration.

4.12.2.1.115.1 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.

4.12.2.1.115.2 Returns

WM_SUCCESS on success, error otherwise.

Note:

application can use print_txpwrlimit API to print the content of the txpwrlimit structure.

4.12.2.1.116 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.

4.12.2.1.116.1 Parameters

in	*reassoc_control*	Reassociation enable/disable

4.12.2.1.117 void wlan_uap_set_beacon_period (const uint16_t beacon_period)

API to set the beacon period of uAP

4.12.2.1.117.1 Parameters

in	*beacon_period*	Beacon period in TU (1 TU = 1024 micro seconds)

Note:

Please call this API before calling uAP start API.

4.12.2.1.118 int wlan_uap_set_bandwidth (const uint8_t bandwidth)

API to set the bandwidth of uAP

4.12.2.1.118.1 Parameters

in	*bandwidth*	Wi-Fi AP Bandwidth (20MHz/40MHz) 1: 20 MHz 2: 40 MHz

4.12.2.1.118.2 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.

4.12.2.1.119 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.

4.12.2.1.119.1 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

4.12.2.1.119.2 Returns

WM_SUCCESS if successful otherwise failure.

-WM_FAIL if command fails.

Note:

Please call this API before calling uAP start API.

4.12.2.1.120 void wlan_uap_ctrl_deauth (const bool enable)

API to control the deauth during uAP channel switch

4.12.2.1.120.1 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.

4.12.2.1.121 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.

4.12.2.1.122 void wlan_uap_set_htcapinfo (const uint16_t ht_cap_info)

API to set the HT Capability Information of uAP

4.12.2.1.122.1 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.

4.12.2.1.123 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.

4.12.2.1.123.1 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.

4.12.2.1.124 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.

4.12.2.1.125 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.

4.12.2.1.126 void wlan_sta_ampdu_rx_enable (void)

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

4.12.2.1.127 void wlan_sta_ampdu_rx_disable (void)

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

4.12.2.1.128 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.

4.12.2.1.129 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.

4.12.2.1.130 void wlan_uap_ampdu_rx_enable (void)

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

4.12.2.1.131 void wlan_uap_ampdu_rx_disable (void)

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

4.12.2.1.132 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.

4.12.2.1.132.1 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.

4.12.2.1.133 void wlan_enable_wpa2_enterprise_ap_only ()

Use this API if application want to allow station connection to WPA2 Enterprise ap profiles only.

If called the in scan result only the WPA2 Enterprise AP will be listed and station network profile only with WPA2 Enterprise security will be allowed to add to network profile list.

4.12.2.1.134 int wlan_set_rts (int rts)

Set the rts threshold of sta in WLAN firmware.

4.12.2.1.134.1 Parameters

in	*rts*	the value of rts threshold configuration.

4.12.2.1.134.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.135 int wlan_set_uap_rts (int rts)

Set the rts threshold of uap in WLAN firmware.

4.12.2.1.135.1 Parameters

in	*rts*	the value of rts threshold configuration.

4.12.2.1.135.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.136 int wlan_set_frag (int frag)

Set the fragment threshold of sta in WLAN firmware.

4.12.2.1.136.1 Parameters

in	*frag*	the value of fragment threshold configuration.

4.12.2.1.136.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.137 int wlan_set_uap_frag (int frag)

Set the fragment threshold of uap in WLAN firmware.

4.12.2.1.137.1 Parameters

in	*frag*	the value of fragment threshold configuration.

4.12.2.1.137.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.138 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.

4.12.2.1.138.1 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

4.12.2.1.138.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.139 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.

4.12.2.1.139.1 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.

4.12.2.1.140 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.

4.12.2.1.140.1 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.

4.12.2.1.140.2 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.

4.12.2.1.141 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.

4.12.2.1.141.1 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.

4.12.2.1.141.2 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.

4.12.2.1.142 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.

4.12.2.1.142.1 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.

4.12.2.1.142.2 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.

4.12.2.1.143 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.

4.12.2.1.143.1 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.

4.12.2.1.143.2 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.

4.12.2.1.144 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.

4.12.2.1.144.1 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.

4.12.2.1.144.2 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.

4.12.2.1.145 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.

4.12.2.1.145.1 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.

4.12.2.1.145.2 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.

4.12.2.1.146 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.

4.12.2.1.146.1 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.

4.12.2.1.146.2 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.

4.12.2.1.147 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.

4.12.2.1.147.1 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.

4.12.2.1.147.2 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.

4.12.2.1.148 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.

4.12.2.1.148.1 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.

4.12.2.1.148.2 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.

4.12.2.1.149 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.

4.12.2.1.149.1 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.

4.12.2.1.149.2 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.

4.12.2.1.150 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.

4.12.2.1.150.1 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.

4.12.2.1.150.2 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.

4.12.2.1.151 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.

4.12.2.1.151.1 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.

4.12.2.1.151.2 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.

4.12.2.1.152 int wlan_host_11k_cfg (int enable_11k)

enable/disable host 11k feature

4.12.2.1.152.1 Parameters

in	*enable_11k*	the value of 11k configuration.

4.12.2.1.152.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.153 int wlan_host_11k_neighbor_req (t_u8 * ssid)

host send neighbor report request

4.12.2.1.153.1 Parameters

in	*ssid*	the SSID for neighbor report

Note:

ssid parameter is optional

4.12.2.1.153.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.154 int wlan_tx_ampdu_prot_mode (tx_ampdu_prot_mode_para * prot_mode, t_u16 action)

Set/Get Tx ampdu prot mode.

4.12.2.1.154.1 Parameters

in,out	*prot_mode*	Tx ampdu prot mode
in	*action*	Command action

4.12.2.1.154.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.155 int wlan_mef_set_auto_arp (t_u8 mef_action)

This function set auto ARP configuration.

4.12.2.1.155.1 Parameters

in	*mef_action*	To be 0–discard and not wake host, 1–discard and wake host 3–allow and wake host.

4.12.2.1.155.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.156 int wlan_mef_set_auto_ping (t_u8 mef_action)

This function set auto ping configuration.

4.12.2.1.156.1 Parameters

in	*mef_action*	To be 0–discard and not wake host, 1–discard and wake host 3–allow and wake host.

4.12.2.1.156.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.157 int wlan_config_mef (int type, t_u8 mef_action)

This function set/delete mef entries configuration.

4.12.2.1.157.1 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.

4.12.2.1.157.2 Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

4.12.2.1.158 int wlan_set_ipv6_ns_mef (t_u8 mef_action)

Use this API to enable IPv6 Neighbor Solicitation offload in Wi-Fi firmware

4.12.2.1.158.1 Parameters

in	*mef_action*	0–discard and not wake host, 1–discard and wake host 3–allow and wake host.

4.12.2.1.158.2 Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

4.12.2.1.159 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

4.12.2.1.159.1 Parameters

in	*threshold*	Threshold rssi value to be set

4.12.2.1.160 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.

4.12.2.1.160.1 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

4.12.2.1.160.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.161 t_u32 wlan_get_entp_cert_files (int cert_type, t_u8 ** data)

This function get enterprise certificate data from "wlan" global structure *

4.12.2.1.161.1 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

4.12.2.1.161.2 Returns

size of raw data

4.12.2.1.162 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.

4.12.2.1.163 int wlan_net_monitor_cfg (wlan_net_monitor_t * monitor)

Send the net monitor config parameter to FW.

4.12.2.1.163.1 Parameters

in	*monitor*	Monitor config parameter

4.12.2.1.163.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.164 void wlan_register_monitor_user_callback (int(*)(void *buffer, t_u16 data_len) monitor_data_recv_callback)

This function registers callback which are used to deliver monitor data to user.

4.12.2.1.164.1 Parameters

in	*monitor_data_recv_ callback*	Callback to deliver monitor data and data length to user. Memory layout of buffer: offset(byte) items 0 rssi 1 802.11 mac header 1 + 'size of 802.11 mac header' frame body

4.12.2.1.164.2 Returns

void

4.12.2.1.165 void wlan_deregister_net_monitor_user_callback ()

This function deregisters monitor callback.

4.12.2.1.165.1 Returns

void

4.12.2.1.166 uint8_t wlan_check_11n_capa (unsigned int channel)

Check if 11n(2G or 5G) is supported by hardware or not.

4.12.2.1.166.1 Parameters

in	*channel*	Channel number.

4.12.2.1.166.2 Returns

true if 11n is supported or false if not.

4.12.2.1.167 uint8_t wlan_check_11ac_capa (unsigned int channel)

Check if 11ac(2G or 5G) is supported by hardware or not.

4.12.2.1.167.1 Parameters

in	*channel*	Channel number.

4.12.2.1.167.2 Returns

true if 11ac is supported or false if not.

4.12.2.1.168 uint8_t wlan_check_11ax_capa (unsigned int channel)

Check if 11ax(2G or 5G) is supported by hardware or not.

4.12.2.1.168.1 Parameters

in	*channel*	Channel number.

4.12.2.1.168.2 Returns

true if 11ax is supported or false if not.

4.12.2.1.169 int wlan_get_signal_info (wlan_rssi_info_t * signal)

Get rssi information.

4.12.2.1.169.1 Parameters

out	*signal*	rssi infomation get report buffer

4.12.2.1.169.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.170 int wlan_set_rg_power_cfg (t_u16 region_code)

set region power table

4.12.2.1.170.1 Parameters

in	*region_code*	region code

4.12.2.1.170.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.171 int wlan_get_turbo_mode (t_u8 * mode)

Get Turbo mode.

4.12.2.1.171.1 Parameters

out	*mode*	turbo mode 0: disable turbo mode 1: turbo mode 1 2: turbo mode 2 3: turbo mode 3

4.12.2.1.171.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.172 int wlan_get_uap_turbo_mode (t_u8 * mode)

Get UAP Turbo mode.

4.12.2.1.172.1 Parameters

out	*mode*	turbo mode 0: disable turbo mode 1: turbo mode 1 2: turbo mode 2 3: turbo mode 3

4.12.2.1.172.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.173 int wlan_set_turbo_mode (t_u8 mode)

Set Turbo mode.

4.12.2.1.173.1 Parameters

in	*mode*	turbo mode 0: disable turbo mode 1: turbo mode 1 2: turbo mode 2 3: turbo mode 3

4.12.2.1.173.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.174 int wlan_set_uap_turbo_mode (t_u8 mode)

Set UAP Turbo mode.

4.12.2.1.174.1 Parameters

in	*mode*	turbo mode 0: disable turbo mode 1: turbo mode 1 2: turbo mode 2 3: turbo mode 3

4.12.2.1.174.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.175 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.

4.12.2.1.175.1 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

4.12.2.1.176 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.

4.12.2.1.176.1 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

4.12.2.1.176.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.177 int wlan_set_country_ie_ignore (uint8_t * ignore)

Set ignore region code

4.12.2.1.177.1 Parameters

in	*ignore*	0: Don't ignore 1: ignore

4.12.2.1.177.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.178 int wlan_set_region_code (unsigned int region_code)

Set region code

4.12.2.1.178.1 Parameters

in	*region_code*	

4.12.2.1.178.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.179 int wlan_get_region_code (unsigned int * region_code)

Get region code

4.12.2.1.179.1 Parameters

out	*region_code*	pointer

4.12.2.1.179.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.180 int wlan_set_11d_state (int bss_type, int state)

Set STA/uAP 80211d feature enable/disable

4.12.2.1.180.1 Parameters

in	*bss_type*	0: STA, 1: uAP
in	*state*	0: disable, 1: enable

4.12.2.1.180.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.181 int wlan_single_ant_duty_cycle (t_u16 enable, t_u16 nbTime, t_u16 wlanTime)

Set single ant duty cycle.

4.12.2.1.181.1 Parameters

in	*enable*	
in	*nbTime*	
in	*wlanTime*	

4.12.2.1.181.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.1.182 int wlan_dual_ant_duty_cycle (t_u16 enable, t_u16 nbTime, t_u16 wlanTime, t_u16 wlanBlockTime)

Set dual ant duty cycle.

4.12.2.1.182.1 Parameters

in	*enable*	
in	*nbTime*	
in	*wlanTime*	
in	*wlanBlockTime*	

4.12.2.1.182.2 Returns

WM_SUCCESS if successful otherwise failure.

4.12.2.2 Macro documentation

4.12.2.2.1 #define ACTION_GET (0U)

Action GET

4.12.2.2.2 #define ACTION_SET (1)

Action SET

4.12.2.2.3 #define IEEEtypes_SSID_SIZE 32U

Maximum SSID length

4.12.2.2.4 #define IEEEtypes_ADDRESS_SIZE 6

MAC Address length

4.12.2.2.5 #define WLAN_RESCAN_LIMIT 5U

The number of times that the WLAN Connection Manager will look for a network before giving up.

4.12.2.2.6 #define WLAN_RECONNECT_LIMIT 5U

The number of times that the WLAN Connection Manager will attempt a reconnection with the network before giving up.

4.12.2.2.7 #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

4.12.2.2.8 #define WLAN_NETWORK_NAME_MAX_LENGTH 32U

The space reserved for storing network names, wlan_network

4.12.2.2.9 #define WLAN_PSK_MIN_LENGTH 8U

The space reserved for storing PSK (password) phrases.

4.12.2.2.10 #define WLAN_PSK_MAX_LENGTH 65U

Max WPA2 passphrase can be upto 63 ASCII chars or 64 hexadecimal digits

4.12.2.2.11 #define WLAN_PASSWORD_MIN_LENGTH 8U

Min WPA3 password can be upto 8 ASCII chars

4.12.2.2.12 #define WLAN_PASSWORD_MAX_LENGTH 255U

Max WPA3 password can be upto 255 ASCII chars

4.12.2.2.13 #define IDENTITY_MAX_LENGTH 64U

Max WPA2 Enterprise identity can be upto 256 characters

4.12.2.2.14 #define PASSWORD_MAX_LENGTH 128U

Max WPA2 Enterprise password can be upto 256 unicode characters

4.12.2.2.15 #define MAX_USERS 8U

Max identities for EAP server users

4.12.2.2.16 #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.

4.12.2.2.17 #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

4.12.2.2.18 #define HASH_MAX_LENGTH 40U

MAX CA Cert hash len

4.12.2.2.19 #define DOMAIN_MATCH_MAX_LENGTH 64U

MAX domain len

4.12.2.2.20 #define WLAN_MAX_KNOWN_NETWORKS CONFIG_WLAN_KNOWN_NETWORKS

The size of the list of known networks maintained by the WLAN Connection Manager

4.12.2.2.21 #define WLAN_PMK_LENGTH 32

Length of a pairwise master key (PMK). It's always 256 bits (32 Bytes)

4.12.2.2.22 #define WLAN_ERROR_NONE 0

The operation was successful.

4.12.2.2.23 #define WLAN_ERROR_PARAM 1

The operation failed due to an error with one or more parameters.

4.12.2.2.24 #define WLAN_ERROR_NOMEM 2

The operation could not be performed because there is not enough memory.

4.12.2.2.25 #define WLAN_ERROR_STATE 3

The operation could not be performed in the current system state.

4.12.2.2.26 #define WLAN_ERROR_ACTION 4

The operation failed due to an internal error.

4.12.2.2.27 #define WLAN_ERROR_PS_ACTION 5

The operation to change power state could not be performed

4.12.2.2.28 #define WLAN_ERROR_NOT_SUPPORTED 6

The requested feature is not supported

4.12.2.2.29 #define WLAN_MGMT_ACTION MBIT(13)

BITMAP for Action frame

4.12.2.2.30 #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)
```

4.12.2.3 Typedef Documentation

4.12.2.3.1 typedef wifi_pkt_stats_t wlan_pkt_stats_t

Wi-Fi firmware stat from wifi_pkt_stats_t

4.12.2.3.2 typedef wifi_scan_channel_list_t wlan_scan_channel_list_t

Configuration for Wireless scan channel list from wifi_scan_channel_list_t

4.12.2.3.3 typedef wifi_scan_params_v2_t wlan_scan_params_v2_t

Configuration for wireless scanning parameters v2 from wifi_scan_params_v2_t

4.12.2.3.4 typedef wifi_cal_data_t wlan_cal_data_t

Configuration for Wireless Calibration data from wifi_cal_data_t

4.12.2.3.5 typedef wifi_flt_cfg_t wlan_flt_cfg_t

Configuration for Memory Efficient Filters in Wi-Fi firmware from wifi_flt_cfg_t

4.12.2.3.6 typedef wifi_wowlan_ptn_cfg_t wlan_wowlan_ptn_cfg_t

Configuration for wowlan pattern parameters from wifi_wowlan_ptn_cfg_t

4.12.2.3.7 typedef wifi_tcp_keep_alive_t wlan_tcp_keep_alive_t

Configuration for TCP Keep alive parameters from wifi_tcp_keep_alive_t

4.12.2.3.8 typedef wifi_ds_rate wlan_ds_rate

Configuration for TX Rate and Get data rate from wifi_ds_rate

4.12.2.3.9 typedef wifi_ed_mac_ctrl_t wlan_ed_mac_ctrl_t

Configuration for ED MAC Control parameters from wifi_ed_mac_ctrl_t

4.12.2.3.10 typedef wifi_bandcfg_t wlan_bandcfg_t

Configuration for Band from wifi_bandcfg_t

4.12.2.3.11 typedef wifi_cw_mode_ctrl_t wlan_cw_mode_ctrl_t

Configuration for CW Mode parameters from wifi_cw_mode_ctrl_t

4.12.2.3.12 typedef wifi_chanlist_t wlan_chanlist_t

Configuration for Channel list from wifi_chanlist_t

4.12.2.3.13 typedef wifi_txpwrlimit_t wlan_txpwrlimit_t

Configuration for TX Pwr Limit from wifi_txpwrlimit_t

4.12.2.3.14 typedef wifi_net_monitor_t wlan_net_monitor_t

Configuration for Net monitor from wifi_net_monitor_t

4.12.2.3.15 typedef wifi_rssi_info_t wlan_rssi_info_t

Configuration for RSSI information wifi_rssi_info_t

4.12.2.4 Enumeration type documentation

4.12.2.4.1 enum wm_wlan_errno

Enum for wlan errors

4.12.2.4.1.1 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

4.12.2.4.2 enum wlan_event_reason

WLAN Connection Manager event reason

4.12.2.4.2.1 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.
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_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
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.

4.12.2.4.3 enum wlan_wakeup_event_t

Wakeup events for which wakeup will occur

4.12.2.4.3.1 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

4.12.2.4.4 enum wlan_connection_state

WLAN station/micro-AP/Wi-Fi Direct Connection/Status state

4.12.2.4.4.1 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.

4.12.2.4.5 enum wlan_ps_mode

Station Power save mode

4.12.2.4.5.1 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
WLAN_WNM	WNM power save mode
WLAN_WNM_DEEP_SLEEP	WNM and Deep sleep power save mode

4.12.2.4.6 enum wlan_security_type

Network security types

4.12.2.4.6.1 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_WPA3_SAE	The network uses WPA3 security with SAE.
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_EAP_TLS	The network uses WPA2 Enterprise EAP-TLS security The identity field in wlan_network structure is 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_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.

4.12.2.4.7 enum address_types

Address types to be used by the element wlan_ip_config.addr_type below

4.12.2.4.7.1 Enumerator

ADDR_TYPE_STATIC	static IP address
ADDR_TYPE_DHCP	Dynamic IP address
ADDR_TYPE_LLA	Link level address

4.13 wlan_11d.h file reference

WLAN module 11d API.

4.13.1 Function documentation

4.13.1.1 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.

4.13.1.1.1 Parameters

in	*state*	1: enable, 0: disable

4.13.1.1.2 Returns

-WM_FAIL if operation was failed.

WM_SUCCESS if operation was successful.

4.13.1.2 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.

4.13.1.2.1 Parameters

in	*state*	1: enable, 0: disable

4.13.1.2.2 Returns

-WM_FAIL if operation was failed.

WM_SUCCESS if operation was successful.

4.14 wlan_tests.h file reference

WLAN Connection Manager Tests.

4.14.1 Function documentation

4.14.1.1 void print_txpwrlimit (wlan_txpwrlimit_t * txpwrlimit)

Print the TX PWR Limit table received from Wi-Fi firmware

4.14.1.1.1 Parameters

in	*txpwrlimit*	A wlan_txpwrlimit_t struct holding the the TX PWR Limit table received from Wi-Fi firmware.

4.15 wm_net.h file reference

Network Abstraction Layer.

4.15.1 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

4.15.2 Function documentation

4.15.2.1 int net_dhcp_hostname_set (char * hostname)

Set hostname for network interface

4.15.2.1.1 Parameters

in	*hostname*	Hostname to be set.

Note:

NULL is a valid value for hostname.

4.15.2.1.2 Returns

WM_SUCESS

4.15.2.2 void net_stop_dhcp_timer (void)

Deactivate the dhcp timer

4.15.2.3 static int net_socket_blocking (int sock, int state)[inline], [static]

Set socket blocking option as on or off

4.15.2.3.1 Parameters

in	*sock*	socket number to be set for blocking option.
in	*state*	set blocking on or off

4.15.2.3.2 Returns

WM_SUCESS otherwise standard LWIP error codes.

4.15.2.4 static int net_get_sock_error (int sock)[inline], [static]

Get error number from provided socket

4.15.2.4.1 Parameters

in	*sock*	socket number to get error number.

4.15.2.4.2 Returns

error number.

4.15.2.5 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)

4.15.2.5.1 Parameters

in	*cp*	IPv4 host address in dotted-decimal notation.

4.15.2.5.2 Returns

IPv4 address in binary form

4.15.2.6 void net_wlan_set_mac_address (unsigned char * stamac, unsigned char * uapmac)

set MAC hardware address to lwip network interface

4.15.2.6.1 Parameters

in	*stamac*	sta MAC address.
in	*uapmac*	uap MAC address.

4.15.2.7 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

4.15.2.7.1 Parameters

in	*buf*	input stack buffer.
in	*in_offset*	offset to skip.

4.15.2.7.2 Returns

the payload pointer after skip a number of bytes

4.15.2.8 static void net_stack_buffer_free (void * buf)[inline], [static]

Free a buffer allocated from stack memory

4.15.2.8.1 Parameters

in	*buf*	stack buffer pointer.

4.15.2.9 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

4.15.2.9.1 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

4.15.2.9.2 Returns

copy status based on stack definition.

4.15.2.10 static void* net_stack_buffer_get_payload (void * buf)[inline], [static]

Get the data payload inside the stack buffer.

4.15.2.10.1 Parameters

in	*buf*	input stack buffer.

4.15.2.10.2 Returns

the payload pointer of the stack buffer.

4.15.2.11 static int net_gethostbyname (const char * cp, struct hostent ** hentry)[inline], [static]

Get network host entry

4.15.2.11.1 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.

4.15.2.11.2 Returns

WM_SUCESS if operation successful.

-WM_FAIL if operation fails.

4.15.2.12 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

4.15.2.12.1 Parameters

in	*addr*	IP address in network byte order.
out	*cp*	buffer in which IPv4 dotted-decimal string is returned.

4.15.2.13 static bool net_is_ip_or_ipv6 (const uint8_t * buffer)[inline], [static]

Check whether buffer is IPv4 or IPV6 packet type

4.15.2.13.1 Parameters

in	*buffer*	pointer to buffer where packet to be checked located.

4.15.2.13.2 Returns

true if buffer packet type matches with IPv4 or IPv6, false otherwise.

4.15.2.14 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.

4.15.2.14.1 Parameters

in	*sock*	socket descriptor

4.15.2.14.2 Returns

[out] interface handle

4.15.2.15 int net_wlan_init (void)

Initialize TCP/IP networking stack

4.15.2.15.1 Returns

WM_SUCCESS on success

-WM_FAIL otherwise

4.15.2.16 int net_wlan_deinit (void)

DiInitialize TCP/IP networking stack

4.15.2.16.1 Returns

WM_SUCCESS on success

-WM_FAIL otherwise

4.15.2.17 struct netif* net_get_sta_interface (void)

Get STA interface netif structure pointer

4.15.2.17.1 Returns

A pointer to STA interface netif structure

4.15.2.18 struct netif* net_get_uap_interface (void)

Get uAP interface netif structure pointer

4.15.2.18.1 Returns

A pointer to uAP interface netif structure

4.15.2.19 int net_get_if_name_netif (char * pif_name, struct netif * iface)

Get interface name for given netif

4.15.2.19.1 Parameters

out	*pif_name*	Buffer to store interface name
in	*iface*	Interface to get the name

4.15.2.19.2 Returns

WM_SUCCESS on success

-WM_FAIL otherwise

4.15.2.20 int net_alloc_client_data_id ()

Get client data index for storing private data in * netif.

4.15.2.20.1 Returns

allocated client data index, -1 if error or not supported.

4.15.2.21 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.

4.15.2.21.1 Returns

station interface handle

4.15.2.22 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.

4.15.2.22.1 Returns

micro-AP interface handle

4.15.2.23 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.

4.15.2.23.1 Parameters

in	*intrfc_handle*	interface handle

4.15.2.24 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.

4.15.2.24.1 Parameters

in	*intrfc_handle*	interface handle

4.15.2.25 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.

4.15.2.25.1 Parameters

in	*intrfc_handle*	interface handle

4.15.2.26 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.

4.15.2.26.1 Parameters

in	*intrfc_handle*	interface handle

4.15.2.27 int net_configure_address (struct net_ip_config * addr, void * intrfc_handle)

Configure IP address for interface

4.15.2.27.1 Parameters

in	*addr*	Address that needs to be configured.
in	*intrfc_handle*	Handle for network interface to be configured.

4.15.2.27.2 Returns

WM_SUCCESS on success or an error code.

4.15.2.28 void net_configure_dns (struct net_ip_config * ip, unsigned int role)

Configure DNS server address

4.15.2.28.1 Parameters

in	*ip*	IP address of the DNS server to set
in	*role*	Network wireless BSS Role

4.15.2.29 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.

4.15.2.29.1 Parameters

out	*addr*	net_ip_config
in	*intrfc_handle*	interface handle

4.15.2.29.2 Returns

WM_SUCCESS on success or error code.

4.15.2.30 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.

4.15.2.30.1 Parameters

out	*if_name*	interface name pointer
in	*intrfc_handle*	interface handle

4.15.2.30.2 Returns

WM_SUCCESS on success or error code.

4.15.2.31 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.

4.15.2.31.1 Parameters

out	*ip*	ip address pointer
in	*intrfc_handle*	interface handle

4.15.2.31.2 Returns

WM_SUCCESS on success or error code.

4.15.2.32 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.

4.15.2.32.1 Parameters

in	*nm*	Subnet Mask pointer
in	*intrfc_handle*	interface

4.15.2.32.2 Returns

WM_SUCCESS on success or error code.

4.15.2.33 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.

4.15.2.34 void net_stat (void)

Display network statistics

4.15.3 Enumeration type documentation

4.15.3.1 enum net_address_types

Address types to be used by the element net_ip_config.addr_type below

4.15.3.1.1 Enumerator

NET_ADDR_TYPE_STATIC	static IP address
NET_ADDR_TYPE_DHCP	Dynamic IP address
NET_ADDR_TYPE_LLA	Link level address

4.16 wm_os.h file reference

OS Abstraction Layer.

4.16.1 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.

4.16.2 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().

4.16.2.1 Function documentation

4.16.2.1.1 unsigned os_ticks_get (void)

Get current OS tick counter value

4.16.2.1.1.1 Returns

32 bit value of ticks since boot-up

4.16.2.1.2 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.

4.16.2.1.2.1 Returns

Time in micro-secs since bootup

4.16.2.1.3 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.

4.16.2.1.3.1 Parameters

in	*msecs*	Milliseconds

4.16.2.1.3.2 Returns

Number of OS ticks corresponding to msecs

4.16.2.1.4 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.

4.16.2.1.4.1 Parameters

in	*ticks*	OS ticks

4.16.2.1.4.2 Returns

Number of milliseconds corresponding to ticks

4.16.2.1.5 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.

4.16.2.1.5.1 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.

4.16.2.1.5.2 Returns

WM_SUCCESS if thread was created successfully

-WM_FAIL if thread creation failed

4.16.2.1.6 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.

4.16.2.1.6.1 Parameters

in	*thandle*	Pointer to the thread handle of the thread to be deleted. If self deletion is required NULL should be passed.

4.16.2.1.6.2 Returns

WM_SUCCESS if operation success

-WM_FAIL if operation fails

4.16.2.1.7 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()

4.16.2.1.7.1 Parameters

in	*ticks*	Number of ticks to sleep

4.16.2.1.8 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.

4.16.2.1.8.1 Parameters

in	*thandle*	Pointer to thread handle

4.16.2.1.9 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.

4.16.2.1.9.1 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.

4.16.2.1.9.2 Returns

WM_SUCCESS if queue creation was successful

-WM_FAIL if queue creation failed

4.16.2.1.10 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.

4.16.2.1.10.1 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.

4.16.2.1.10.2 Returns

WM_SUCCESS if send operation was successful

-WM_E_INVAL if invalid parameters are passed

-WM_FAIL if send operation failed

4.16.2.1.11 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.

4.16.2.1.11.1 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.

4.16.2.1.11.2 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.

4.16.2.1.12 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.

4.16.2.1.12.1 Parameters

in	*qhandle*	Pointer to handle of the queue to be deleted.

4.16.2.1.12.2 Returns

Currently always returns WM_SUCCESS

4.16.2.1.13 int os_queue_get_msgs_waiting (os_queue_t * qhandle)

Return the number of messages stored in queue.

4.16.2.1.13.1 Parameters

in	*qhandle*	Pointer to handle of the queue to be queried.

4.16.2.1.13.2 Returns

Number of items in the queue

-WM_E_INVAL if invalid parameters are passed

4.16.2.1.14 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.

4.16.2.1.14.1 Parameters

in	*func*	The callback function

4.16.2.1.14.2 Returns

WM_SUCCESS on success

-WM_FAIL on error

4.16.2.1.15 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.

4.16.2.1.15.1 Parameters

in	*func*	The callback function

4.16.2.1.15.2 Returns

WM_SUCCESS on success

-WM_FAIL on error

4.16.2.1.16 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().

4.16.2.1.16.1 Parameters

in	*func*	The callback function

4.16.2.1.16.2 Returns

WM_SUCCESS on success

-WM_FAIL on error

4.16.2.1.17 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().

4.16.2.1.17.1 Parameters

in	*func*	Callback function

4.16.2.1.17.2 Returns

WM_SUCCESS on success

-WM_FAIL on error

4.16.2.1.18 int os_mutex_create (os_mutex_t * mhandle, const char * name, int flags)

Create mutex

This function creates a mutex.

4.16.2.1.18.1 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.

4.16.2.1.18.2 Returns

WM_SUCCESS on success

-WM_FAIL on error

4.16.2.1.19 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.

4.16.2.1.19.1 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.

4.16.2.1.19.2 Returns

WM_SUCCESS when mutex is acquired

-WM_E_INVAL if invalid parameters are passed

-WM_FAIL on failure

4.16.2.1.20 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.

4.16.2.1.20.1 Parameters

in	*mhandle*	Pointer to the mutex handle

4.16.2.1.20.2 Returns

WM_SUCCESS when mutex is released

-WM_E_INVAL if invalid parameters are passed

-WM_FAIL on failure

4.16.2.1.21 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.

4.16.2.1.21.1 Parameters

out	*mhandle*	Pointer to a mutex handle
in	*name*	Name of the mutex as NULL terminated string

4.16.2.1.21.2 Returns

WM_SUCCESS on success

-WM_E_INVAL on invalid parameter.

-WM_FAIL on error

4.16.2.1.22 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().

4.16.2.1.22.1 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.

4.16.2.1.22.2 Returns

WM_SUCCESS when recursive mutex is acquired

-WM_FAIL on failure

4.16.2.1.23 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()

4.16.2.1.23.1 Parameters

in	*mhandle*	Pointer to the mutex handle

4.16.2.1.23.2 Returns

WM_SUCCESS when mutex is released

-WM_FAIL on failure

4.16.2.1.24 int os_mutex_delete (os_mutex_t * mhandle)

Delete mutex

This function deletes a mutex.

4.16.2.1.24.1 Parameters

in	*mhandle*	Pointer to the mutex handle

Note:

A mutex should not be deleted if other tasks are blocked on it.

4.16.2.1.24.2 Returns

WM_SUCCESS on success

4.16.2.1.25 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

4.16.2.1.25.1 Parameters

in	*wait_time*	Timeout specified in no. of OS ticks

4.16.2.1.25.2 Returns

WM_SUCCESS when notification is successful

-WM_FAIL on failure or timeout

4.16.2.1.26 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

4.16.2.1.26.1 Parameters

in	*task*	Task handle to be notified

4.16.2.1.26.2 Returns

WM_SUCCESS when notification is successful

-WM_FAIL on failure or timeout

4.16.2.1.27 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.

4.16.2.1.27.1 Parameters

out	*mhandle*	Pointer to a semaphore handle
in	*name*	Name of the semaphore

4.16.2.1.27.2 Returns

WM_SUCCESS on success

-WM_FAIL on error

4.16.2.1.28 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.

4.16.2.1.28.1 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().

4.16.2.1.28.2 Returns

WM_SUCCESS on success

-WM_FAIL on error

4.16.2.1.29 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.

4.16.2.1.29.1 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.

4.16.2.1.29.2 Returns

WM_SUCCESS when semaphore is acquired

-WM_E_INVAL if invalid parameters are passed

-WM_FAIL on failure

4.16.2.1.30 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.

4.16.2.1.30.1 Parameters

in	*mhandle*	Pointer to a semaphore handle

4.16.2.1.30.2 Returns

WM_SUCCESS when semaphore is released

-WM_E_INVAL if invalid parameters are passed

-WM_FAIL on failure

4.16.2.1.31 int os_semaphore_getcount (os_semaphore_t * mhandle)

Get semaphore count

This function returns the current value of a semaphore.

4.16.2.1.31.1 Parameters

in	*mhandle*	Pointer to a semaphore handle

4.16.2.1.31.2 Returns

current value of the semaphore

4.16.2.1.32 int os_semaphore_delete (os_semaphore_t * mhandle)

Delete a semaphore

This function deletes the semaphore.

4.16.2.1.32.1 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)

4.16.2.1.32.2 Returns

WM_SUCCESS on success

4.16.2.1.33 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.

4.16.2.1.33.1 Parameters

in	*plock*	Pointer to a reader-writer lock handle
in	*mutex_name*	Name of the mutex
in	*lock_name*	Name of the lock

4.16.2.1.33.2 Returns

WM_SUCCESS on success

-WM_FAIL on error

4.16.2.1.34 void os_rwlock_delete (os_rw_lock_t * lock)

Delete a reader-write lock

This function deletes a reader-writer lock.

4.16.2.1.34.1 Parameters

in	*lock*	Pointer to the reader-writer lock handle

4.16.2.1.35 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.

4.16.2.1.35.1 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.

4.16.2.1.35.2 Returns

WM_SUCCESS on success

-WM_FAIL on error

4.16.2.1.36 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().

4.16.2.1.36.1 Parameters

in	*lock*	Pointer to the reader-writer lock handle

4.16.2.1.37 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.

4.16.2.1.37.1 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.

4.16.2.1.37.2 Returns

WM_SUCCESS on success

-WM_FAIL on error

4.16.2.1.38 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().

4.16.2.1.38.1 Parameters

in	*lock*	pointer to the reader-writer lock handle

4.16.2.1.38.2 Returns

WM_SUCCESS if unlock operation successful.

-WM_FAIL if unlock operation failed.

4.16.2.1.39 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.

4.16.2.1.39.1 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

4.16.2.1.39.2 Returns

WM_SUCCESS if timer created successfully

-WM_FAIL if timer creation fails

4.16.2.1.40 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().

4.16.2.1.40.1 Parameters

in	*timer_t*	Pointer to a timer handle

4.16.2.1.40.2 Returns

WM_SUCCESS if timer activated successfully

-WM_E_INVAL if invalid parameters are passed

-WM_FAIL if timer fails to activate

4.16.2.1.41 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.

4.16.2.1.41.1 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

4.16.2.1.41.2 Returns

WM_SUCCESS on success

-WM_E_INVAL if invalid parameters are passed

-WM_FAIL on failure

4.16.2.1.42 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.

4.16.2.1.42.1 Parameters

in	*timer_t*	Pointer to a timer handle

4.16.2.1.42.2 Returns

true if timer is active

false if time is not active

4.16.2.1.43 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().

4.16.2.1.43.1 Parameters

in	*timer_t*	Pointer to timer handle. The timer handle is received in the timer callback.

4.16.2.1.43.2 Returns

The timer context i.e. the callback argument passed to os_timer_create().

4.16.2.1.44 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().

4.16.2.1.44.1 Parameters

in	*timer_t*	Pointer to a timer handle

4.16.2.1.44.2 Returns

WM_SUCCESS on success

-WM_E_INVAL if invalid parameters are passed

-WM_FAIL on failure

4.16.2.1.45 int os_timer_deactivate (os_timer_t * timer_t)

Deactivate timer

This function deactivates (or stops) a timer that was previously started.

4.16.2.1.45.1 Parameters

in	*timer_t*	handle populated by os_timer_create()

4.16.2.1.45.2 Returns

WM_SUCCESS on success

-WM_E_INVAL if invalid parameters are passed

-WM_FAIL on failure

4.16.2.1.46 int os_timer_delete (os_timer_t * timer_t)

Delete timer

This function deletes a timer.

4.16.2.1.46.1 Parameters

in	*timer_t*	Pointer to a timer handle

4.16.2.1.46.2 Returns

WM_SUCCESS on success

-WM_E_INVAL if invalid parameters are passed

-WM_FAIL on failure

4.16.2.1.47 void* os_mem_alloc (size_t size)

Allocate memory

This function allocates memory dynamically.

4.16.2.1.47.1 Parameters

in	*size*	Size of the memory to be allocated

4.16.2.1.47.2 Returns

Pointer to the allocated memory

NULL if allocation fails

4.16.2.1.48 void* os_mem_calloc (size_t size)

Allocate memory and zero it

This function allocates memory dynamically and sets the memory contents to zero.

4.16.2.1.48.1 Parameters

in	*size*	Size of the memory to be allocated

4.16.2.1.48.2 Returns

Pointer to the allocated memory

NULL if allocation fails

4.16.2.1.49 void os_mem_free (void * ptr)

Free Memory

This function frees dynamically allocated memory using any of the dynamic allocation primitives.

4.16.2.1.49.1 Parameters

in	*ptr*	Pointer to the memory to be freed

4.16.2.1.50 void os_disable_all_interrupts (void)

Disables all interrupts at NVIC level

4.16.2.1.51 void os_enable_all_interrupts (void)

Enable all interrupts at NVIC lebel

4.16.2.1.52 static void os_lock_schedule (void)[inline], [static]

Disable all tasks schedule

4.16.2.1.53 static void os_unlock_schedule (void)[inline], [static]

Enable all tasks schedule

4.16.2.1.54 static void os_srand (uint32_t seed)[inline], [static]

This function initialize the seed for rand generator

4.16.2.1.54.1 Parameters

in	*seed*	Seed for random number generator

4.16.2.1.55 static uint32_t os_rand ()[inline], [static]

This function generate a random number

4.16.2.1.55.1 Returns

a uint32_t random numer

4.16.2.1.56 static uint32_t os_rand_range (uint32_t low, uint32_t high)[inline], [static]

This function generate a random number in a range

4.16.2.1.56.1 Parameters

in	*low*	range low
in	*high*	range high

4.16.2.1.56.2 Returns

a uint32_t random numer

4.16.2.2 Macro documentation

4.16.2.2.1 #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.

4.16.2.2.1.1 Returns

The current value of microsecond counter. Force a context switch

4.16.2.2.2 #define os_ticks_to_unblock() xTaskGetUnblockTime()

Get ticks to next thread wakeup

4.16.2.2.3 #define os_thread_stack_define(stackname, stacksize) os_thread_stack_t stackname = {(stacksize) / (sizeof(portSTACK_TYPE))}

Helper macro to define the stack size (in bytes) before a new thread is created using the function os_thread_create().

4.16.2.2.4 #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().

4.16.2.2.5 #define OS_WAIT_FOREVER portMAX_DELAY

Wait Forever

4.16.2.2.6 #define OS_NO_WAIT 0

Do Not Wait

4.16.2.2.7 #define OS_MUTEX_INHERIT 1

Priority Inheritance Enabled

4.16.2.2.8 #define OS_MUTEX_NO_INHERIT 0

Priority Inheritance Disabled

4.16.2.2.9 #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.

4.16.2.2.10 #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.

4.16.2.3 Typedef Documentation

4.16.2.3.1 typedef int(* cb_fn) (os_rw_lock_t *plock, unsigned int wait_time)

This is prototype of reader callback

4.16.2.4 Enumeration type documentation

4.16.2.4.1 enum os_timer_reload_t

OS Timer reload options

4.16.2.4.1.1 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.

4.16.2.4.2 enum os_timer_activate_t

OS Timer Activate Options

4.16.2.4.2.1 Enumerator

OS_TIMER_AUTO_ACTIVATE	Start the timer on creation.
OS_TIMER_NO_ACTIVATE	Do not start the timer on creation.

4.17 wm_utils.h file reference

Utility functions.

4.17.1 Detailed description

Collection of some common helper functions

4.17.2 Function documentation

4.17.2.1 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.

4.17.2.1.1 Parameters

in	*ibuf*	input buffer
out	*obuf*	output buffer
in	*max_olen*	Maximum output buffer length

4.17.2.1.2 Returns

length of the binary string

4.17.2.2 void bin2hex (uint8_t * src, char * dest, unsigned int src_len, unsigned int dest_len)

Convert given binary array to equivalent hex representation.

4.17.2.2.1 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

4.17.2.3 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.

4.17.2.3.1 Parameters

in	*func*	Function pointer of type random_hdlr_t

4.17.2.3.2 Returns

WM_SUCCESS if successful

-WM_E_NOSPC if there is no space available for additional handlers

4.17.2.4 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()

4.17.2.4.1 Parameters

in	*func*	Function pointer of type random_hdlr_t used during registering

4.17.2.4.2 Returns

WM_SUCCESS if successful

-WM_E_INVAL if the passed pointer is invalid

4.17.2.5 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.

4.17.2.5.1 Parameters

in	*func*	Function pointer of type random_hdlr_t

4.17.2.5.2 Returns

WM_SUCCESS if successful

-WM_E_NOSPC if there is no space available for additional handlers

4.17.2.6 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()

4.17.2.6.1 Parameters

in	*func*	Function pointer of type random_hdlr_t used during registering

4.17.2.6.2 Returns

WM_SUCCESS if successful

-WM_E_INVAL if the passed pointer is invalid

4.17.2.7 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().

4.17.2.8 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.

4.17.2.8.1 Returns

Random seed

4.17.2.9 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.

4.17.2.9.1 Parameters

out	*buf*	The buffer to be populated with random data
in	*size*	The number of bytes of the random sequence required

4.17.2.10 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.

4.17.2.10.1 Parameters

in	*s*	Pointer to string to be duplicated

4.17.2.10.2 Returns

Pointer to newly allocated string which is duplicate of input string

NULL on error

4.17.2.11 uint32_t soft_crc32 (const void * data__, int data_size, uint32_t crc)

Calculate CRC32 using software algorithm

4.17.2.11.1 Precondition

soft_crc32_init()

soft_crc32() allows the user to calculate CRC32 values of arbitrary sized buffers across multiple calls.

4.17.2.11.2 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.

4.17.2.11.3 Returns

Calculated CRC32 value

4.17.2.12 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}

4.17.2.12.1 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.

4.17.2.13 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}

4.17.2.13.1 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.

4.17.2.13.2 Returns

'true' If verification successful.

'false' If verification fails.

4.17.3 Macro documentation

4.17.3.1 #define dump_hex(...)

```
Value:    do                \
    {                \
    } while (0)
```

4.17.3.2 #define dump_hex_ascii(...)

```
Value:    do                    \
    {                    \
    } while (0)
```

4.17.3.3 #define dump_ascii(...)

```
Value:    do                \
    {                \
    } while (0)
```

4.17.3.4 #define print_ascii(...)

```
Value:    do                \
    {                \
    } while (0)
```

4.17.3.5 #define dump_json(...)

```
Value:    do                \
    {                \
    } while (0)
```

4.17.4 Typedef Documentation

4.17.4.1 typedef uint32_t(* random_hdlr_t) (void)

Function prototype for a random entropy/seed generator

4.17.4.1.1 Returns

a 32bit random number

4.18 wmcrypto.h file reference

Crypto Functions.

4.18.1 Detailed description

This provides crypto wrapper functions that selectively call the WMSDKA or mbed TLS crypto functions based on the selected configuration.

4.18.2 Function documentation

4.18.2.1 void* nxp_dh_setup_key (uint8_t * public_key, uint32_t public_len, uint8_t * private_key, uint32_t private_len, DH_PG_PARAMS * dh_params)

4.18.2.1.1 Parameters

public_key	Pointers to public key generated
public_len	Length of public key
private_key	Pointers to private key generated randomly
private_len	Length of private key
dh_params	Parameters for DH algorithm

4.18.2.1.2 Returns

0 on success, -1 on failure

4.18.2.2 int nxp_dh_compute_key (void * dh, uint8_t * shared_key, uint32_t shared_len, uint8_t * public_key, uint32_t public_len, uint8_t * private_key, uint32_t private_len, DH_PG_PARAMS * dh_params)

4.18.2.2.1 Parameters

dh	DH key
shared_key	Pointer to agreed shared key generated
shared_len	Length of agreed shared key
public_key	Pointer to public key generated
public_len	Length of public key
private_key	Pointer to private key generated randomly
private_len	Length of private key
dh_params	Parameters for DH algorithm

4.18.2.2.2 Returns

0 on success, -1 on failure

4.18.2.3 void nxp_dh_free (void * dh_context)

4.18.2.3.1 Parameters

dh_context	DH key

4.18.2.3.2 Returns

None

4.18.2.4 uint32_t nxp_sha1_vector (size_t nmsg, const uint8_t * msg[], const size_t msglen[], uint8_t * mac, size_t maclen)

4.18.2.4.1 Parameters

nmsg	Number of elements in the data vector
msg	Pointers to the data areas
msglen	Lengths of the data blocks
mac	Buffer for the hash
maclen	Length of hash buffer

4.18.2.4.2 Returns

0 on success, -1 of failure

4.18.2.5 uint32_t nxp_sha256_vector (size_t nmsg, const uint8_t * msg[], const size_t msglen[], uint8_t * mac, size_t maclen)

4.18.2.5.1 Parameters

nmsg	Number of elements in the data vector
msg	Pointers to the data areas
msglen	Lengths of the data blocks
mac	Buffer for the hash
maclen	Length of hash buffer

4.18.2.5.2 Returns

0 on success, -1 on failure

4.18.2.6 void nxp_sha256 (size_t num_elem, const uint8_t * addr[], const size_t * len, uint8_t * mac)

4.18.2.6.1 Parameters

num_elem	Number of elements in the data vector

addr	Pointers to the data areas
len	Lengths of the data blocks
mac	Buffer for the hash

4.18.2.6.2 Returns

0 on success, -1 on failure

4.18.2.7 uint32_t nxp_hmac_sha256 (const uint8_t * key, uint32_t keylen, uint8_t * msg, uint32_t msglen, uint8_t * mac, uint32_t maclen)

4.18.2.7.1 Parameters

key	Key for HMAC operations
keylen	Length of the key in bytes
msg	Pointers to the data areas
msglen	Lengths of the data blocks
mac	Buffer for the hash (32 bytes)
maclen	Length of hash buffer

4.18.2.7.2 Returns

0 on success, -1 on failure

4.18.2.8 int nxp_kdf (uint8_t * key, uint32_t key_len, uint8_t * result, uint32_t result_len)

4.18.2.8.1 Parameters

key	Input key to generate authentication key (KDK)
key_len	Length of input key
result	result buffer
result_len	Length of result buffer

4.18.2.8.2 Returns

0 on success, 1 otherwise

4.18.2.9 int nxp_aes_wrap (uint8_t * plain_txt, uint32_t txt_len, uint8_t * cip_txt, uint8_t * kek, uint32_t kek_len, uint8_t * iv)

4.18.2.9.1 Parameters

plain_txt	Plaintext key to be wrapped
txt_len	Length of the plain key in bytes (16 bytes)
cip_txt	Wrapped key

kek	Key encryption key (KEK)
kek_len	Length of KEK in bytes (must be divisible by 16)
iv	Encryption IV for CBC mode (16 bytes)

4.18.2.9.2 Returns

0 on success, -1 on failure

4.18.2.10 int nxp_aes_unwrap (uint8_t * cip_txt, uint32_t txt_len, uint8_t * plain_txt, uint8_t * kek, uint32_t kek_len, uint8_t * iv)

4.18.2.10.1 Parameters

cip_txt	Wrapped key to be unwrapped
txt_len	Length of the wrapped key in bytes (16 bytes)
plain_txt	Plaintext key
kek	Key encryption key (KEK)
kek_len	Length of KEK in bytes (must be divisible by 16)
iv	Encryption IV for CBC mode (16 bytes)

4.18.2.10.2 Returns

0 on success, -1 on failure

4.18.2.11 int nxp_aes_wrap_ext (uint8_t * plain_txt, uint32_t plain_len, uint8_t * cip_txt, uint8_t * kek, uint32_t kek_Len, uint8_t * iv)

4.18.2.11.1 Parameters

plain_txt	Plaintext key to be wrapped
plain_len	Length of the plain key in bytes (16 bytes)
cip_txt	Wrapped key
kek	Key encryption key (KEK)
kek_Len	Length of KEK in bytes (must be divisible by 16)
iv	Encryption IV for CBC mode (16 bytes)

4.18.2.11.2 Returns

0 on success, -1 on failure

4.18.2.12 int nxp_aes_unwrap_ext (uint8_t * cip_txt, uint32_t cip_Len, uint8_t * plain_txt, uint8_t * kek, uint32_t key_len, uint8_t * iv)

4.18.2.12.1 Parameters

cip_txt	Wrapped key to be unwrapped
cip_Len	Length of the wrapped key in bytes (16 bytes)
plain_txt	Plaintext key
kek	Key encryption key (KEK)
key_len	Length of KEK in bytes (must be divisible by 16)
iv	Encryption IV for CBC mode (16 bytes)

4.18.2.12.2 Returns

0 on success, -1 on failure

4.18.2.13 void nxp_crypto_hmac_md5 (uint8_t * input, int len, uint8_t * hash, char * hash_key)

4.18.2.13.1 Parameters

input	Pointer to input data
len	Length of input data
hash	Pointer to hash
hash_key	Pointer to hash key

4.18.2.13.2 Returns

None

4.18.2.14 void nxp_crypto_md5 (uint8_t * input, int len, uint8_t * hash, int hlen)

4.18.2.14.1 Parameters

input	Pointer to input data
len	Length of input data
hash	Pointer to hash
hlen	Pointer to hash len

4.18.2.14.2 Returns

None

4.18.2.15 void nxp_crypto_pass_to_key (char * password, unsigned char * ssid, int ssidlength, int iterations, int output_len, unsigned char * output)

4.18.2.15.1 Parameters

password	Pointer to password
ssid	Pointer to ssid
ssidlength	Length of ssid
iterations	No if iterations
output_len	Length of output data
output	Pointer to output data

4.18.2.15.2 Returns

None

4.18.3 Macro documentation

4.18.3.1 #define SHA256_DIGEST_SIZE (256 / 8)

Digest size

4.18.3.2 #define SHA256_BLOCK_SIZE (512 / 8)

Block size

4.19 wmerrno.h file reference

Error Management.

4.19.1 Macro documentation

4.19.1.1 #define MOD_UNUSED_3 2

Unused

4.19.1.2 #define MOD_HTTPD 3

HTTPD module index

4.19.1.3 #define MOD_AF 4

Application framework module index

4.19.1.4 #define MOD_FTFS 5

FTFS module index

4.19.1.5 #define MOD_RFGET 6

RFGET module index

4.19.1.6 #define MOD_JSON 7

JSON module index

4.19.1.7 #define MOD_TELNETD 8

TELNETD module index

4.19.1.8 #define MOD_SMDNS 9

SIMPLE MDNS module index

4.19.1.9 #define MOD_EXML 10

EXML module index

4.19.1.10 #define MOD_DHCPD 11

DHCPD module index

4.19.1.11 #define MOD_MDNS 12

MDNS module index

4.19.1.12 #define MOD_SYSINFO 13

SYSINFO module index

4.19.1.13 #define MOD_UNUSED_1 14

Unused module index

4.19.1.14 #define MOD_CRYPTO 15

CRYPTO module index

4.19.1.15 #define MOD_HTTPC 16

HTTP-CLIENT module index

4.19.1.16 #define MOD_PROV 17

PROVISIONING module index

4.19.1.17 #define MOD_SPI 18

SPI module index

4.19.1.18 #define MOD_PSM 19

PSM module index

4.19.1.19 #define MOD_TTCP 20

TTCP module index

4.19.1.20 #define MOD_DIAG 21

DIAGNOSTICS module index

4.19.1.21 #define MOD_UNUSED_2 22

Unused module index

4.19.1.22 #define MOD_WPS 23

WPS module index

4.19.1.23 #define MOD_WLAN 24

WLAN module index

4.19.1.24 #define MOD_USB 25

USB module index

4.19.1.25 #define MOD_WIFI 26

WIFI driver module index

4.19.1.26 #define MOD_CRIT_ERR 27

Critical error module index

4.19.1.27 #define MOD_ERR_LAST 50

Last module index .Applications can define their own modules beyond this

4.19.1.28 #define WM_E_INSMALL

Value: `40 /* A finer version for WM_E_INVAL, where it clearly specifies that input is much smaller than minimum \ requirement */`

4.20 wmlog.h file reference

This file contains macros to print logs.

4.20.1 Detailed description

SPDX-License-Identifier: BSD-3-Clause

4.21 wmstats.h file reference

Wireless Microcontroller statistics.

4.21.1 Enumeration type documentation

4.21.1.1 enum wm_reboot_reason_t

Define return values

4.22 wmtime.h file reference

Time Management Subsystem.

4.22.1 Detailed description

This module provides the time management APIs. Internally wireless microcontroller manages time in terms of seconds since epoch (1 Jan 1970 00:00). This representation is known as "posix time". These routines can be used to do conversions between human readable time and posix time as well as to set the system time to desired value.

4.22.2 Function documentation

4.22.2.1 time_t http_date_to_time (const unsigned char * date)

Convert HTTP date format to POSIX time format

4.22.2.1.1 Parameters

in	*date*	HTTP date format

4.22.2.1.2 Returns

success or failure as: -WM_FAIL: Conversion failed. Invalid format/data else valid time_t value

4.22.2.2 int wmtime_time_set (const struct tm * tm)

Set the date and time

4.22.2.2.1 Parameters

in	*tm*	The rtc value is updated with the values in tm structure

4.22.2.2.2 Returns

success or failure as: 0: Success -1: Failed validation of tm structure

4.22.2.3 int wmtime_time_get (struct tm * tm)

Get date and time

4.22.2.3.1 Parameters

out	*tm*	tm structure is updated to get the current value in rtc

4.22.2.3.2 Returns

success or failure as: 0: Success non-zero: Internal error

4.22.2.4 int wmtime_time_set_posix (time_t time)

Set the date and time using posix time

4.22.2.4.1 Parameters

in	*time*	The rtc value is updated with the value present in time

4.22.2.4.2 Returns

success or failure as: 0: Success non-zero: Internal error

4.22.2.5 time_t wmtime_time_get_posix (void)

Get date and time in posix format

4.22.2.5.1 Returns

time_t value from RTC

4.22.2.6 struct tm* gmtime_r (const time_t * time, struct tm * result)

Convert to tm structure from POSIX/Unix time (Seconds since epoch)

4.22.2.6.1 Parameters

in	*time*	This is POSIX time that is to be converted into tm
out	*result*	This should point to pre-allocated tm instance

4.22.2.6.2 Returns

pointer to struct tm; NULL in case of error

4.22.2.7 time_t mktime (struct tm * tm)

Converts to POSIX/Unix time from tm structure

4.22.2.7.1 Parameters

in	*tm*	This is tm instance that is to be converted into time_t format

4.22.2.7.2 Returns

time_t POSIX/Unix time equivalent

4.22.2.8 char* asctime (const struct tm * tm)

Converts the broken-down time value tm into a null-terminated string.

4.22.2.8.1 Parameters

in	*tm*	This is tm instance that is to be converted string.

4.22.2.8.2 Returns

Pointer to a statically allocated string which contains the date and time format as follows "Tue Mar 24 09:20:14 2015". The statically allocated string might be overwritten by subsequent calls to any of the date and time functions.

4.22.2.9 int wmtime_init (void)

Initialize time subsystem including RTC. Sets system time to 1/1/1970 00:00 (i.e. epoch 0)

4.22.2.9.1 Returns

WM_SUCCESS on success, zero otherwise

4.22.2.10 bool is_wmtime_init_done ()

Get current wmtime initialization status

4.22.2.10.1 Returns

true if initialized, false if not.

4.23 wmtypes.h file reference

Consolidated Header for Data types.

4.23.1 Detailed description

SPDX-License-Identifier: BSD-3-Clause

5 Note about the source code in the document

The example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2023 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials must be provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

6 Revision history

Table 1. Revision history

Document ID	Release date	Description
RM00282 v.1.0	12 December 2023	• Initial version

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Tables

Contents

Date of release: 12 December 2023
Document identifier: RM00282