

Introduction

Atmel® SmartConnect ATWILC1000 is an IEEE® 802.11b/g/n link controller SoC for applications in the Internet-Of-Things. It is an ideal add-on to existing powerful MCU/MPU solutions bringing Wi-Fi® and Ethernet interface capabilities through a UART-to-Wi-Fi or SPI-to-Wi-Fi interface.

Features

- Wi-Fi IEEE 802.11 b/g/n STA, AP and Wi-Fi Direct® modes
- Wi-Fi Protected Setup (WPS)
- Support of WEP, WPA/WPA2 personal, and WPA/WPA2 Enterprise security
- Ultra-low cost IEEE 802.11b/g/n RF/PH/MAC SoC
- Low power consumption with different power saving modes
- SPI, I²C, and UART support
- Ethernet data interface
- Low footprint host driver with the following capabilities:
 - Can run on 8, 16, and 32 bit MCU
 - Little and Big endian support
 - Consumes about 8KB of code memory and 1KB of data memory on host MCU
- Concurrency support in the following modes:
 - Station – Station
 - Station – AP
 - Station – P2P client
 - Station – P2P GO










Table of Contents

1	Overview	6
1.1	Host Driver Architecture	6
1.1.1	WLAN API.....	6
1.1.2	Host Interface	6
1.1.3	Board Support Package.....	7
1.1.4	Serial Bus Interface	7
1.2	WILC System Architecture	8
1.2.2	Bus Interface.....	8
1.2.3	CPU	8
1.2.4	IEEE 802.11 MAC Hardware	8
1.2.5	Program Memory	9
1.2.6	Data Memory	9
1.2.7	Shared Packet Memory	9
1.2.8	IEEE 802.11 MAC Firmware.....	9
1.2.9	Memory Manager.....	9
1.2.10	Power Management.....	9
2	WILC Initialization and Simple Application	10
2.1	BSP Initialization	10
2.2	WILC Host Driver Initialization.....	10
2.3	WILC Event Handling	10
2.3.2	Asynchronous Events	11
2.3.3	Interrupt Handling	11
2.4	Code Example.....	12
3	WILC Configuration	13
3.1	Device Parameters.....	13
3.1.1	Firmware and Driver Version	13
3.2	WILC Modes of Operation	14
3.2.1	Idle Mode.....	14
3.2.2	Wi-Fi Station Mode	15
3.2.3	Wi-Fi Direct (P2P) Mode.....	15
3.2.4	Wi-Fi Hotspot (AP) Mode.....	15
3.3	Network Parameters.....	16
3.3.1	Device Name	16
3.3.2	Wi-Fi MAC Address	16
3.4	Power Saving Parameters.....	16
3.4.1	Power Saving Modes.....	17
3.4.2	Configuring Listen Interval and DTIM Monitoring.....	18
4	Wi-Fi Station Mode.....	20
4.1	Scan Configuration Parameters	20
4.1.1	Scan Region	20
4.1.2	Scan Options	20
4.2	Wi-Fi Scan.....	20
4.3	On Demand Wi-Fi Connection.....	21
4.4	Wi-Fi Security	22
4.5	Example Code.....	22

5	Wi-Fi AP Mode.....	24
5.1	Overview	24
5.2	Setting WILC AP Mode.....	24
5.3	Capabilities.....	24
5.4	Sequence Diagram.....	24
5.5	AP Mode Code Example	25
6	Wi-Fi Direct P2P Mode.....	26
6.1	Overview	26
6.2	WILC P2P Capabilities	26
6.3	WILC P2P Limitations.....	26
6.4	WILC P2P States	26
6.5	WILC P2P Listen State.....	26
6.6	WILC P2P Connection State	26
6.7	WILC P2P Disconnection State.....	27
6.8	P2P Mode Code Example	28
7	Wi-Fi Protected Setup.....	30
7.1.1	WPS Configuration Methods.....	30
7.1.2	WPS Limitations	30
7.1.3	WPS Control Flow	31
7.1.4	WPS Code Example	32
8	Concurrency.....	33
8.1	Limitations	33
8.2	Controlling Second Interface	33
8.3	Station-Station Concurrency.....	33
8.3.1	Concurrent WPS.....	34
8.4	Station-AP Concurrency	35
8.5	Station-P2P Client Concurrency.....	37
9	Data Send/Receive.....	39
9.1	Send Ethernet Frame	39
9.2	Receive Ethernet Frame.....	39
9.3	Concurrency Send.....	40
9.4	Concurrency Receive	40
10	Host Interface Protocol	41
10.1	Chip Initialization Sequence	42
10.2	Transfer Sequence Between HIF Layer and WILC Firmware.....	43
10.2.1	Frame Transmit	43
10.2.2	Frame Receive	43
10.3	HIF Message Header Structure.....	44
10.4	HIF Layer APIs	45
10.5	Scan Code Example.....	46
11	WILC SPI Protocol	51
11.1	Introduction.....	51
11.1.1	Command Format.....	52
11.1.2	Response Format	56
11.1.3	Data Packet Format.....	57
11.1.4	Error Recovery Mechanism	58

11.1.5	Clockless Registers Access.....	60
11.2	Message Flow for Basic Transactions.....	60
11.2.1	Read Single Word.....	60
11.2.2	Read Internal Register (for Clockless Registers).....	60
11.2.3	Read Block.....	61
11.2.4	Write Single Word.....	62
11.2.5	Write Internal Register (for Clockless Registers).....	62
11.2.6	Write Block.....	62
11.3	SPI Level Protocol Example.....	64
11.3.1	TX (Send Request).....	64
11.3.2	RX (Receive Response).....	74
12	ATWILC1000 Firmware Download	87
Appendix A	API Reference	89
A.1	WLAN Module	89
A.1.1	Defines	89
A.1.2	Enumeration/Typedef	91
A.1.3	Function.....	105
A.2	BSP	130
A.2.1	Defines	130
A.2.2	Data Types	130
A.2.3	Function.....	130
A.2.4	Enumeration/Typedef	132
13	Document Revision History	134

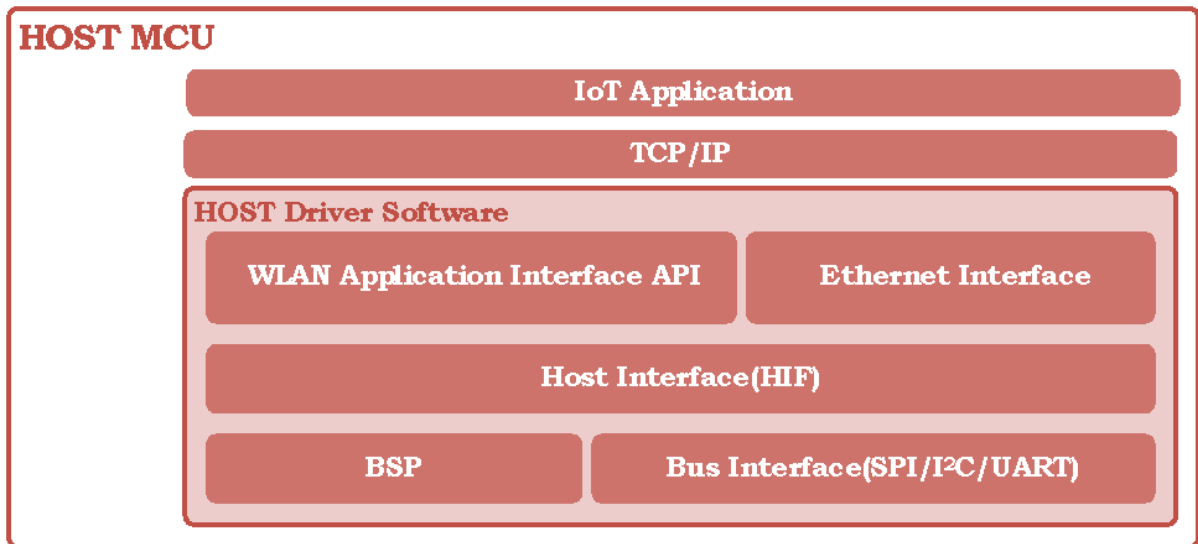
Icon Key Identifiers

	TIP	Useful Tips and Techniques
	INFO	Delivers Contextual Information About a Specific Topic
	IMPORTANT	Note to Quality and Performance
	TO DO	Objectives to be Completed
	EXECUTE	Actions to be Executed Out of the Target
	RESULT	The Expected Result of an Assignment Step
	CAUTION	Procedure Which Can Result in Minor Equipment Damage
	WARNING	Procedure With Potential Equipment Damage
	DANGER	Procedure With Imminent Equipment Destruction

1 Overview

1.1 Host Driver Architecture

Figure 1-1. Host Driver Software Architecture



WILC host driver software is a C library which provides the host MCU application with necessary APIs to perform necessary WLAN and Ethernet operations. [Figure 1-1](#) shows the architecture of the WILC host driver software which runs on the host MCU. The components of the host driver are described in the following sub-sections.

1.1.1 WLAN API

This module provides an interface to the application for all Wi-Fi operations and any non-IP related operations. This includes the following services:

- Wi-Fi STA management operations
 - Wi-Fi Scan
 - Wi-Fi Connection management (Connect, Disconnect, Connection status, etc.)
 - WPS activation/deactivation
- Wi-Fi AP enable/disable
- Wi-Fi Direct enable/disable
- Wi-Fi power save control API
- Wi-Fi monitoring (Sniffer) mode

This interface is defined in the file: `m2m_wifi.h`.

1.1.2 Host Interface

The Host Interface (HIF) is responsible for handling the communication between the host driver and the WILC firmware. This includes interrupt handling, DMA, and HIF command/response management. The host driver communicates with the firmware in a form of commands and responses formatted by the HIF layer.

The interface is defined in the file: `m2m_hif.h`.

The detailed description of the HIF design is provided in [Chapter 10](#).

1.1.3 Board Support Package

The Board Support Package (BSP) abstracts the functionality of a specific host MCU platform. This allows the driver to be portable to a wide range of hardware and hosts. Abstraction includes: pin assignment, power on/off sequence, reset sequence and peripheral definitions (Push buttons, LEDs, etc.).

The minimum required BSP functionality is defined in the file: `nm_bsp.h`.

1.1.4 Serial Bus Interface

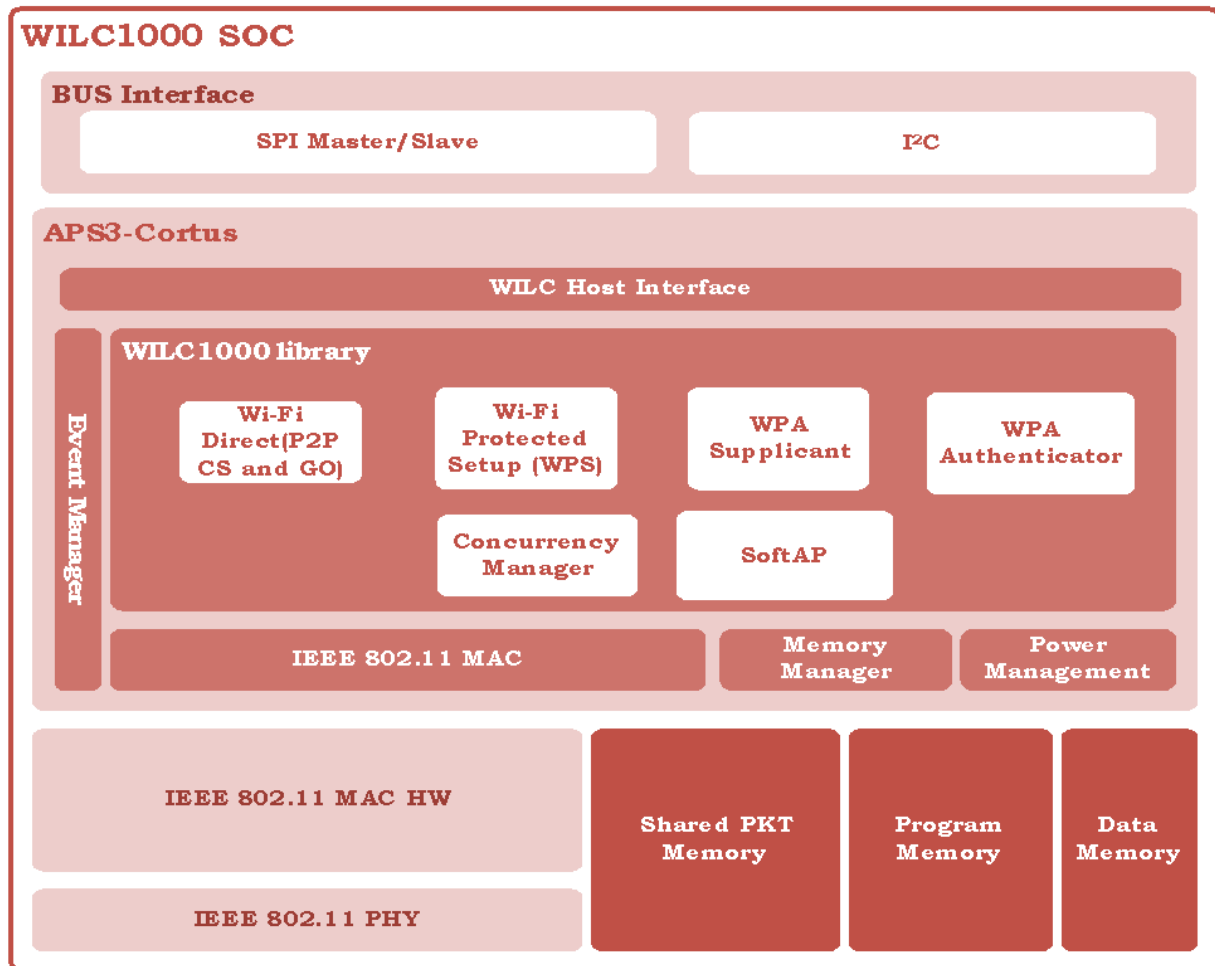
The Serial Bus Interface module abstracts the hardware associated with implementing the bus between the Host and the WILC. The serial bus interface abstracts I²C, SPI, or UART bus interface. The basic bus access operations (Read and Write) are implemented in this module as appropriate for the interface type and the specific hardware.

The bus interface APIs are defined in the file: `nm_bus_wrapper.h`.

1.2 WILC System Architecture

Figure 1-2 shows the WILC system architecture. In addition to its built-in Wi-Fi IEEE-802.11 physical layer and RF front end, the WILC ASIC has an embedded APS3S-Cortus 32-bit CPU to run the WILC firmware. The firmware comprises the Wi-Fi IEEE-802.11 MAC layer and embedded protocol stacks which offload the host MCU. The components of the system are described in the following sub-sections.

Figure 1-2. WILC System Architecture



1.2.2 Bus Interface

Hardware logic for the supported bus types for WILC communications.

1.2.3 CPU

The SoC contains an APS3S-Cortus 32-bit CPU running at 40MHz clock speed which executes the embedded WILC firmware.

1.2.4 IEEE 802.11 MAC Hardware

The SoC contains a hardware accelerator to ensure fast and compliant implementation of the IEEE 802.11 MAC layer and associated timing. It offloads IEEE 802.11 MAC functionality from firmware to improve performance and boost the MAC throughput. The accelerator includes hardware encryption/decryption of Wi-Fi traffic and traffic filtering mechanisms to avoid unnecessary processing in software.

1.2.5 Program Memory

128KB Instruction RAM is provided for execution of the WILC firmware code.

1.2.6 Data Memory

64KB Data RAM is provided for WILC firmware data storage.

1.2.7 Shared Packet Memory

128KB memory is provided for TX/RX packet management. It is shared between the MAC hardware and the CPU. This memory is managed by the Memory Manager SW component.

1.2.8 IEEE 802.11 MAC Firmware

The system supports IEEE 802.11 b/g/n Wi-Fi MAC including WEP and WPA/WPA2 security supplicant. Between the MAC hardware and firmware, a full range of IEEE 802.11 features are implemented and supported including beacon generation and reception, control packet generation and reception and packet aggregation and de-aggregation.

1.2.9 Memory Manager

The memory manager is responsible for the allocation and de-allocation of memory chunks in both shared packet memory and data memory.

1.2.10 Power Management

The Power Management module is responsible for handling different power saving modes supported by the WILC and coordinating these modes with the Wi-Fi transceiver.

EAP-TTLS/MSCHAPV2.0

This module implements the authentication protocol EAP-TTLS/MsChapv2.0 used for establishing a Wi-Fi connection with an AP by with WPA-Enterprise security.

WI-FI PROTECTED SETUP

For WPS protocol implementation, see Chapter 7: Wi-Fi Protected Setup for details.

WI-FI DIRECT

For Wi-Fi Direct protocol implementation, see Chapter 6: Wi-Fi Direct P2P Mode for details.

CRYPTO LIBRARY

The Crypto Library contains a set of cryptographic algorithms used by common security protocols. This library has an implementation of the following algorithms:

- **SHA-1** Hash algorithm
- **SHA-256** Hash algorithm
- **DES** Encryption (used only for MsChapv2.0 digest calculation)
- **MS-CHAPv2.0** (used as the EAP-TTLS inner authentication algorithm)
- **AES-128, AES-256** Encryption (used for securing WPS and TLS traffic)
- **BigInt** module for large integer arithmetic (for Public Key Cryptographic computations)
- **RSA** Public Key cryptography algorithms (includes RSA Signature and RSA Encryption algorithms)

2 WILC Initialization and Simple Application

After powering-up the WILC device, a set of synchronous initialization sequences must be executed, for the correct operation of the Wi-Fi functions. This chapter aims to explain the different steps required during the initialization phase of the system. After initialization, the host MCU application is required to call the WILC driver entry point to handle events from WILC firmware.

- BSP Initialization
- WILC Host Driver Initialization
- Call WILC driver entry point



IMPORTANT

Failure to complete any of the initializations steps will result in failure in WILC startup.

2.1 BSP Initialization

The BSP is initialized by calling the `nm_bsp_init` API. The BSP initialization routine performs the following steps:

- Resets the WILC (see the note below) using corresponding host MCU control GPIOs
- Initializes the host MCU GPIO which connects to WILC interrupt line. It configures the GPIO as an interrupt source to the host MCU. During runtime, WILC interrupts the host to notify the application of events and data pending inside WILC firmware.
- Initializes the host MCU delay function used within `nm_bsp_sleep` implementation

Note: Refer to the ATWILC1000 datasheet [R03] for more information about WILC hardware reset sequence.

2.2 WILC Host Driver Initialization

The WILC host driver is initialized by calling the `m2m_wifi_init` API. The Host driver initialization routine performs the following steps:

- Initializes the bus wrapper, I²C, SPI, or UART, depending on the host driver software bus interface configuration compilation flag `USE_I2C`, `USE_SPI`, or `USE_UART` respectively
- Registers an application-defined Wi-Fi event handler
- Initializes the driver and ensures that the current WILC firmware matches the current driver version
- Initializes the host interface and the Wi-Fi layer and registers the BSP Interrupt



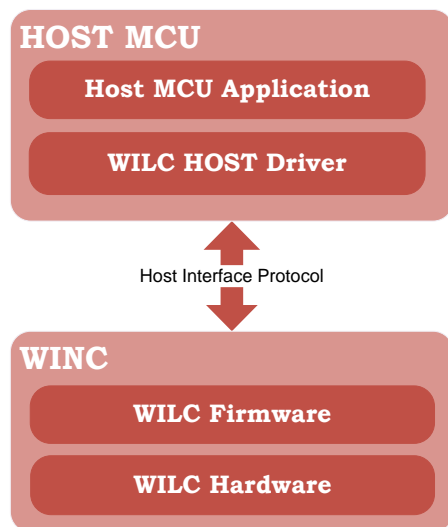
INFO

A Wi-Fi event handler is required for the correct operation of any WILC application.

2.3 WILC Event Handling

The WILC host driver API allows the host MCU application to interact with the WILC firmware. To facilitate interaction, the WILC driver implements the Host Interface (HIF) Protocol described in Chapter 10. The HIF protocol defines how to serialize and de-serializes API requests and response callbacks over the serial bus interface: I²C, UART, or SPI.

Figure 2-1. WILC System Architecture



WILC host driver API provides services to the host MCU applications that are mainly divided in two major categories: Wi-Fi control services and Ethernet interface services. The Wi-Fi control services allow actions such as channel scanning, network identification, connection and disconnection. Ethernet interface services allow application to transfer Ethernet frames once a Wi-Fi connection has been established.

2.3.2 Asynchronous Events

Some WILC host driver APIs are synchronous function calls, where the result is ready by the return of the function. However, most WILC host driver API functions are asynchronous. This means that when the application calls an API to request a service, the call is non-blocking and returns immediately, most often before the requested action is completed. When completed, a notification is provided in the form of a HIF protocol message from the WILC firmware to the host which, in turn, is delivered to the application via a callback (see the note below) function. Asynchronous operation is essential when the requested service such as Wi-Fi connection may take significant time to complete. In general, the WILC firmware uses asynchronous events to signal the host driver about status change or pending data.

The HIF uses “push” architecture, where data and events are pushed from WILC firmware to the host MCU in FCFS manner. For instance, suppose that WILC received two Ethernet packets, then HIF shall deliver the frame data in two HIF protocol messages in the order they were received. HIF does not allow reading packet 2 data before packet 1 data.

Note: The callback is C function which contains an application-defined logic. The callback is registered using the WILC host driver registration API to handle the result of the requested service.

2.3.3 Interrupt Handling

The HIF interrupts the host MCU when one or more events are pending in WILC firmware. The host MCU application is a big state machine which processes received data and events when WILC driver calls the event callback function(s). In order to receive event callbacks, the host MCU application is required to call the `m2m_wifi_handle_events` API to let the host driver retrieve and process the pending events from the WILC firmware. It's recommended to call this function either:

- Host MCU application polls the API in main loop or a dedicated task
- Or at least once when host MCU receives an interrupt from WILC firmware



INFO

All the application-defined event callback functions registered with WILC driver run in the context `m2m_wifi_handle_events` API.

The above HIF architecture allows WILC host driver to be flexible to run in the following configurations:

- Host MCU with no operating system configuration: In this configuration, the MCU main loop is responsible to handle deferred work from interrupt handler.
- Host MCU with operating system configuration: In this configuration, a dedicated task or thread is required to call `m2m_wifi_handle_events` to handle deferred work from interrupt handler.



INFO

Host driver entry point `m2m_wifi_handle_events` is non-reentrant. In the operating system configuration, it is required to protect the host driver from reentrance by a synchronization object.



TIP

When host MCU is polling `m2m_wifi_handle_events`, the API checks for pending unhandled interrupt from WILC. If no interrupt is pending, it returns immediately. If an interrupt is pending, `m2m_wifi_handle_events` reads all the pending the HIF message sequentially and dispatches the HIF message content to the respective registered callback. If a callback is not registered to handle the type of message, the HIF message content is discarded.

2.4 Code Example

The code example below shows the initialization flow as described in previous sections.

```
static void wifi_cb(uint8_t u8MsgType, void *pvMsg)
{
}

int main (void)
{
    tstrWifiInitParam param;
    nm_bsp_init();

    m2m_memset((uint8*)&param, 0, sizeof(param));
    param.pfAppWifiCb = wifi_cb;

    /*intilize the WILC Driver*/
    ret = m2m_wifi_init(&param);
    if (M2M_SUCCESS != ret){
        M2M_ERR("Driver Init Failed <%d>\n",ret);
        while(1);
    }

    while(1){
        /* Handle the app state machine plus the WILC event handler */
        while(m2m_wifi_handle_events(NULL) != M2M_SUCCESS) {
        }
    }
}
```

3 WILC Configuration

WILC firmware has a set of configurable parameters that control its behavior. There is a set of APIs provided to host MCU application to configure these parameters. The configuration APIs are categorized according to their functionality into: device, network, and power saving parameters.

Any parameters left unset by the host MCU application shall use their default values assigned during the initialization of the WILC firmware. A host MCU application needs to configure its parameters when coming out of cold boot or when a specific configuration change is required.

3.1 Device Parameters

3.1.1 Firmware and Driver Version

During startup, the host driver requests the firmware version through `nm_get_firmware_info` API which returns the structure `tstrM2mRev` containing the minimum supported driver version and the current WILC firmware version.



IMPORTANT

If the current driver version is less than the minimum driver version required by WILC firmware, the driver initialization will fail.

The version parameters provided are:

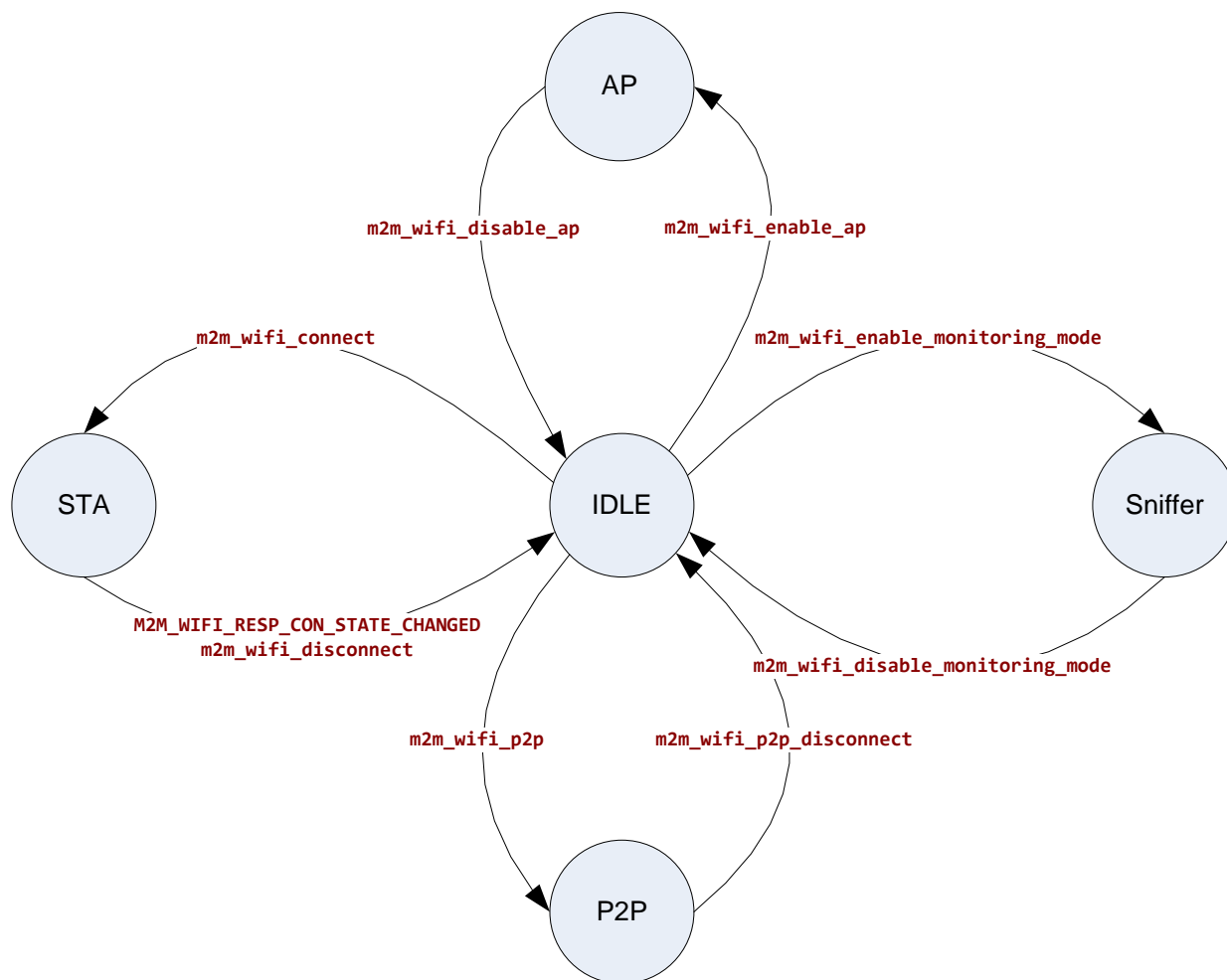
- `M2M_FIRMWARE_VERSION_MAJOR_NO`: Firmware Major release version number
- `M2M_FIRMWARE_VERSION_MINOR_NO`: Firmware Minor release version number
- `M2M_FIRMWARE_VERSION_PATCH_NO`: Firmware patch release version number
- `M2M_DRIVER_VERSION_MAJOR_NO`: Driver Major release version number
- `M2M_DRIVER_VERSION_MINOR_NO`: Driver Minor release version number
- `M2M_DRIVER_VERSION_PATCH_NO`: Driver patch release version number

3.2 WILC Modes of Operation

The WILC firmware supports the following modes of operation:

- Idle Mode
- Wi-Fi STA Mode
- Wi-Fi Direct (P2P)
- Wi-Fi Hotspot (AP)
- Sniffer mode (Monitoring mode)

Figure 3-1. WILC Modes of Operation



3.2.1 Idle Mode

After the host MCU application calls the WILC driver initialization `m2m_wifi_init` API, WILC remains in idle mode waiting for any command to change the mode or to update the configuration parameters. In this mode WILC will enter power save in which it disables the IEEE 802.11 radio and all unneeded peripherals and suspends the WILC CPU. If WILC receives any configuration commands from the host MCU, WILC will update the configuration, send back the response to the host MCU and then go back the power save mode.

3.2.2 Wi-Fi Station Mode

WILC enters station (STA) mode when the host MCU requests connection to an AP using the `m2m_wifi_connect` API. WILC exits STA mode when it receives a disconnect request from the Wi-Fi AP conveyed to the host MCU application via the event callback `M2M_WIFI_RESP_CON_STATE_CHANGED` or when the host MCU application decides to terminate the connection via `m2m_wifi_disconnect` API. WILC firmware ignores mode change requests while in this mode until WILC exits the mode.



TIP

The supported API functions in this mode use the HIF command types: `tenuM2mConfigCmd` and `tenuM2mStaCmd`. See the full list of commands in the header file `m2m_types.h`.

For more information about this mode, refer to Chapter 4: Wi-Fi Station Mode.

3.2.3 Wi-Fi Direct (P2P) Mode

In Wi-Fi direct mode, WILC can discover, negotiate and connect wirelessly to Wi-Fi Direct capable peer devices. To enter P2P mode, host MCU application calls `m2m_wifi_p2p` API. To exit P2P mode, the application calls `m2m_wifi_p2p_disconnect` API. WILC firmware ignores mode change requests while in this mode until WILC exits the mode.



TIP

The supported API functions in this mode use the HIF command types: `tenuM2mP2pCmd` and `tenuM2mConfigCmd`. See the full list in the header file `m2m_types.h`.

For more information about this mode, refer to Chapter 6: Wi-Fi Direct P2P Mode.

3.2.4 Wi-Fi Hotspot (AP) Mode

In AP mode, WILC allows Wi-Fi stations to connect to WILC. To enter AP mode, host MCU application calls `m2m_wifi_enable_ap` API. To exit AP mode, the application calls `m2m_wifi_disable_ap` API. WILC firmware ignores mode change requests while in this mode until WILC exits the mode.



TIP

The supported API functions in this mode use the HIF command types: `tenuM2mApCmd` and `tenuM2mConfigCmd`. See the full list of commands in the header file `m2m_types.h`.

For more information about this mode, refer to Chapter 5: Wi-Fi AP Mode:

3.3 Network Parameters

3.3.1 Device Name

The device name is used for the Wi-Fi Direct (P2P) mode only. Host MCU application can set the WILC P2P device name using the `m2m_wifi_set_device_name` API.



INFO

If no device name is provided, the default device name is the WILC MAC address in colon hexadecimal notation e.g. aa:bb:cc:dd:ee:ff.

3.3.2 Wi-Fi MAC Address

The WILC firmware provides two methods to assign the WILC MAC address:

- **Assignment from host MCU:** when host MCU application calls the `m2m_wifi_set_mac_address` API after initialization using `m2m_wifi_init` API
- **Assignment from WILC OTP (One Time Programmable) memory:** WILC supports an internal MAC address assignment method through a built-in OTP memory. If MAC address is programmed in the WILC OTP memory, the WILC working MAC address defaults to the OTP MAC address unless the host MCU application sets a different MAC address programmatically after initialization using the API `m2m_wifi_set_mac_address`.



INFO

OTP MAC address is programmed in WILC OTP memory at manufacturing time.

For more details, refer to description of the following APIs in Section/Chapter 0.

API Reference:

- `m2m_wifi_get_otp_mac_address`
- `m2m_wifi_set_mac_address`
- `m2m_wifi_get_mac_address`



TIP

Use `m2m_wifi_get_otp_mac_address` API to check if there is a valid programmed MAC address in WILC OTP memory. The host MCU application can also use the same API to read the OTP MAC address octets. `m2m_wifi_get_otp_mac_address` API not to be confused with the `m2m_wifi_get_mac_address` API which reads the *working WILC MAC address* in WILC firmware regardless from whether it is assigned from the host MCU or from WILC OTP.

3.4 Power Saving Parameters

When a Wi-Fi station is idle, it disables the Wi-Fi radio and enters power saving mode. The AP is required to buffer data while stations are in power save mode and transmit data later when stations wake up. The AP transmits a beacon frame periodically to synchronize the network every *beacon period*. A station which is in power save wakes up periodically to receive the beacon and monitor the signaling information included in the beacon. The beacon conveys information to the station about unicast data which belong to the station and currently buffered inside the AP while the station was sleeping. The beacon also provides information to the station when the AP is going to send broadcast/multicast data.

3.4.1 Power Saving Modes

WILC firmware supports multiple power saving modes which provide flexibility to the host MCU application to tweak the system power consumption. The host MCU can configure the WILC power saving policy using the `m2m_wifi_set_sleep_mode` and `m2m_wifi_set_lsn_int` APIs. WILC supports the following power saving modes:

- `M2M_PS_MANUAL`
- `M2M_PS_AUTOMATIC`
- `M2M_PS_H_AUTOMATIC`
- `M2M_PS_DEEP_AUTOMATIC`



TIP

`M2M_PS_DEEP_AUTOMATIC` mode recommended for most applications.

3.4.1.1 `M2M_PS_MANUAL`

This is a fully host-driven power saving mode.

- WILC sleeps when the host instructs it to do so using the `m2m_wifi_request_sleep` API. During WILC sleep, the host MCU can decide to sleep also for extended durations.
- WILC wakes up when the host MCU application requests services from WILC by calling any host driver API function, e.g., Wi-Fi or data operation



IMPORTANT

In `M2M_PS_MANUAL` mode, when WILC sleeps due to `m2m_wifi_request_sleep` API. WILC does not wakeup to receive and monitor AP beacon. Beacon monitoring is resumed when host MCU application wakes up the WILC.

For an active Wi-Fi connection, the AP may decide to drop the connection if WILC is absent because it sleeps for long time duration. If connection is dropped, WILC detects the disconnection on the next wake up cycle and notifies the host to reconnect to the AP again. In order to maintain an active Wi-Fi connection for extended durations, the host MCU application should wake up the WILC periodically so that WILC can send a keep-alive Wi-Fi frame to the AP. The host should choose the sleep period carefully to satisfy the tradeoff between keeping the Wi-Fi connection uninterrupted and minimizing the system power consumption.

This mode is useful for applications which send notifications very rarely due to a certain trigger. It fits also applications which send notifications periodically with a very long spacing between notifications. Careful power planning is required when using this mode. If the host MCU decides to sleep for very long period, it may use `M2M_PS_MANUAL` or may power off WILC (see the note below) completely. The advantage of this mode compared to powering off WILC is that `M2M_PS_MANUAL` saves the time required for WILC firmware to boot since the firmware is always loaded in WILC memory. The real pros and cons depend on the nature of the application. In some applications, the sleep duration could be long enough to be a power-efficient decision to power off WILC and power it on again and reconnect to the AP when host MCU wakes up. In other situations, a latency-sensitive application may choose to use `M2M_PS_MANUAL` to avoid WILC firmware boot latency on the expense of slightly increased power consumption.

During WILC sleep, WILC in `M2M_PS_MANUAL` mode saves more power than `M2M_PS_DEEP_AUTOMATIC` mode since in the former mode WILC skips beacon monitoring while the latter it wakes up to receive beacons. The comparison should also include the effect of host MCU sleep duration: if host MCU sleep period is too large, the Wi-Fi connection may drop frequently and the power advantage of `M2M_PS_MANUAL` is lost due to the power consumed in Wi-Fi reconnection. In contrast,

`M2M_PS_DEEP_AUTOMATIC` can keep the Wi-Fi connection for long durations at the expense of waking up WILC to monitor the AP beacon.

Note: Refer to WILC datasheet in [R03] for hardware power off sequence.

3.4.1.2 M2M_PS_AUTOMATIC

This mode is deprecated and kept for backward compatibility and development reasons. It should not be used in new implementations.

3.4.1.3 M2M_PS_H_AUTOMATIC

This mode implements the Wi-Fi standard power saving method in which WILC will sleep and wakeup periodically to monitor AP beacons. In contrast to `M2M_PS_MANUAL`, this mode does not involve the host MCU application.

In this mode, when WILC enters sleep state, it only turns off the IEEE 802.11 radio, MAC and PHY. All system clocks and the APS3S-Cortus CPU are on.

This mode is useful for a low-latency packet transmission because WILC clocks are on and ready to transmit packets immediately unlike the `M2M_PS_DEEP_AUTOMATIC` which may require time to wake up the WILC to transmit a packet if WILC was sleep mode.

`M2M_PS_H_AUTOMATIC` mode is very similar to `M2M_PS_DEEP_AUTOMATIC` except that the former power consumption is higher than the latter since WILC system clock is on.

3.4.1.4 M2M_PS_DEEP_AUTOMATIC

Like `M2M_PS_HS_AUTOMATIC`, this mode implements the Wi-Fi standard power saving method. However, when WILC enters sleep state, system clock is turned off.

Before sleep, the WILC programs a hardware timer (running on an internal low-power oscillator) with a sleep period determined by the WILC firmware power management module.

While sleeping, the WILC will wake up if one of the following events happens:

- Expiry of the hardware sleep timer. WILC wakes up to receive the upcoming beacon from AP.
- WILC wakes up (see the note below) when the host MCU application requests services from WILC by calling any host driver API function, e.g. Wi-Fi or data operation.

Note: The wakeup sequence is handled internally in the WILC host driver in the `hif_chip_wake` API. Refer to the reference Chapter 15 for more information.

3.4.2 Configuring Listen Interval and DTIM Monitoring

WILC allows the host MCU application to tweak system power consumption by configuring beacon monitoring parameters. The AP sends beacons periodically every beacon period (e.g. 100ms). The beacon contains a *TIM element* which informs the station about presence of unicast data for the station buffer in the AP. The station negotiates with the AP a *listen interval* which is how many beacon periods the station can sleep before it wakes up to receive data buffer in AP. The AP beacon also contains the *DTIM* which contains information to the station about the presence of broadcast/multicast data. Which the AP is ready to transmit following this beacon after normal channel access rules (CSMA/CA).

The WILC driver allows the host MCU application to configure beacon monitoring parameters as follows:

- **Configure DTIM monitoring:** I.e. enable or disable reception of broadcast/multicast data using the API:
 - `m2m_wifi_set_sleep_mode(desired_mode, 1)` to receive broadcast data
 - `m2m_wifi_set_sleep_mode(desired_mode, 0)` to ignore broadcast data
- **Configure the listen interval:** Using the `m2m_wifi_set_lsn_int` API

**TIP**

Listen interval value provided to the `m2m_wifi_set_lsn_int` API is expressed in the unit of beacon period.

4 Wi-Fi Station Mode

This chapter provides information about WILC Wi-Fi station (STA) mode described in section 3.2.2: Wi-Fi Station Mode. Wi-Fi station mode involves scan operation; association to an AP using parameters (SSID and credentials) provided by host MCU or using AP parameters stored in WILC non-volatile storage (default connection). The chapter also provides information about supported security modes along with code examples.

4.1 Scan Configuration Parameters

4.1.1 Scan Region

The number of RF channels supported varies by geographical region. For example, 14 channels are supported in Asia while 11 channels are supported in North America. By default the WILC initial region configuration is equal to 14 channels (Asia), but this can be changed by setting the scan region using: the `m2m_wifi_set_scan_region` API.

4.1.2 Scan Options

During Wi-Fi scan operation, WILC sends probe request Wi-Fi frames and waits for some time on the current Wi-Fi channel to receive probe response frames from nearby APs before it switches to the next channel. Increasing the scan wait time has a positive effect on the number of access points detected during scan. However, it has a negative effect on the power consumption and overall scan duration. WILC firmware default scan wait time is optimized to provide the tradeoff between power consumption and scan accuracy. WILC firmware provides flexible configuration options to the host MCU application to increase the scan time. For more detail, refer to the `m2m_wifi_set_scan_options` API.

4.2 Wi-Fi Scan

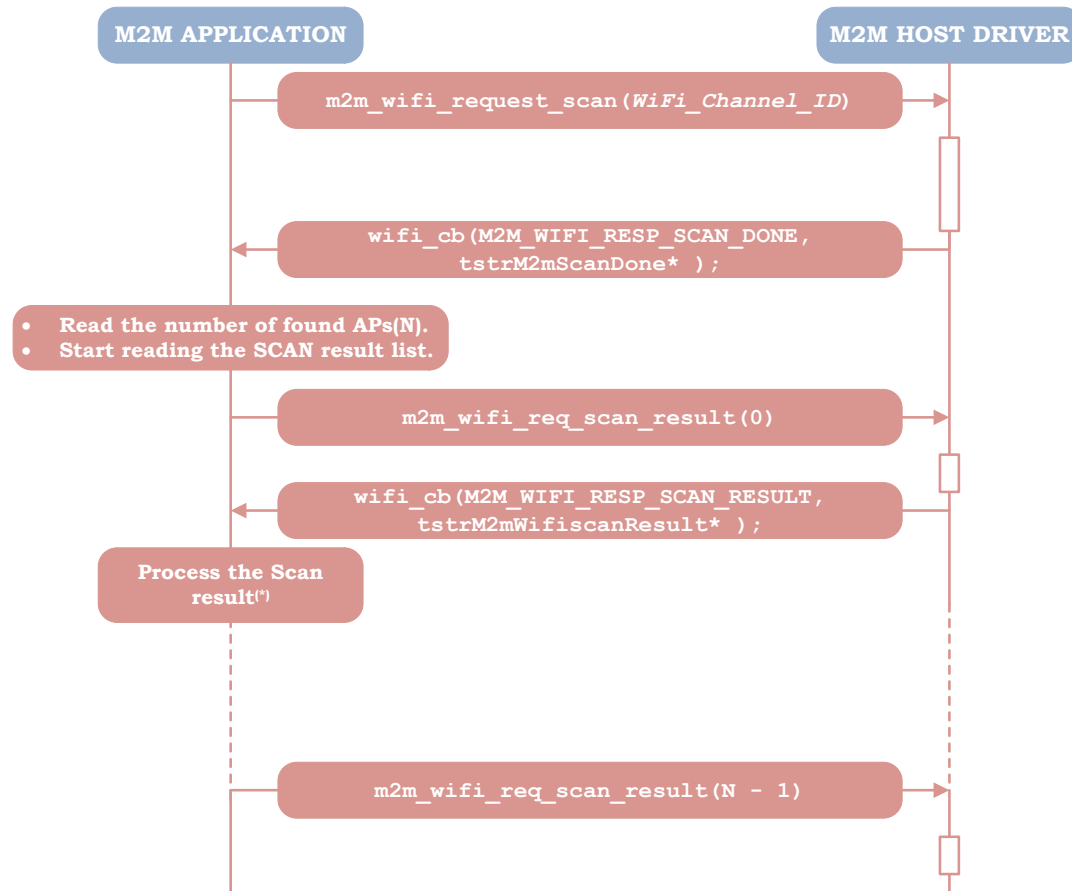
A Wi-Fi scan operation can be initiated by calling the `m2m_wifi_request_scan` API. The scan can be performed on all 2.4GHz Wi-Fi channels or on a specific requested channel.

The scan response time depends on the scan options. For instance, if the host MCU application requests to scan all channels, the scan time will be equal to `NoOfChannels (14) *`

`M2M_SCAN_MIN_NUM_SLOTS* M2M_SCAN_MIN_SLOT_TIME` (refer to the 0 for how to customize the scan parameters).

The scan operation is illustrated in [Figure 4-1](#).

Figure 4-1. Wi-Fi Scan Operation



4.3 On Demand Wi-Fi Connection

The host MCU application may establish a Wi-Fi connection on demand if all the required connection parameters (SSID, security credentials, etc.) are known to the application. To start a Wi-Fi connection on demand, the application shall call the API `m2m_wifi_connect`.

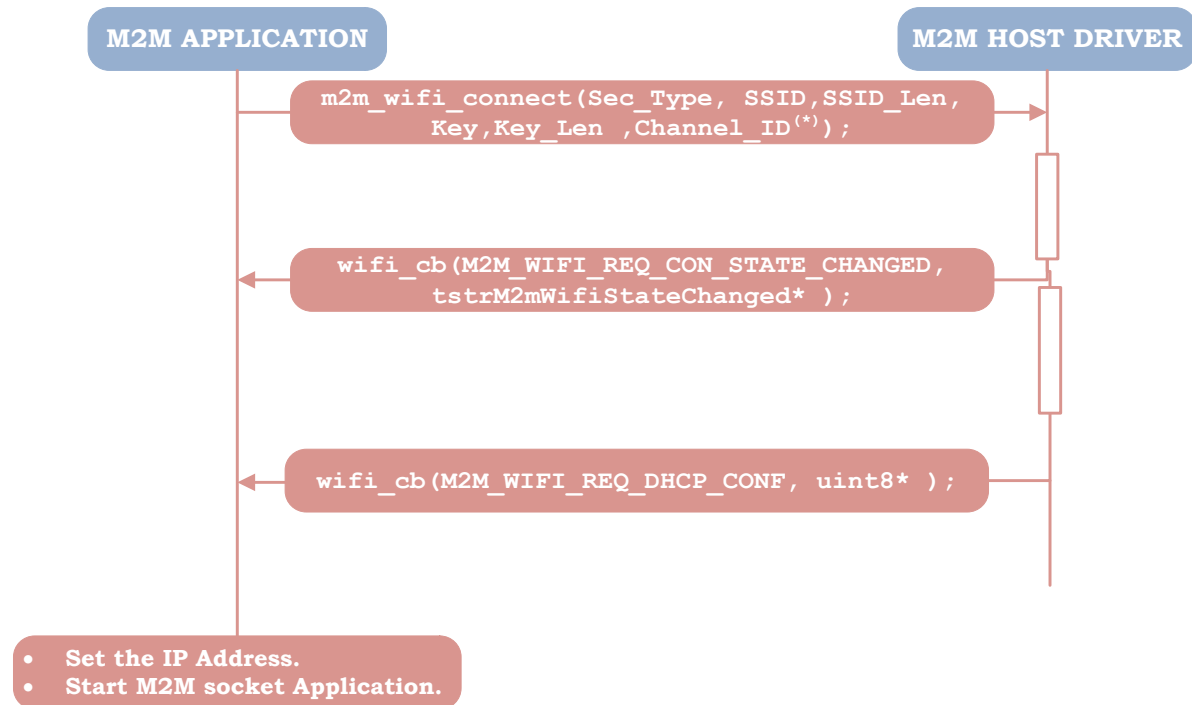


TIP

Using `m2m_wifi_connect` implies that the host MCU application has prior knowledge of the connection parameters. For instance, connection parameters can be stored on non-volatile storage attached to the host MCU.

The Wi-Fi on demand connection operation is described in [Figure 4-2](#).

Figure 4-2. On Demand Wi-Fi Connector



4.4 Wi-Fi Security

The following types of security are supported in WILC Wi-Fi STA mode.

- M2M_WIFI_SEC_OPEN
- M2M_WIFI_SEC_WEP
- M2M_WIFI_SEC_WPA_PSK (WPA/WPA2-Personal Security Mode i.e. Passphrase)
- M2M_WIFI_SEC_802_1X (WPA-Enterprise security)



INFO

The currently supported 802.1x authentication algorithm is EAP-TTLS with MsChapv2.0 authentication.

4.5 Example Code

```

#define M2M_802_1X_USR_NAME      "user_name"
#define M2M_802_1X_PWD          "password"
#define AUTH_CREDENTIALS        {M2M_802_1X_USR_NAME, M2M_802_1X_PWD }

int main (void)
{
    tstrWifiInitParam param;
    tstr1xAuthCredentials gstrCred1x = AUTH_CREDENTIALS;

    nm_bsp_init();

    m2m_memset((uint8*)&param, 0, sizeof(param));
    param.pfAppWifiCb = wifi_event_cb;

    /* intilize the WILC Driver
    */
    ret = m2m_wifi_init(&param);
}
    
```

```

if (M2M_SUCCESS != ret)
{
    M2M_ERR("Driver Init Failed <rd>\n",ret);
    while(1);
}

/* Connect to a WPA-Enterprise AP
*/
m2m_wifi_connect("DEMO_AP", sizeof("DEMO_AP"), M2M_WIFI_SEC_802_1X,
    (uint8*)&gstrCred1x, M2M_WIFI_CH_ALL);

while(1)
{
    /******
    /* Handle the app state machine plus the WILC event handler      */
    /******
    while(m2m_wifi_handle_events(NULL) != M2M_SUCCESS)
    {
    }
}
}

```

5 Wi-Fi AP Mode

5.1 Overview

This chapter provides an overview of WILC Access Point (AP) mode and describes how to setup this mode and configure its parameters.

5.2 Setting WILC AP Mode

WILC AP mode configuration parameters should be set first using `tstrM2MAPConfig` structure.

There are two functions to enable/disable AP mode.

- `sint8 m2m_wifi_enable_ap(CONST tstrM2MAPConfig* pstrM2MAPConfig)`
- `sint8 m2m_wifi_disable_ap(void);`

For more information about structure and APIs, refer to the API reference in 0.

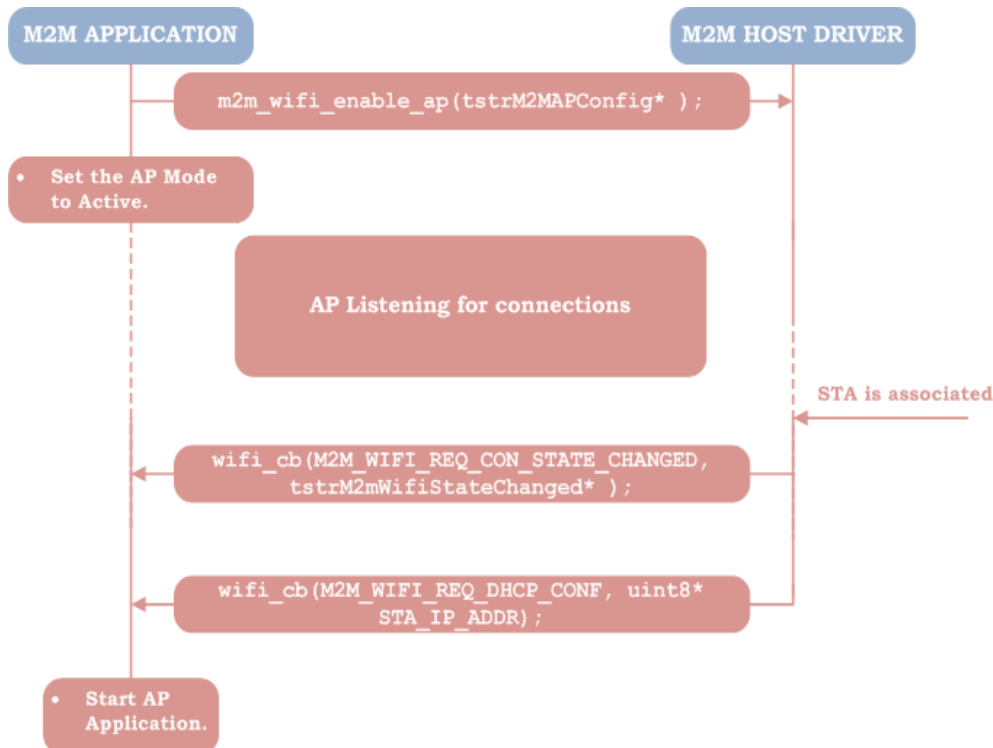
5.3 Capabilities

- The AP supports up to 8 associated stations and up to 7 in case of concurrency (see Section Station-AP Concurrency).
- Supports all modes of security (Open, WEP, and WPA/WPA2)
- Can be started concurrently with a station interface (see section Station-AP Concurrency).

5.4 Sequence Diagram

Once the AP mode has been established, no data interface exists until after a station associates to the AP. Therefore the application needs to wait until it receives a notification via an event callback. This process is shown in [Figure 5-1](#).

Figure 5-1. WILC AP Mode Establishment



5.5 AP Mode Code Example

The following example shows how to configure WILC AP Mode with “WILC_SSID” as broadcasted SSID on channel one with open security and an IP address equals 192.168.1.1.

```
#include "m2m_wifi.h"
#include "m2m_types.h"
void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    switch(u8WiFiEvent)
    {
        break;
        default:
        break;
    }
}
int main()
{
    tstrWifiInitParam param;

    /* Platform specific initializations. */

    param.pfAppWifiCb = wifi_event_cb;
    if (!m2m_wifi_init(&param))
    {
        tstrM2MAPConfig apConfig;
        strcpy(apConfig.au8SSID, "WILC_SSID");           // Set SSID
        apConfig.u8SsidHide = SSID_MODE_VISIBLE;        // Set SSID to be broadcasted
        apConfig.u8ListenChannel = 1;                   // Set Channel

        apConfig.u8SecType = M2M_WIFI_SEC_WEP;          // Set Security to WEP
        apConfig.u8KeyIndx = 0;                          // Set WEP Key Index
        apConfig.u8KeySz = WEP_40_KEY_STRING_SIZE;      // Set WEP Key Size
        strcpy(apConfig.au8WepKey, "1234567890");        // Set WEP Key
        // Start AP mode
        m2m_wifi_enable_ap(&apConfig);
        while(1)
        {
            m2m_wifi_handle_events(NULL);
        }
    }
}
```

6 Wi-Fi Direct P2P Mode

6.1 Overview

Wi-Fi Direct or “Peer to Peer” (P2P) allows two wireless devices to discover each other, negotiate on which device will act as a group owner, form a group including WPS key generation and make a connection. The WILC supports a subset of this functionality that allows the WILC firmware to connect to other P2P capable devices that are prepared to become the group owner.

6.2 WILC P2P Capabilities

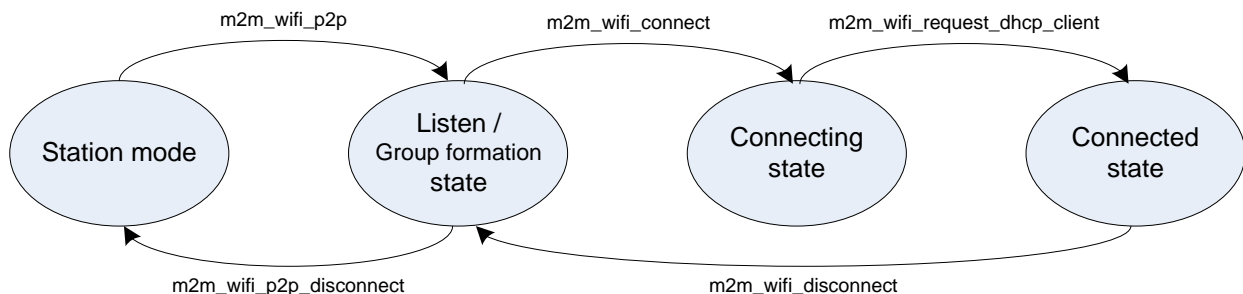
- P2P client mode is supported
- P2P device discovery
- P2P listen state

6.3 WILC P2P Limitations

- GO mode is not supported (P2P negotiation with GO intent set to 1)
- No support for GO-NOA Notice-Of-Absence
- Power save is disabled during P2P mode
- WILC cannot initiate the P2P connection; the other device must be the initiator

6.4 WILC P2P States

Figure 6-1. P2P Mode State Diagram



WILC P2P device can be in any of the above mentioned states based on the function call executed; a brief of each of these states will be explained in the following sections.

6.5 WILC P2P Listen State

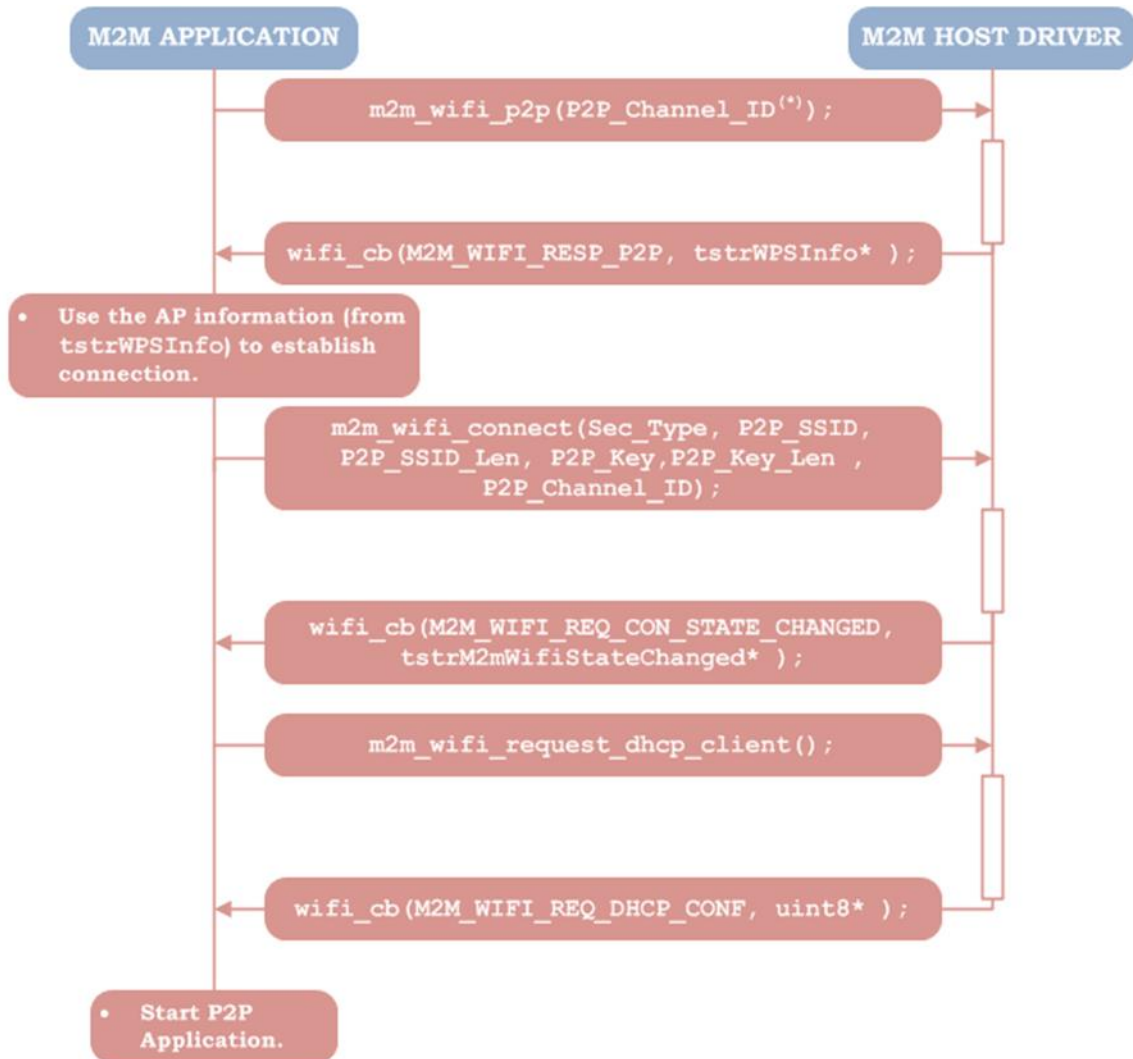
The WILC device becomes discoverable to other P2P devices on a predefined listen channel, ready to accept any connection initiations. To enter the listen state, the user must call the `m2m_wifi_p2p` function to set the WILC firmware in the listening state at a certain listen channel defined through the `MAIN_WLAN_CHANNEL`.

6.6 WILC P2P Connection State

The peer P2P device will initiate group owner (GO) negotiation and the WILC device will always decline to become group owner. Assuming the peer device **will take the GO role**, the WILC will then perform a WPS exchange to establish a mutual shared key. The information about the remote device (which is now acting as an AP), is received by an event via the Wi-Fi callback with the P2P GO information. The Application can then use this information to connect to the GO in the same manner that the WILC

connects to any conventional AP (using the `m2m_wifi_connect` function). The following sequence diagram shows the above connection flow for the WILC P2P device:

Figure 6-2. P2P Connection Flow



6.7 WILC P2P Disconnection State

To terminate the P2P connection, the GO can send a disconnection that is received through the Wi-Fi callback with the event `M2M_WIFI_RESP_CON_STATE_CHANGED`. However, this will not change the P2P listen state, unless a P2P disable request is made.

6.8 P2P Mode Code Example

```
#include "driver/include/m2m_wifi.h"
#include "driver/source/nmasic.h"

#define MAIN_WLAN_DEVICE_NAME    "WILC1000_P2P" /* < P2P Device Name */
#define MAIN_WLAN_CHANNEL        (6) /* < Channel number */

static void wifi_cb(uint8_t u8MsgType, void *pvMsg)
{
    switch (u8MsgType)
    {
        case M2M_WIFI_RESP_CON_STATE_CHANGED:
        {
            tstrM2mWifiStateChanged *pstrWifiState = (tstrM2mWifiStateChanged *)pvMsg;
            if (pstrWifiState->u8CurrState == M2M_WIFI_CONNECTED) {
                printf("Wi-Fi P2P connected\r\n");
            } else if (pstrWifiState->u8CurrState == M2M_WIFI_DISCONNECTED) {
                printf("Wi-Fi disconnected\r\n");
            }
            break;
        }
        default:
        {
            break;
        }
    }
}

int main(void)
{
    tstrWifiInitParam param;
    int8_t ret;

    // Initialize the BSP.
    nm_bsp_init();

    // Initialize Wi-Fi parameters structure.
    memset((uint8_t *)&param, 0, sizeof(tstrWifiInitParam));

    // Initialize Wi-Fi driver with data and status callbacks.
    param.pfAppWifiCb = wifi_cb;
    ret = m2m_wifi_init(&param);
    if (M2M_SUCCESS != ret) {
        printf("main: m2m_wifi_init call error!(%d)\r\n", ret);
        while (1) {
        }
    }

    // Set device name to be shown in peer device.
    ret = m2m_wifi_set_device_name((uint8_t *)MAIN_WLAN_DEVICE_NAME,
                                   strlen(MAIN_WLAN_DEVICE_NAME));

    if (M2M_SUCCESS != ret) {
        printf("main: m2m_wifi_set_device_name call error!\r\n");
        while (1) {
        }
    }
}
```

```

    }
}

// Bring up P2P mode with channel number.
ret = m2m_wifi_p2p(MAIN_WLAN_CHANNEL);
if (M2M_SUCCESS != ret) {
    printf("main: m2m_wifi_p2p call error!\r\n");
    while (1) {
    }
}

printf("P2P mode started. You can connect to %s.\r\n", (char *)MAIN_WLAN_DEVICE_NAME);
while (1) {
    /* Handle pending events from network controller. */
    while (m2m_wifi_handle_events(NULL) != M2M_SUCCESS) {
    }
}
return 0;
}
}

```

7 Wi-Fi Protected Setup

Most modern Access Points support Wi-Fi Protected Setup (WPS) method, typically using the push button method. From the user's perspective WPS is a simple mechanism to make a device connect securely to an AP without remembering passwords or passphrases. WPS uses asymmetric cryptography to form a temporary secure link which is then used to transfer a passphrase (and other information) from the AP to the new station. After the transfer, secure connections are made as for normal static PSK configuration.

7.1.1 WPS Configuration Methods

There are two authentication methods that can be used with WPS:

1. **PBC (Push button) method:** a physical button is pressed on the AP which puts the AP into WPS mode for a limited period of time. WPS is initiated on the ATWILC1000 by calling `m2m_wifi_wps` with input parameter `WPS_PBC_TRIGGER`.
2. **PIN method:** The AP is always available for WPS initiation but requires proof that the user has knowledge of an 8-digit PIN, usually printed on the body of the AP. Because WILC is often used in "headless" devices (no user interface) it is necessary to reverse this process and force the AP to use a PIN number provided with the WILC device. Some APs allow the PIN to be changed through configuration. WPS is initiated on the ATWILC1000 by calling `m2m_wifi_wps` with input parameter `WPS_PIN_TRIGGER`. Given the difficulty of this approach it is not recommended for most applications.

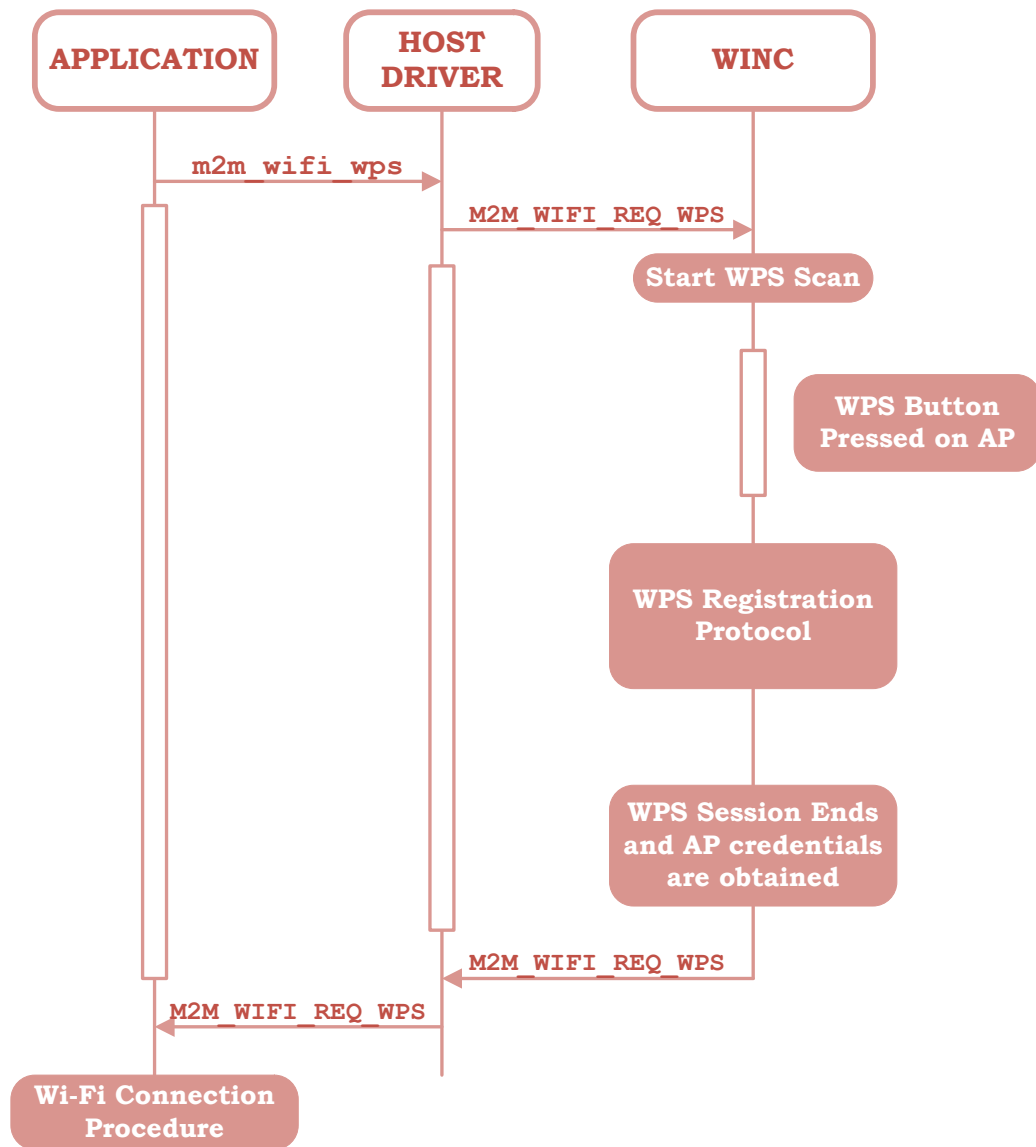
The flow of messages and actions for WPS operation is shown in [Figure 7-1](#).

7.1.2 WPS Limitations

- WPS is used to transfer the WPA/WPA2 key only; other security types are not supported
- The WPS standard will reject the session (WPS response fail) if the WPS button is pressed on more than one AP in the same proximity, and the application should try after a couple of minutes
- If no WPS button is pressed on the AP, the WPS scan will timeout after two minutes since the initial WPS trigger
- The WPS is responsible to deliver the connection parameters to the application, the connection procedure and the connection parameters validity is the application responsibility

7.1.3 WPS Control Flow

Figure 7-1. WPS Operation for Push Button Trigger



7.1.4 WPS Code Example

```
void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    if(u8WiFiEvent == M2M_WIFI_REQ_WPS)
    {
        tstrM2MWPSInfo *pstrWPS = (tstrM2MWPSInfo*)pvMsg;
        if(pstrWPS->u8AuthType != 0)
        {
            printf("WPS SSID          : %s\n",pstrWPS->au8SSID);
            printf("WPS PSK           : %s\n",pstrWPS->au8PSK);
            printf("WPS SSID Auth Type : %s\n",
                pstrWPS->u8AuthType == M2M_WIFI_SEC_OPEN ? "OPEN" : "WPA/WPA2");
            printf("WPS Channel        : %d\n",pstrWPS->u8Ch + 1);

            // Establish Wi-Fi connection
            m2m_wifi_connect((char*)pstrWPS->au8SSID, (uint8)m2m_strlen(pstrWPS->au8SSID),
                pstrWPS->u8AuthType, pstrWPS->au8PSK, pstrWPS->u8Ch);
        }
        else
        {
            printf("(ERR) WPS Is not enabled OR Timedout\n");
        }
    }
}

int main()
{
    tstrWifiInitParam param;

    // Platform specific initializations.

    // Driver initialization.
    param.pfAppWifiCb = wifi_event_cb;
    if(!m2m_wifi_init(&param))
    {
        // Trigger WPS in Push button mode.
        m2m_wifi_wps(WPS_PBC_TRIGGER, NULL);

        while(1)
        {
            m2m_wifi_handle_events(NULL);
        }
    }
}
```


8 Concurrency

ATWILC1000 firmware supports different modes of concurrent operations as follows:

- Station-Station
- Station-AP
- Station-P2P client

8.1 Limitations

- Single channel concurrency, this means the two logical interfaces should operate on the same channel
- Single MAC Address, on current HW revision, the two interfaces will share the same MAC address, this limitation shouldn't make a conflict on the AIR because the two interfaces would always work on different basic service set (BSS)

8.2 Controlling Second Interface

Second interface is controlled with the same APIs that is control the first interface however the driver first needs to set which interface is currently under control using the API "sint8 m2m_wifi_set_control_ifc(uint8 u8IfcId)", this API takes two values to the u8IfcId either 1 to control the first interface and this is the default value or 2 to control the second interface.

If the function "m2m_wifi_set_control_ifc" is never called, all the control functions would go for the first interface by default.

8.3 Station-Station Concurrency

In this mode of concurrency, driver would be able to connect to two different APs operating in any security modes but should be operating on the same channel.



TIP

It is recommended to wait for the first connection status before trying to connect on the second interface as the next example does.

Below is a code example to connect to two different Aps:

```
#define DEMO_WLAN_SSID           "Demo_AP"
#define DEMO_WLAN_AUTH           M2M_WIFI_SEC_WPA_PSK
#define DEMO_WLAN_PSK            "1234567890"

#define DEMO_WLAN_SSID_1         "Demo_AP_1"
#define DEMO_WLAN_AUTH_1         M2M_WIFI_SEC_WPA_PSK
#define DEMO_WLAN_PSK_1          "1234567890"

static void wifi_cb(uint8_t u8MsgType, void *pvMsg)
{
    switch (u8MsgType)
    {
        case M2M_WIFI_RESP_CON_STATE_CHANGED:
        {
            static int interfaceNo = 1;
            tstrM2mWifiStateChanged *pstrWifiState = (tstrM2mWifiStateChanged *)pvMsg;
            if (pstrWifiState->u8CurrState == M2M_WIFI_CONNECTED) {
                printf("Wi-Fi interface [%d] connected\r\n", interfaceNo);
            } else if (pstrWifiState->u8CurrState == M2M_WIFI_DISCONNECTED) {
                printf("Wi-Fi interface [%d] disconnected\r\n", interfaceNo);
            }
            if(interfaceNo == 1)
            {
                printf("Trying to connect on interface 2\r\n");
                m2m_wifi_set_control_ifc(2);
            }
        }
    }
}
```

```

        ret = m2m_wifi_connect((char *)DEMO_WLAN_SSID_1,
sizeof(DEMO_WLAN_SSID_1),
        DEMO_WLAN_AUTH_1, (char *)DEMO_WLAN_PSK, M2M_WIFI_CH_ALL)
        if (M2M_SUCCESS != ret) {
            printf("main: m2m_wifi_p2p call error!\r\n");
            while (1) {
            }
        }
        interfaceNo = 2;
    }
    break;
}
default:
{
    break;
}
}
}

int main(void)
{
    tstrWifiInitParam param;
    int8_t ret;

    // Initialize the BSP.
    nm_bsp_init();

    // Initialize Wi-Fi parameters structure.
    memset((uint8_t *)&param, 0, sizeof(tstrWifiInitParam));

    // Initialize Wi-Fi driver with data and status callbacks.
    param.pfAppWifiCb = wifi_cb;
    ret = m2m_wifi_init(&param);
    if (M2M_SUCCESS != ret) {
        printf("main: m2m_wifi_init call error!(%d)\r\n", ret);
        while (1) {
        }
    }
    printf("Trying to connect on interface 1\r\n");
    // connect to the first AP on interface 1
    ret = m2m_wifi_connect((char *)DEMO_WLAN_SSID, sizeof(DEMO_WLAN_SSID),
DEMO_WLAN_AUTH, (char *)DEMO_WLAN_PSK, M2M_WIFI_CH_ALL)
    if (M2M_SUCCESS != ret) {
        printf("main: m2m_wifi_p2p call error!\r\n");
        while (1) {
        }
    }
    while (1) {
        /* Handle pending events from network controller. */
        while (m2m_wifi_handle_events(NULL) != M2M_SUCCESS) {
        }
    }
    return 0;
}

```

8.3.1 Concurrent WPS

WPS session could be started on both interfaces to get the connection parameters as described in 7: Wi-Fi Protected Setup, but it is not allowed to the sessions to overlap, the first session should be completed either be success or failure before trying to start a new session on the second interface.

```

void wifi_event_cb(uint8 u8WifiEvent, void * pVMsg)
{
    static int interfaceNo = 1;
    if(u8WifiEvent == M2M_WIFI_REQ_WPS)
    {
        tstrM2MWPSInfo *pstrWPS = (tstrM2MWPSInfo*)pVMsg;
        if(pstrWPS->u8AuthType != 0)
    }
}

```

```

        {
            printf("WPS SSID          : %s\n", pstrWPS->au8SSID);
            printf("WPS PSK           : %s\n", pstrWPS->au8PSK);
            printf("WPS SSID Auth Type : %s\n",
                pstrWPS->u8AuthType == M2M_WIFI_SEC_OPEN ? "OPEN" : "WPA/WPA2");
            printf("WPS Channel      : %d\n", pstrWPS->u8Ch + 1);
            //set the control interface
            m2m_wifi_set_control_ifc(interfaceNo);
            // Establish Wi-Fi connection
            m2m_wifi_connect((char*)pstrWPS->au8SSID, (uint8)m2m_strlen(pstrWPS->au8SSID),
                pstrWPS->u8AuthType, pstrWPS->au8PSK, pstrWPS->u8Ch);
        }
        else
        {
            printf("(ERR) WPS Is not enabled OR Timedout\n");
        }
    }
    else if(u8WifiEvent == M2M_WIFI_RESP_CON_STATE_CHANGED)
    {
        tstrM2mWifiStateChanged *pstrWifiState = (tstrM2mWifiStateChanged *)pvMsg;
        if (pstrWifiState->u8CurrState == M2M_WIFI_CONNECTED) {
            printf("Wi-Fi interface [%d] connected\r\n", interfaceNo);
        } else if (pstrWifiState->u8CurrState == M2M_WIFI_DISCONNECTED) {
            printf("Wi-Fi interface [%d] disconnected\r\n", interfaceNo);
        }
        if(interfaceNo == 1)
        {
            printf("Start WPS on interface 2\r\n");
            m2m_wifi_set_control_ifc(2);
            // Trigger WPS in Push button mode.
            m2m_wifi_wps(WPS_PBC_TRIGGER, NULL);
            interfaceNo = 2;
        }
        break;
    }
}

int main()
{
    tstrWifiInitParam    param;

    // Platform specific initializations.

    // Driver initialization.
    param.pfAppWifiCb     = wifi_event_cb;
    if(!m2m_wifi_init(&param))
    {
        // Trigger WPS in Push button mode.
        m2m_wifi_wps(WPS_PBC_TRIGGER, NULL);

        while(1)
        {
            m2m_wifi_handle_events(NULL);
        }
    }
}

```

8.4 Station-AP Concurrency

In this mode of concurrency, driver would be able to connect to one AP using one interface and start an AP on the second interface regardless which happens first, keeping in mind the following facts and limitations:

1. AP should be started on the second interface regardless of the station interface would connect after or before the AP start.

2. If the AP started first then the station interface connect to an AP on a different channel, the AP will send a de-authentication frame to all the associated stations and move immediately to the same channel of the station interface so that the previously associated station would be able to connect on the new channel.
3. If the station interface is connected to an AP then the AP mode, the AP would start on the same channel of the station mode regardless of the channel number passed in the AP start request.

```
#define DEMO_WLAN_SSID "Demo_AP"
#define DEMO_WLAN_AUTH M2M_WIFI_SEC_WPA_PSK
#define DEMO_WLAN_PSK "1234567890"
static void wifi_cb(uint8_t u8MsgType, void *pvMsg)
{
    static int StartAP = 1;
    switch (u8MsgType)
    {
        case M2M_WIFI_RESP_CON_STATE_CHANGED:
        {
            tstrM2MAPConfig strM2MAPConfig;
            tstrM2mWifiStateChanged *pstrWifiState = (tstrM2mWifiStateChanged *)pvMsg;
            if (pstrWifiState->u8CurrState == M2M_WIFI_CONNECTED) {
                printf("Wi-Fi interface [%d] connected\r\n");
            } else if (pstrWifiState->u8CurrState == M2M_WIFI_DISCONNECTED) {
                printf("Wi-Fi interface [%d] disconnected\r\n");
            }
            if(StartAP == 1)
            {
                StartAP = 0;
                m2m_wifi_set_control_ifc(2);
                strcpy(strM2MAPConfig.au8WepKey, "1234567890");
                strM2MAPConfig.u8KeySz = WEP_40_KEY_STRING_SIZE;
                strM2MAPConfig.u8KeyIndx = 0;
                strcpy(strM2MAPConfig.au8SSID, "WILC1000_AP");
                strM2MAPConfig.u8ListenChannel = M2M_WIFI_CH_11;
                strM2MAPConfig.u8SecType = M2M_WIFI_SEC_WEP;
                strM2MAPConfig.u8SsidHide = 0;

                m2m_wifi_enable_ap(&strM2MAPConfig);
            }
            break;
        }
        default:
        {
            break;
        }
    }
}

int main(void)
{
    tstrWifiInitParam param;
    int8_t ret;

    // Initialize the BSP.
    nm_bsp_init();

    // Initialize Wi-Fi parameters structure.
    memset((uint8_t *)&param, 0, sizeof(tstrWifiInitParam));

    // Initialize Wi-Fi driver with data and status callbacks.
    param.pfAppWifiCb = wifi_cb;
    ret = m2m_wifi_init(&param);
    if (M2M_SUCCESS != ret) {
        printf("main: m2m_wifi_init call error!(%d)\r\n", ret);
        while (1) {
        }
    }
    printf("Trying to connect on interface 1\r\n");
    // connect to the first AP on interface 1
    ret = m2m_wifi_connect((char *)DEMO_WLAN_SSID, sizeof(DEMO_WLAN_SSID),
```

```

    DEMO_WLAN_AUTH, (char *)DEMO_WLAN_PSK, M2M_WIFI_CH_ALL)
    if (M2M_SUCCESS != ret) {
        printf("main: m2m_wifi_p2p call error!\r\n");
        while (1) {
        }
    }
    while (1) {
        /* Handle pending events from network controller. */
        while (m2m_wifi_handle_events(NULL) != M2M_SUCCESS) {
        }
    }
    return 0;
}

```

8.5 Station-P2P Client Concurrency

In this mode of concurrency, driver would be able to connect to an AP using one interface and start P2P connection on the second interface regardless which happens first, keeping in mind the following facts and limitations:

1. P2P connection should be started on the second interface regardless of the station interface connect after or before the P2P connection.
2. If the station interface is connected to an AP then the P2P connection, WILC firmware will enforce the GO the channel number of the station interface during the group negotiation frames.
3. If the P2P connection happened first, the station interface should connect on the same channel of the P2P group otherwise the P2P connection will be dropped.

```

#define DEMO_WLAN_SSID "Demo_AP"
#define DEMO_WLAN_AUTH M2M_WIFI_SEC_WPA_PSK
#define DEMO_WLAN_PSK "1234567890"

static void wifi_cb(uint8_t u8MsgType, void *pvMsg)
{
    switch (u8MsgType)
    {
        case M2M_WIFI_RESP_CON_STATE_CHANGED:
        {
            static int interfaceNo = 1;
            tstrM2mWifiStateChanged *pstrWifiState = (tstrM2mWifiStateChanged *)pvMsg;
            if (pstrWifiState->u8CurrState == M2M_WIFI_CONNECTED) {
                printf("Wi-Fi interface [%d] connected\r\n", interfaceNo);
            } else if (pstrWifiState->u8CurrState == M2M_WIFI_DISCONNECTED) {
                printf("Wi-Fi interface [%d] disconnected\r\n", interfaceNo);
            }
            if (interfaceNo == 1)
            {
                printf("start P2P on interface 2\r\n");
                m2m_wifi_set_control_ifc(2);
                m2m_wifi_p2p(11);
                interfaceNo = 2;
            }
            break;
        }
        default:
        {
            break;
        }
    }
}

int main(void)
{
    tstrWifiInitParam param;
    int8_t ret;
}

```

```

// Initialize the BSP.
nm_bsp_init();

// Initialize Wi-Fi parameters structure.
memset((uint8_t *)&param, 0, sizeof(tstrWifiInitParam));

// Initialize Wi-Fi driver with data and status callbacks.
param.pfAppWifiCb = wifi_cb;
ret = m2m_wifi_init(&param);
if (M2M_SUCCESS != ret) {
    printf("main: m2m_wifi_init call error!(%d)\r\n", ret);
    while (1) {
    }
}
printf("Trying to connect on interface 1\r\n");
// connect to the first AP on interface 1
ret = m2m_wifi_connect((char *)DEMO_WLAN_SSID, sizeof(DEMO_WLAN_SSID),
DEMO_WLAN_AUTH, (char *)DEMO_WLAN_PSK, M2M_WIFI_CH_ALL)
if (M2M_SUCCESS != ret) {
    printf("main: m2m_wifi_p2p call error!\r\n");
    while (1) {
    }
}
while (1) {
    /* Handle pending events from network controller. */
    while (m2m_wifi_handle_events(NULL) != M2M_SUCCESS) {
    }
}
return 0;
}

```

9 Data Send/Receive

In ATWILC1000 the data interface between the host driver and the upper layer is Ethernet frames, In order to use socket interface TCP/IP layer should be ported over the ATWILC1000 Ethernet interface.

9.1 Send Ethernet Frame

The API “m2m_wifi_send_ethernet_pkt” is used to transmit Ethernet frame over the AIR.



INFO

If the Wi-Fi is not connected to an AP, the frame will be dropped by the firmware and will not be transmitted over the AIR.

The function “m2m_wifi_send_ethernet_pkt” is a synchronous function when it returns with successful code it means the frame has been transferred from the host driver to the firmware but it doesn't mean that the frame has transmitted over the AIR, also there is no way to make sure that the frame is delivered to its final target successfully or it has been lost over the AIR, this should be handled by upper layer protocol e.g. TCP layer.

If the function returns error code M2M_ERR_MEM_ALLOC this means the chip is temporally out of buffers and the frame is not transferred to the chip memory, it is up to the application to wait and retry sending till the function returns success code.



INFO

Frame allocation and freeing is the a[application responsibility once the function “m2m_wifi_send_ethernet_pkt” returns the application can free the frame or reuse the buffer.

9.2 Receive Ethernet Frame

At the initialization an Ethernet callback function must be registered and a receive buffer must be allocated to be used as a receive buffer to the HIF and the registered callback function must add a handling to the “M2M_WIFI_RESP_ETHERNET_RX_PACKET” notification in order to receive Ethernet frames, see the below code example.

```
void ethernet_demo_cb(uint8 u8MsgType,void * pvMsg,void * pvCtrlBf)
{
    if(u8MsgType == M2M_WIFI_RESP_ETHERNET_RX_PACKET)
    {
        int i=0;
        uint8 au8RemoteIpAddr[4];
        uint8 *au8packet = (uint8*)pvMsg;
        tstrM2mIpCtrlBuf *PstrM2mIpCtrlBuf =(tstrM2mIpCtrlBuf *)pvCtrlBf;
        printk("Ethernet Frame Received buffer[%u] , Size = %d , Ifc ID
= %d\n",pvMsg,PstrM2mIpCtrlBuf->u16DataSize,PstrM2mIpCtrlBuf->u8IfcId);
    }
}

int main()
{
    tstrWifiInitParam param;
    rx_buff = linux_wlan_malloc(15*1024);
    m2m_memset((uint8*)&param, 0, sizeof(param));
    param.pfAppWifiCb = m2m_wifi_state;
    param.strEthInitParam.pfAppEthCb = ethernet_demo_cb;
    param.strEthInitParam.au8ethRcvBuf = rx_buff;
    param.strEthInitParam.u16ethRcvBufSize = 1600;
```

```
    ret = m2m_wifi_init(&param);  
    return ret;  
}
```

After the return of the callback function the HIF will reuse the registered buffer and will overwrite the data inside so the application should either to move the frame from the buffer or to update the buffer info using the API “m2m_wifi_set_receive_buffer” before the return of the callback function.

If the received frame is larger than the provided buffer the HIF will receive part of the frame and sets the “u16RemainigDataSize” of the structure tstrM2mIpCtrlBuf to the remaining size of the current frame then after the return of the callback function the HIF will receive the other part(s) and give a callback function on each part till the end of the frame.

9.3 Concurrency Send

If the concurrency is used Application can send frames on the second interface using the API “m2m_wifi_send_ethernet_pkt_ifc1”, the API has the same characteristics as of “m2m_wifi_send_ethernet_pkt” with an exception it sends the frame on interface 2.

9.4 Concurrency Receive

If the concurrency is used application can distinguish between the frames received on interface 1 and frames received on interface 2 using the parameter “u8Ifcld” included in the structure “tstrM2mIpCtrlBuf” , everything else can be used from Receive Ethernet Frame, see Section [9.2](#).

10 Host Interface Protocol

Communication between the user application and the WILC device is facilitated by driver software. This driver implements the Host Interface Protocol and exposes an API to the application with various services. The services are broadly in two categories: Wi-Fi device control and Ethernet data. The Wi-Fi device control services allow actions such as channel scanning, network identification, connection and disconnection. The data services allow data transfer once a connection has been established.

The host driver implements services asynchronously. This means that when the application calls an API to request a service action, the call is non-blocking and returns immediately, often before the action is completed. Where appropriate, notification that an action has completed is provided in a subsequent message from the WILC device to the Host which is delivered to the application via a callback function. More generally, the WILC firmware uses asynchronous events to signal the host driver of certain status changes. Asynchronous operation is essential where functions (such as Wi-Fi connection) make take significant time.

When an API is called, a sequence of layers is activated formatting the request and arranging to transfer it to the WILC device through the serial protocol.

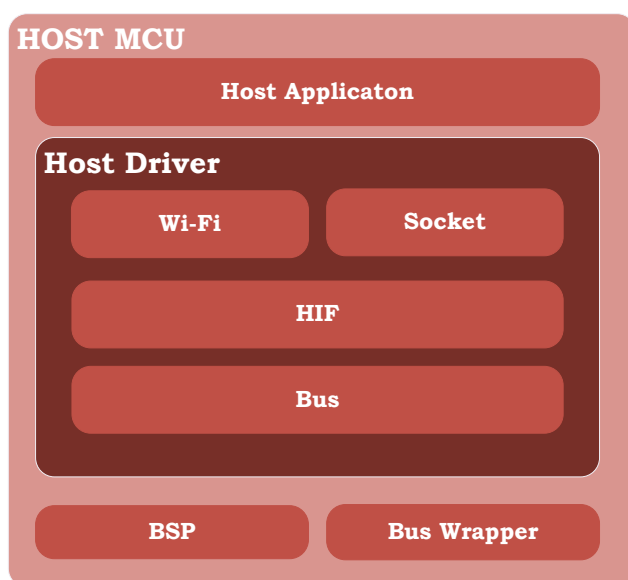


IMPORTANT

Dealing with HIF messages in host MCU application is an advanced topic. For most applications, it is recommended to use Wi-Fi. This layer hides the complexity of the HIF APIs.

After the application sends request, the Host Driver (Wi-Fi) formats the request and sends it to the HIF layer which then interrupts the WILC device announcing that a new request will be posted. Upon receipt, the WILC firmware parses the request and starts the required operation.

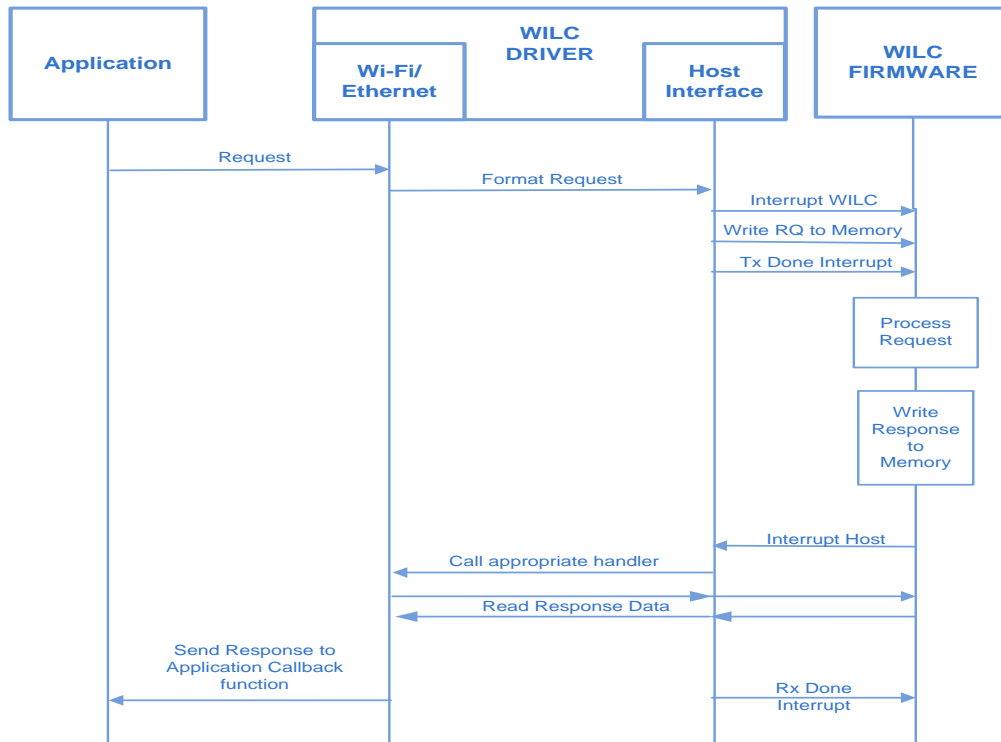
Figure 10-1. WILC Driver Layers



The Host Interface Layer is responsible for handling communication between the host MCU and the WILC device. This includes Interrupt handling, DMA control and management of communication logic between firmware driver at host and WILC firmware.

The Request/Response sequence between the Host and the WILC chip is shown in [Figure 10-2](#).

Figure 10-2. The Request/Response Sequence Diagram



10.1 Chip Initialization Sequence

Table 10-1 shows the sequence and the registers needed to initialize the ATWILC1000 HW.

Table 10-1.

Step	Description
Read chip ID to make sure the bus and the chip are working fine	Read the register 0x1000 , the return value should be 0x1002xx
Download the firmware into the chip memory	Refer to Chapter 12: ATWILC1000 Firmware Download for details
Disable the boot ROM	Write in register 0xC0000 value 0x71
Reset the state register	Write in register NMI_STATE_REG value 0
Set MUX to enable CPU reset from the GLOBAL RESET register	Write in register 0x1118 value 1
Set NMI_VMM_CORE_CFG to SPI bus	Write in register NMI_VMM_CORE_CFG value 1
Rest the chip CPU	Toggle bit(10) from 0 to 1 in register NMI_GLB_RESET_0
Poll on state register to make sure firmware is started successfully	Read register NMI_STATE_REG and compere the value to M2M_FINISH_INIT_STATE
Set MUX to enable IRQN pin output	Set Bit 8 in register NMI_PIN_MUX_0
Enable IRQ on IRQN pin	Set bit 16 ion register NMI_INTR_ENABLE

10.2 Transfer Sequence Between HIF Layer and WILC Firmware

The following sections shows the individual steps taken during a HIF frame transmit (HIF message to the WILC) and a HIF frame receive (HIF message from the WILC).

10.2.1 Frame Transmit

The following diagram shows the steps and states involved in sending a message from the host to the WILC device:

Figure 10-3. HIF Frame Transmit to WILC

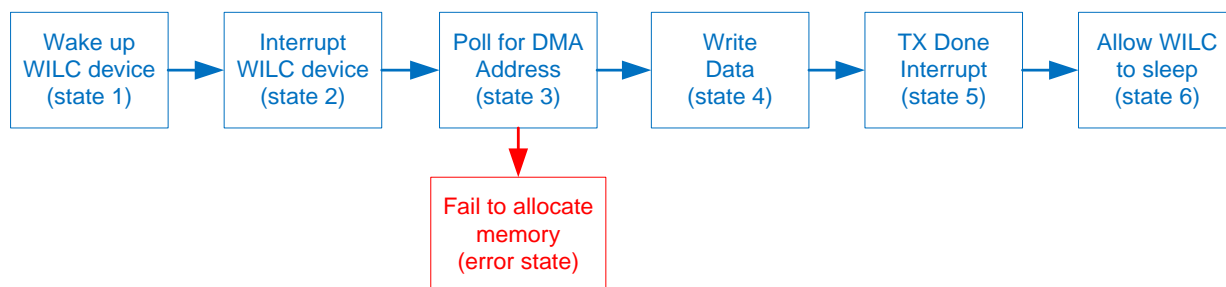


Table 10-2.

Step	Description
Step (1) Wake up the WILC device	Wakeup the device to be able to receive Host requests
Step (2) Interrupt the WILC device	Prepare and Set the HIF layer header to NMI_STATE_REG register (4 Bytes header describing the sent packet). Set BIT [1] of WIFI_HOST_RCV_CTRL_2 register to raise an interrupt to the WILC chip.
Step (3) Poll for DMA address	Wait until the WILC chip clears BIT [1] of WIFI_HOST_RCV_CTRL_2 register. Get the DMA address (for the allocated memory) from register 0x150400.
Step (4) Write Data	Write the Data Blocks in sequence, the HIF header then the Control buffer (if any) then the Data buffer (if any)
Step (5) TX Done Interrupt	Announce finishing writing the data by setting BIT [1] of WIFI_HOST_RCV_CTRL_3 register
Step (6) Allow WILC device to sleep	Allow the WILC device to enter sleep mode again (if it wishes)

10.2.2 Frame Receive

Figure 10-4 shows the steps and states involved in sending a message from the WILC device to the host:

Figure 10-4. HIF Frame Receive from WILC to Host

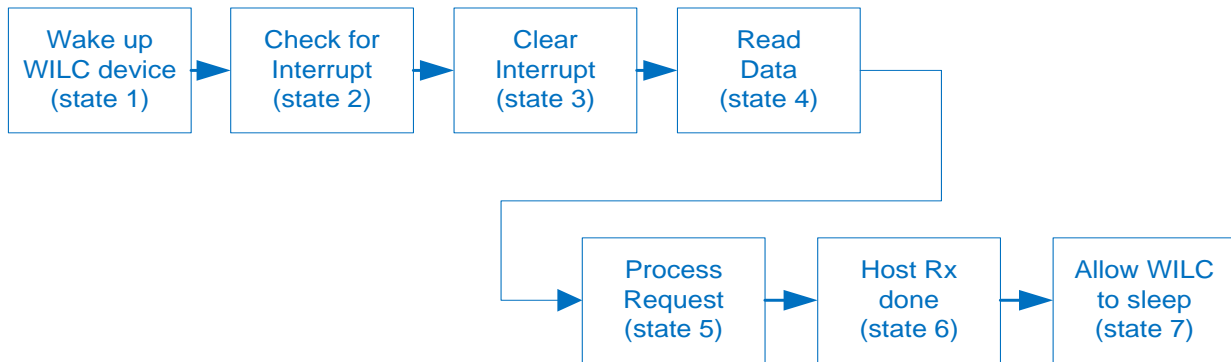
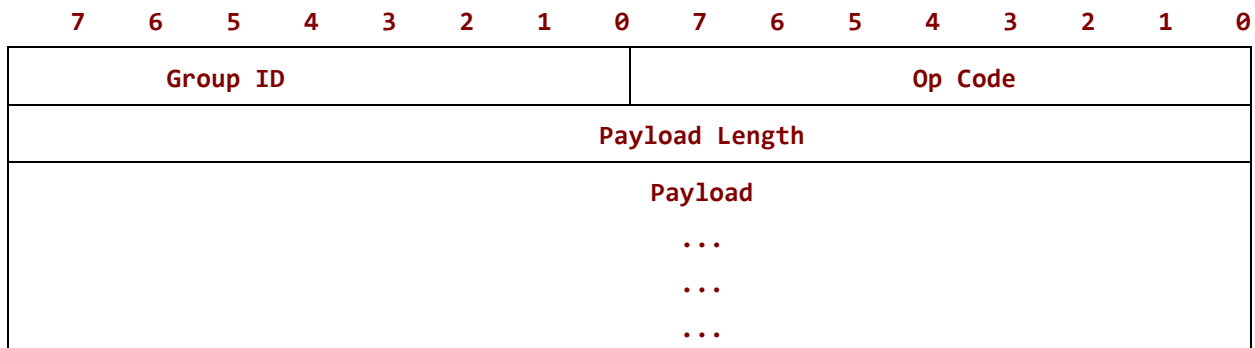


Table 10-3.

Step	Description
Step (1) Wake up the WILC device	Wakeup the device to be able to receive Host requests.
Step (2) Check for Interrupt	Monitor BIT[0] of WIFI_HOST_RCV_CTRL_0 register. Disable the host from receiving interrupts (until this one has been processed).
Step (3) Clear interrupt	Write zero to BIT[0] of WIFI_HOST_RCV_CTRL_0 register.
Step (4) Read Data	Get the address of the data block from WIFI_HOST_RCV_CTRL_1 register. Read Data block with size obtained from WIFI_HOST_RCV_CTRL_0 register BIT[13] <-> BIT[2].
Step (5) Process Request	Parse the HIF header at the start of the Data and forward the Data to the appropriate registered Callback function.
Step (6) HOST RX Done	Raise an interrupt for the chip to free the memory holding the data by setting BIT[1] of WIFI_HOST_RCV_CTRL_0 register. Enable Host interrupt reception again.
Step (7) Allow WILC device to sleep	Allow the WILC device to enter sleep mode again (if it wishes).

10.3 HIF Message Header Structure

The HIF message is the data structure exchanged back and forth between the Host Interface and WILC firmware. The HIF message header structure consists of three fields:



- **The Group ID (8-bits):** A group ID is the category of the message. Valid categories are `M2M_REQ_GRP_WIFI`, `M2M_REQ_GRP_HIF` corresponding to Wi-Fi and HIF respectively. A group ID can be assigned one of the values enumerated in `tenuM2mReqGrp`.
- **Op Code (8-bit):** Is a command number. Valid command number is a value enumerated in: `tenuM2mConfigCmd` and `tenuM2mStaCmd`, `tenuM2mApCmd` and `tenuM2mP2pCmd` corresponding to configuration, STA mode AP mode and P2P mode commands. See the full list of commands in the header file `m2m_types.h`.
- **Payload Length (16-bits):** The payload length in bytes (does not include header).

10.4 HIF Layer APIs

The interface between the application and the driver will be done at the higher layer API interface (Wi-Fi) as explained previously, the driver upper layer uses a lower layer API to access the services of the Host Interface Protocol. This section describes the Host Interface APIs that the upper layers use.

The following API functions are described:

- `hif_chip_wake`
- `hif_chip_sleep`
- `hif_register_cb`
- `hif_isr`
- `hif_receive`
- `hif_send`

For all functions the return value is either `M2M_SUCCESS` (zero) in case of success or a negative value in case of failure.

`sint8 hif_chip_wake(void) :`

This function wakes the WILC chip from sleep mode using clockless register access. It sets BIT[1] of register 0x01 and sets the value of `WAKE_REG` register to `WAKE_VALUE`.

`sint8 hif_chip_sleep(void) :`

This function enables sleep mode for the WILC chip by setting the `WAKE_REG` register to a value of `SLEEP_VALUE` and clearing BIT[1] of register 0x01.

`sint8 hif_register_cb(uint8 u8Grp, tpfHifCallBack fn) :`

This function set the callback function for different components (e.g. `M2M_WIFI`, `M2M_HIF`, `M2M_OTA` ...etc.). A callback is registered by upper layers to receive specific events of a specific message group.

`sint8 hif_isr(void) :`

This is the Host interface interrupt service routine. It handles interrupts generated by the WILC chip and parses the HIF header to call back the appropriate handler.

`sint8 hif_receive(uint32 u32Addr, uint8 *pu8Buf, uint16 u16Sz, uint8 isDone) :`

This function causes the Host driver to read data from the WILC chip. The location and length of the data must be known in advance and specified. This will typically have been extracted from an earlier part of a transaction.

`sint8 hif_send(uint8 u8Gid, uint8 u8Opcode, uint8 *pu8CtrlBuf, uint16 u16CtrlBufSize, uint8 *pu8DataBuf, uint16 u16DataSize, uint16 u16DataOffset) :`

This function causes the Host driver to send data to the WILC chip. The WILC chip will have been prepared for reception according to the flow described in the previous section.

10.5 Scan Code Example

The following code example illustrates the Request/Response flow on a Wi-Fi Scan request: For more details on the code examples, refer to [R02].

- The application requests a Wi-Fi scan

```
{  
    m2m_wifi_request_scan(M2M_WIFI_CH_ALL);  
}
```

- The Host driver Wi-Fi layer formats the request and forward it to HIF (Host Interface) layer

```
sint8 m2m_wifi_request_scan(uint8 ch)  
{  
    tstrM2MScan strtmp;  
    sint8 s8Ret = M2M_ERR_SCAN_IN_PROGRESS;  
    strtmp.u8ChNum = ch;  
    s8Ret = hif_send(M2M_REQ_GRP_WIFI, M2M_WIFI_REQ_SCAN, (uint8*)&strtmp,  
        sizeof(tstrM2MScan), NULL, 0, 0);  
    return s8Ret;  
}
```

- The HIF layer sends the request to the WILC chip

```
sint8 hif_send(uint8 u8Gid, uint8 u8Opcode, uint8 *pu8CtrlBuf, uint16 u16CtrlBufSize,  
    uint8 *pu8DataBuf, uint16 u16DataSize, uint16 u16DataOffset)  
{  
    sint8 ret = M2M_ERR_SEND;  
    volatile tstrHifHdr strHif;  
  
    strHif.u8Opcode = u8Opcode & (~NBIT7);  
    strHif.u8Gid = u8Gid;  
    strHif.u16Length = M2M_HIF_HDR_OFFSET;  
    if(pu8DataBuf != NULL)  
    {  
        strHif.u16Length += u16DataOffset + u16DataSize;  
    }  
    else  
    {  
        strHif.u16Length += u16CtrlBufSize;  
    }  
  
    /* TX STEP (1) */  
    ret = hif_chip_wake();  
    if(ret == M2M_SUCCESS)  
    {  
        volatile uint32 reg, dma_addr = 0;  
        volatile uint16 cnt = 0;  
  
        reg = 0UL;  
        reg |= (uint32)u8Gid;  
        reg |= ((uint32)u8Opcode << 8);  
        reg |= ((uint32)strHif.u16Length << 16);  
        ret = nm_write_reg(NMI_STATE_REG, reg);  
        if(M2M_SUCCESS != ret) goto ERR1;  
        reg = 0;  
        /* TX STEP (2) */  
        reg |= (1 << 1);  
        ret = nm_write_reg(WIFI_HOST_RCV_CTRL_2, reg);  
    }  
}
```

```

if(M2M_SUCCESS != ret) goto ERR1;
dma_addr = 0;
for(cnt = 0; cnt < 1000; cnt++)
{
    ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_2,(uint32 *)&reg);
    if(ret != M2M_SUCCESS) break;
    if (!(reg & 0x2))
    {
        /* TX STEP (3) */
        ret = nm_read_reg_with_ret(0x150400,(uint32 *)&dma_addr);
        if(ret != M2M_SUCCESS) {
            /*in case of read error clear the dma address and return error*/
            dma_addr = 0;
        }
        /*in case of success break */
        break;
    }
}
if (dma_addr != 0)
{
    volatile uint32    u32CurrAddr;
    u32CurrAddr = dma_addr;
    strHif.u16Length=NM_BSP_B_L_16(strHif.u16Length);
    /* TX STEP (4) */
    ret = nm_write_block(u32CurrAddr, (uint8*)&strHif, M2M_HIF_HDR_OFFSET);
    if(M2M_SUCCESS != ret) goto ERR1;
    u32CurrAddr += M2M_HIF_HDR_OFFSET;
    if(pu8CtrlBuf != NULL)
    {
        ret = nm_write_block(u32CurrAddr, pu8CtrlBuf, u16CtrlBufSize);
        if(M2M_SUCCESS != ret) goto ERR1;
        u32CurrAddr += u16CtrlBufSize;
    }
    if(pu8DataBuf != NULL)
    {
        u32CurrAddr += (u16DataOffset - u16CtrlBufSize);
        ret = nm_write_block(u32CurrAddr, pu8DataBuf, u16DataSize);
        if(M2M_SUCCESS != ret) goto ERR1;
        u32CurrAddr += u16DataSize;
    }
    reg = dma_addr << 2;
    reg |= (1 << 1);
    /* TX STEP (5) */
    ret = nm_write_reg(WIFI_HOST_RCV_CTRL_3, reg);
    if(M2M_SUCCESS != ret) goto ERR1;
}
else
{
    /* ERROR STATE */
    M2M_DBG("Failed to alloc rx size\r\n");
    ret = M2M_ERR_MEM_ALLOC;
    goto ERR1;
}
}
else
{
    M2M_ERR("(HIF)Fail to wakeup the chip\r\n");
}

```

```

        goto ERR1;
    }

    /* TX STEP (6) */
    ret = hif_chip_sleep();
ERR1:
    return ret;
}

```

- The WILC chip processes the request and interrupts the host after finishing the operation
- The HIF layer then receives the response

```

static sint8 hif_isr(void)
{
    sint8 ret = M2M_ERR_BUS_FAIL;
    uint32 reg;
    volatile tstrHifHdr strHif;

    /* RX STEP (1) */
    ret = hif_chip_wake();
    if(ret == M2M_SUCCESS)
    {
        /* RX STEP (2) */
        ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_0, &reg);
        if(M2M_SUCCESS == ret)
        {
            /* New interrupt has been received */
            if(reg & 0x1)
            {
                uint16 size;
                nm_bsp_interrupt_ctrl(0);
                /*Clearing RX interrupt*/
                ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_0, &reg);
                if(ret != M2M_SUCCESS) goto ERR1;
                reg &= ~(1<<0);

                /* RX STEP (3) */
                ret = nm_write_reg(WIFI_HOST_RCV_CTRL_0, reg);
                if(ret != M2M_SUCCESS) goto ERR1;
                /* read the rx size */
                ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_0, &reg);
                if(M2M_SUCCESS != ret)
                {
                    M2M_ERR("(hif) WIFI_HOST_RCV_CTRL_0 bus fail\n");
                    nm_bsp_interrupt_ctrl(1);
                    goto ERR1;
                }
                gu8HifSizeDone = 0;
                size = (uint16)((reg >> 2) & 0xfff);
                if (size > 0) {
                    uint32 address = 0;
                    /**
                     * start bus transfer
                     */

                    /* RX STEP (4) */
                    ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_1, &address);
                    if(M2M_SUCCESS != ret)
                    {
                        M2M_ERR("(hif) WIFI_HOST_RCV_CTRL_1 bus fail\n");
                        nm_bsp_interrupt_ctrl(1);
                        goto ERR1;
                    }
                    ret = nm_read_block(address, (uint8*)&strHif, sizeof(tstrHifHdr));
                    strHif.u16Length = NM_BSP_B_L_16(strHif.u16Length);
                    if(M2M_SUCCESS != ret)

```



```

    {
        M2M_ERR("(hif) address bus fail\n");
        nm_bsp_interrupt_ctrl(1);
        goto ERR1;
    }
    if(strHif.u16Length != size)
    {
        if((size - strHif.u16Length) > 4)
        {
            M2M_ERR("(hif) Corrupted packet Size = %u <L = %u, G = %u, OP
= %02X>\n",
                size, strHif.u16Length, strHif.u8Gid, strHif.u8Opcode);
            nm_bsp_interrupt_ctrl(1);
            ret = M2M_ERR_BUS_FAIL;
            goto ERR1;
        }
    }

    /* RX STEP (5) */
    if(M2M_REQ_GRP_WIFI == strHif.u8Gid)
    {
        if(pfWifiCb)
            pfWifiCb(strHif.u8Opcode, strHif.u16Length - M2M_HIF_HDR_OFFSET,
                address + M2M_HIF_HDR_OFFSET);
    }
    else if(M2M_REQ_GRP_IP == strHif.u8Gid)
    {
        if(pfIpCb)
            pfIpCb(strHif.u8Opcode, strHif.u16Length - M2M_HIF_HDR_OFFSET,
                address + M2M_HIF_HDR_OFFSET);
    }
    else if(M2M_REQ_GRP_OTA == strHif.u8Gid)
    {
        if(pfOtaCb)
            pfOtaCb(strHif.u8Opcode, strHif.u16Length - M2M_HIF_HDR_OFFSET,
                address + M2M_HIF_HDR_OFFSET);
    }
    else
    {
        M2M_ERR("(hif) invalid group ID\n");
        ret = M2M_ERR_BUS_FAIL;
        goto ERR1;
    }

    /* RX STEP (6) */
    if(!gu8HifSizeDone)
    {
        M2M_ERR("(hif) host app didn't set RX Done\n");
        ret = hif_set_rx_done();
    }
}
else
{
    ret = M2M_ERR_RCV;
    M2M_ERR("(hif) Wrong Size\n");
    goto ERR1;
}
}
else
{
    #ifndef WIN32
        M2M_ERR("(hif) False interrupt %lx", reg);
    #endif
}
}

```

```

        else
        {
            M2M_ERR("(hif) Fail to Read interrupt reg\n");
            goto ERR1;
        }
    }
    else
    {
        M2M_ERR("(hif) FAIL to wakeup the chip\n");
        goto ERR1;
    }
    /* RX STEP (7) */
    ret = hif_chip_sleep();
ERR1:
    return ret;
}

```

- The appropriate handler is layer Wi-Fi (called from HIF layer)

```

static void m2m_wifi_cb(uint8 u8OpCode, uint16 u16DataSize, uint32 u32Addr)
{
    // ...code eliminated...
    else if (u8OpCode == M2M_WIFI_RESP_SCAN_DONE)
    {
        tstrM2mScanDone strState;
        gu8scanInProgress = 0;
        if(hif_receive(u32Addr, (uint8*)&strState, sizeof(tstrM2mScanDone), 0) == M2M_SUCCESS)
        {
            gu8ChNum = strState.u8NumofCh;
            if (gpfAppWifiCb)
                gpfAppWifiCb(M2M_WIFI_RESP_SCAN_DONE, &strState);
        }
    }
    // ...code eliminated...
}

```

- The Wi-Fi layer sends the response to the application through its callback function

```

if (u8MsgType == M2M_WIFI_RESP_SCAN_DONE)
{
    tstrM2mScanDone *pstrInfo = (tstrM2mScanDone*) pvMsg;
    if( (gu8IsWiFiConnected == M2M_WIFI_DISCONNECTED) &&
        (gu8WPS == WPS_DISABLED) && (gu8Prov == PROV_DISABLED) )
    {
        gu8Index = 0;
        gu8Sleep = PS_WAKE;
        if (pstrInfo->u8NumofCh >= 1)
        {
            m2m_wifi_req_scan_result(gu8Index);
            gu8Index++;
        }
        else
        {
            m2m_wifi_request_scan(M2M_WIFI_CH_ALL);
        }
    }
}

```

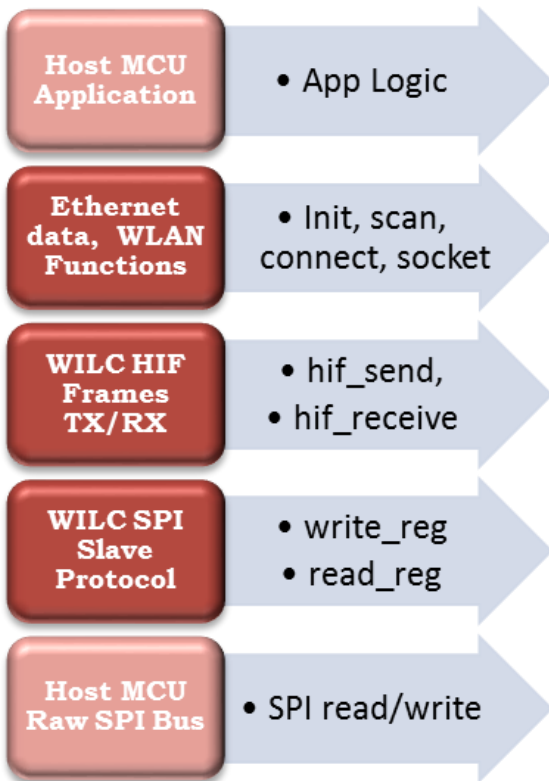
11 WILC SPI Protocol

WILC main interface is SPI. The WILC device employs a protocol to allow exchange of formatted binary messages between WILC firmware and host MCU application. The WILC protocol uses raw bytes exchanged on SPI bus to form high level structures like requests and callbacks.

The WILC SPI protocol consists of three layers:

- **Layer 1:** WILC SPI slave protocol, which allows the host MCU application to perform register/memory read and write operation in the ATWILC1000 device using raw SPI data exchange.
- **Layer 2:** Host MCU application uses the register and memory read and write capabilities to exchange host interface frames with the WILC firmware. It also provides asynchronous callback from the WILC firmware to the host MCU through interrupts and host interface RX frames. This layer was discussed earlier in chapter 15.
- **Layer 3:** Allows the host MCU application to exchange high level messages (e.g. Wi-Fi scan or Ethernet data received) with the WILC firmware to employ in the host MCU application logic.

Figure 11-1. WILC SPI Protocol Layers



11.1 Introduction

The WILC SPI Protocol is implemented as a command-response transaction and assumes one party is the master and the other is the slave. The roles correspond to the master and slave devices on the SPI bus. Each message has an identifier in the first byte indicating the type of message:

- Command
- Response
- Data

In the case of Command and Data messages, the last byte is used as data integrity check.

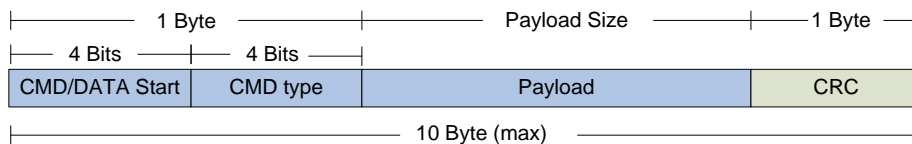
The format of Command and Response and Data frames is described in the following sections. The following points apply:

- There is a response for each command
- Transmitted/received data is divided into packets with fixed size
- For a write transaction (*Slave is receiving data packets*), the slave should reply by a response for each data packet
- For a RD transaction (*master is receiving data packets*), the master doesn't send response. If there is an error, the master should request retransmission on the lost data packet.
- Protection of commands and data packets by CRC is optional

11.1.1 Command Format

The following frame formation is used for commands where the host supports a DMA address of three bytes.

Figure 11-2.



The first byte contains two fields:

- The CMD/Data Start field indicates that this is a Command frame
- The CMD type field specifies the command to be executed

The **CMD type** may be one of 15 commands:

- DMA write
- DMA read
- Internal register write
- Internal register read
- Transaction termination
- Repeat data Packet
- DMA extended write
- DMA extended read
- DMA single-word write
- DMA single-word read
- Soft reset

The **Payload** field contains command specific data and its length depends on the CMD type.

The **CRC** field is optional and generally computed in software.

The **Payload** field can be one of four types each having a different length:

- A: 3 bytes
- B: 5 bytes
- C: 6 bytes
- D: 7 bytes

Type A commands include:

- DMA single-word RD

- internal register RD
- Transaction termination command
- Repeat Data PKT command
- Soft reset command

Type B commands include:

- DMA RD Transaction
- DMA WR Transaction

Type C commands include:

- DMA Extended RD transaction
- DMA Extended WR transaction
- Internal register WR

Type D commands include:

- DMA single-word WR

Full details of the frame format fields are provided in [Table 11-1](#):

Table 11-1.

Field	Size	Description
CMD Start	4 bits	Command Start : 4'b1100
CMD Type	4 bits	Command type: 4'b0001: DMA write transaction 4'b0010: DMA read transaction 4'b0011: Internal register write 4'b0100: Internal register read 4'b0101: Transaction termination 4'b0110: Repeat data Packet command 4'b0111: DMA extended write transaction 4'b1000: DMA extended read transaction 4'b1001: DMA single-word write 4'b1010: DMA single-word read 4'b1111: soft reset command

Payload	<p>A: 3 B: 5 C: 6 D: 7</p>	<p>The Payload field may be of Type A,B,C or D</p> <p><u>Type A (length 3)</u></p> <p>1- DMA single-word RD Param: Read Address: Payload bytes:</p> <ul style="list-style-type: none"> B0: ADDRESS[23:16] B1: ADDRESS[15:8] B2: ADDRESS[7:0] <p>2- internal register RD Param: Offset address (2 bytes): Payload bytes:</p> <ul style="list-style-type: none"> B0: OFFSET-ADDR[15:8] B1: OFFSET-ADDR[7:0] B2: 0 <p>3- Transaction termination command Param: none Payload bytes:</p> <ul style="list-style-type: none"> B0: 0 B1: 0 B2: 0 <p>4- Repeat Data PKT command Param: none Payload bytes:</p> <ul style="list-style-type: none"> B0: 0 B1: 0 B2: 0 <p>5- Soft reset command Param: none Payload bytes:</p> <ul style="list-style-type: none"> B0: 0xFF B1: 0xFF B2: 0xFF <p><u>Type B (length 5)</u></p> <p>1- DMA RD Transaction Params:</p> <ul style="list-style-type: none"> DMA Start Address : 3 bytes DMA count : 2 bytes <p>Payload bytes:</p> <ul style="list-style-type: none"> B0: ADDRESS[23:16] B1: ADDRESS[15:8] B2: ADDRESS[7:0] B3: COUNT[15:8] B4: COUNT[7:0] <p>2- DMA WR Transaction Params:</p> <ul style="list-style-type: none"> DMA Start Address : 3 bytes DMA count : 2 bytes <p>Payload bytes:</p> <ul style="list-style-type: none"> B0: ADDRESS[23:16] B1: ADDRESS[15:8] B2: ADDRESS[7:0] B3: COUNT[15:8]
---------	--	---

Field	Size	Description
		<ul style="list-style-type: none"> B4: COUNT[7:0] <p><u>Type C (length 6)</u></p> <p>1- DMA Extended RD transaction</p> <p><i>Params:</i></p> <ul style="list-style-type: none"> DMA Start Address : 3 bytes DMA extended count: 3 bytes <p><i>Payload bytes:</i></p> <ul style="list-style-type: none"> B0: ADDRESS[23:16] B1: ADDRESS[15:8] B2: ADDRESS[7:0] B3: COUNT[23:16] B4: COUNT[15:8] B5: COUNT[7:0] <p>2- DMA Extended WR transaction</p> <p><i>Params:</i></p> <ul style="list-style-type: none"> DMA Start Address : 3 bytes DMA extended count: 3 bytes <p><i>Payload bytes:</i></p> <ul style="list-style-type: none"> B0: ADDRESS[23:16] B1: ADDRESS[15:8] B2: ADDRESS[7:0] B3: COUNT[23:16] B4: COUNT[15:8] B5: COUNT[7:0] <p>3- Internal register WR*</p> <p><i>Params:</i></p> <ul style="list-style-type: none"> Offset address: 3 bytes Write Data: 3 bytes <p>* "clocked or clockless registers"</p> <p><i>Payload bytes:</i></p> <ul style="list-style-type: none"> B0: OFFSET-ADDR[15:8] B1: OFFSET-ADDR [7:0] B2: DATA[31:24] B3: DATA [23:16] B4: DATA [15:8] B5: DATA [7:0] <p><u>Type D (length 7)</u></p> <p>1- DMA single-word WR</p> <p><i>Params:</i></p> <ul style="list-style-type: none"> Address: 3 bytes DMA Data: 4 bytes <p><i>Payload bytes:</i></p> <ul style="list-style-type: none"> B0: ADDRESS[23:16] B1: ADDRESS[15:8] B2: ADDRESS[7:0] B3: DATA[31:24] B4: DATA [23:16] B5: DATA [15:8] B6: DATA [7:0]

Field	Size	Description
CRC7	1 byte	Optional data integrity field comprising two subfields: bit 0: fixed value '1' bits 1-7: 7 bit CRC value computed using polynomial $G(x) = X^7 + X^3 + 1$ with seed value: 0x7F

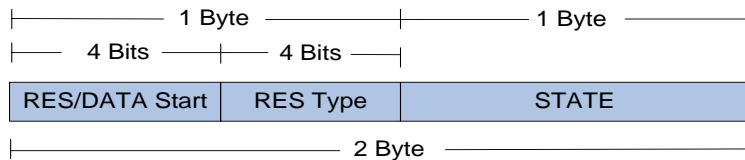
Table 11-2 summarizes the different commands according to the payload type (DMA address = 3-bytes).

Table 11-2.

Payload Type	Payload size	Command packet size "with CRC"	Commands
Type A	3-Bytes	5-Bytes	1- DMA Single-Word Read 2- Internal Register Read 3- Transaction Termination 4- Repeat Data Packet 5- Soft Reset
Type B	5-Bytes	7-Bytes	1- DMA Read 2- DMA Write
Type C	6-Bytes	8-Bytes	1- DMA Extended Read 2- DMA Extended Write 3- Internal Register Write
Type D	7-Bytes	9-Bytes	1- DMA Single-Word Write

11.1.2 Response Format

The following frame formation is used for responses sent by the WILC device as the result of receiving a Command or certain Data frames. The Response message has a fixed length of two bytes.



The first byte contains two four bit fields which identify the response message and the response type.

The second byte indicates the status of the WILC after receiving and, where possible, executing the command/data. This byte contains two sub fields:

- B0-B3: Error state
- B4-B7: DMA state

States that may be indicated are:

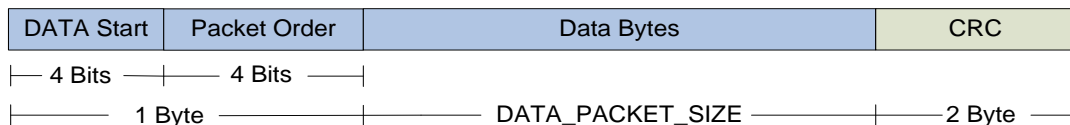
- DMA state:
 - DMA ready for any transaction
 - DMA engine is busy
- Error state:
 - No error
 - Unsupported command
 - Receiving unexpected data packet
 - Command CRC7 error

Table 11-3.

Field	Size	Description
Res Start	4 bits	Response Start: 4'b1100
Response Type	4 bits	If the response packet is for Command: <ul style="list-style-type: none"> Contains of copy of the Command Type field in the Command If the response packet is for received Data Packet: <ul style="list-style-type: none"> 4'b0001: First data packet is received 4'b0010: Receiving data packets 4'b0011: Last data packet is received 4'b1111: Reserved value
State	1 byte	This field is divided into two subfields: <div style="text-align: center;"> <pre> graph TD State[State] --- DMAState[DMA State] State --- ErrorState[Error State] DMAState --- DMABits[4 Bits] ErrorState --- ErrorBits[4 bits] </pre> </div> DMA State: <ul style="list-style-type: none"> 4'b0000: DMA ready for any transaction 4'b0001: DMA engine is busy Error State: <ul style="list-style-type: none"> 4'b0000: No error 4'b0001: Unsupported command 4'b0010: Receiving unexpected data packet 4'b0011: Command CRC7 error 4'b0100: Data CRC16 error 4'b0101: Internal general error

11.1.3 Data Packet Format

The Data Packet Format is used in either direction (master to slave or slave to master) to transfer opaque data. A Command frame is used either to inform the slave that a data packet is about to be sent or to request the slave to send a data packet to the master. In the case of master to slave, the slave sends a response after the command and each subsequent data frame. The format of a data packet is shown below.

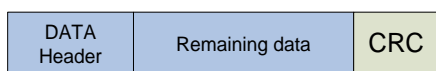


To support DMA hardware a large data transfer may be fragmented into multiple smaller Data Packets. This is controlled by the value of DATA_PACKET_SIZE which is agreed between the master and slave in software and is a fixed value such as 256B, 512B, 1KB (default), 2KB, 4KB, or 8KB. If a transfer has a length m which exceeds DATA_PACKET_SIZE the sender must split into n frames where frames $1..n-1$ will be length DATA_PACKET_SIZE and frame n will be length:

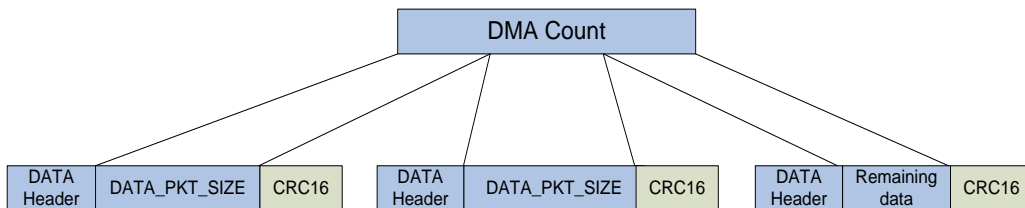
$(m - (n-1) * \text{DATA_PACKET_SIZE})$. This is shown diagrammatically below:

- **If DMA count \leq DATA_PACKET_SIZE**

The data packet is "DATA_Header + DMA count + optional CRC16 ", i.e. No padding.



- If DMA count > DATA_PACKET_SIZE



If **remaining data** < DATA_PACKET_SIZE, the **last data packet** is:

“DATA_Header + remaining data + optional CRC16 “, i.e. No padding

The frame fields are describe in detail in [Table 11-4](#).

Table 11-4.

Field	Size	Description
Data Start	4 bits	4'b1111 (Default) (Can be changed to any value by programming DATA_START_CTRL register)
Packet Order	4 bits	4'b0001: First packet in this transaction 4'b0010: Neither the first or the last packet in this transaction 4'b0011: Last packet in this transaction 4'b1111: Reserved
Data Bytes	DATA_PACKET_SIZE	User data
CRC16	2 bytes	Optional data integrity field comprising a 16 bit CRC value encoded in two bytes. The most significant 8 bits are transmitted first in the frame. The CRC16 value is computed on data bytes only based on the polynomial: $G(x) = X^{16} + X^{12} + X^5 + 1$, seed value: 0xFFFF

11.1.4 Error Recovery Mechanism

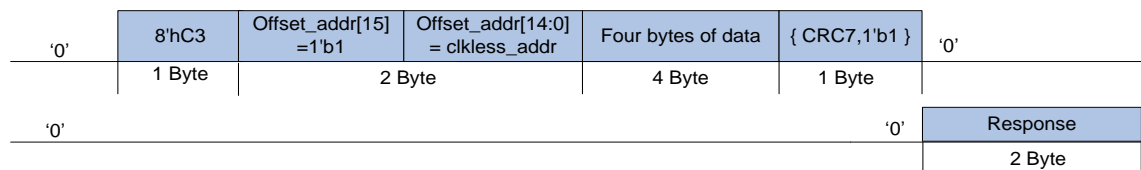
Error Type	Recovery Mechanism
Master:	
CRC error in command	<ol style="list-style-type: none"> 1. Error response received from slave. 2. Retransmit the command.
CRC error in received data	<ol style="list-style-type: none"> 1. Issue a repeat command for the data packet that has a CRC error. 2. Slave sends a response to the previous command. 3. Slave keeps the start DMA address of the previous data packet, so it can retransmit it. 4. Receive the data packet again.

Error Type	Recovery Mechanism
No response is received from slave	<ul style="list-style-type: none"> Synchronization is lost between master and slave The worst case is when slave is in receiving data state Solution: master should wait for max DATA_PACKET_SIZE period then generate a soft reset command
Unexpected response	Retransmit the command
TX/RX Data count error	Retransmit the command
No response to soft reset command	<ul style="list-style-type: none"> Transmit all ones till master receives a response of all ones from the slave Then deactivate the output data line
Slave:	
Unsupported command	<ul style="list-style-type: none"> Send response with error Returns to command monitor state
Receive command CRC error	<ul style="list-style-type: none"> Send response with error waits for command retransmission
Received data CRC error	<ul style="list-style-type: none"> Send response with error wait for retransmission of the data packet
Internal general error	The master should soft reset the slave
TX/RX Data count error	<ul style="list-style-type: none"> Only the master can detect this error Slave operates with the data count received till the count finishes or the master terminates the transaction In both cases the master should retry the command from the beginning
No response to soft reset command	<ol style="list-style-type: none"> First received 4'b1001, it decides data start. Then received packet order 4'b1111 that is reserved value. Then monitors for 7 bytes all ones to decide Soft Reset action. The slave should activate the output data line. Waits for deactivation for the received line. The slave then deactivates the output data line and returns to the CMD/DATA start monitor state.
General NOTE	<ul style="list-style-type: none"> The slave should monitor the received line for command reception in any time When a CMD start is detected, the slave will receive 8 bytes then return again to the command reception state When the slave is transmitting data, it should also monitor for command reception When the slave is receiving data, it will monitor for command reception between the data packets Therefore issuing a soft reset command, should be detected in all cases

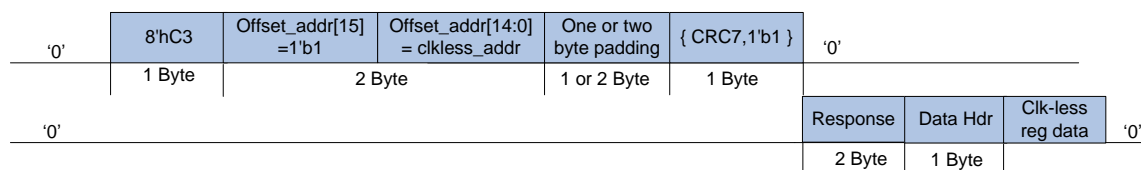
11.1.5 Clockless Registers Access

Clockless register access allows a host device to access registers on the WILC device while it is held in a reset state. This type of access can only be done using the “internal register read” and “internal register write” commands. For clockless access, bit 15 of the Offset_addr in the command should be ‘1’ to differentiate between clockless and clocked access mode.

For clock-less register write: The protocol master should wait for the response as shown below.



For clock-less register read: According to the interface, the protocol slave may not send CRC16. One or two byte padding depends on three or four byte DMA addresses.

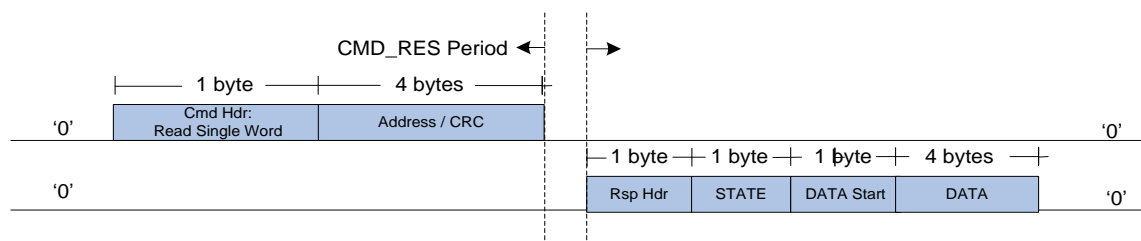


11.2 Message Flow for Basic Transactions

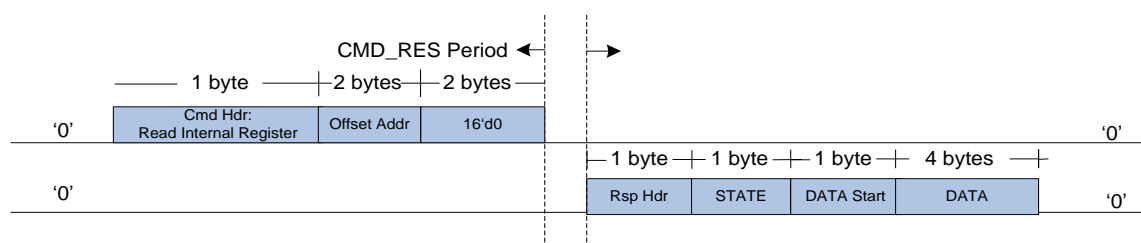
This section shows the essential message exchanges and timings associated with the following commands:

- Read Single Word
- Read Internal Register (clockless)
- Read Block
- Write Single Word
- Write Internal Register (clockless)
- Write Block

11.2.1 Read Single Word



11.2.2 Read Internal Register (for Clockless Registers)



11.2.3 Read Block

Normal Transaction

Master: Issues a DMA read transaction and waits for a response.

Slave: Sends a response after CMD_RES_PERIOD.

Master: Waits for a data packet start.

Slave: Sends the data packets, separated by DATA_DATA_PERIOD (see note below) where DATA_DATA_PERIOD is controlled by software and has one of these values:

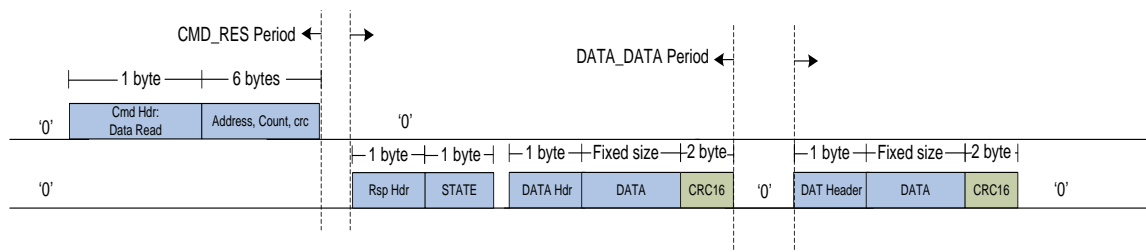
NO_DELAY (default), 4_BYTE_PERIOD, 8_BYTE_PERIOD and 16_BYTE_PERIOD

Slave: Continues sending till the count ends.

Master: Receive data packets. No response is sent for data packets but a termination/retransmit command may be sent if there is an error.

Note: Actually the period between data packets is "DATA_DATA_PERIOD + DMA access time." The master should monitor for DATA_START directly after DATA_DATA_PERIOD

The message sequence for this case is shown below:



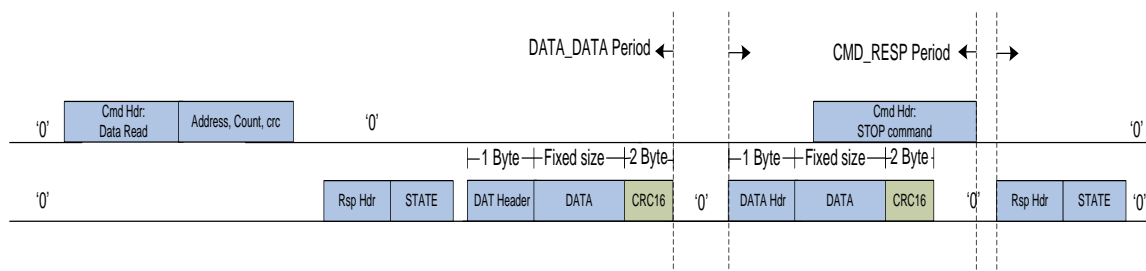
Termination Command Is Issued

Master: Can issue a termination command at any time during the transaction.

Master: Should monitor for RES_START after CMD_RESP_PERIOD.

Slave: Should cut off the current running data packet "if any".

Slave: Should respond to the termination command after CMD_RESP_PERIOD from the end of the termination command packet.



Repeat Command Is Issued

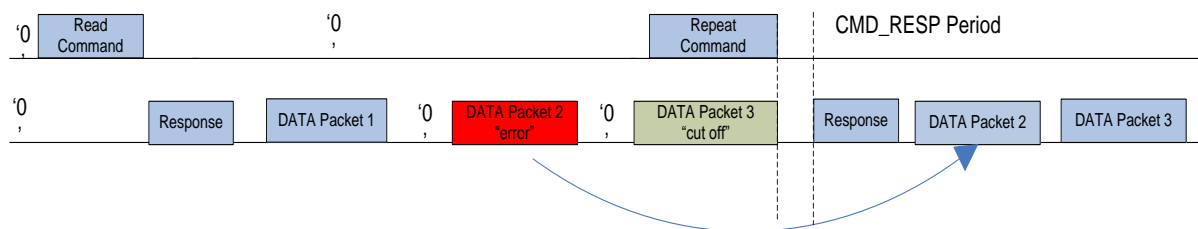
Master: Can issue a repeat command at any time during the transaction.

Master: Should monitor for RES_START after CMD_RESP_PERIOD.

Slave: Should cut off the current running data packet, if any.

Slave: should respond to the repeat command after CMD_RESP_PERIOD from the end of the repeat command packet.

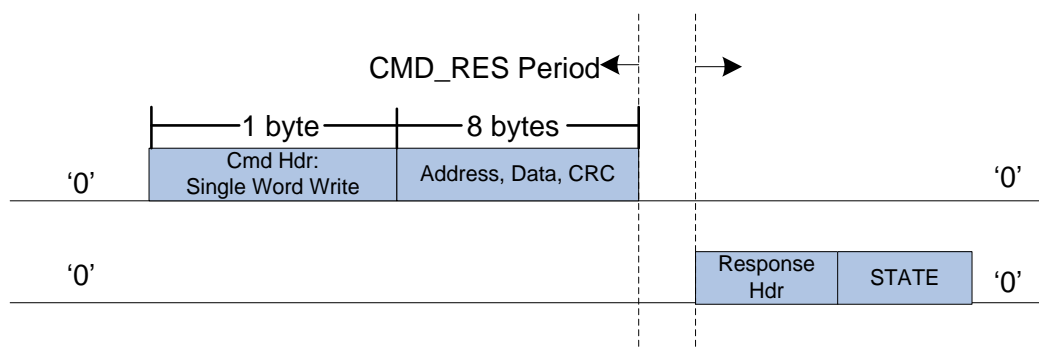
Slave: Resends the data packet that has an error then continues the transaction as normal.



11.2.4 Write Single Word

Master: Issues DMA single-word write command, including the data.

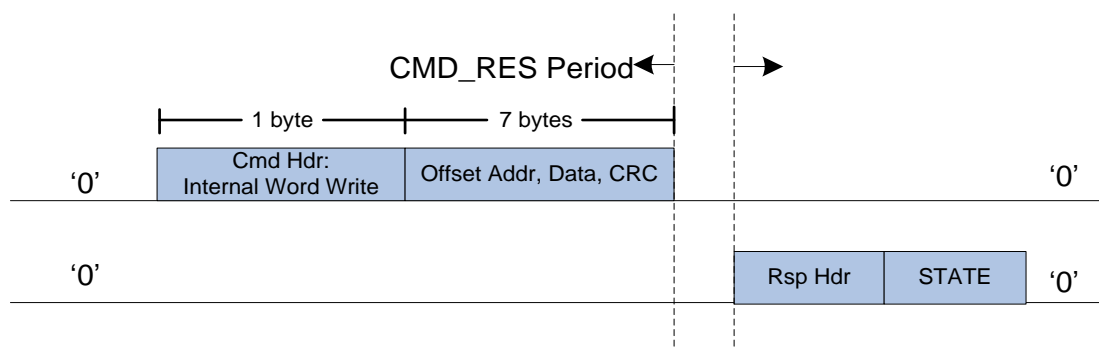
Slave: Takes the data and sends a command response.



11.2.5 Write Internal Register (for Clockless Registers)

Master: Issues an internal register write command, including the data.

Slave: Takes the data and sends a command response.



11.2.6 Write Block

Case 1: Master Waits for a Command Response

Master: Issues a DMA write command and waits for a response.

Slave: Sends response after CMD_RES_PERIOD.

Master: Sends the data packets after receiving response.

Slave: Sends a response packet for each data packet received after DATA_RES_PERIOD.

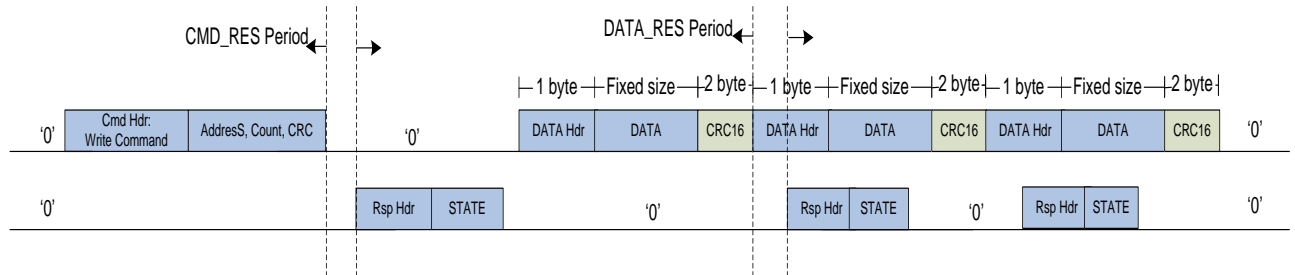
Master: Does not wait for the data response before sending the following data packet.

Note: CMD_RES_PERIOD is controlled by SW taking one of the values:

NO_DELAY (default), 1_BYTE_PERIOD, 2_BYTE_PERIOD and 3_BYTE_PERIOD.

The master should monitor for RES_START after CMD_RES_PERIOD.

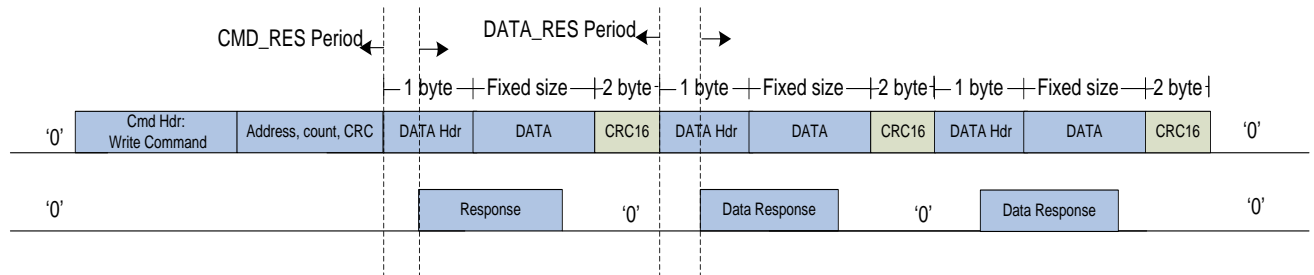
Note: DATA_RES_PERIOD is controlled by SW taking one of the values:
NO_DELAY (default), 1_BYTE_PERIOD, 2_BYTE_PERIOD and 3_BYTE_PERIOD.



Case 2: Master does not wait for a command response:

Master: Sends the data packets directly after the command but it still monitors for a command response after CMD_RESP_PERIOD.

Master: Retransmits the data packets if there is an error in the command.



11.3 SPI Level Protocol Example

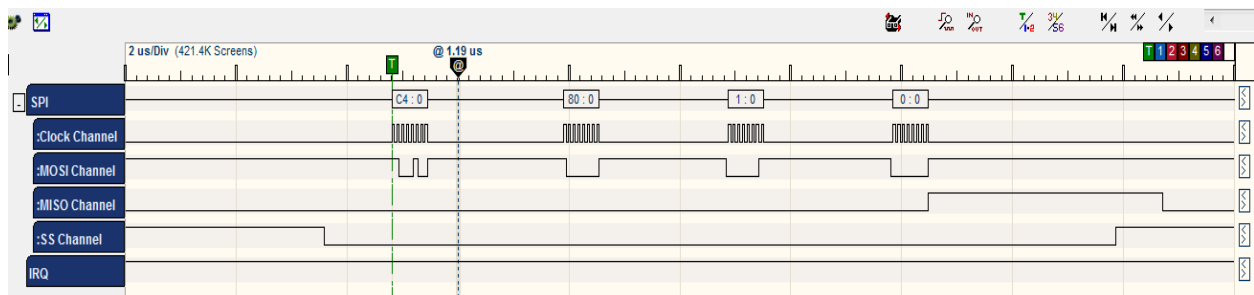
In order to illustrate how WILC SPI protocol works, SPI Bytes from the scan request example were dumped and the sequence is described below.

11.3.1 TX (Send Request)

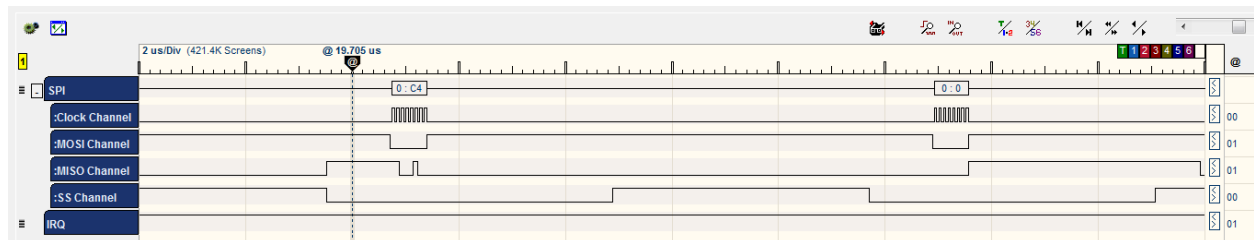
First step in `hif_send()` API is to wake up the chip:

```
sint8 nm_clkless_wake(void)
{
    ret = nm_read_reg_with_ret(0x1, &reg);
    /* Set bit 1 */
    ret = nm_write_reg(0x1, reg | (1 << 1));
    // Check the clock status
    ret = nm_read_reg_with_ret(clk_status_reg_adr, &clk_status_reg);
    // Tell Firmware that Host waked up the chip
    ret = nm_write_reg(WAKE_REG, WAKE_VALUE);
    return ret;
}
```

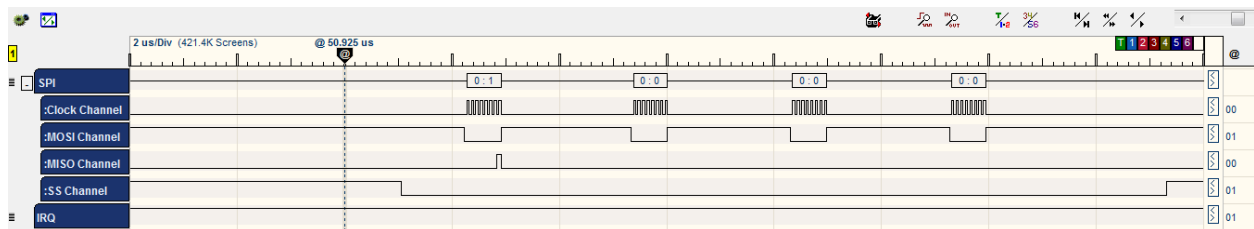
Command	<code>CMD_INTERNAL_READ: 0xC4</code>	<code>/* internal register read */</code>
	<code>BYTE [0] = CMD_INTERNAL_READ</code>	
	<code>BYTE [1] = address >> 8;</code>	<code>/* address = 0x01 */</code>
	<code>BYTE [1] = (1 << 7);</code>	<code>/* clockless register */</code>
	<code>BYTE [2] = address;</code>	
	<code>BYTE [3] = 0x00;</code>	



WILC acknowledges the command by sending three bytes [C4] [0] [F3].



Then WILC chip sends the value of the register 0x01 which equals 0x01.

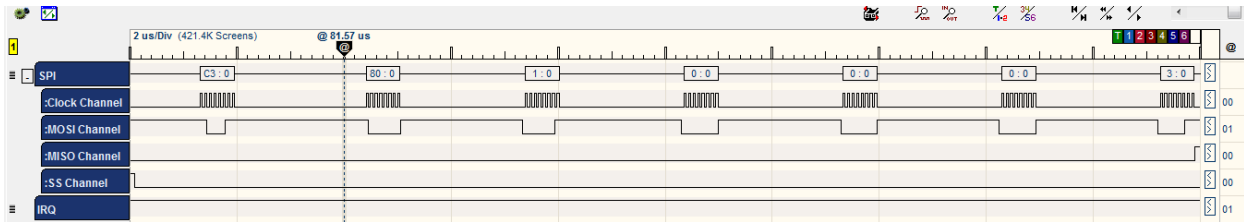


Command	<code>CMD_INTERNAL_WRITE: C3</code>	<code>/* internal register write */</code>
	<code>BYTE [0] = CMD_INTERNAL_WRITE</code>	
	<code>BYTE [1] = address >> 8;</code>	<code>/* address = 0x01 */</code>

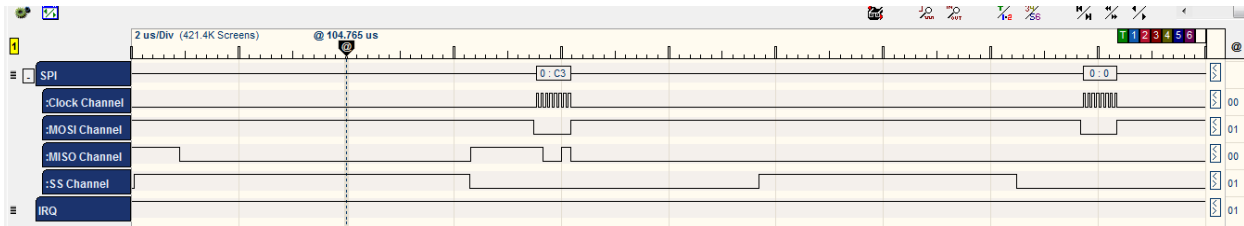

```

BYTE [1] |= (1 << 7);          /* clockless register */
BYTE [2] = address;
BYTE [3] = u32data >> 24;      /* Data = 0x03 */
BYTE [4] = u32data >> 16;
BYTE [5] = u32data >> 8;
BYTE [6] = u32data;

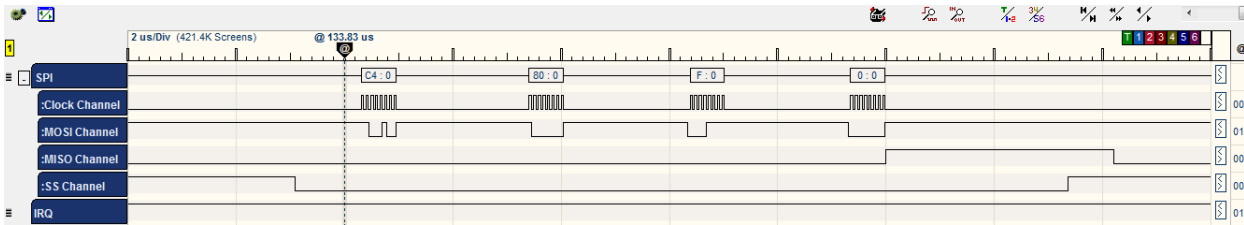
```



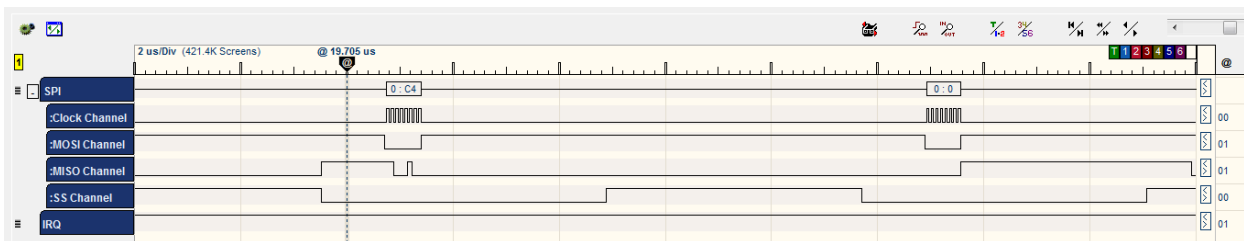
WILC acknowledges the command by sending 2 bytes [C3] [0].



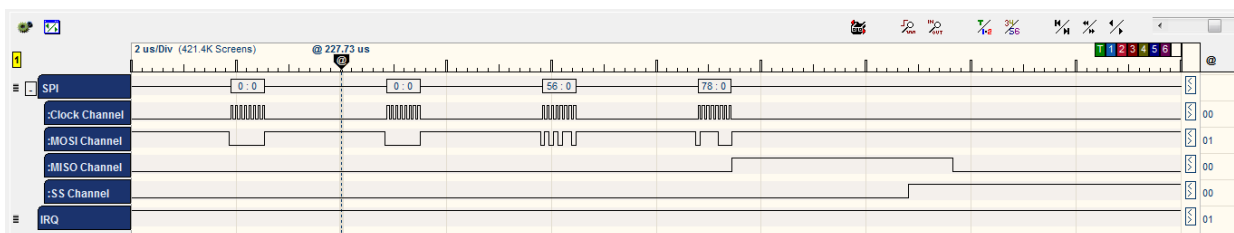
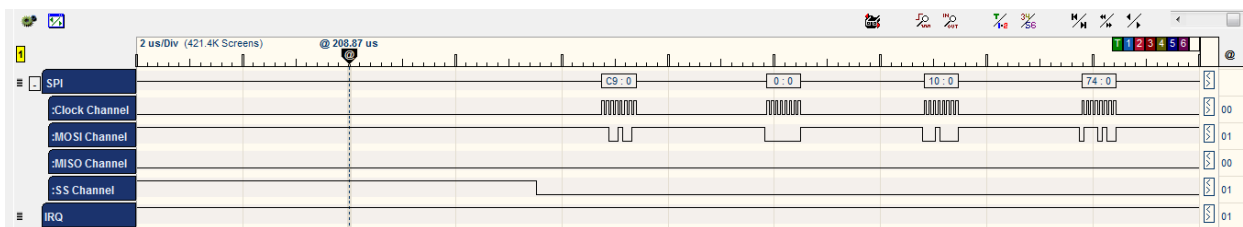
Command **CMD_INTERNAL_READ: 0xC4** /* internal register read */
 BYTE [0] = CMD_INTERNAL_READ
 BYTE [1] = address >> 8; /* address = 0x0F */
 BYTE [1] |= (1 << 7); /* clockless register */
 BYTE [2] = address;
 BYTE [3] = 0x00;



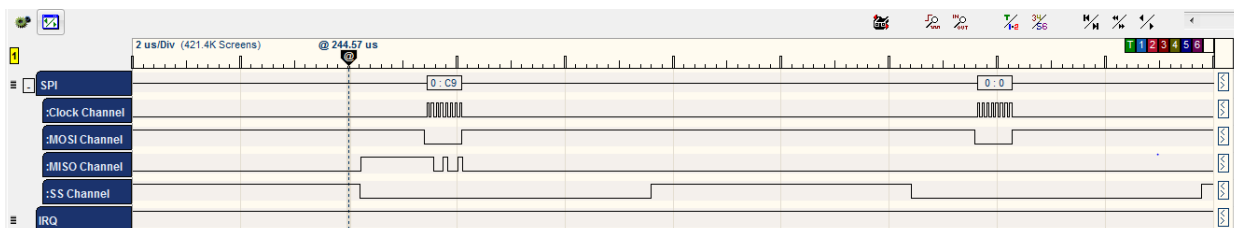
WILC acknowledges the command by sending 3 bytes [C4] [0] [F3].



Command **CMD_SINGLE_WRITE: 0xC9** /* single word write */
 BYTE [0] = CMD_SINGLE_WRITE
 BYTE [1] = address >> 16; /* WAKE_REG address = 0x1074 */
 BYTE [2] = address;
 BYTE [3] = address;
 BYTE [4] = u32data >> 24; /* WAKE_VALUE Data = 0x5678 */
 BYTE [5] = u32data >> 16;
 BYTE [6] = u32data >> 8;
 BYTE [7] = u32data;



The chip acknowledges the command by sending 2 bytes [C9] [0].



At this point, HIF finishes executing the clockless wakeup of the WILC chip.

The HIF layer Prepares and Sets the HIF layer header to NMI_STATE_REG register (4 | 8 Byte header describing the packet to be sent).

Set BIT [1] of WIFI_HOST_RCV_CTRL_2 register to raise an interrupt to the chip.

```

sint8 hif_send(uint8 u8Gid, uint8 u8Opcode, uint8 *pu8CtrlBuf, uint16 u16CtrlBufSize,
               uint8 *pu8DataBuf, uint16 u16DataSize, uint16 u16DataOffset)
{
    volatile tstrHifHdr strHif;
    volatile uint32 reg;
    strHif.u8Opcode = u8Opcode & (~NBIT7);
    strHif.u8Gid = u8Gid;
    strHif.u16Length = M2M_HIF_HDR_OFFSET;
    strHif.u16Length += u16CtrlBufSize;
    ret = nm_clkless_wake();

    reg = 0UL;
    reg |= (uint32)u8Gid;
    reg |= ((uint32)u8Opcode << 8);
    reg |= ((uint32)strHif.u16Length << 16);
    ret = nm_write_reg(NMI_STATE_REG, reg);
    reg = 0;
    reg |= (1 << 1);
    ret = nm_write_reg(WIFI_HOST_RCV_CTRL_2, reg);
}

```

```

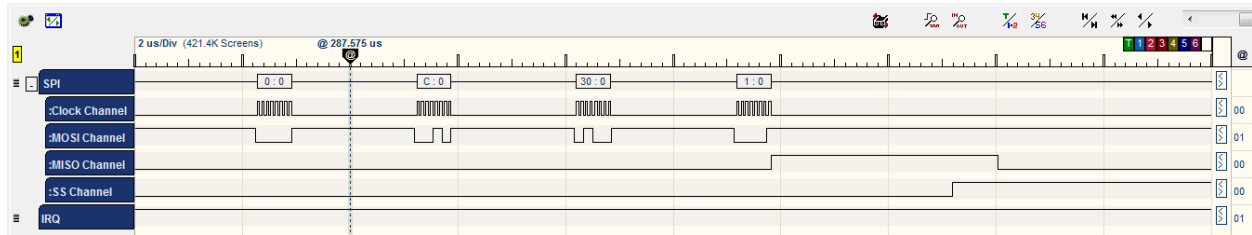
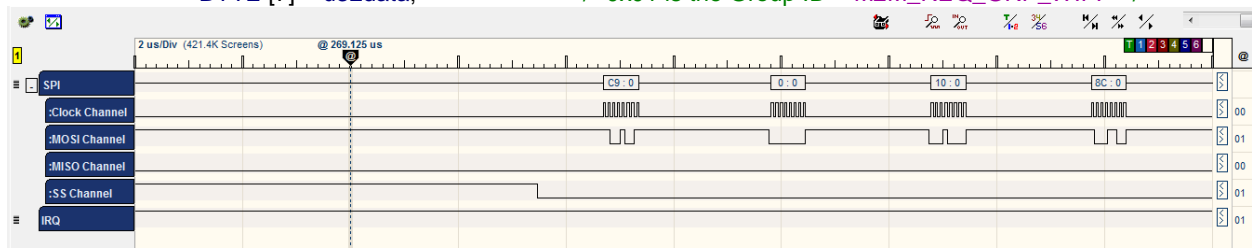
Command    CMD_SINGLE_WRITE:0XC9          /* single word write */
           BYTE [0] = CMD_SINGLE_WRITE
           BYTE [1] = address >> 16;      /* NMI_STATE_REG address = 0x180c */
           BYTE [2] = address >> 8;
           BYTE [3] = address;
           BYTE [4] = u32data >> 24;      /* Data = 0x000C3001 */
           BYTE [5] = u32data >> 16;      /* 0x0C is the length and equals 12 */

```

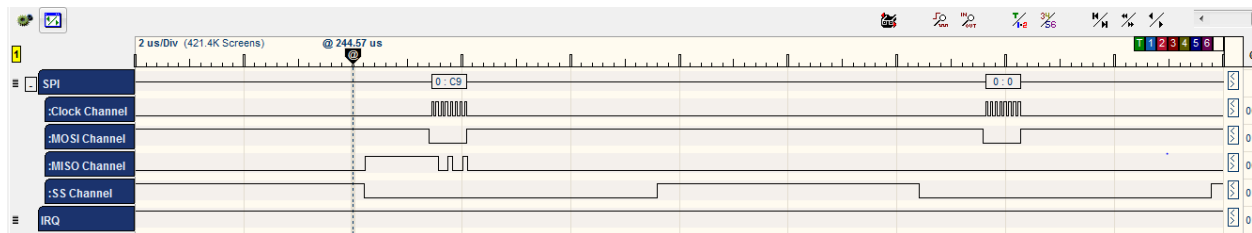
```

*/
BYTE [6] = u32data >> 8;          /* 0x30 is the Opcode=M2M_WIFI_REQ_SET_SCAN_REGION
BYTE [7] = u32data;                /* 0x01 is the Group ID = M2M_REQ_GRP_WIFI */

```



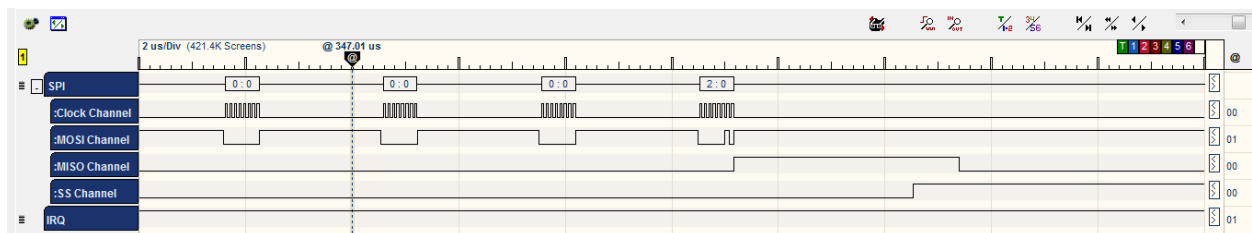
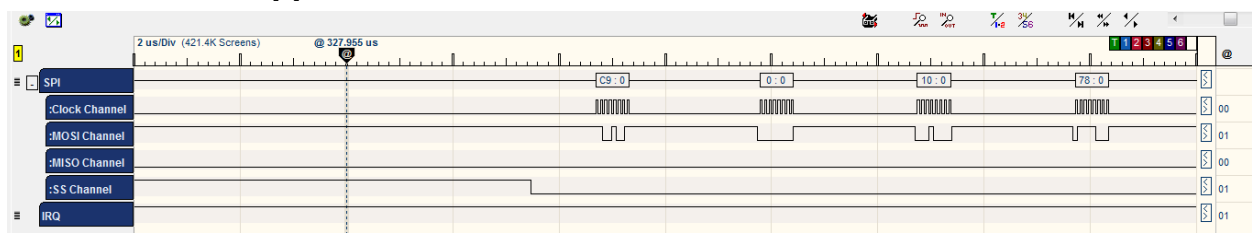
WILC acknowledges the command by sending two bytes [C9] [0].



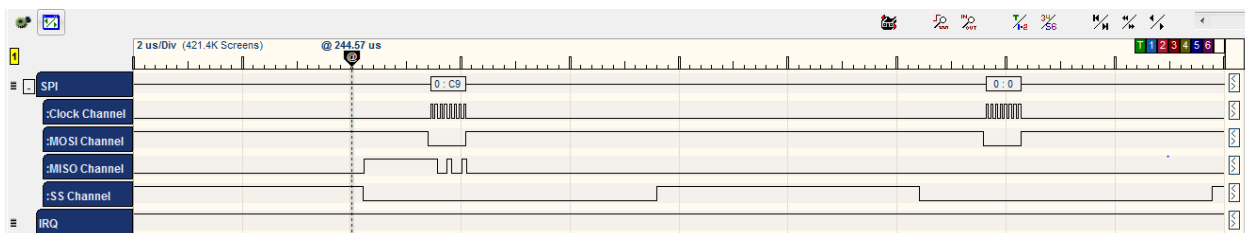
```

Command      CMD_SINGLE_WRITE:0XC9      /* single word write */
BYTE [0] = CMD_SINGLE_WRITE
BYTE [1] = address >> 16;              /* WIFI_HOST_RCV_CTRL_2address = 0x1087*/
BYTE [2] = address >> 8;
BYTE [3] = address;
BYTE [4] = u32data >> 24;              /* Data = 0x02 */
BYTE [5] = u32data >> 16;
BYTE [6] = u32data >> 8;
BYTE [7] = u32data;

```



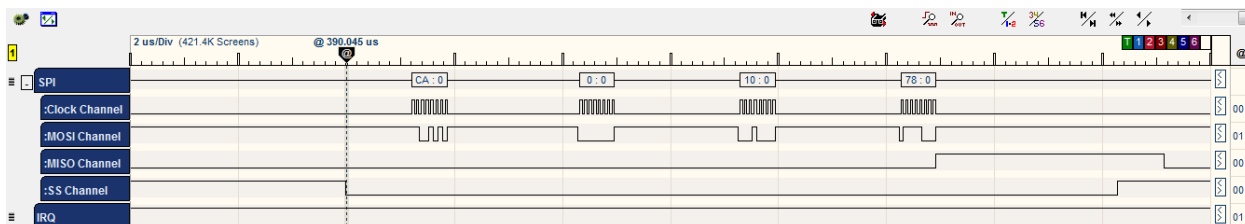
WILC acknowledges the command by sending two bytes [C9] [0].



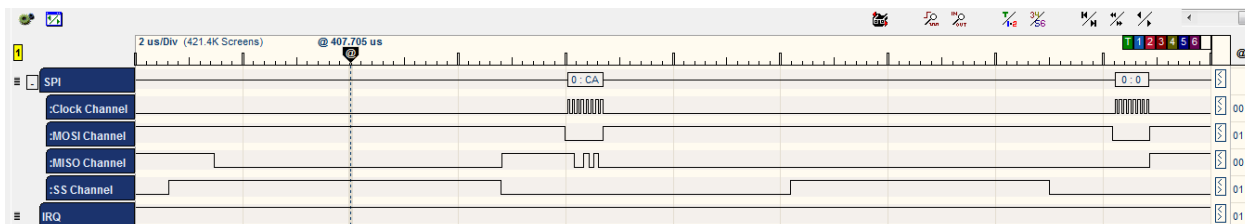
Then HIF polls for DMA address.

```
for (cnt = 0; cnt < 1000; cnt++)
{
    ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_2,(uint32 *)&reg);
    if(ret != M2M_SUCCESS) break;
    if (!(reg & 0x2))
    {
        ret = nm_read_reg_with_ret(0x150400,(uint32 *)&dma_addr);
        /*in case of success break */
        break;
    }
}
```

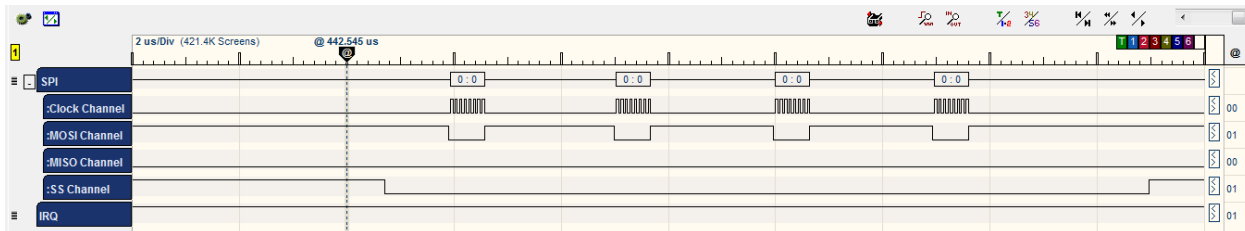
Command **CMD_SINGLE_READ: 0xCA** /* single word (4 bytes) read */
BYTE [0] = CMD_SINGLE_READ
BYTE [1] = address >> 16; /* WIFI_HOST_RCV_CTRL_2 address = 0x1078 */
BYTE [2] = address >> 8;
BYTE [3] = address;



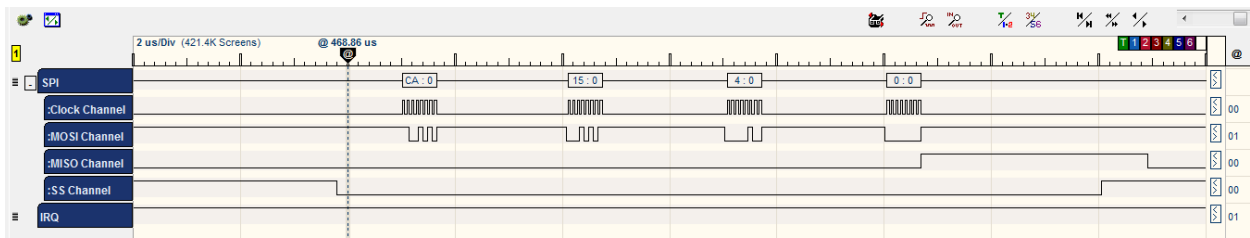
WILC acknowledges the command by sending three bytes [CA] [0] [F3].



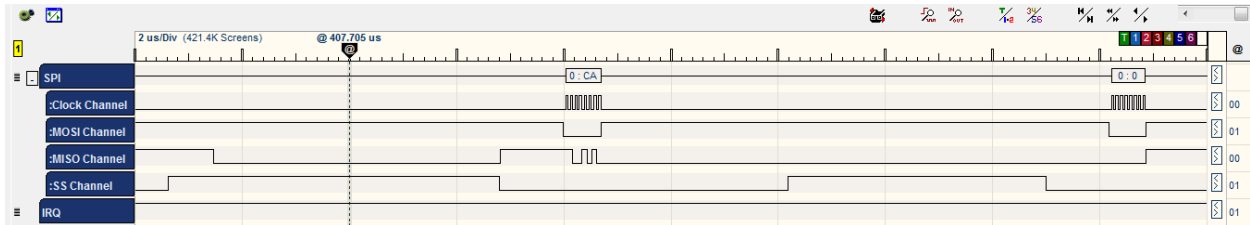
Then WILC chip send the value of the register 0x1078 which equals 0x00.



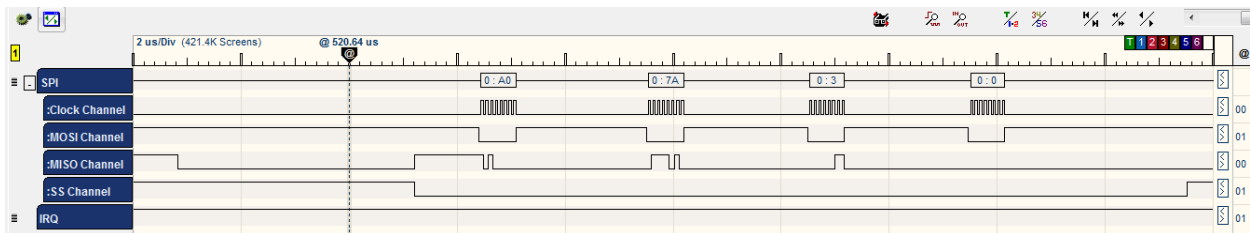
Command **CMD_SINGLE_READ: 0xCA** /* single word (4 bytes) read */
BYTE [0] = CMD_SINGLE_READ
BYTE [1] = address >> 16; /* address = 0x1504 */
BYTE [2] = address >> 8;
BYTE [3] = address;



WILC acknowledges the command by sending three bytes [CA] [0] [F3].



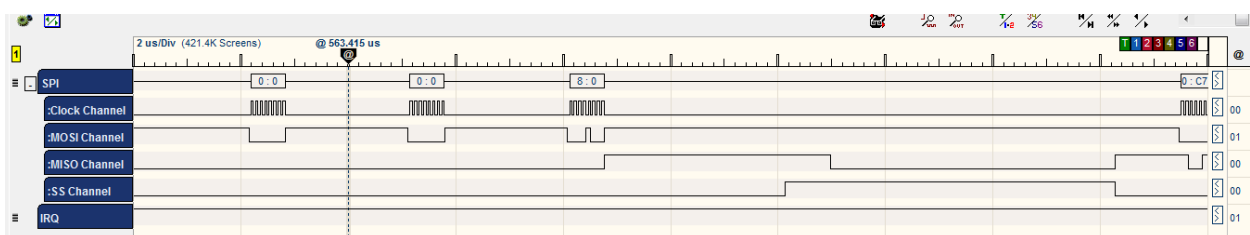
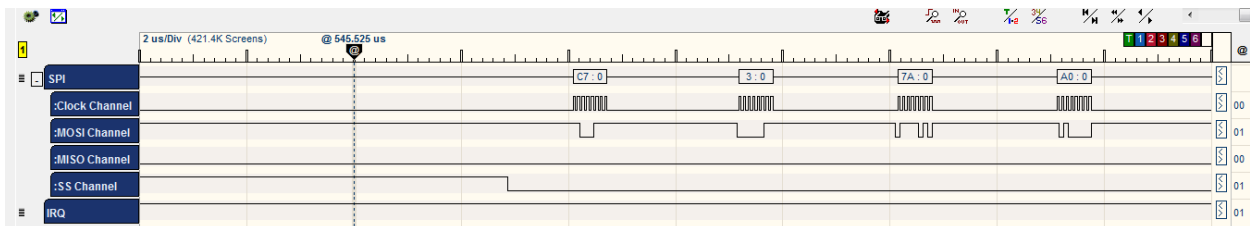
Then WILC chip send the value of the register 0x1504 which equals 0x037AA0.



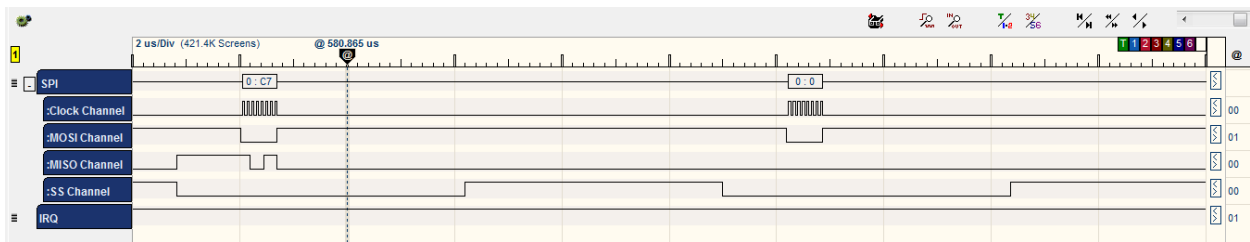
WILC writes the HIF header to the DMA memory address.

```
u32CurrAddr = dma_addr;
strHif.u16Length=NM_BSP_B_L_16(strHif.u16Length);
ret = nm_write_block(u32CurrAddr, (uint8*)&strHif, M2M_HIF_HDR_OFFSET);
```

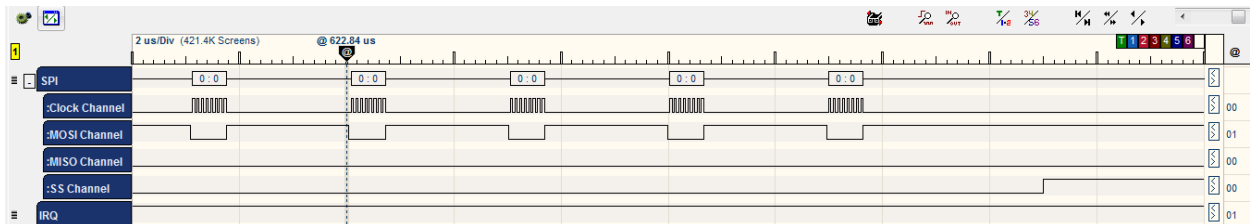
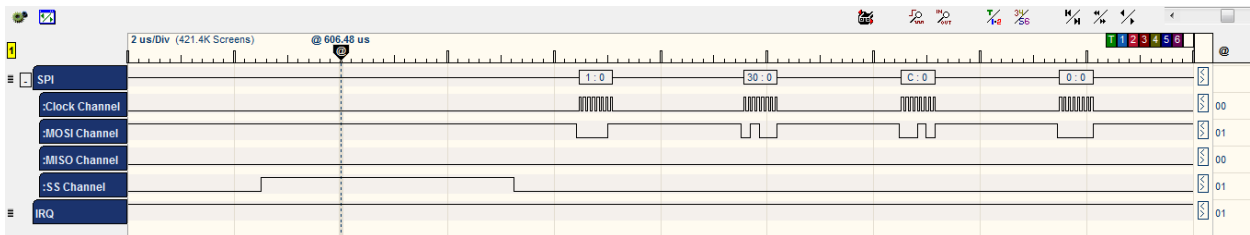
```
Command    CMD_DMA_EXT_WRITE:0xC7          /* DMA extended write */
           BYTE [0] = CMD_DMA_EXT_WRITE
           BYTE [1] = address >> 16; /* address = 0x037AA0 */
           BYTE [2] = address >> 8;
           BYTE [3] = address;
           BYTE [4] = size >> 16;
           BYTE [5] = size >> 8;      /* size = 0x08 */
           BYTE [6] = size;
```



WILC acknowledges the command by sending three bytes [C7] [0] [F3].



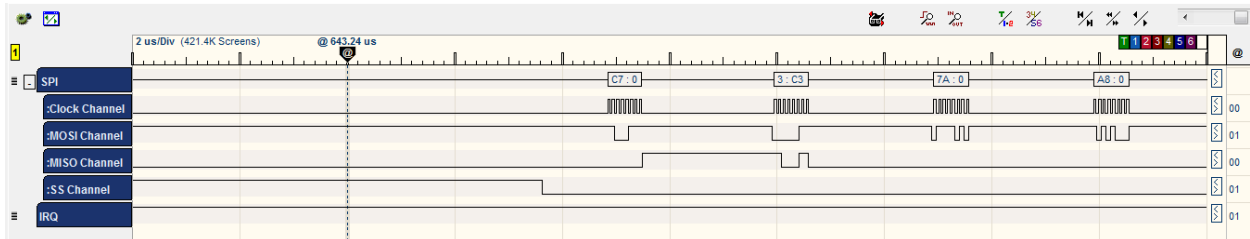
The HIF layer writes the Data.

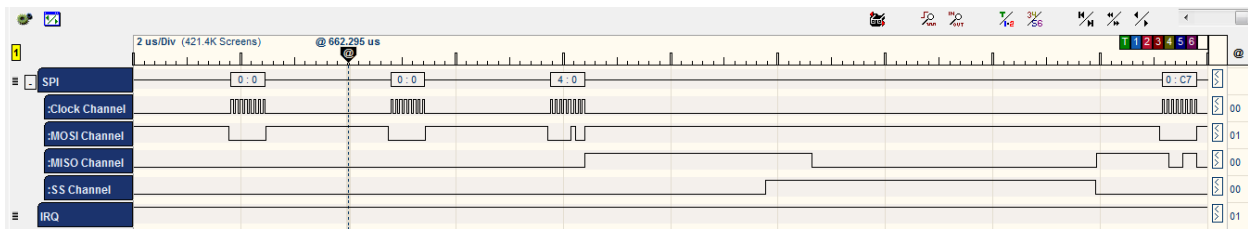


HIF writes the Control Buffer data (part of the framing of the request).

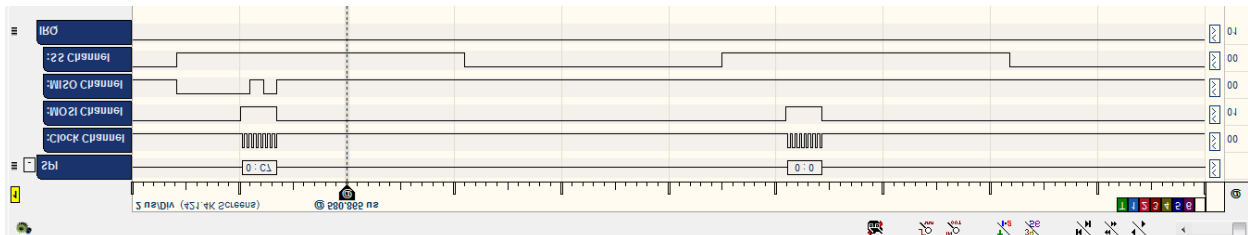
```
if (pu8CtrlBuf != NULL)
{
    ret = nm_write_block(u32CurrAddr, pu8CtrlBuf, u16CtrlBufSize);
    if (M2M_SUCCESS != ret) goto ERR1;
    u32CurrAddr += u16CtrlBufSize;
}
```

```
Command    CMD_DMA_EXT_WRITE:0xC7          /* DMA extended write */
BYTE [0] = CMD_DMA_EXT_WRITE
BYTE [1] = address >> 16;          /* address = 0x037AA8 */
BYTE [2] = address >> 8;
BYTE [3] = address;
BYTE [4] = size >> 16;              /* size = 0x04 */
BYTE [5] = size >> 8;
BYTE [6] = size;
```

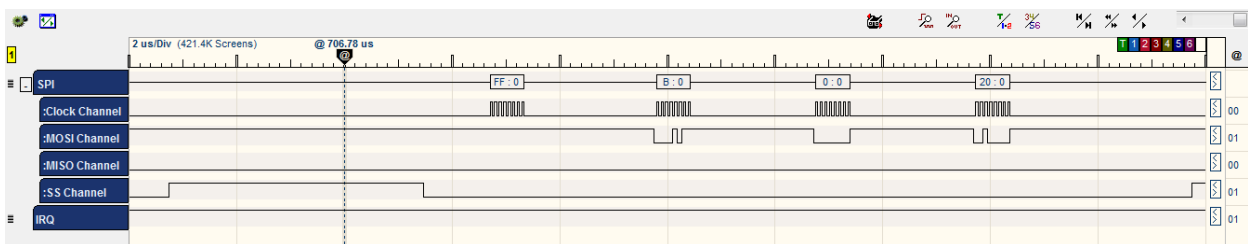




WILC acknowledges the command by sending three bytes [C7] [0] [F3].



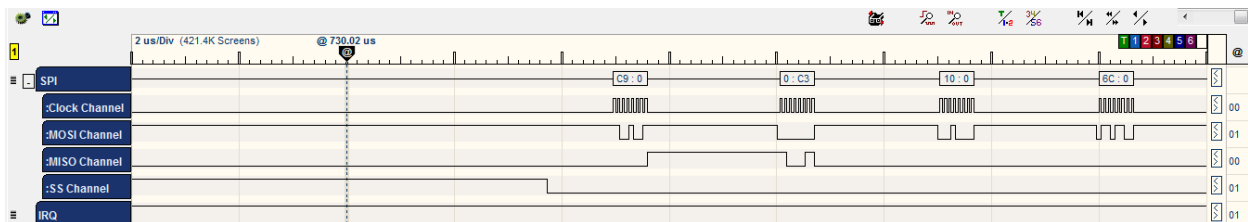
HIF layer writes the Data.

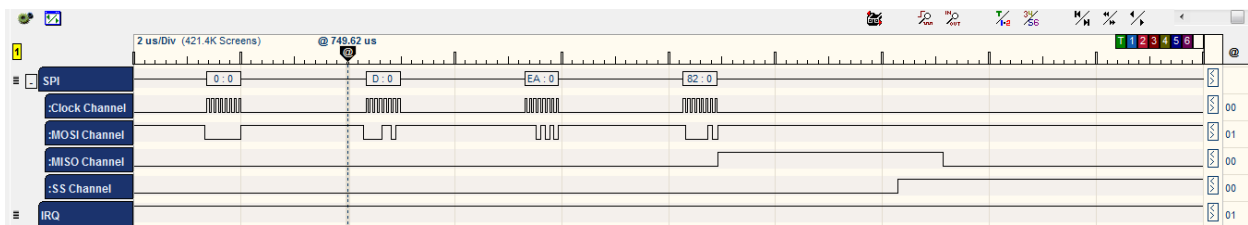


Finally, HIF finished writing the request data to memory and is going to interrupt the chip announcing that host TX is done.

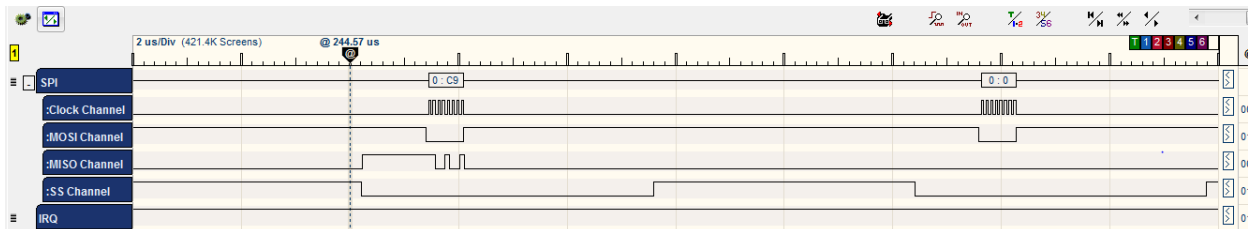
```
reg = dma_addr << 2;
reg |= (1 << 1);
ret = nm_write_reg(WIFI_HOST_RCV_CTRL_3, reg);
```

Command **CMD_SINGLE_WRITE:0XC9** /* single word write */
 BYTE [0] = CMD_SINGLE_WRITE
 BYTE [1] = address >> 16; /* WIFI_HOST_RCV_CTRL_3 address = 0x106C */
 BYTE [2] = address >> 8;
 BYTE [3] = address;
 BYTE [4] = u32data >> 24; /* Data = 0x000DEA82 */
 BYTE [5] = u32data >> 16;
 BYTE [6] = u32data >> 8;
 BYTE [7] = u32data;





WILC acknowledges the command by sending two bytes [C9] [0].



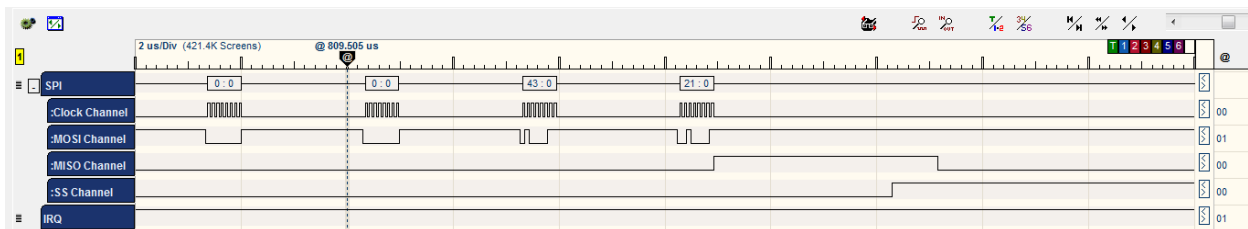
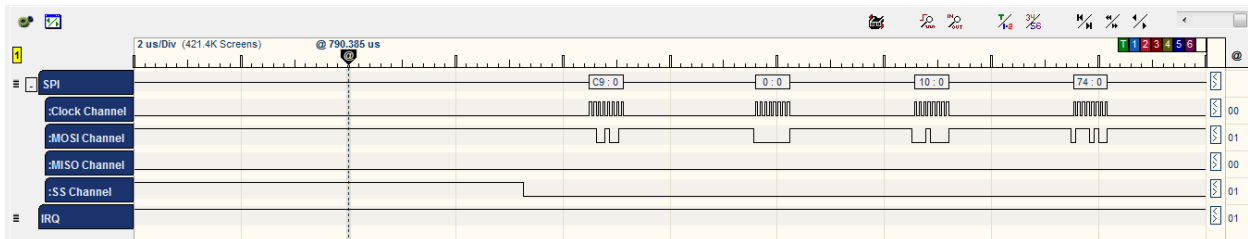
HIF layer allows the chip to enter sleep mode again.

```

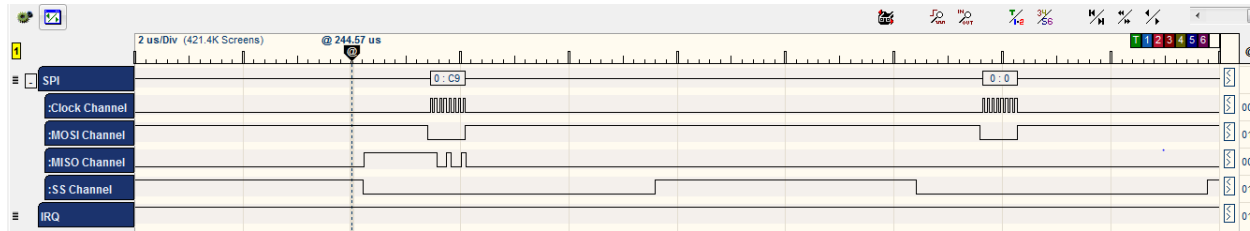
sint8 hif_chip_sleep(void)
{
    sint8 ret = M2M_SUCCESS;
    uint32 reg = 0;
    ret = nm_write_reg(WAKE_REG, SLEEP_VALUE);
    /* Clear bit 1 */
    ret = nm_read_reg_with_ret(0x1, &reg);
    if(reg&0x2)
    {
        reg &= ~(1 << 1);
        ret = nm_write_reg(0x1, reg);
    }
}

```

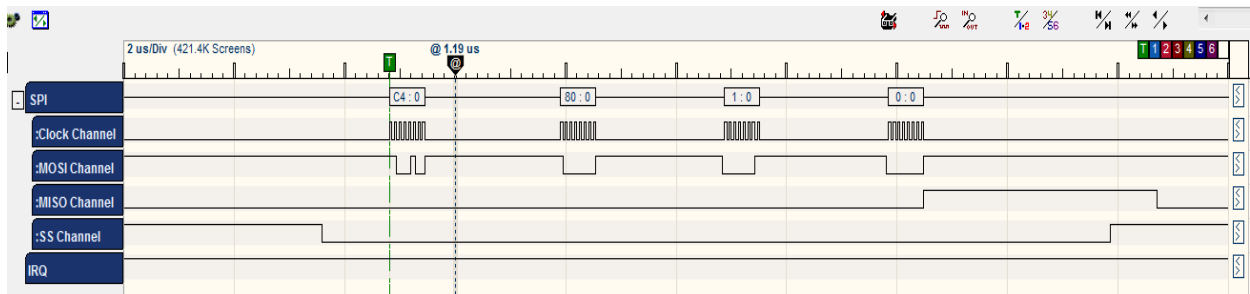
Command **CMD_SINGLE_WRITE:0XC9** /* single word write */
 BYTE [0] = **CMD_SINGLE_WRITE**
 BYTE [1] = **address >> 16;** /* **WAKE_REG** address = 0x1074 */
 BYTE [2] = **address >> 8;**
 BYTE [3] = **address;**
 BYTE [4] = **u32data >> 24;** /* **SLEEP_VALUE** Data = 0x4321 */
 BYTE [5] = **u32data >> 16;**
 BYTE [6] = **u32data >> 8;**
 BYTE [7] = **u32data;**



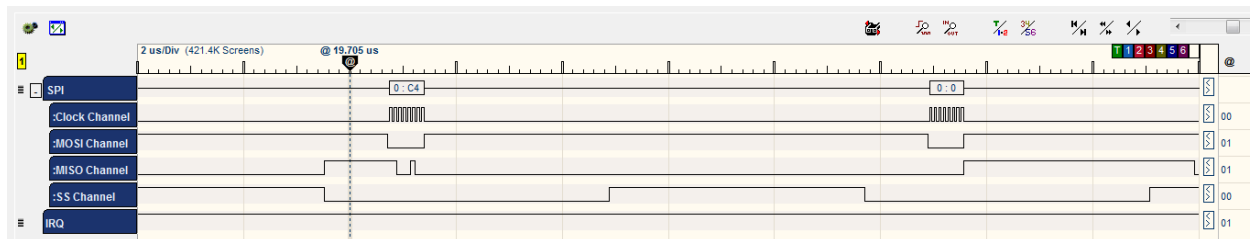
WILC acknowledges the command by sending two bytes [C9] [0].



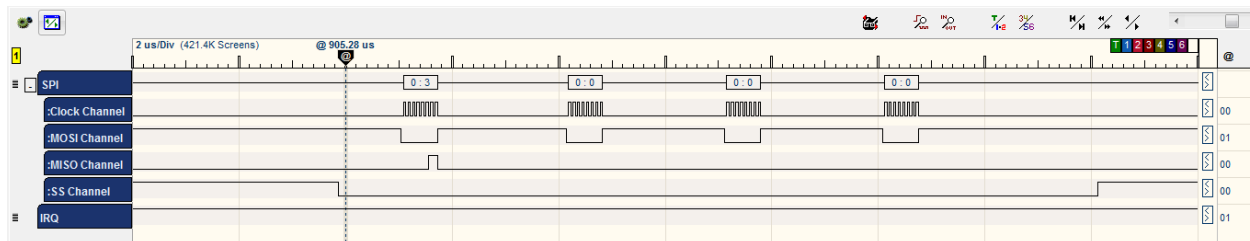
```
Command      CMD_INTERNAL_READ:      0xC4      /* internal register read */
BYTE [0] = CMD_INTERNAL_READ
BYTE [1] = address >> 8;          /* address = 0x01 */
BYTE [1] |= (1 << 7);            /* clockless register */
BYTE [2] = address;
BYTE [3] = 0x00;
```



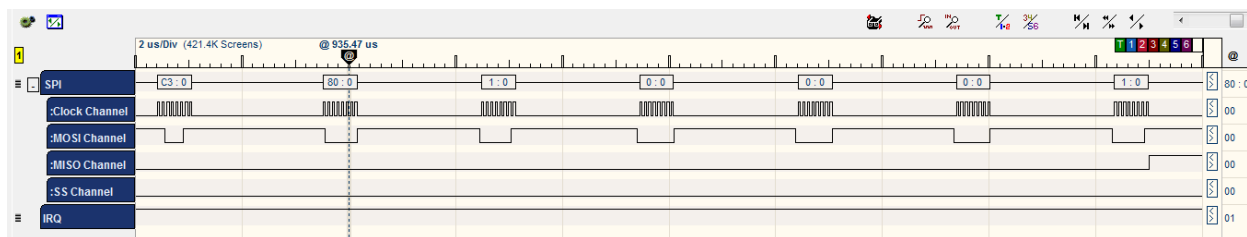
WILC acknowledges the command by sending three bytes [C4] [0] [F3].



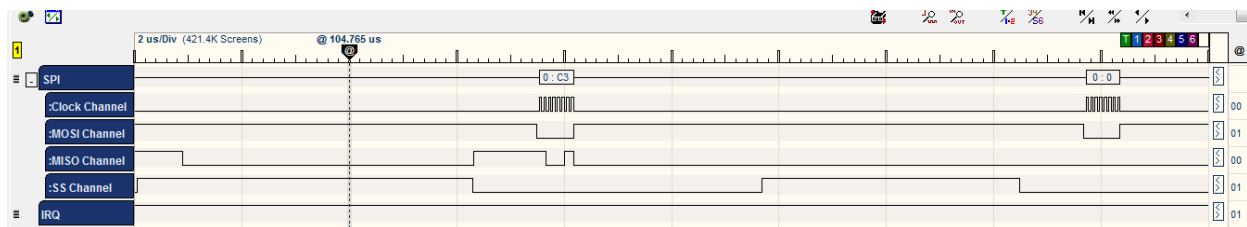
Then WILC chip sends the value of the register 0x01 which equals 0x03.



```
Command      CMD_INTERNAL_WRITE:    C3      /* internal register write */
BYTE [0] = CMD_INTERNAL_WRITE
BYTE [1] = address >> 8;          /* address = 0x01 */
BYTE [1] |= (1 << 7);            /* clockless register */
BYTE [2] = address;
BYTE [3] = u32data >> 24;        /* Data = 0x01 */
BYTE [4] = u32data >> 16;
BYTE [5] = u32data >> 8;
BYTE [6] = u32data;
```



The WILC chip acknowledges the command by sending two bytes [C3] [0].



At this point, the HIF layer has finished posting the scan Wi-Fi request to the WILC chip and the request is being processed by the chip.

11.3.2 RX (Receive Response)

After finishing the required operation (scan Wi-Fi) the WILC will interrupt the Host announcing that the request has been processed.

Host will handle this interrupt to receive the response.

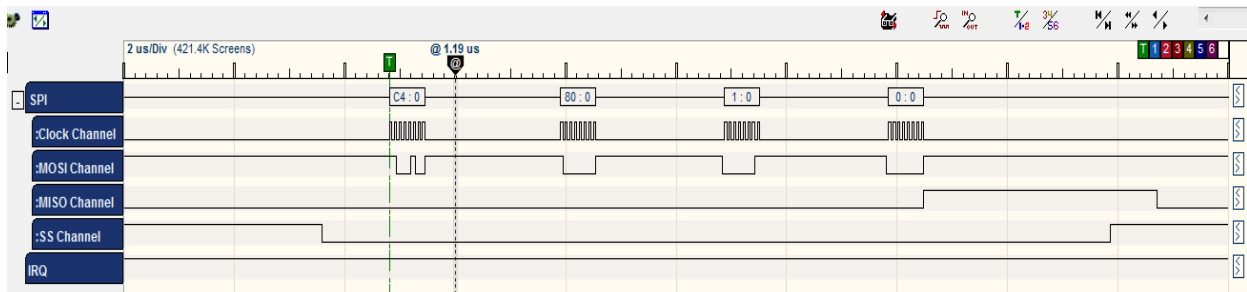
First step in `hif_isr ()` is to wake up the WILC chip.

```

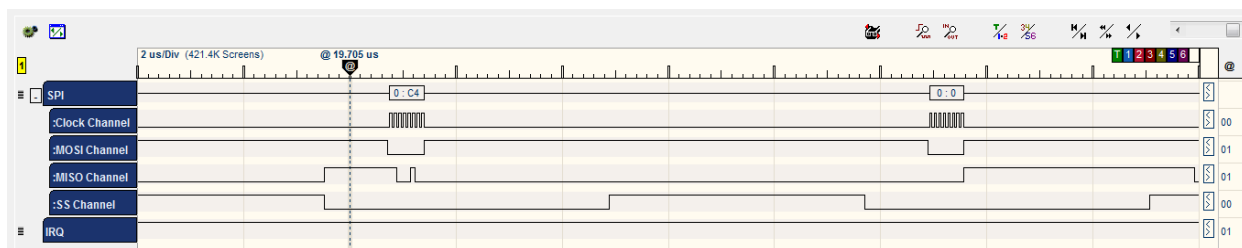
sint8 nm_clkless_wake(void)
{
    ret = nm_read_reg_with_ret(0x1, &reg);
    /* Set bit 1 */
    ret = nm_write_reg(0x1, reg | (1 << 1));
    // Check the clock status
    ret = nm_read_reg_with_ret(clk_status_reg_adr, &clk_status_reg);
    // Tell Firmware that Host waked up the chip
    ret = nm_write_reg(WAKE_REG, WAKE_VALUE);
    return ret;
}

```

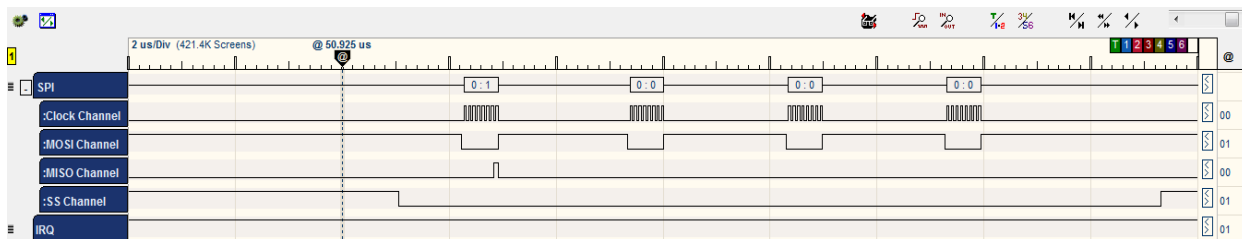
Command `CMD_INTERNAL_READ: 0xC4` /* internal register read */
`BYTE [0] = CMD_INTERNAL_READ`
`BYTE [1] = address >> 8;` /* address = 0x01 */
`BYTE [1] |= (1 << 7);` /* clockless register */
`BYTE [2] = address;`
`BYTE [3] = 0x00;`



WILC acknowledges the command by sending three bytes [C4] [0] [F3].



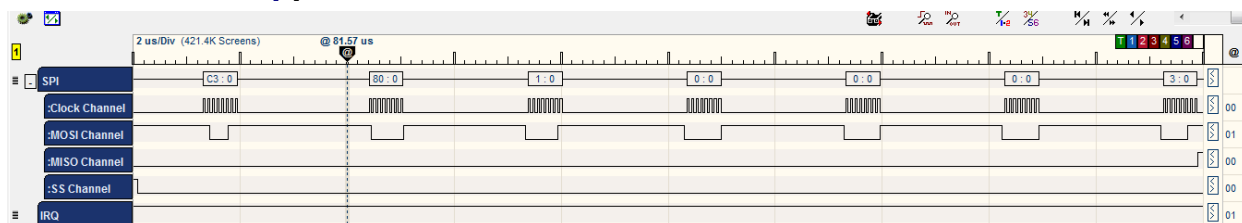
Then WILC chip sends the value of the register 0x01 which equals 0x01.



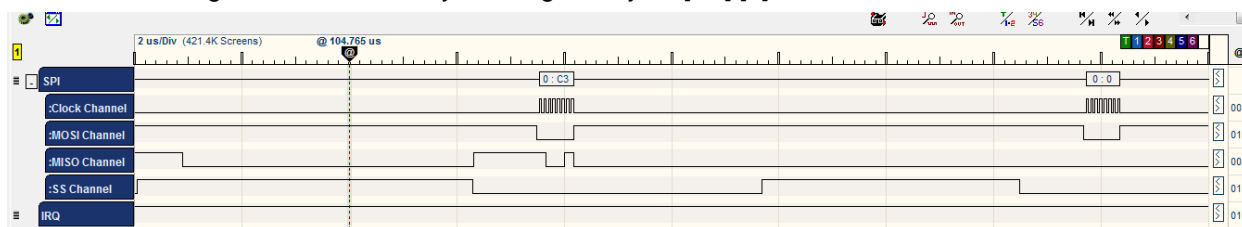
```

Command      CMD_INTERNAL_WRITE:      C3          /*      internal register write */
BYTE [0] = CMD_INTERNAL_WRITE
BYTE [1] = address >> 8;                /*      address = 0x01          */
BYTE [1] |= (1 << 7);                   /*      clockless register     */
BYTE [2] = address;
BYTE [3] = u32data >> 24;                /*      Data = 0x03            */
BYTE [4] = u32data >> 16;
BYTE [5] = u32data >> 8;
BYTE [6] = u32data;

```



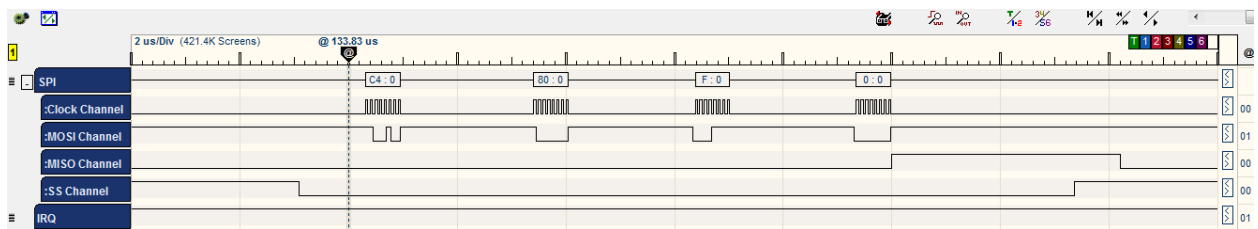
WILC acknowledges the command by sending two bytes [C3] [0].



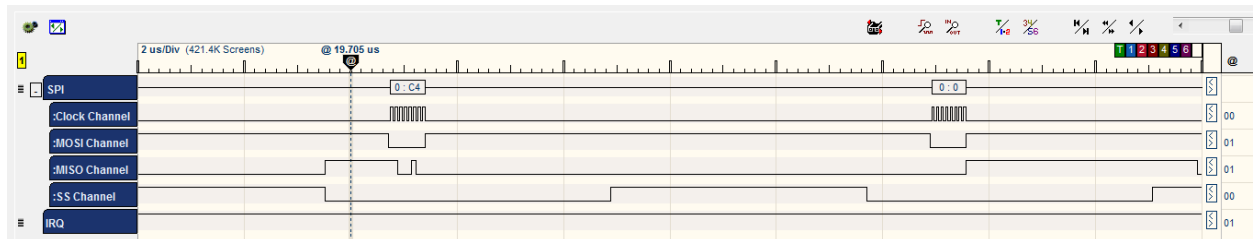
```

Command      CMD_INTERNAL_READ:      0xC4        /*      internal register read  */
BYTE [0] = CMD_INTERNAL_READ
BYTE [1] = address >> 8;                /*      address = 0x0F          */
BYTE [1] |= (1 << 7);                   /*      clockless register     */
BYTE [2] = address;
BYTE [3] = 0x00;

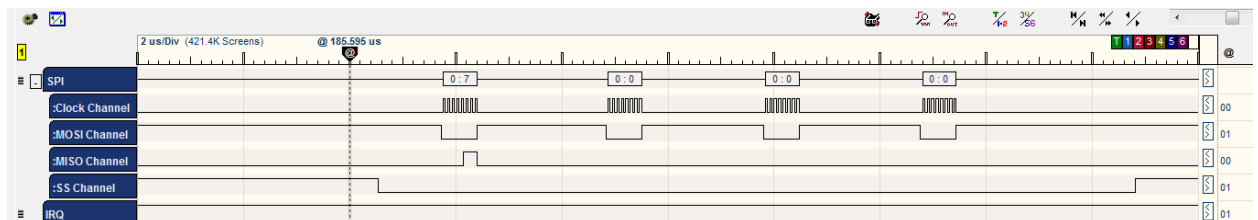
```



WILC acknowledges the command by sending three bytes [C4] [0] [F3].



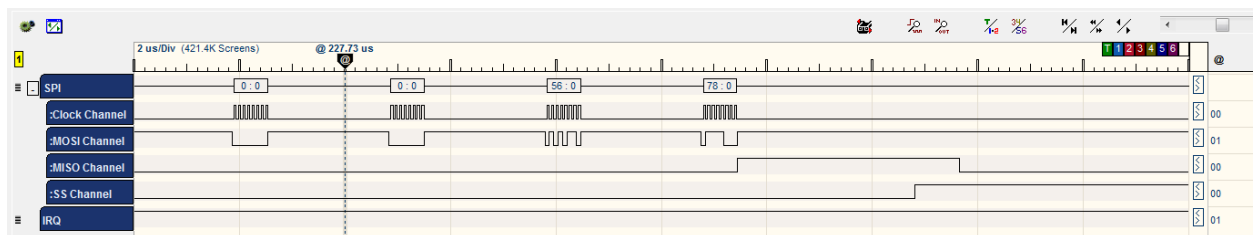
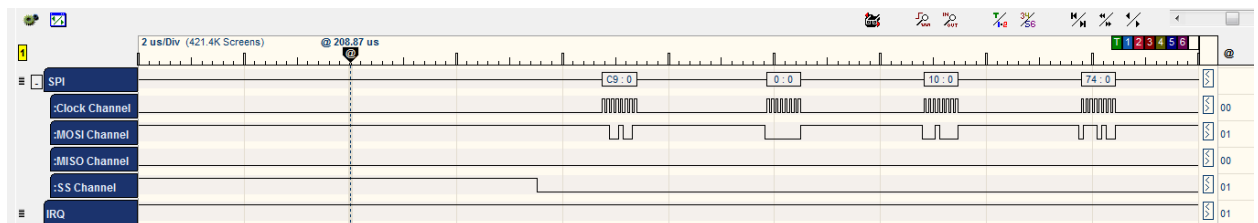
Then WILC chip sends the value of the register 0x01 which equals 0x07.



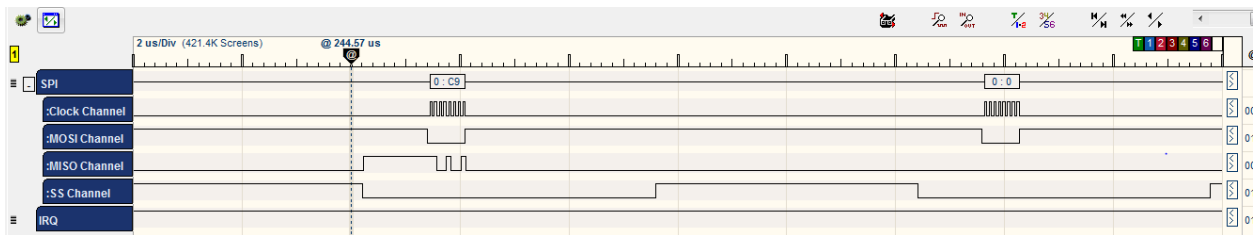
```

Command      CMD_SINGLE_WRITE:0XC9          /* single word write */
BYTE [0] = CMD_SINGLE_WRITE
BYTE [1] = address >> 16;           /* WAKE_REG address = 0x1074 */
BYTE [2] = address >> 8;
BYTE [3] = address;
BYTE [4] = u32data >> 24;           /* WAKE_VALUE Data = 0x5678 */
BYTE [5] = u32data >> 16;
BYTE [6] = u32data >> 8;
BYTE [7] = u32data;

```



The chip acknowledges the command by sending two bytes [C9] [0].

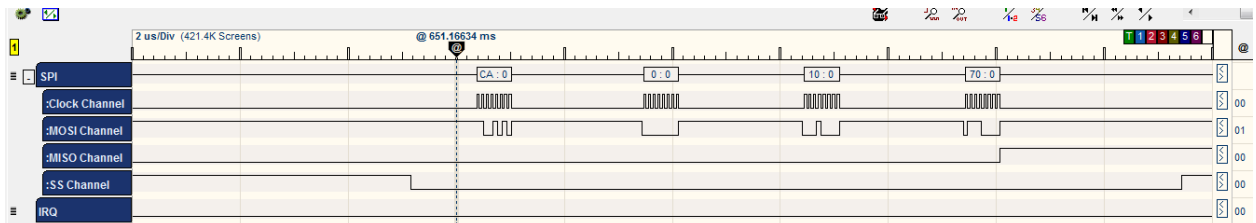


Read register WIFI_HOST_RCV_CTRL_0 to check if there is new interrupt, and if so, clear it (as it will be handled now).

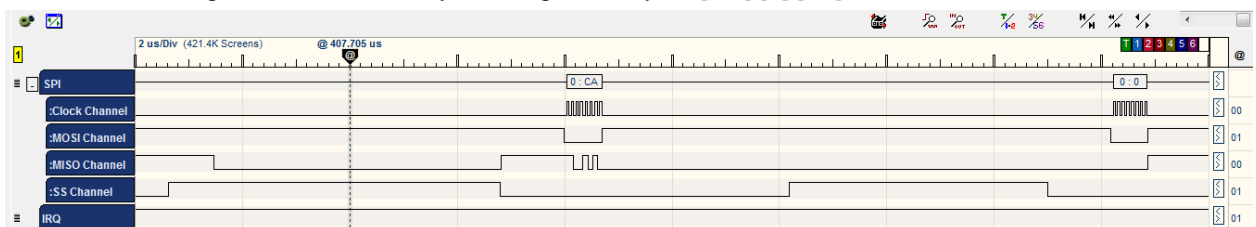
```
static sint8 hif_isr(void)
{
    sint8 ret ;
    uint32 reg;
    volatile tstrHifHdr strHif;

    ret = hif_chip_wake();
    ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_0, &reg);
    if(reg & 0x1) /* New interrupt has been received */
    {
        uint16 size;
        /*Clearing RX interrupt*/
        ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_0,&reg);
        reg &= ~(1<<0);
        ret = nm_write_reg(WIFI_HOST_RCV_CTRL_0,reg);
    }
}
```

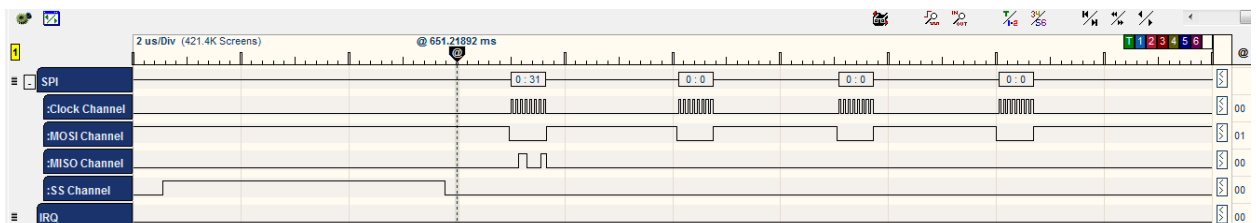
Command **CMD_SINGLE_READ: 0xCA** /* single word (4 bytes) read */
 BYTE [0] = **CMD_SINGLE_READ**
 BYTE [1] = address >> 16; /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
 BYTE [2] = address >> 8;
 BYTE [3] = address;



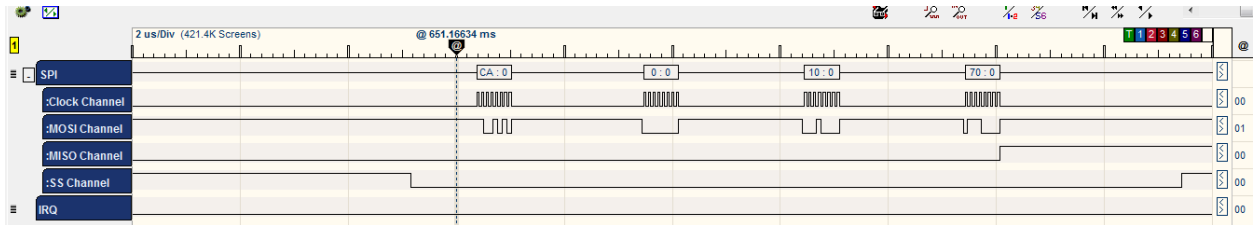
WILC acknowledges the command by sending three bytes [CA] [0] [F3].



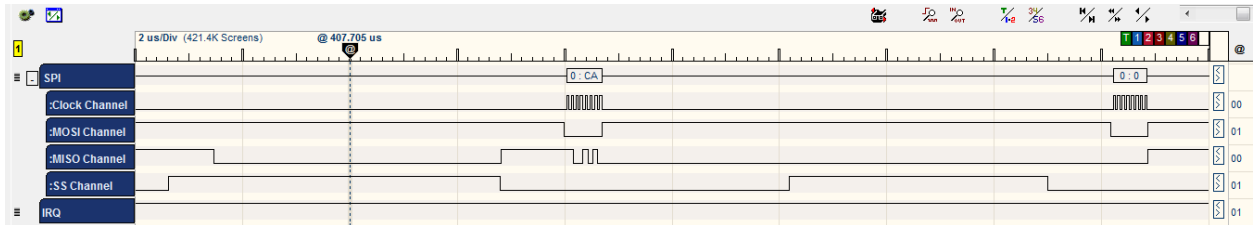
Then WILC chip sends the value of the register 0x1070 which equals 0x31.



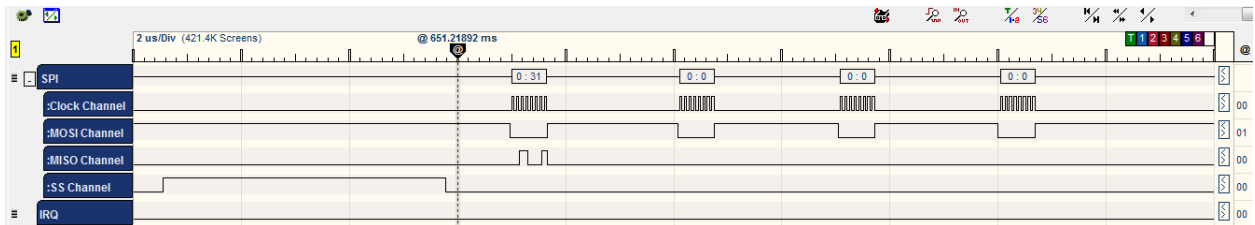
Command **CMD_SINGLE_READ: 0xCA** /* single word (4 bytes) read */
 BYTE [0] = CMD_SINGLE_READ
 BYTE [1] = address >> 16; /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
 BYTE [2] = address >> 8;
 BYTE [3] = address;



WILC acknowledges the command by sending three bytes [CA] [0] [F3].

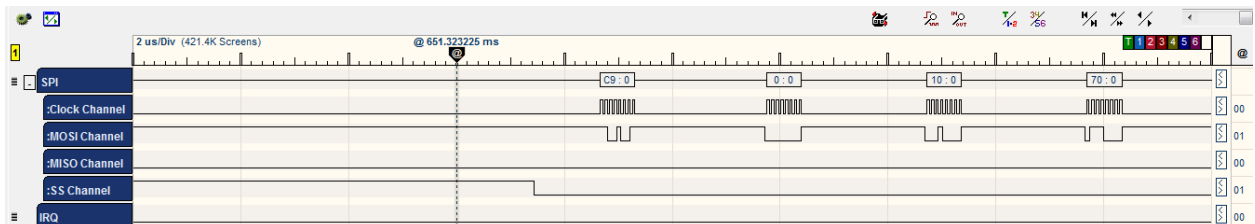


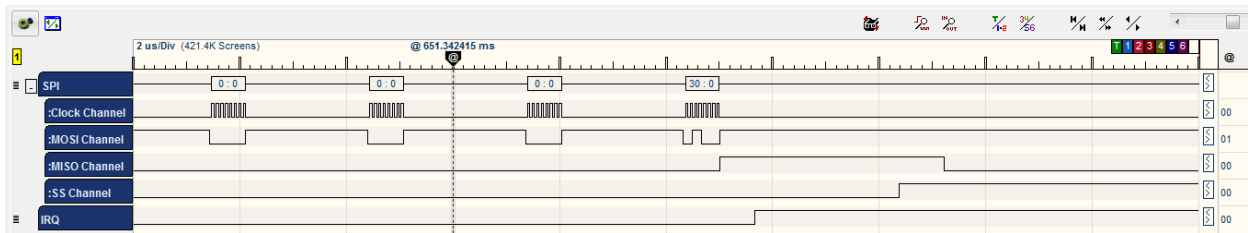
Then WILC chip sends the value of the register 0x1070 which equals 0x31.



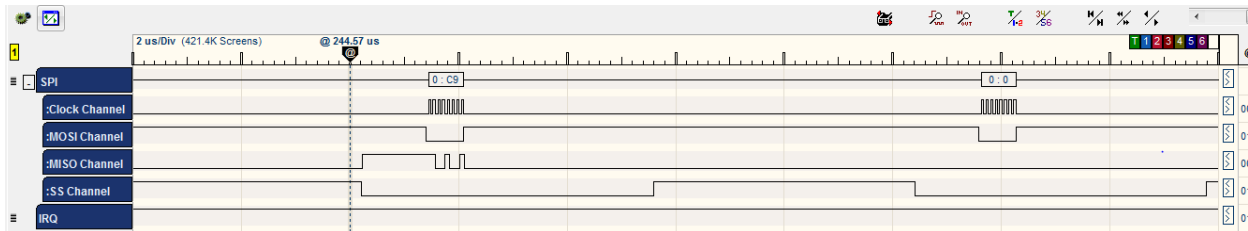
Clear the WILC Interrupt.

Command **CMD_SINGLE_WRITE: 0xC9** /* single word write */
 BYTE [0] = CMD_SINGLE_WRITE
 BYTE [1] = address >> 16; /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
 BYTE [2] = address >> 8;
 BYTE [3] = address;
 BYTE [4] = u32data >> 24; /* Data = 0x30 */
 BYTE [5] = u32data >> 16;
 BYTE [6] = u32data >> 8;
 BYTE [7] = u32data;





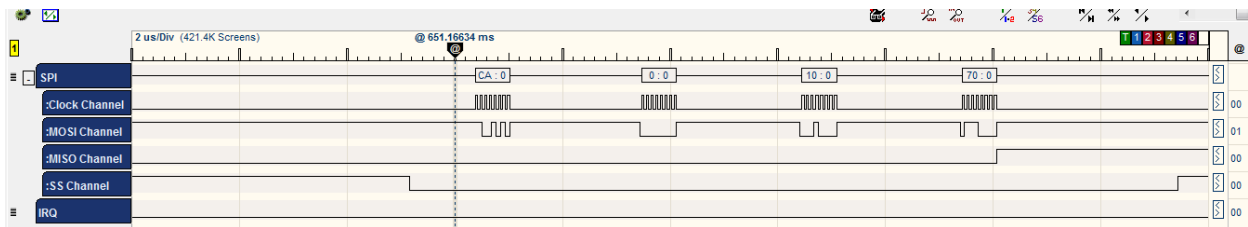
The chip acknowledges the command by sending two bytes [C9] [0].



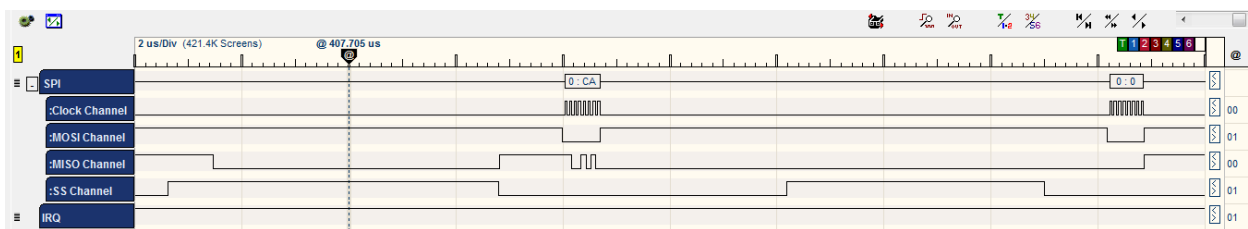
Then HIF reads the data size.

```
/* read the rx size */
ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_0, &reg);
```

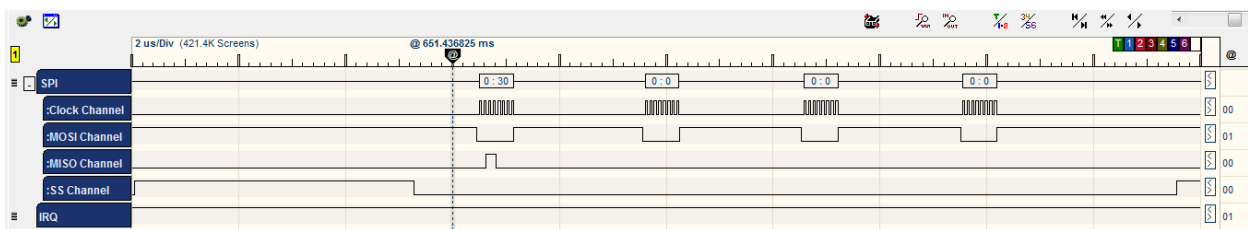
Command **CMD_SINGLE_READ: 0xCA** /* single word (4 bytes) read */
 BYTE [0] = CMD_SINGLE_READ
 BYTE [1] = address >> 16; /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
 BYTE [2] = address >> 8;
 BYTE [3] = address;



WILC acknowledges the command by sending three bytes [CA] [0] [F3].



Then WILC chip sends the value of the register 0x1070 which equals 0x30.



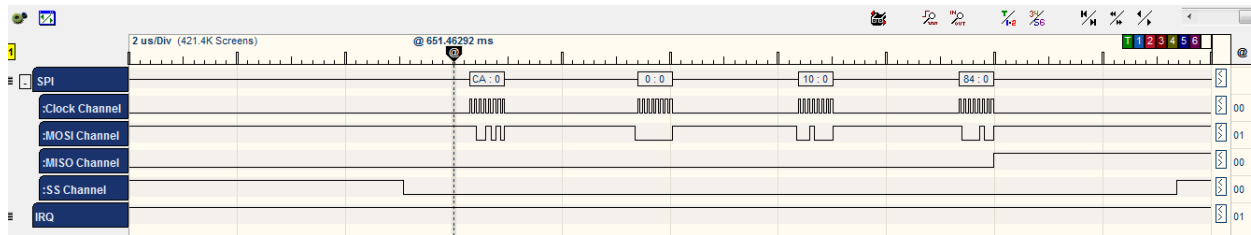
HIF reads hif header address.

```
/** start bus transfer**/
ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_1, &address);
```

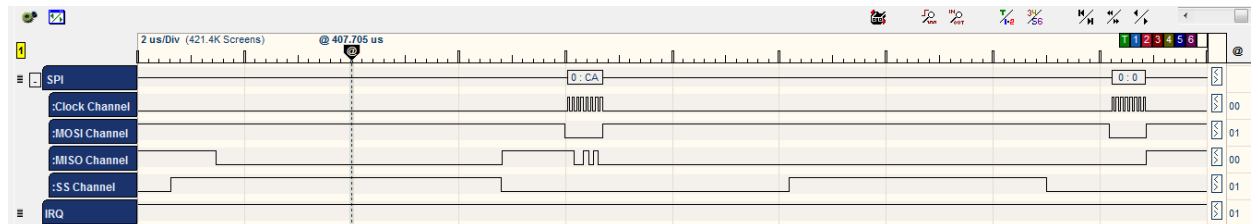
```

Command    CMD_SINGLE_READ: 0xCA          /* single word (4 bytes) read */
           BYTE [0] = CMD_SINGLE_READ
           BYTE [1] = address >> 16;    /* WIFI_HOST_RCV_CTRL_1 address = 0x1084 */
           BYTE [2] = address >> 8;
           BYTE [3] = address;

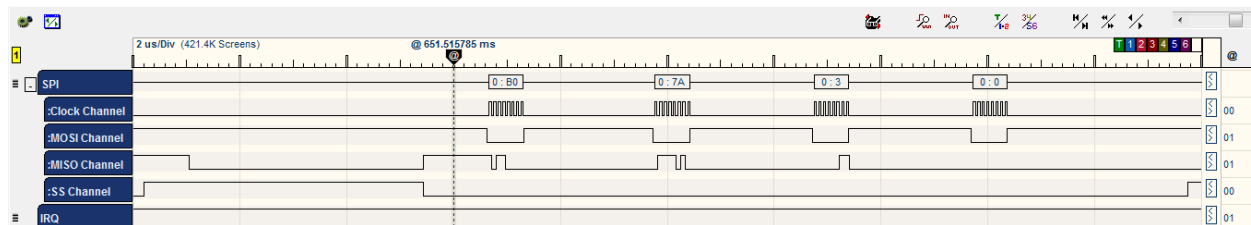
```



WILC acknowledges the command by sending three bytes [CA] [0] [F3].



Then WILC chip sends the value of the register 0x1078 which equals 0x037AB0.



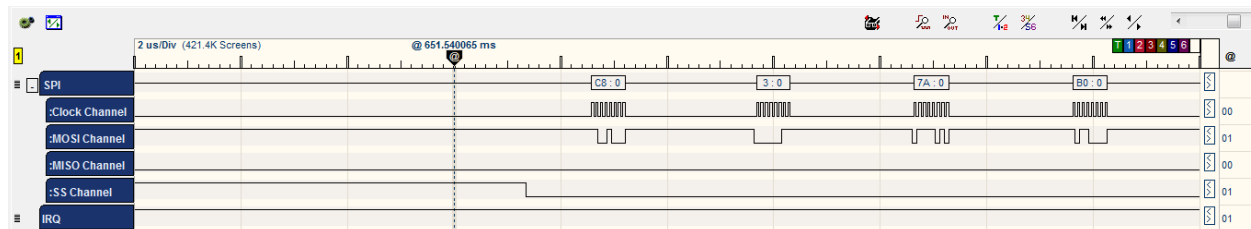
HIF reads the hif header data (as a block).

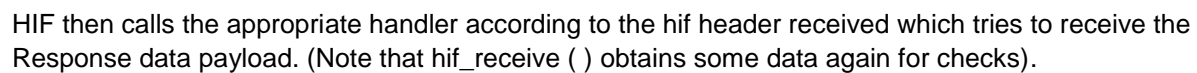
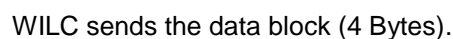
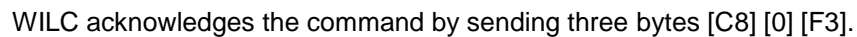
```
ret = nm_read_block(address, (uint8*)&strHif, sizeof(tstrHifHdr));
```

```

Command    CMD_DMA_EXT_READ: C8          /* dma extended read */
           BYTE [0] = CMD_DMA_EXT_READ
           BYTE [1] = address >> 16;    /* address = 0x037AB0 */
           BYTE [2] = address >> 8;
           BYTE [3] = address;
           BYTE [4] = size >> 16;
           BYTE [5] = size >> 8;
           BYTE [6] = size;

```

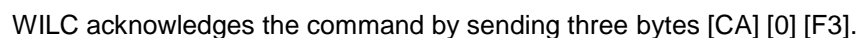


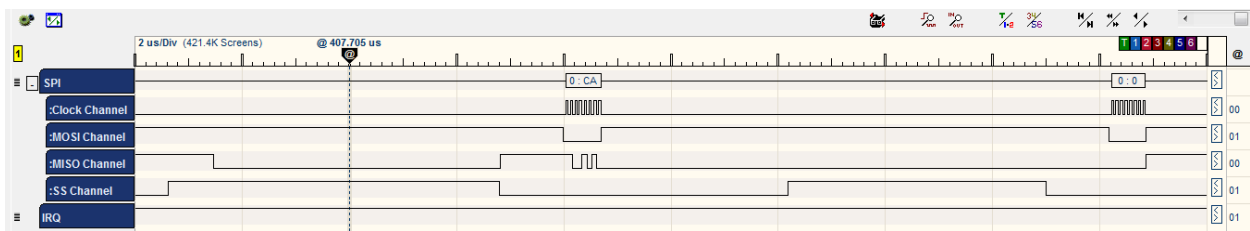


```

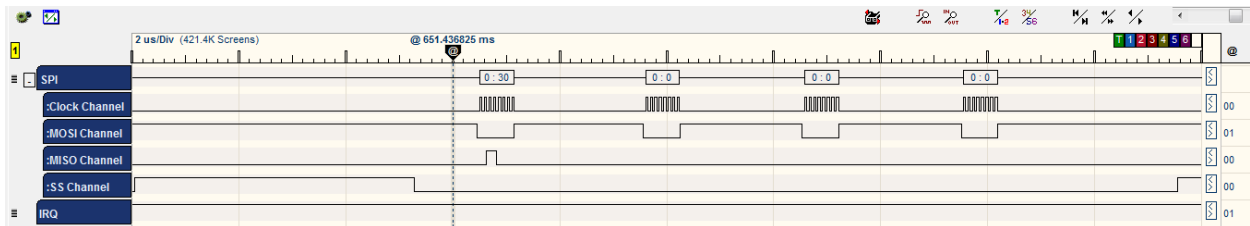
Command      CMD_SINGLE_READ:    0xCA          /* single word (4 bytes) read          */
             BYTE [0] = CMD_SINGLE_READ
             BYTE [1] = address >> 16; /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
             BYTE [2] = address >> 8;
             BYTE [3] = address;

```

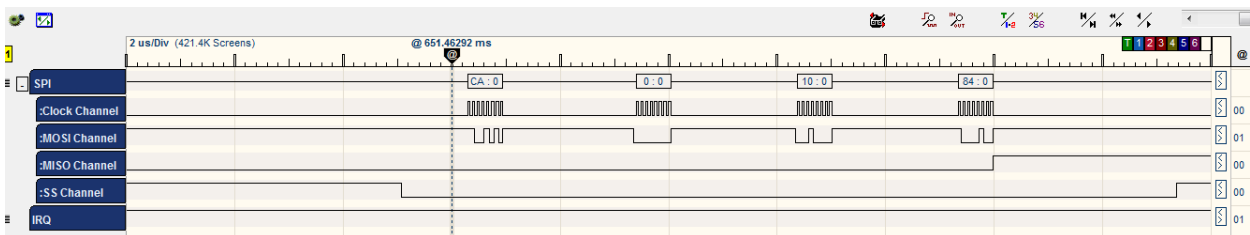




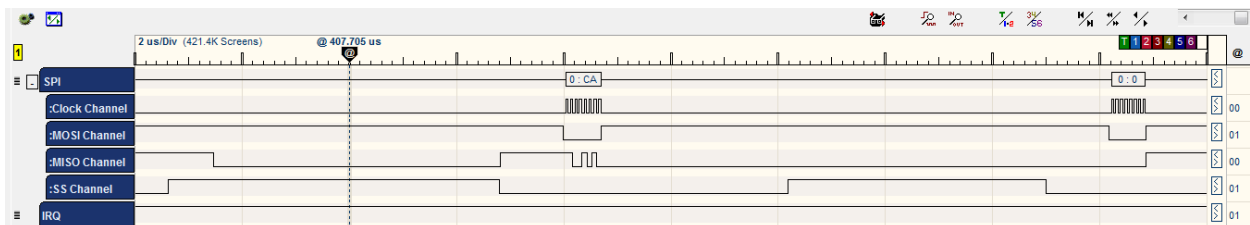
Then WILC chip sends the value of the register 0x1070 which equals 0x30.



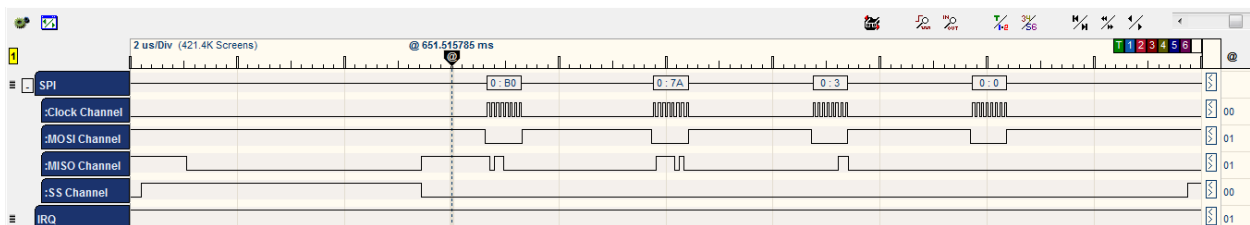
Command **CMD_SINGLE_READ: 0xCA** /* single word (4 bytes) read */
 BYTE [0] = **CMD_SINGLE_READ**
 BYTE [1] = address >> 16; /* WIFI_HOST_RCV_CTRL_1 address = 0x1084 */
 BYTE [2] = address >> 8;
 BYTE [3] = address;



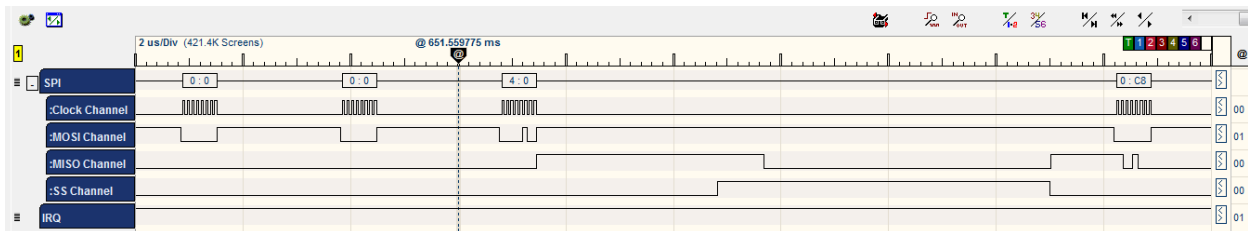
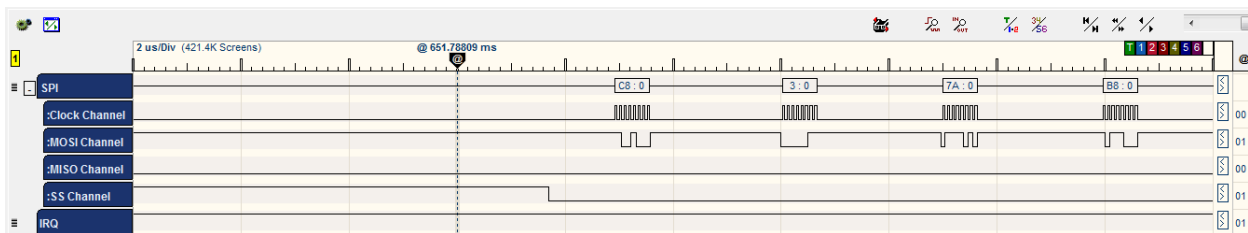
WILC acknowledges the command by sending three bytes [CA] [0] [F3].



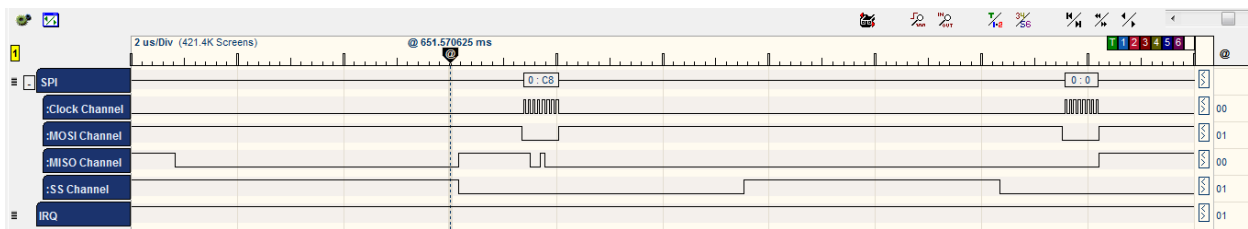
Then WILC chip sends the value of the register 0x1078 which equals 0x037AB0.



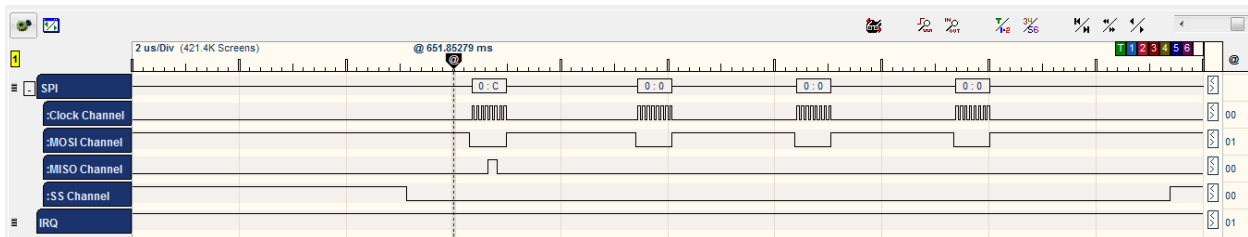
Command **CMD_DMA_EXT_READ: C8** /* dma extended read */
 BYTE [0] = **CMD_DMA_EXT_READ**
 BYTE [1] = address >> 16; /* address = 0x037AB8 */
 BYTE [2] = address >> 8;
 BYTE [3] = address;
 BYTE [4] = size >> 16;
 BYTE [5] = size >> 8;
 BYTE [6] = size;



WILC acknowledges the command by sending three bytes [C8] [0] [F3].



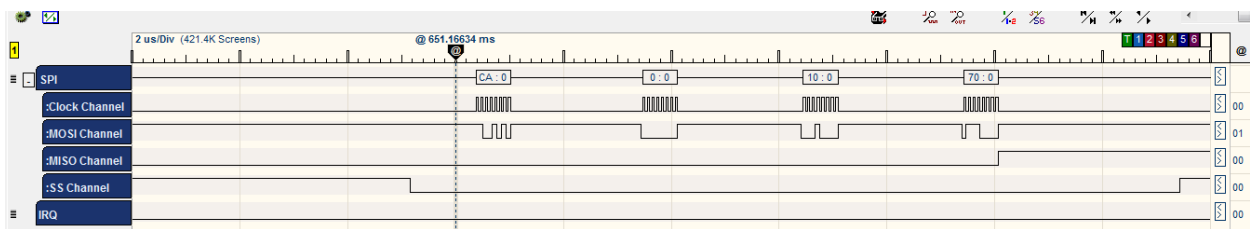
WILC sends the data block (4 Bytes).



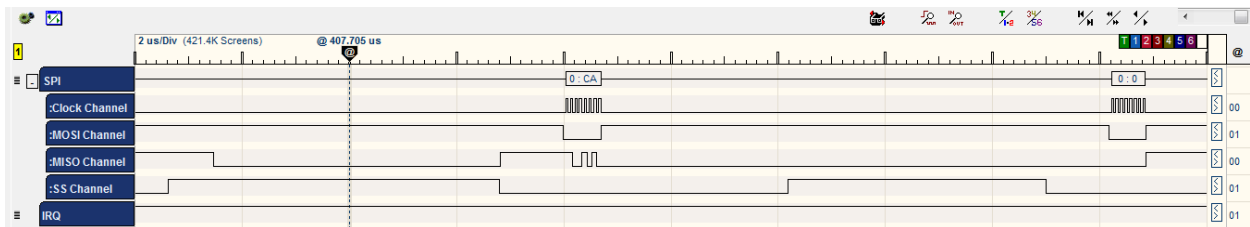
Now, after HIF layer received the response, it interrupts the chip to announce host RX is done.

```
static sint8 hif_set_rx_done(void)
{
    uint32 reg;
    sint8 ret = M2M_SUCCESS;
    ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_0,&reg);
    /* Set RX Done */
    reg |= (1<<1);
    ret = nm_write_reg(WIFI_HOST_RCV_CTRL_0,reg);
}
```

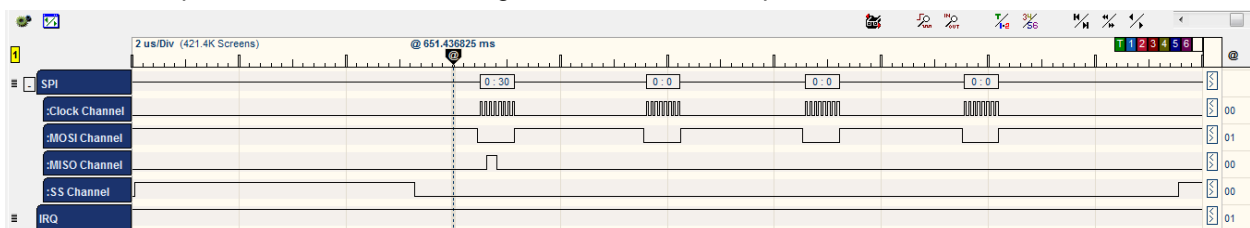
Command	CMD_SINGLE_READ: 0xCA	/* single word (4 bytes) read	*/
	BYTE [0] = CMD_SINGLE_READ		
	BYTE [1] = address >> 16;	/* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */	
	BYTE [2] = address >> 8;		
	BYTE [3] = address;		



WILC acknowledges the command by sending three bytes [CA] [0] [F3].



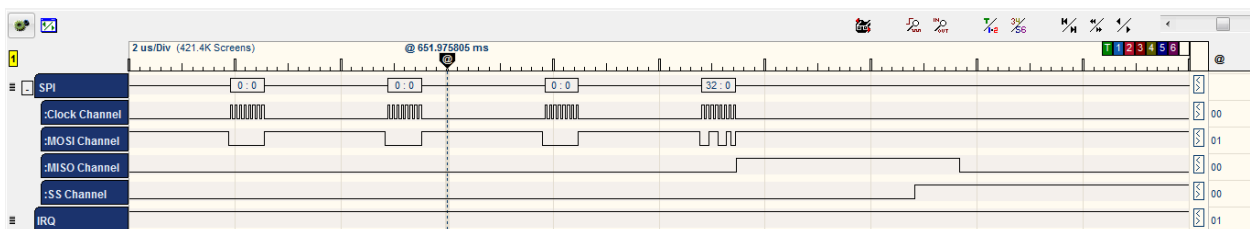
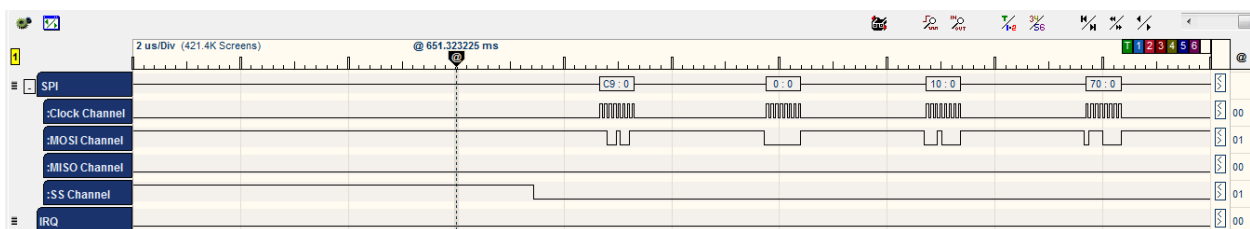
Then WILC chip sends the value of the register 0x1070 which equals 0x30.



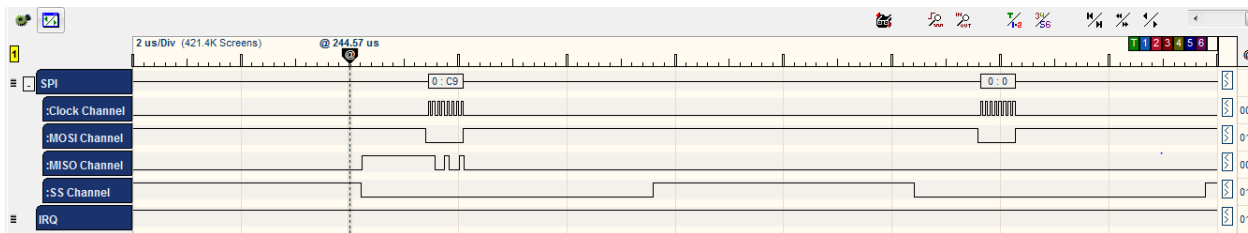
```

Command      CMD_SINGLE_WRITE:0XC9          /* single word write */
BYTE [0] = CMD_SINGLE_WRITE
BYTE [1] = address >> 16;                  /* WIFI_HOST_RCV_CTRL_0 address= 0x1070 */
BYTE [2] = address >> 8;
BYTE [3] = address;
BYTE [4] = u32data >> 24;                  /* Data = 0x32 */
BYTE [5] = u32data >> 16;
BYTE [6] = u32data >> 8;
BYTE [7] = u32data;

```



The chip acknowledges the command by sending two bytes [C9] [0].



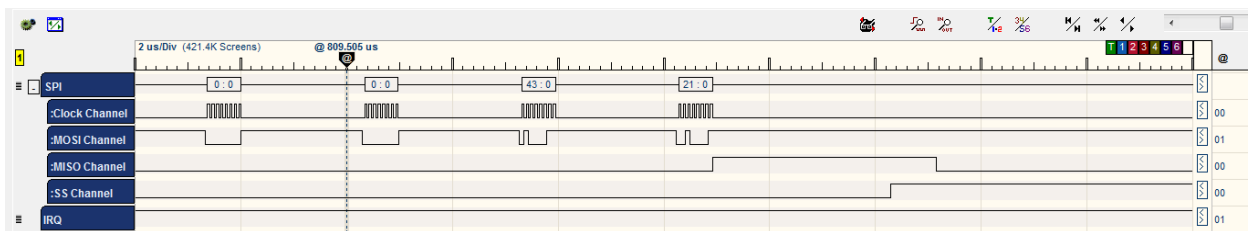
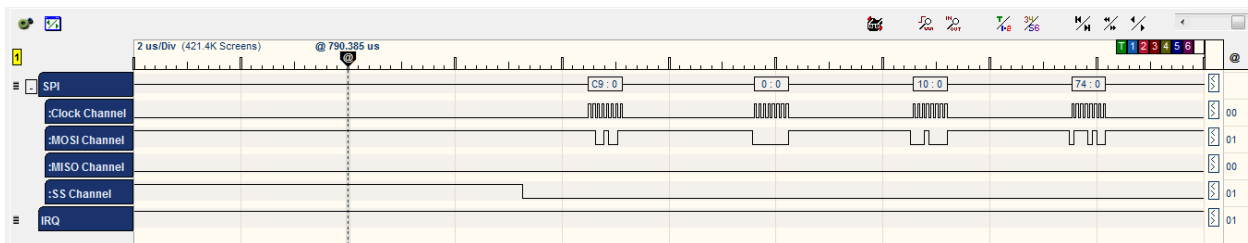
The HIF layer allows the chip to enter sleep mode again.

```

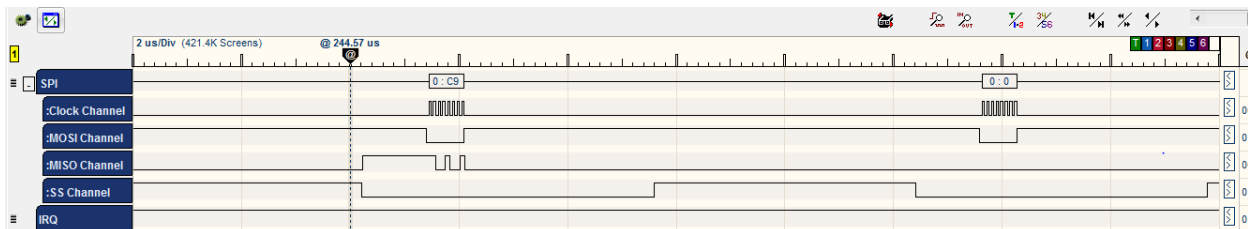
sint8 hif_chip_sleep(void)
{
    sint8 ret = M2M_SUCCESS;
    uint32 reg = 0;
    ret = nm_write_reg(WAKE_REG, SLEEP_VALUE);
    /* Clear bit 1 */
    ret = nm_read_reg_with_ret(0x1, &reg);
    if(reg & 0x2)
    {
        reg &= ~(1 << 1);
        ret = nm_write_reg(0x1, reg);
    }
}

```

Command CMD_SINGLE_WRITE:0XC9 /* single word write */
 BYTE [0] = CMD_SINGLE_WRITE
 BYTE [1] = address >> 16; /* WAKE_REG address = 0x1074 */
 BYTE [2] = address >> 8;
 BYTE [3] = address;
 BYTE [4] = u32data >> 24; /* SLEEP_VALUE Data = 0x4321 */
 BYTE [5] = u32data >> 16;
 BYTE [6] = u32data >> 8;
 BYTE [7] = u32data;



WILC acknowledges the command by sending two bytes [C9] [0].

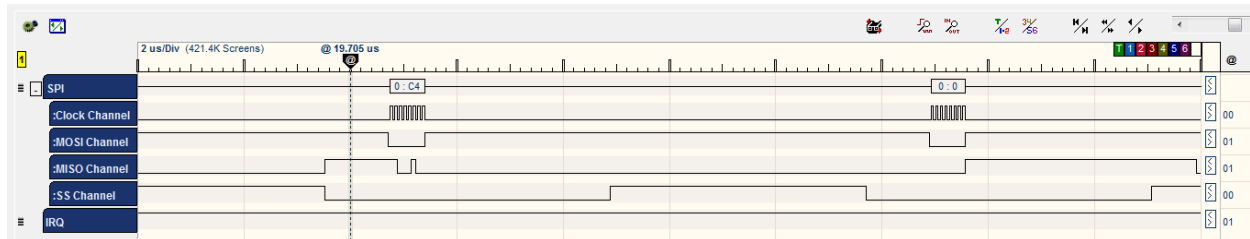


```

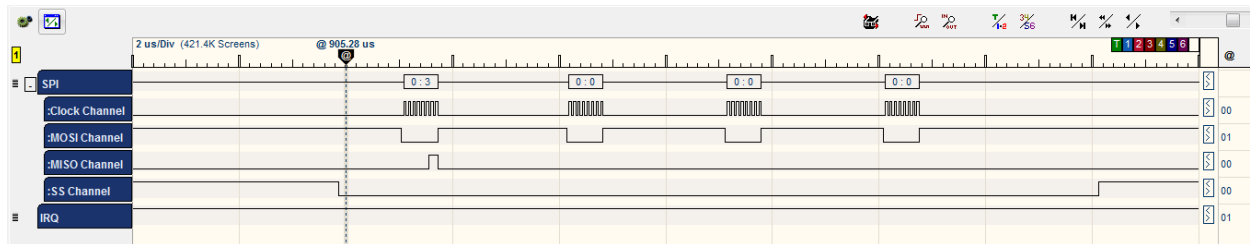
Command      CMD_INTERNAL_READ:      0xC4      /* internal register read */
BYTE [0] = CMD_INTERNAL_READ
BYTE [1] = address >> 8;             /* address = 0x01 */
BYTE [1] |= (1 << 7);                /* clockless register */
BYTE [2] = address;
BYTE [3] = 0x00;

```

WILC acknowledges the command by sending three bytes [C4] [0] [F3].



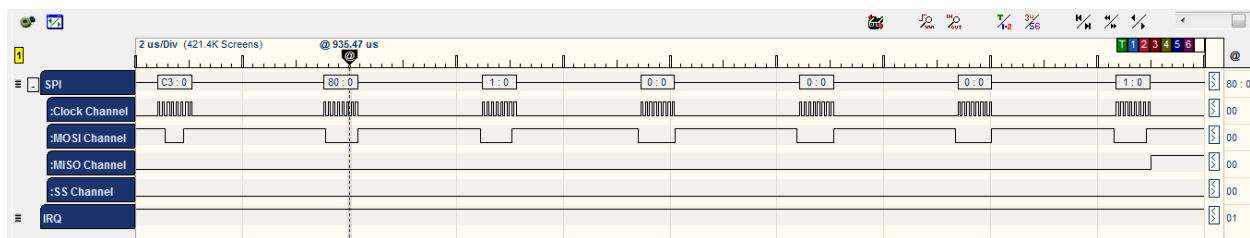
Then WILC chip sends the value of the register 0x01 which equals 0x03.



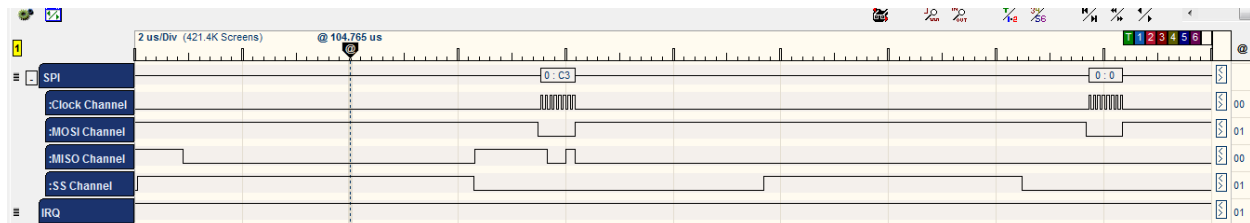
```

Command      CMD_INTERNAL_WRITE:      C3      /* internal register write */
BYTE [0] = CMD_INTERNAL_WRITE
BYTE [1] = address >> 8;             /* address = 0x01 */
BYTE [1] |= (1 << 7);                /* clockless register */
BYTE [2] = address;
BYTE [3] = u32data >> 24;             /* Data = 0x01 */
BYTE [4] = u32data >> 16;
BYTE [5] = u32data >> 8;
BYTE [6] = u32data;

```



The WILC chip acknowledges the command by sending two bytes [C3] [0].



Scan Wi-Fi request has been sent to the WILC chip and the response is sent to the host successfully.

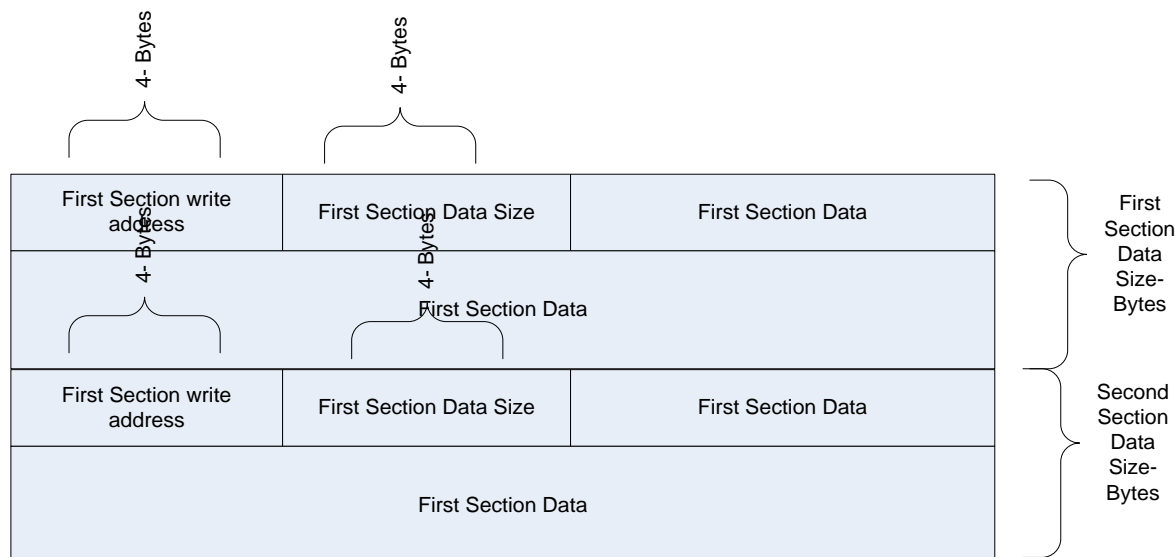
12 ATWILC1000 Firmware Download

ATWILC1000 HW doesn't have an internal Flash to store the firmware, so the firmware has to be stored on the host Flash and to be downloaded to the ATWILC1000 ram once at the driver initialization

The firmware binary is delivered as a part of the ATWILC1000 SW release at the file firmware.h where the firmware binary array is defined as "const char firmware[]"

The firmware binary is composed of a number of sections; each section is composed as follows:

1. The first 4-bytes is the write address of the current section in the ATWILC1000 chip memory.
2. 4-Bytes the current section data size.
3. Section data with length equals to the section size.



IMPORTANT

The Driver should repeat the pervious pattern till the end of the array and shouldn't assume a predefined number of sections, as the number of sections could be changed from release to release.



IMPORTANT

On some platforms Flash memory is not directly connected to the DMA engine, In that case the driver should copy the firmware in chunks to the host ram then write those chunks to the SPI interface.

Writing the firmware on the SPI bus should follow the same sequence at Data Packet Format.

Below is a code sample of downloading the firmware with ram chunks of 32 bytes, this could be changed to any chunk size.

```
sint8 firmware_download(void)
{
    sint8 ret = M2M_SUCCESS;
    uint32 u32SecSize,u32SecAddress;
    uint8_t* pu8FirmwareBuffer;
    sint32 BuffIndex =0,CurrentSecSize = 0;
    uint8_t u8TransferChunk[32],ChunkSize = 32;

    pu8FirmwareBuffer = (uint8_t*)firmware;
    M2M_DBG("firmware size = %d\n",sizeof(firmware));
```

```

while((uint32_t)pu8FirmwareBuffer < (uint32_t)(firmware+sizeof(firmware)))
{
    /*get text section address and size*/
    u32SecAddress = (((uint32_t)(pu8FirmwareBuffer[3]))<<24)|(((uint32_t)(pu8FirmwareBuffer[2]))<<16)|
    (((uint32_t)(pu8FirmwareBuffer[1]))<<8)|(((uint32_t)(pu8FirmwareBuffer[0]))<<0);
    u32SecSize = (((uint32_t)(pu8FirmwareBuffer[7]))<<24)|(((uint32_t)(pu8FirmwareBuffer[6]))<<16)|
    (((uint32_t)(pu8FirmwareBuffer[5]))<<8)|(((uint32_t)(pu8FirmwareBuffer[4]))<<0);
    M2M_DBG("write sec %x size %d\n",u32SecAddress,u32SecSize);
    CurrentSecSize = u32SecSize;
    ChunkSize = 32;
    BuffIndex = 8;
    while(CurrentSecSize>0)
    {
        if(CurrentSecSize < ChunkSize)
            ChunkSize = CurrentSecSize;

        m2m_memcpy(u8TransferChunk,pu8FirmwareBuffer+BuffIndex,ChunkSize);
        nm_write_block(u32SecAddress,u8TransferChunk,ChunkSize);
        u32SecAddress += ChunkSize;
        BuffIndex += ChunkSize;
        CurrentSecSize -= ChunkSize;
    }
    pu8FirmwareBuffer += BuffIndex;
}
return ret;
}

```



INFO

There is only one firmware for all mode of operations of ATWILC1000 (Station, AP, and P2P).

Appendix A API Reference

A.1 WLAN Module

A.1.1 Defines

Define	Definition	Value
#define M2M_FIRMWARE_VERSION_MAJOR_NO	Firmware Major release version number	18
#define M2M_FIRMWARE_VERSION_MINOR_NO	Firmware Minor release version number	0
#define M2M_FIRMWARE_VERSION_PATCH_NO	Firmware patch release version number	2
#define M2M_DRIVER_VERSION_PATCH_NO	Driver patch release version number	0
#define M2M_BUFFER_MAX_SIZE (1600UL)	Maximum size for the shared packet buffer	-4
#define M2M_MAC_ADDRES_LEN	The size for 802.11 MAC address	6
#define M2M_ETHERNET_HDR_OFFSET	The offset of the Ethernet header within the WLAN TX Buffer	34
#define M2M_ETHERNET_HDR_LEN	Length of the Ethernet header in bytes	14
#define M2M_MAX_SSID_LEN	Maximum size for the Wi-Fi SSID including the NULL termination	33
#define M2M_MAX_PSK_LEN	Maximum size for the WPA PSK including the NULL termination	65
#define M2M_DEVICE_NAME_MAX	Maximum Size for the device name including the NULL termination	48
#define M2M_LISTEN_INTERVAL	The STA uses the Listen Interval parameter to indicate to the AP how many beacon intervals it shall sleep before it retrieves the queued frames	1
#define M2M_1X_PWD_MAX	The maximum size of the password including the NULL termination. It is used for RADIUS authentication in case of connecting the device to an AP secured with WPA-Enterprise.	41
#define M2M_CUST_IE_LEN_MAX	The maximum size of IE (Information Element).	252
#define M2M_CONFIG_CMD_BASE	The base value of all the host configuration commands opcodes	1
#define M2M_SERVER_CMD_BASE	The base value of all the power save mode host commands codes	20
#define M2M_STA_CMD_BASE	The base value of all the station mode host commands opcodes	40
#define M2M_AP_CMD_BASE	The base value of all the Access Point mode host commands opcodes	70
#define M2M_P2P_CMD_BASE	The base value of all the P2P mode host commands opcodes	90
#define M2M_OTA_CMD_BASE	The base value of all the Over the Air (OTA) mode host commands opcodes	100

Define	Definition	Value
#define WEP_40_KEY_SIZE	Indicate the wep key size in bytes for 40 bit hex passphrase	((uint8)5)
#define WEP_104_KEY_SIZE	Indicate the wep key size in bytes for 104 bit hex passphrase	((uint8)13)
#define WEP_40_KEY_STRING_SIZE	Indicate the wep key size in bytes for 40 bit string passphrase	(uint8)10)
#define WEP_104_KEY_STRING_SIZE	Indicate the wep key size in bytes for 104 bit string passphrase	((uint8)26)
#define WEP_KEY_MAX_INDEX	Indicate the max key index value for WEP authentication	(uint8)4
#define M2M_SCAN_MIN_NUM_SLOTS	The min. number of scan slots performed by the WILC firmware	2
#define M2M_SCAN_MIN_SLOT_TIME	The min. duration in milliseconds of a scan slots performed by the WILC firmware	(20)
#define M2M_SCAN_FAIL	Indicate that the WILC firmware has failed to perform the scan operation	((uint8)1)
#define M2M_JOIN_FAIL	Indicate that the WILC firmware has failed to join the BSS	((uint8)2)
#define M2M_AUTH_FAIL	Indicate that the WILC firmware has failed to authenticate with the AP	((uint8)3)
#define M2M_ASSOC_FAIL	Indicate that the WILC firmware has failed to associate with the AP	((uint8)4)
#define M2M_SCAN_ERR_WIFI	Currently not used	((sint8)-2)
#define M2M_SCAN_ERR_IP	Currently not used	((sint8)-3)
#define M2M_SCAN_ERR_AP	Currently not used	((sint8)-4)
#define M2M_SCAN_ERR_P2P	Currently not used	((sint8)-5)
#define M2M_SCAN_ERR_WPS	Currently not used	((sint8)-6)
#define M2M_DEFAULT_CONN_EMPTY_LIST	A failure response that indicates an empty network list as a result to the function call m2m_default_connect	((sint8)-20)
#define M2M_DEFAULT_CONN_SCAN_MISMATCH	A failure response that indicates that no one of the cached networks was found in the scan results, as a result to the function call m2m_default_connect	((sint8)-21)
#define M2M_WIFI_FRAME_TYPE_ANY	Set monitor mode to receive any of the frames types	0xFF
#define M2M_WIFI_FRAME_SUB_TYPE_ANY	Set monitor mode to receive frames with any sub type	0xFF

A.1.2 Enumeration/Typedef

```
enum tenuM2mConfigCmd
```

This enum contains all the host commands used to configure the WILC firmware

Enumerator Values	
M2M_WIFI_REQ_RESTART	Reserved for Firmware use not allowed from host driver
M2M_WIFI_REQ_SET_MAC_ADDRESS	Set the WILC mac address (will overwrite production eFused boards)
M2M_WIFI_REQ_CURRENT_RSSI	Request the current connected AP RSSI
M2M_WIFI_RESP_CURRENT_RSSI	Response to M2M_WIFI_REQ_CURRENT_RSSI with the RSSI value
M2M_WIFI_REQ_SET_DEVICE_NAME	Set the WILC device name property
M2M_WIFI_REQ_CUST_INFO_ELEMENT	Add Custom Element to Beacon Management Frame

```
enum tenuM2mStaCmd
```

This enum contains all the WILC commands while in Station mode.

Enumerator Values	
M2M_WIFI_REQ_CONNECT	Connect with AP command
M2M_WIFI_REQ_GET_CONN_INFO	Request connection information command
M2M_WIFI_RESP_CONN_INFO	Request connection information response
M2M_WIFI_REQ_DISCONNECT	Request to disconnect from AP command
M2M_WIFI_RESP_CON_STATE_CHANGE D	Connection state changed response
M2M_WIFI_REQ_SLEEP	Set PS mode command
M2M_WIFI_REQ_SCAN	Request scan command
M2M_WIFI_REQ_WPS_SCAN	Request WPS scan command
M2M_WIFI_RESP_SCAN_DONE	Scan complete notification response
M2M_WIFI_REQ_SCAN_RESULT	Request Scan results command
M2M_WIFI_RESP_SCAN_RESULT	Request Scan results response
M2M_WIFI_REQ_WPS	Request WPS start command
M2M_WIFI_REQ_START_WPS	This command is for internal use by the WILC and should not be used by the host driver
M2M_WIFI_REQ_DISABLE_WPS	Request to disable WPS command
M2M_WIFI_REQ_LSN_INT	Set Wi-Fi listen interval
M2M_WIFI_REQ_SEND_ETHER- NET_PACKET	Send Ethernet packet in bypass mode

Enumerator Values	
M2M_WIFI_RESP_ETHER-NET_RX_PACKET	Receive Ethernet packet in bypass mode
M2M_WIFI_REQ_SET_SCAN_OPTION	Set Scan options “slot time, slot number, etc.”
M2M_WIFI_REQ_SET_SCAN_REGION	Set scan region
M2M_WIFI_REQ_DOZE	Used to force the WILC to sleep in manual PS mode
M2M_WIFI_REQ_SET_MAC_MCAST	Set the WILC multicast filters

enum tenuM2mP2pCmd

This enum contains all the WILC commands while in P2P mode.

Enumerator Values	
M2M_WIFI_REQ_P2P_INT_CONNECT	This command is for internal use by the WILC and should not be used by the host driver
M2M_WIFI_REQ_ENABLE_P2P	Enable P2P mode command
M2M_WIFI_REQ_DISABLE_P2P	Disable P2P mode command
M2M_WIFI_REQ_P2P_REPOST	This command is for internal use by the WILC and should not be used by the host driver

enum tenuM2mApCmd

This enum contains all the WILC commands while in AP mode.

Enumerator Values	
M2M_WIFI_REQ_ENABLE_AP	Enable AP mode command
M2M_WIFI_REQ_DISABLE_AP	Disable AP mode command
M2M_WIFI_REQ_AP_ASSOC_INFO	Command to get Info about the associated stations
M2M_WIFI_RESP_AP_ASSOC_INFO	Response to get Info about the associated stations

enum tenuM2mConnState

Wi-Fi Connection State.

Enumerator Values	
M2M_WIFI_DISCONNECTED	Wi-Fi state is disconnected
M2M_WIFI_CONNECTED	Wi-Fi state is connected
M2M_WIFI_UNDEF	Undefined Wi-Fi State

enum tenuM2mSecType

Wi-Fi Supported Security types.

Enumerator Values	
M2M_WIFI_SEC_INVALID	Invalid security type
M2M_WIFI_SEC_OPEN	Wi-Fi network is not secured
M2M_WIFI_SEC_WPA_PSK	Wi-Fi network is secured with WPA/WPA2 personal (PSK)
M2M_WIFI_SEC_WEP	Security type WEP (40 or 104) OPEN OR SHARED
M2M_WIFI_SEC_802_1X	Wi-Fi network is secured with WPA/WPA2 Enterprise.IEEE802.1x user-name/password authentication

enum tenuM2mSsidMode

Wi-Fi Supported SSID types.

Enumerator Values	
SSID_MODE_VISIBLE	SSID is visible to others
SSID_MODE_HIDDEN	SSID is hidden

enum tenuM2mScanCh

Wi-Fi RF Channels.

Enumerator values
M2M_WIFI_CH_1
M2M_WIFI_CH_2
M2M_WIFI_CH_3
M2M_WIFI_CH_4
M2M_WIFI_CH_5
M2M_WIFI_CH_6
M2M_WIFI_CH_7
M2M_WIFI_CH_8
M2M_WIFI_CH_9
M2M_WIFI_CH_10
M2M_WIFI_CH_11
M2M_WIFI_CH_12
M2M_WIFI_CH_13
M2M_WIFI_CH_14
M2M_WIFI_CH_ALL

enum tenuM2mScanRegion

Wi-Fi RF Channels.

Enumerator values
NORTH_AMERICA
ASIA

enum tenuPowerSaveModes

Power save Modes.

Enumerator Values	
M2M_NO_PS	Power save is disabled
M2M_PS_AUTOMATIC	Power save is done automatically by the WILC. This mode doesn't disable all of the WILC modules and use higher amount of power than the H_AUTO-MATIC and the DEEP_AUTOMATIC modes.
M2M_PS_H_AUTOMATIC	Power save is done automatically by the WILC. Achieve higher power save than the AUTOMATIC mode by shutting down more parts of the WILC firm-ware.
M2M_PS_DEEP_AUTOMATIC	Power save is done automatically by the WILC. Achieve the highest possible power save.
M2M_PS_MANUAL	Power save is done manually by the user

enum tenuWPSTrigger

WPS triggering methods.

Enumerator Values	
WPS_PIN_TRIGGER	WPS is triggered in PIN method
WPS_PBC_TRIGGER	WPS is triggered via push button

enum tenuControllInterface

Values used to set the interface currently under control, Used in case of concurrency.

Enumerator Values	
INTERFACE_1	Interface 1
INTERFACE_2	Interface 2

enum tenuWifiFrameType

Enumeration for Wi-Fi MAC frame type codes (2-bit), the following types are used to identify the type of frame sent or received. Each frame type constitutes a number of frame subtypes as defined in **tenuSubTypes** to specify the exact type of frame. Values are defined as per the IEEE 802.11 standard.

Remarks:

The following frame types are useful for advanced user usage when CONF_MGMT is defined and the user application requires monitoring the frame transmission and reception.

See also:

- **tenuSubTypes**

Enumerator Values	
MANAGEMENT	Wi-Fi Management frame (Probe Req/Resp, Beacon, Association Req/Resp, etc.)
CONTROL	Wi-Fi Control frame (eg. ACK frame)
DATA_BASICTYPE	Wi-Fi Data frame
RESERVED	

enum tenuSubTypes

Enumeration for Wi-Fi MAC Frame subtype code (6-bit). The frame subtypes fall into one of the three frame type groups as defined in **tenuWifiFrameType** (MANAGEMENT, CONTROL, and DATA). Values are defined as per the IEEE 802.11 standard.

Remarks:

The following sub-frame types are useful for advanced user usage when CONF_MGMT is defined and the application developer requires monitoring the frame transmission and reception.

See also:

- **tenuWifiFrameType**

Enumerator Values
Sub-Types related to Management Sub-Types
ASSOC_REQ
ASSOC_RSP
REASSOC_REQ
REASSOC_RSP
PROBE_REQ
PROBE_RSP
BEACON
ATIM
DISASOC
AUTH
DEAUTH
ACTION
Sub-Types related to Control
PS_POLL
RTS
CTS

Enumerator Values
ACK
CFEND
CFEND_ACK
BLOCKACK_REQ
BLOCKACK
Sub-Types related to Data
DATA
DATA_ACK
DATA_POLL
DATA_POLL_ACK
NULL_FRAME
CFACK
CFPOLL
CFPOLL_ACK
QOS_DATA
QOS_DATA_ACK
QOS_DATA_POLL
QOS_DATA_POLL_ACK
QOS_NULL_FRAME
QOS_CFPOLL
QOS_CFPOLL_ACK

enum tenuInfoElementId

Enumeration for the Wi-Fi Information Element (IE) IDs, which indicates the specific type of IEs. IEs are management frame information included in management frames. Values are defined as per the IEEE 802.11 standard.

Enumerator Values	
ISSID	Service Set Identifier (SSID)
ISUPRATES	Supported Rates
IFHPARMS	FH parameter set
IDSPARMS	DS parameter set
ICFPARMS	CF parameter set
ITIM	Traffic Information Map
IIBPARMS	IBSS parameter set
ICOUNTRY	Country element

Enumerator Values	
IEDCAPARAMS	EDCA parameter set
ITSPEC	Traffic Specification
ITCLAS	Traffic Classification
ISCHED	Schedule
ICTEXT	Challenge Text
IPOWERCONSTRAINT	Power Constraint
IPOWERCAPABILITY	Power Capability
ITPCREQUEST	TPC Request
ITPCREPORT	TPC Report
ISUPCHANNEL	
ICHSWANNOUNC	Channel Switch Announcement
IMEASUREMENTREQUEST	Measurement request
IMEASUREMENTREPORT	Measurement report
IQUIET	Quiet element Info
IIBSSDFS	IBSS DFS
IERPINFO	ERP Information
ITSDELAY	TS Delay
ITCLASPROCESS	TCLAS Processing
IHTCAP	HT Capabilities
IQOSCAP	QoS Capability
IRSNELEMENT	RSN Information Element
IEXSUPRATES	Extended Supported Rates
IEXCHSWANNOUNC	Extended Ch Switch Announcement
IHTOPERATION	HT Information
ISECCHOFF	Secondary Channel Offset
I2040COEX	20/40 Coexistence IE
I2040INTOLCHREPORT	20/40 Intolerant channel report
IOBSSSCAN	OBSS Scan parameters
IEXTCAP	Extended capability
IWMM	WMM® parameters
IWPAELEMENT	WPA Information Element

```
typedef struct tstr1xAuthCredentials
```

Credentials for the user to authenticate with the AAA server (WPA-Enterprise Mode IEEE802.1x).

Data Field	Description
uint8 au8UserName[M2M_1X_USR_NAME_MAX]	User Name. It must be Null terminated string.
uint8 au8Passwd[M2M_1X_PWD_MAX]	Password corresponding to the user name. It must be Null terminated string.

```
typedef struct tstrEthInitParam
```

Structure to hold Ethernet interface parameters. Structure should be defined, based on the application's functionality. Before a call is made to the initialize the Wi-Fi operations, set the structure's attributes and pass it as a parameter (part of the Wi-Fi configuration structure **tstrWifiInitParam**) to the **m2m_wifi_init** function.

Applications shouldn't need to define this structure, if the bypass mode is not defined.

Data Field	Definition
tpfAppWifiCb pfAppWifiCb	Not used
tpfAppEthCb pfAppEthCb	Callback for Ethernet interface
uint8* au8ethRcvBuf	Pointer to Receive Buffer of Ethernet Packet
uint16 u16ethRcvBufSize	Size of Receive Buffer for Ethernet Packet

See also:

- **tpfAppEthCb tpfAppWifiCb**
- **m2m_wifi_init**

Warning:

Make sure that bypass mode is defined before using **tstrEthInitParam**.

```
typedef struct tstrM2MAPConfig
```

AP Configuration structure. This structure holds the configuration parameters for the M2M AP mode. It should be set by the application when it requests to enable the M2M AP operation mode. The M2M AP mode currently supports only OPEN and WEP security.

Data Field	Definition
uint8 au8SSID[M2M_MAX_SSID_LEN]	Configuration parameters for the Wi-Fi AP.AP SSID
uint8 u8ListenChanel	Wi-Fi RF Channel which the AP will operate on
uint8 u8KeyIdx	Wep key Index start from 0 to 3
uint8 u8KeySz	Wep key Size WEP_40_KEY_STRING_SIZE or WEP_104_KEY_STRING_SIZE
uint8 au8Wep-Key[WEP_104_KEY_STRING_SIZE+1]	Wep key null terminated
uint8 u8SecType	Security type: Open or WEP only in the current implementation
uint8 u8SsidHide	SSID Status "Hidden(1)/Visible(0)"
uint8 u8IsPMKUsed	For internal use by the driver, shouldn't be set by the application

Data Field	Definition
uint8 au8PSK[M2M_MAX_PSK_LEN]	Pre-Shared key of the AP, used if the “u8SecType” is set to M2M_WIFI_SEC_WPA_PSK

```
typedef struct tstrM2mClientState
```

PS Client State.

Data Field	Definition
uint8 u8State	PS Client State
uint8 __PAD24__[3]	Padding bytes for forcing 4-byte alignment

```
typedef struct tstrM2MConnInfo
```

M2M Provisioning Information obtained from the HTTP Provisioning server.

Data Field	Definition
char acSSID[M2M_MAX_SSID_LEN]	AP connection SSID name
uint8 u8SecType	Security type
uint8 au8IPAddr[4]	Connection IP address
sint8 s8RSSI	Connection RSSI signal
uint8 __PAD8__	Padding bytes for forcing 4-byte alignment

```
typedef struct tstrM2MDeviceNameConfig
```

Device name.

It is assigned by the application. It is used mainly for Wi-Fi Direct device discovery and WPS device information.

Data Field	Definition
uint8 au8DeviceName[M2M_DEVICE_NAME_MAX]	NULL terminated device name

```
typedef struct tstrM2MAssocEntryInfo
```

Holds the assoc info of an entry in AP mode.

Data Field	Definition
uint8 BSSID[6];	MAC address of the associated station
sint8 s8RSSI;	RSSI of this station

```
typedef struct tstrM2MAPAssocInfo
```

Holds the assoc info of all entries associated in AP mode.

Data Field	Definition
uint8 u8NoConnSta;	No. Of currently associated stations in AP mode
tstrM2MAssocEntryInfo astrM2MAssocEntryInfo[8];	Structure holds info per station

```
typedef struct tstrM2mlpCtrlBuf
```

Structure holding the incoming buffer's data size information, indicating the data size of the buffer and the remaining buffer's data size. The data of the buffer which holds the packet sent to the host when in the bypass mode, is placed in the **tstrEthInitParam** structure in the au8ethRcvBuf attribute. This following information is retrieved in the host when an event **M2M_WIFI_RESP_ETHERNET_RX_PACKET** is received in the Wi-Fi callback function tpfAppWifiCb.

The application is expected to use this structure's information to determine if there is still incoming data to be received from the firmware.

See also:

- **tpfAppEthCb**
- **tstrEthInitParam**

Warning:

Make sure that bypass mode is defined before using **tstrM2mlpCtrlBuf**.

Data Field	Definition
uint16 u16DataSize	Size of the received data in bytes
uint16 u16RemainigDataSize	Size of the remaining data bytes to be delivered to host

```
typedef struct tstrM2MMulticastMac
```

M2M add/remove multicast mac address.

Data Field	Definition
uint8 au8macaddress[M2M_MAC_ADDRES_LEN]	Mac address needed to be added or removed from filter
uint8 u8AddRemove	Set by 1 to add or 0 to remove from filter
uint8 __PAD8__	Padding bytes for forcing 4-byte alignment

```
typedef struct tstrM2MP2PConnect
```

Set the device to operate in the Wi-Fi Direct (P2P) mode.

Data Field	Definition
uint8 u8ListenChannel	P2P Listen Channel (1, 6, or 11)
uint8 __PAD24__[3]	Padding bytes for forcing 4-byte alignment

```
typedef struct tstrM2MProvisionInfo
```

M2M Provisioning Information obtained from the HTTP Provisioning server.

Data Field	Definition
uint8 au8SSID[M2M_MAX_SSID_LEN]	Provisioned SSID
uint8 au8Password[M2M_MAX_PSK_LEN]	Provisioned Password
uint8 u8SecType	Wi-Fi Security type OPEN/WPA
uint8 u8Status	Provisioning status. It must be checked before reading the provisioning information. It may be: M2M_SUCCESS (Provision successful) M2M_FAIL (Provision Failed)

typedef struct tstrM2MProvisionModeConfig

M2M Provisioning Mode Configuration.

Data Field	Definition
tstrM2MAPConfig strApConfig	Configuration parameters for the Wi-Fi AP
char acHttpServerDomainName[64]	The device domain name for HTTP provisioning
uint8 u8EnableRedirect	A flag to enable/disable HTTP redirect. Feature for the HTTP Provisioning server. If the Redirect is enabled, all HTTP traffic (http://URL) from the device associated with WILC AP will be redirected to the HTTP Provisioning Web page. 0: Disable HTTP Redirect 1: Enable HTTP Redirect
uint8 __PAD24__[3]	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2mPs

Power Save Configuration.

See also:

- **tenuPowerSaveModes**

Data Field	Definition
uint8 u8PsType	Power save operating mode tenuPowerSaveModes
uint8 u8BcastEn	1 Enabled -> Listen to the broadcast data 0 Disabled -> Ignore the broadcast data
uint8 __PAD16__[2]	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2mReqScanResult

Scan Result Request. The Wi-Fi Scan results list is stored in Firmware.

The application can request a certain scan result by its index.

Data Field	Definition
uint8 u8Index	Index of the desired scan result
uint8 __PAD24__[3]	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2MScan

Wi-Fi Scan Request.

See also:

- **tenuM2mScanCh**

Data Field	Definition
uint8 u8ChNum	The Wi-Fi RF Channel number
uint8 __PAD24__[3]	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2mScanDone

Wi-Fi Scan Result.

Data Field	Definition
uint8 u8NumofCh	Number of found APs
sint8 s8ScanState	Scan status
uint8 __PAD16__[2]	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2MScanOption

Wi-Fi Scan Request.

Data Field	Definition
uint8 u8NumOfSlot	The min number of slots is 2 for every channel, every slot the SoC will send Probe Req on air, and wait/listen for PROBE RESP/BEACONS for the u16slotTime
uint8 u8SlotTime	the time that the SoC will wait on every channel listening to the frames on air when that time increased number of AP will increased in the scan results min time is 10ms and the max is 250ms
uint8 __PAD16__[2]	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2mSetMacAddress

Sets the MAC address from application. The WILC load the mac address from the effuse by default to the WILC configuration memory, but that function is used to let the application overwrite the configuration memory with the mac address from the host.

Note: It is recommended to call this only once before calling connect request and after the m2m_wifi_init.

Data Field	Definition
uint8 au8Mac[6]	MAC address array
uint8 __PAD16__[2]	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2mSlpReqTime

Manual power save request sleep time.

Data Field	Definition
uint32 u32SleepTime	Sleep time in ms

typedef struct tstrM2mWifiConnect

Wi-Fi Connect Request.

Data Field	Definition
tstrM2MWifiSecInfo strSec	Security parameters for authenticating with the AP
uint16 u16Ch	RF Channel for the target SSID from 0 to 13
uint8 au8SSID[M2M_MAX_SSID_LEN]	SSID of the desired AP. It must be NULL terminated string
uint8 __PAD__[__CONN_PAD_SIZE__]	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2mWifiscanResult

Wi-Fi Scan Result.

Information corresponding to an AP in the Scan Result list identified by its order (index) in the list.

Data Field	Definition
uint8 u8index	AP index in the scan result list
sint8 s8rssi	AP signal strength
uint8 u8AuthType	AP authentication type
uint8 u8ch	AP RF channel
uint8 au8BSSID[6]	BSSID of the AP
uint8 au8SSID[M2M_MAX_SSID_LEN]	AP SSID
uint8 _PAD8_	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2MWifiSecInfo

Authentication credentials to connect to a Wi-Fi network.

Data Field	Definition
tuniM2MWifiAuth uniAuth	Union holding all possible authentication parameters corresponding the current security types

Data Field	Definition
uint8 u8SecType	Wi-Fi network security type. See tenuM2mSecType for supported security types.
uint8 __PAD__[__PADDING__]	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2mWifiStateChanged

Wi-Fi Connection State.

See also:

- **M2M_WIFI_DISCONNECTED**
- **M2M_WIFI_CONNECTED**
- **M2M_WIFI_REQ_CON_STATE_CHANGED**

Data Field	Definition
uint8 u8CurrState Current Wi-Fi connection state	WLAN frame length
uint8 u8ErrCode	Error type
uint8 __PAD16__[2]	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2mWifiWepParams

WEP security key parameters.

Data Field	Definition
uint8 u8KeyIndx	Wep key Index
uint8 u8KeySz	Wep key Size
uint8 au8WepKey[WEP_104_KEY_STRING_SIZE+1]	WEP Key represented as a NULL terminated ASCII string
uint8 __PAD24__[3]	Padding bytes to keep the structure word aligned

typedef struct tstrM2MWPSConnect

WPS configuration parameters.

See also:

- **tenuWPSTrigger**

Data Field	Definition
uint8 u8TriggerType	WPS triggering method (Push button or PIN)
char acPinNumber[8]	WPS PIN No (for PIN method)
uint8 __PAD24__[3]	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2MWPSInfo

WPS Result.

This structure is passed to the application in response to a WPS request.

If the WPS session is completed successfully, the structure will have Non-ZERO authentication type.

If the WPS Session fails (due to error or timeout) the authentication type is set to ZERO.

See also:

- **tenuM2mSecType**

Data Field	Definition
uint8 u8Ch	RF Channel for the AP
uint8 au8SSID[M2M_MAX_SSID_LEN]	SSID obtained from WPS
uint8 au8PSK[M2M_MAX_PSK_LEN]	PSK for the network obtained from WPS

A.1.3 Function

- **m2m_wifi_init**
 - NMI_API sint8 m2m_wifi_init (tstrWifilnitParam *pWifilnitParam)

Synchronous initialization function for the WILC driver. This function initializes the driver by, registering the call back function for M2M_WIFI layer (also the call back function for bypass mode/monitoring mode if defined), initializing the host interface layer and the bus interfaces.

Wi-Fi callback registering is essential to allow the handling of the events received, in response to the asynchronous Wi-Fi operations.

Following are the possible Wi-Fi events that are expected to be received through the call back function (provided by the application) to the M2M_WIFI layer are:

- **M2M_WIFI_RESP_CON_STATE_CHANGED**
- **M2M_WIFI_RESP_CONN_INFO**
- **M2M_WIFI_REQ_WPS**
- **M2M_WIFI_RESP_SCAN_DONE**
- **M2M_WIFI_RESP_SCAN_RESULT**
- **M2M_WIFI_RESP_CURRENT_RSSI**
- **M2M_WIFI_RESP_CLIENT_INFO**

Example:

In case bypass mode is defined:

- **M2M_WIFI_RESP_ETHERNET_RX_PACKET**

In case Monitoring mode is used:

- **M2M_WIFI_RESP_WIFI_RX_PACKET**

Any application using the WILC driver must call this function at the start of its main function.

Parameters:

In	<i>pWifilnitParam</i>	This is a pointer to the <i>tstrWifilnitParam</i> structure which holds the pointer to the application WIFI layer call back function, monitoring mode call back and <i>tstrEthlInitParam</i> structure containing bypass mode parameters
----	-----------------------	--

Precondition:

Prior to this function call, application developers must provide a call back function responsible for receiving all the Wi-Fi events that are received on the M2M_WIFI layer.

Warning:

Failure to successfully complete function indicates that the driver couldn't be initialized and a fatal error will prevent the application from proceeding.

See also:

- **m2m_wifi_deinit**
- **tenuM2mStaCmd**

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- **m2m_wifi_deinit**
 - NMI_API sint8 m2m_wifi_deinit (void *arg)

Synchronous de-initialization function to the ATWILC1000 driver. Deinitializes the host interface and frees any resources used by the M2M_WIFI layer. This function must be called in the application closing phase, to ensure that all resources have been correctly released. No arguments are expected to be passed in.

Parameters:

In	arg	Generic argument. Not used in current implementation
----	-----	--

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- **m2m_wifi_handle_events**
 - NMI_API sint8 m2m_wifi_handle_events (void * arg)

Synchronous M2M event handler function, responsible for handling interrupts received from the WILC firmware. Application developers should call this function periodically in-order to receive the events that are to be handled by the callback functions implemented by the application.

Precondition:

Prior to receiving Wi-Fi interrupts, the WILC driver should have been successfully initialized by calling the **m2m_wifi_init** function.

Warning:

Failure to successfully complete this function indicates bus errors and hence a fatal error that will prevent the application from proceeding.

Returns:

The function returns **M2M_SUCCESS** for successful interrupt handling and a negative value otherwise.

- **m2m_wifi_connect**
 - NMI_API sint8 m2m_wifi_connect (char *pcSsid, uint8 u8SsidLen, uint8 u8SecType, void *pvAuthInfo, uint16 u16Ch)

Asynchronous Wi-Fi connection function to a specific AP. Prior to a successful connection, the application developers must know the SSID of the AP, the security type, the authentication information parameters and the channel number to which the connection will be established. The connection status is known when a response of **M2M_WIFI_RESP_CON_STATE_CHANGED** is received based on the states defined in **tenuM2mConnState**, successful connection is defined by **M2M_WIFI_CONNECTED**.

The only difference between this function and **m2m_wifi_default_connect**, is the connection parameters. Connection using this function is expected to be made to a specific AP and to a specified channel.

Parameters:

In	pcSsid	A buffer holding the SSID corresponding to the requested AP
In	u8SsidLen	Length of the given SSID (not including the NULL termination). A length less than ZERO or greater than the maximum defined SSID M2M_MAX_SSID_LEN will result in a negative error M2M_ERR_FAIL .
In	u8SecType	Wi-Fi security type security for the network. It can be one of the following types: - M2M_WIFI_SEC_OPEN - M2M_WIFI_SEC_WEP - M2M_WIFI_SEC_WPA_PSK - M2M_WIFI_SEC_802_1X A value outside these possible values will result in a negative return error M2M_ERR_FAIL .
In	pvAuthInfo	Authentication parameters required for completing the connection. Its type is based on the Security type. If the authentication parameters are NULL or are greater than the maximum length of the authentication parameters length as defined by M2M_MAX_PSK_LEN a negative error will return M2M_ERR_FAIL (-12) indicating connection failure.
In	u16Ch	Wi-Fi channel number as defined in tenuM2mScanCh enumeration. Channel number greater than M2M_WIFI_CH_14 returns a negative error M2M_ERR_FAIL (-12). Except if the value is M2M_WIFI_CH_ALL (255), since this indicates that the firmware should scan all channels to find the SSID requested to connect to. Failure to find the connection match will return a negative error M2M_DEFAULT_CONN_SCAN_MISMATCH .

Precondition:

Prior to a successful connection request, the Wi-Fi driver must have been successfully initialized through the call of the function.

Warning:

This function must be called in station mode only. Successful completion of this function does not guarantee success of the WIFI connection, and a negative return value indicates only locally detected errors.

See also:

- **tuniM2MWifiAuth**
- **tstr1xAuthCredentials**
- **tstrM2mWifiWepParams**

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- **m2m_wifi_disconnect**
 - NMI_API sint8 m2m_wifi_disconnect (void)

Precondition:

Disconnection must be made to a successfully connected AP. If the WILC is not in the connected state, a call to this function will hold insignificant.

Warning:

This function must be called in station mode only.

See also:

- **m2m_wifi_connect**
- **m2m_wifi_default_connect**

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- `m2m_wifi_get_connection_info`
 - NMI_API sint8 `m2m_wifi_get_connection_info` (void)

Asynchronous connection status retrieval function, retrieves the status information of the currently connected AP. The result is passed to the Wi-Fi notification callback through the event **M2M_WIFI_RESP_CONN_INFO**. Connection information is retrieved from the structure **tstrM2MConnInfo**. Request the status information of the currently connected Wi-Fi AP. The result is passed to the Wi-Fi notification callback with the event **M2M_WIFI_RESP_CONN_INFO**.

Precondition:

- A Wi-Fi notification callback of type `tpfAppWifiCb` MUST be implemented and registered at startup. Registering the callback is done through passing it to the initialization `m2m_wifi_init` function.
- The event **M2M_WIFI_RESP_CONN_INFO** must be handled in the callback to receive the requested connection info

Warning:

Calling this function is valid ONLY in the STA CONNECTED state. Otherwise, the WILC SW shall ignore the request silently.

See also:

- `tpfAppWifiCb`
- `m2m_wifi_init`
- **M2M_WIFI_RESP_CONN_INFO**
- **tstrM2MConnInfo**

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Example:

The code snippet shows an example of how Wi-Fi connection information is retrieved.

```

1 #include "m2m_wifi.h"
2 #include "m2m_types.h"
3
4
5 void wifi_event_cb(uint8 u8WifiEvent, void * pvMsg)
6 {
7     switch(u8WifiEvent)
8     {
9         case M2M_WIFI_RESP_CONN_INFO:
10             {
11                 tstrM2MConnInfo *pstrConnInfo = (tstrM2MConnInfo*)pvMsg;
12
13                 printf("CONNECTED AP INFO\n");
14                 printf("SSID           : %s\n",pstrConnInfo->acSSID);
15                 printf("SEC TYPE       : %d\n",pstrConnInfo->u8SecType);
16                 printf("Signal Strength : %d\n", pstrConnInfo->s8RSSI);
17                 printf("Local IP Address : %d.%d.%d.%d\n",
18                     pstrConnInfo->au8IPAddr[0], pstrConnInfo->au8IPAddr[1],
19                     pstrConnInfo->au8IPAddr[2], pstrConnInfo->au8IPAddr[3]);
20             }
21             break;
22

```

```

28     default:
29         break;
30     }
31 }
32
33 int main()
34 {
35     tstrWifiInitParam    param;
36
37     param.pfAppWifiCb    = wifi_event_cb;
38     if(!m2m_wifi_init(&param))
39     {
40         // connect to the default AP
41         m2m_wifi_default_connect();
42
43         while(1)
44         {
45             m2m_wifi_handle_events(NULL);
46         }
47     }
48 }

```

- m2m_wifi_set_mac_address
 - NMI_API sint8 m2m_wifi_set_mac_address (uint8 au8MacAddress[6])

Synchronous MAC address assigning to the NMC1000. It is used for non-production SW. Assign MAC address to the WILC device.

Parameters:

in	<i>au8MacAddress</i>	MAC Address to be provisioned to the WILC
----	----------------------	---

Returns:

The function returns M2M_SUCCESS for successful operations and a negative value otherwise.

- m2m_wifi_wps
 - NMI_API sint8 m2m_wifi_wps (uint8 u8TriggerType, const char * pcPinNumber)

Asynchronous WPS triggering function. This function is called for the WILC to enter the WPS (Wi-Fi Protected Setup) mode. The result is passed to the Wi-Fi notification callback with the event

M2M_WIFI_REQ_WPS.

Parameters:

In	<i>u8TriggerType</i>	WPS Trigger method. Could be: <ul style="list-style-type: none"> • WPS_PIN_TRIGGER Push button method • WPS_PBC_TRIGGER Pin method
In	<i>pcPinNumber</i>	PIN number for WPS PIN method. It is not used if the trigger type is WPS_PBC_TRIGGER. It must follow the rules stated by the WPS Standard.

Precondition:

- A Wi-Fi notification callback of type (tpfAppWifiCb MUST be implemented and registered at startup. Registering the callback is done through passing it to the m2m_wifi_init.
- The event M2M_WIFI_REQ_WPS must be handled in the callback to receive the WPS status

- The WILC device MUST be in IDLE or STA mode. If AP or P2P mode is active, the WPS will not be performed
- The `m2m_wifi_handle_events` MUST be called to receive the responses in the callback

Warning:

This function is not allowed in AP or P2P modes.

See also:

- `tpfAppWifiCb`
- `m2m_wifi_init`
- `M2M_WIFI_REQ_WPS`
- `tenuWPSTrigger`
- `tstrM2MWPSInfo`

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Example:

The code snippet shows an example of how Wi-Fi WPS is triggered.

```

1 #include "m2m_wifi.h"
2 #include "m2m_types.h"
3
4 void wifi_event_cb(uint8 u8WifiEvent, void * pvMsg)
5 {
6     switch(u8WifiEvent)
7     {
8         case M2M_WIFI_REQ_WPS:
9             {
10                 tstrM2MWPSInfo *pstrWPS = (tstrM2MWPSInfo*)pvMsg;
11                 if(pstrWPS->u8AuthType != 0)
12                 {
13                     printf("WPS SSID          : %s\n",pstrWPS->au8SSID);
14                     printf("WPS PSK          : %s\n",pstrWPS->au8PSK);
15                     printf("WPS SSID Auth Type : %s\n",pstrWPS->u8AuthType ==
M2M_WIFI_SEC_OPEN ? "OPEN" : "WPA/WPA2");
16                     printf("WPS Channel       : %d\n",pstrWPS->u8Ch + 1);
17
18                     // establish Wi-Fi connection
19                     m2m_wifi_connect((char*)pstrWPS->au8SSID,
(uint8)m2m_strlen(pstrWPS->au8SSID),
20                                 pstrWPS->u8AuthType, pstrWPS->au8PSK, pstrWPS->u8Ch);
21                 }
22                 else
23                 {
24                     printf("(ERR) WPS Is not enabled OR Timedout\n");
25                 }
26             }
27             break;
28
29         default:
30             break;
31     }
32 }
33
34 int main()

```

```

35 {
36     tstrWifiInitParam  param;
37
38     param.pfAppWifiCb   = wifi_event_cb;
39     if(!m2m_wifi_init(&param))
40     {
41         // Trigger WPS in Push button mode.
42         m2m_wifi_wps(WPS_PBC_TRIGGER, NULL);
43
44         while(1)
45         {
46             m2m_wifi_handle_events(NULL);
47         }
48     }
49 }

```

- m2m_wifi_wps_disable
 - NMI_API sint8 m2m_wifi_wps_disable (void)

Disable the NMC1000 WPS operation.

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- m2m_wifi_p2p
 - NMI_API sint8 m2m_wifi_p2p (uint8 u8Channel)

Asynchronous Wi-Fi direct (P2P) enabling mode function. The WILC supports P2P in device listening mode ONLY (intent is ZERO). The WILC P2P implementation does not support P2P GO (Group Owner) mode. Active P2P devices (e.g. phones) could find the WILC in the search list. When a device is connected to WILC, a Wi-Fi notification event **M2M_WIFI_RESP_CON_STATE_CHANGED** is triggered.

Parameters:

in	<i>u8Channel</i>	P2P Listen RF channel. According to the P2P standard It must hold only one of the following values 1, 6, or 11.
----	------------------	---

Precondition:

- A Wi-Fi notification callback of type tpfAppWifiCb MUST be implemented and registered at initialization. Registering the callback is done through passing it to the **m2m_wifi_init**.
- The events **M2M_WIFI_RESP_CON_STATE_CHANGED** must be handled in the callback
- The **m2m_wifi_handle_events** MUST be called to receive the responses in the callback

Warning:

This function is not allowed in AP or STA modes.

See also:

- tpfAppWifiCb
- m2m_wifi_init
- M2M_WIFI_RESP_CON_STATE_CHANGED
- tstrM2mWifiStateChanged

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Example:

The code snippet shown an example of how the P2P mode operates.

```
1 #include "m2m_wifi.h"
2 #include "m2m_types.h"
3
4 void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
5 {
6     switch(u8WiFiEvent)
7     {
8         case M2M_WIFI_RESP_CON_STATE_CHANGED:
9             {
10                 tstrM2mWifiStateChanged *pstrWifiState =
11 (tstrM2mWifiStateChanged*)pvMsg;
12                 M2M_INFO("Wi-Fi State :: %s :: ErrCode %d\n", pstrWifiState-
13 >u8CurrState? "CONNECTED":"DISCONNECTED",pstrWifiState->u8ErrCode);
14
15                 // Do something
16             }
17             break;
18
19         default:
20             break;
21     }
22 }
23
24 int main()
25 {
26     tstrWifiInitParam  param;
27
28     param.pfAppWifiCb  = wifi_event_cb;
29     if(!m2m_wifi_init(&param))
30     {
31         // Trigger P2P
32         m2m_wifi_p2p(1);
33
34         while(1)
35         {
36             m2m_wifi_handle_events(NULL);
37         }
38     }
39 }
```

- `m2m_wifi_p2p_disconnect`
 - NMI_API sint8 m2m_wifi_p2p_disconnect (void)

Disable the NMC1000 device Wi-Fi direct mode (P2P).

Precondition:

The P2P mode must have be enabled and active before a disconnect can be called.

See also:

- `m2m_wifi_p2p`

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- `m2m_wifi_enable_ap`
 - `NMI_API sint8 m2m_wifi_enable_ap (CONST tstrM2MAPConfig *pstrM2MAPConfig)`

Asynchronous Wi-Fi hotspot enabling function. The WILC supports AP mode operation with the following facts:

- Up to eight STA could be associated at a time in single mode of operation or seven in case of concurrency
- Open and WEP and WPA2 security types are supported

Parameters:

in	<i>pstrM2MAPConfig</i>	A structure holding the AP configurations
----	------------------------	---

Warning:

This function is not allowed in P2P or STA modes.

Precondition:

- A Wi-Fi notification callback of type `tpfAppWifiCb` MUST be implemented and registered at initialization. Registering the callback is done through passing it to the `m2m_wifi_init`.
- The `m2m_wifi_handle_events` MUST be called to receive the responses in the callback

See also:

- `tpfAppWifiCb`
- `tenuM2mSecType`
- `m2m_wifi_init`
- `tstrM2mWifiStateChanged`
- `tstrM2MAPConfig`

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Example:

The code snippet demonstrates how the AP mode is enabled after the driver is initialized in the application's main function.

```

1 #include "m2m_wifi.h"
2 #include "m2m_types.h"
3
4 void wifi_event_cb(uint8 u8WifiEvent, void * pvMsg)
5 {
6     switch(u8WifiEvent)
7     {
8         .....case M2M_WIFI_RESP_CON_STATE_CHANGED:
9             {
12                printf("One STA has Associated\n");
13            }
14            break;
15
16        default:
17            break;
18    }
19 }
20
21 int main()
22 {

```

```

23     tstrWifiInitParam    param;
24
25     param.pfAppWifiCb    = wifi_event_cb;
26     if(!m2m_wifi_init(&param))
27     {
28         tstrM2MAPConfig    apConfig;
29
30         strcpy(apConfig.au8SSID, "WILC_SSID");
31         apConfig.u8ListenChannel    = 1;
32         apConfig.u8SecType          = M2M_WIFI_SEC_OPEN;
33         apConfig.u8SsidHide         = 0;
34
35         // Trigger AP
36         m2m_wifi_enable_ap(&apConfig);
37
38         while(1)
39         {
40             m2m_wifi_handle_events(NULL);
41         }
42     }
43 }

```

- `m2m_wifi_disable_ap`
 - NMI_API sint8 `m2m_wifi_disable_ap` (void)

Synchronous Wi-Fi hotspot disabling function. Must be called only when the AP is enabled through the **m2m_wifi_enable_ap** function. Otherwise the call to this function will not be useful.

See also:

- **m2m_wifi_enable_ap**

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- `m2m_wifi_ap_get_assoc_info`
 - NMI_API sint8 `m2m_wifi_ap_get_assoc_info`(void)

Asynchronous connection status retrieval function in AP mode, retrieves the status information of the currently associated stations in AP mode. The result is passed to the Wi-Fi notification callback through the event **M2M_WIFI_RESP_AP_ASSOC_INFO**. Association information is retrieved from the structure **tstrM2MAPAssocInfo**. Request the status information of the currently associated stations in AP mode. The result is passed to the Wi-Fi notification callback with the event **M2M_WIFI_RESP_AP_ASSOC_INFO**.

Precondition:

- A Wi-Fi notification callback of type `tpfAppWifiCb` MUST be implemented and registered at startup. Registering the callback is done through passing it to the initialization **m2m_wifi_init** function.
- The event **M2M_WIFI_RESP_AP_ASSOC_INFO** must be handled in the callback to receive the requested connection info

Warning:

Calling this function is valid ONLY in the AP mode. Otherwise, the WILC SW shall ignore the request silently.

See also:

- tpfAppWifiCb
- m2m_wifi_init
- M2M_WIFI_RESP_AP_ASSOC_INFO
- tstrM2MAPAssocInfo

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Example:

The code snippet shows an example of how association information is retrieved.

```
#include "m2m_wifi.h"
#include "m2m_types.h"

void wifi_event_cb(uint8 u8WifiEvent, void * pvMsg)
{
    switch(u8WifiEvent)
    {
        case M2M_WIFI_RESP_AP_ASSOC_INFO:
        {
            tstrM2MAPAssocInfo* pstrAssocInfo =(tstrM2MAPAssocInfo*)pvMsg;
            printk("AP Assoc list received[%d]\n",pstrAssocInfo->u8No-
ConnSta);

            for(i=0;i<pstrAssocInfo->u8NoConnSta;i++)
            {
                printk("STA %x:%x:%x:%x:%x:%x connected RSSI
%d\n",pstrAssocInfo->astrM2MAssocEntryInfo[i].BSSID[0],
pstrAssocInfo->astrM2MAssocEntryInfo[i].BSSID[1],pstrAs-
socInfo->astrM2MAssocEntryInfo[i].BSSID[2],
pstrAssocInfo->astrM2MAssocEntryInfo[i].BSSID[3],pstrAs-
socInfo->astrM2MAssocEntryInfo[i].BSSID[4],
pstrAssocInfo->astrM2MAssocEntryInfo[i].BSSID[5],pstrAs-
socInfo->astrM2MAssocEntryInfo[i].s8RSSI);
            }

        }
        break;
        default:
        break;
    }
}

int main()
{
    tstrWifiInitParam param;

    param.pfAppWifiCb = wifi_event_cb;
    if(!m2m_wifi_init(&param))
    {
        strcpy(strM2MAPConfig.au8WepKey,"1234567890");
        strM2MAPConfig.u8KeySz = WEP_40_KEY_STRING_SIZE;
        strM2MAPConfig.u8KeyIndx = 0;
        strcpy(strM2MAPConfig.au8SSID,"WILC1000_AP");
        strM2MAPConfig.u8ListenChannel = M2M_WIFI_CH_11;
        strM2MAPConfig.u8SecType = M2M_WIFI_SEC_WEP;
    }
}
```

```

        strM2MAPConfig.u8SsidHide = 0;
        //start AP mode
        m2m_wifi_enable_ap(&strM2MAPConfig);

        while(1)
        {
            m2m_wifi_handle_events(NULL);
        }
    }
}

```

- m2m_wifi_set_scan_options
 - NMI_API sint8 m2m_wifi_set_scan_options (uint8 u8NumOfSlot, uint8 u8SlotTime)

Synchronous Wi-Fi scan settings function. This function sets the time configuration parameters for the scan operation.

Parameters:

in	<i>u8NumOfSlot;</i>	The minimum number of slots is 2 for every channel. For every slot the SoC will send Probe Req on air, and wait/listen for PROBE RESP/BEACONS for the u8slot-Time in ms.
in	<i>u8SlotTime;</i>	The time in ms that the SoC will wait on every channel listening for the frames on air when that time increases the number of APs will increase in the scan results Minimum time is 10ms and the maximum is 250ms

See also:

- tenuM2mScanCh
- m2m_wifi_request_scan

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- m2m_wifi_set_scan_region
 - NMI_API sint8 m2m_wifi_set_scan_region (uint8 ScanRegion)

Synchronous Wi-Fi scan region setting function. This function sets the scan region, which will affect the range of possible scan channels. For 2.4GHz supported in the current release, the requested scan region can't exceed the maximum number of channels (14).

Parameters:

in	<i>ScanRegion;</i>	ASIA = 14 NORTH_AMERICA = 11
----	--------------------	------------------------------

See also:

- tenuM2mScanCh
- m2m_wifi_request_scan

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- m2m_wifi_request_scan
 - NMI_API sint8 m2m_wifi_request_scan (uint8 ch)

Asynchronous Wi-Fi scan request on the given channel. The scan status is delivered in the Wi-Fi event callback and then the application is to read the scan results sequentially. The number of APs found (N) is returned in event **M2M_WIFI_RESP_SCAN_DONE** with the number of found APs. The application could read the list of APs by calling the function **m2m_wifi_req_scan_result** N times.

Parameters:

in	ch	RF Channel ID for SCAN operation. It should be set according to <code>tenuM2mScanCh</code> . With a value of <code>M2M_WIFI_CH_ALL(255)</code> , means to scan all channels.
----	----	--

Warning:

This function is not allowed in P2P or AP modes. It works only for STA mode (connected or disconnected).

Precondition:

- A Wi-Fi notification callback of type `tpfAppWifiCb` MUST be implemented and registered at initialization. Registering the callback is done through passing it to the `m2m_wifi_init`.
- The events `M2M_WIFI_RESP_SCAN_DONE` and `M2M_WIFI_RESP_SCAN_RESULT` must be handled in the callback
- The `m2m_wifi_handle_events` function MUST be called to receive the responses in the callback

See also:

- `M2M_WIFI_RESP_SCAN_DONE`
- `M2M_WIFI_RESP_SCAN_RESULT`
- `tpfAppWifiCb`
- `tstrM2mWifiscanResult`
- `tenuM2mScanCh`
- `m2m_wifi_init`
- `m2m_wifi_handle_events`
- `m2m_wifi_req_scan_result`

Returns:

The function returns `M2M_SUCCESS` for successful operations and a negative value otherwise.

Example:

The code snippet demonstrates an example of how the scan request is called from the application's main function and the handling of the events received in response.

```
1 #include "m2m_wifi.h"
2 #include "m2m_types.h"
3
4 void wifi_event_cb(uint8 u8WifiEvent, void * pvMsg)
5 {
6     static uint8    u8ScanResultIdx = 0;
7
8     switch(u8WifiEvent)
9     {
10    case M2M_WIFI_RESP_SCAN_DONE:
11        {
12            tstrM2mScanDone *pstrInfo = (tstrM2mScanDone*)pvMsg;
13
14            printf("Num of AP found %d\n",pstrInfo->u8NumofCh);
15            if(pstrInfo->s8ScanState == M2M_SUCCESS)
16            {
17                u8ScanResultIdx = 0;
18                if(pstrInfo->u8NumofCh >= 1)
19                {
20                    m2m_wifi_req_scan_result(u8ScanResultIdx);
21                    u8ScanResultIdx ++;
```

```

22         }
23         else
24         {
25             printf("No AP Found Rescan\n");
26             m2m_wifi_request_scan(M2M_WIFI_CH_ALL);
27         }
28     }
29     else
30     {
31         printf("(ERR) Scan fail with error <%d>\n",pstrInfo-
>s8ScanState);
32     }
33 }
34 break;
35
36 case M2M_WIFI_RESP_SCAN_RESULT:
37 {
38     tstrM2mWifiscanResult      *pstrScanResult
=(tstrM2mWifiscanResult*)pvMsg;
39     uint8                      u8NumFoundAPs =
m2m_wifi_get_num_ap_found();
40
41     printf(">>%02d RI %d SEC %s CH %02d BSSID
%02X:%02X:%02X:%02X:%02X:%02X SSID %s\n",
42         pstrScanResult->u8index,pstrScanResult->s8rssi,
43         pstrScanResult->u8AuthType,
44         pstrScanResult->u8ch,
45         pstrScanResult->au8BSSID[0], pstrScanResult->au8BSSID[1],
pstrScanResult->au8BSSID[2],
46         pstrScanResult->au8BSSID[3], pstrScanResult->au8BSSID[4],
pstrScanResult->au8BSSID[5],
47         pstrScanResult->au8SSID);
48
49     if(u8ScanResultIdx < u8NumFoundAPs)
50     {
51         // Read the next scan result
52         m2m_wifi_req_scan_result(index);
53         u8ScanResultIdx ++;
54     }
55 }
56 break;
57 default:
58     break;
59 }
60 }
61
62 int main()
63 {
64     tstrWifiInitParam    param;
65
66     param.pfAppWifiCb    = wifi_event_cb;
67     if(!m2m_wifi_init(&param))
68     {
69         // Scan all channels
70         m2m_wifi_request_scan(M2M_WIFI_CH_ALL);
71

```

```

72     while(1)
73     {
74         m2m_wifi_handle_events(NULL);
75     }
76 }

```

- `m2m_wifi_get_num_ap_found`
 - NMI_API uint8 m2m_wifi_get_num_ap_found (void)

Synchronous function to retrieve the number of AP's found in the last scan request. The function read the number of AP's from global variable which updated in the Wi-Fi callback function through the **M2M_WIFI_RESP_SCAN_DONE** event. Function used only in STA mode only.

Precondition:

- `m2m_wifi_request_scan` need to be called first
- A Wi-Fi notification callback of type `tpfAppWifiCb` MUST be implemented and registered at initialization. Registering the callback is done through passing it to the `m2m_wifi_init`.
- The event **M2M_WIFI_RESP_SCAN_DONE** must be handled in the callback to receive the requested connection information

Warning:

- This function must be called only in the Wi-Fi callback function when the events **M2M_WIFI_RESP_SCAN_DONE** or **M2M_WIFI_RESP_SCAN_RESULT** are received. Calling this function in any other place will result in undefined/outdated numbers.

See also:

- `m2m_wifi_request_scan`
- **M2M_WIFI_RESP_SCAN_DONE**
- **M2M_WIFI_RESP_SCAN_RESULT**

Returns:

Return the number of AP's found in the last Scan Request.

Example:

The code snippet demonstrates an example of how the scan request is called from the application's main function and the handling of the events received in response.

```

1  #include "m2m_wifi.h"
2  #include "m2m_types.h"
3
4  void wifi_event_cb(uint8 u8WifiEvent, void * pvMsg)
5  {
6      static uint8    u8ScanResultIdx = 0;
7
8      switch(u8WifiEvent)
9      {
10     case M2M_WIFI_RESP_SCAN_DONE:
11         {
12             tstrM2mScanDone *pstrInfo = (tstrM2mScanDone*)pvMsg;
13
14             printf("Num of AP found %d\n",pstrInfo->u8NumofCh);
15             if(pstrInfo->s8ScanState == M2M_SUCCESS)
16             {
17                 u8ScanResultIdx = 0;
18                 if(pstrInfo->u8NumofCh >= 1)

```

```

19         {
20             m2m_wifi_req_scan_result(u8ScanResultIdx);
21             u8ScanResultIdx ++;
22         }
23         else
24         {
25             printf("No AP Found Rescan\n");
26             m2m_wifi_request_scan(M2M_WIFI_CH_ALL);
27         }
28     }
29     else
30     {
31         printf("(ERR) Scan fail with error <%d>\n",pstrInfo->s8ScanState);
32     }
33 }
34 break;
35
36 case M2M_WIFI_RESP_SCAN_RESULT:
37     {
38         tstrM2mWifiscanResult      *pstrScanResult
=(tstrM2mWifiscanResult*)pvMsg;
39         uint8                      u8NumFoundAPs =
m2m_wifi_get_num_ap_found();
40
41         printf(">>%02d RI %d SEC %s CH %02d BSSID
%02X:%02X:%02X:%02X:%02X:%02X SSID %s\n",
42             pstrScanResult->u8index,pstrScanResult->s8rssi,
43             pstrScanResult->u8AuthType,
44             pstrScanResult->u8ch,
45             pstrScanResult->au8BSSID[0], pstrScanResult->au8BSSID[1],
pstrScanResult->au8BSSID[2],
46             pstrScanResult->au8BSSID[3], pstrScanResult->au8BSSID[4],
pstrScanResult->au8BSSID[5],
47             pstrScanResult->au8SSID);
48
49         if(u8ScanResultIdx < u8NumFoundAPs)
50         {
51             // Read the next scan result
52             m2m_wifi_req_scan_result(index);
53             u8ScanResultIdx ++;
54         }
55     }
56     break;
57 default:
58     break;
59 }
60 }
61
62 int main()
63 {
64     tstrWifiInitParam    param;
65
66     param.pfAppWifiCb    = wifi_event_cb;
67     if(!m2m_wifi_init(&param))
68     {

```



```

69      // Scan all channels
70      m2m_wifi_request_scan(M2M_WIFI_CH_ALL);
71
72      while(1)
73      {
74          m2m_wifi_handle_events(NULL);
75      }
76  }

```

- `m2m_wifi_req_scan_result`
 - NMI_API sint8 `m2m_wifi_req_scan_result` (uint8 index)

Synchronous call to read the AP information from the SCAN Result list with the given index. This function is expected to be called when the response events `M2M_WIFI_RESP_SCAN_RESULT` or `M2M_WIFI_RESP_SCAN_DONE` are received in the Wi-Fi callback function. The response information received can be obtained through the casting to the **`tstrM2mWifiscanResult`** structure.

Parameters:

in	index	Index for the requested result, the index range start from 0 till number of AP's found
----	-------	--

See also:

- **`tstrM2mWifiscanResult`**
- **`m2m_wifi_get_num_ap_found`**
- **`m2m_wifi_request_scan`**

Precondition:

- **`m2m_wifi_request_scan`** needs to be called first, then `m2m_wifi_get_num_ap_found` to get the number of AP's found
- A Wi-Fi notification callback of type `tpfAppWifiCb` MUST be implemented and registered at startup. Registering the callback is done through passing it to the **`m2m_wifi_init`** function.
- The event **`M2M_WIFI_RESP_SCAN_RESULT`** must be handled in the callback to receive the requested connection information

Warning:

Function used in STA mode only. The scan results are updated only if the scan request is called. Calling this function only without a scan request will lead to firmware errors. Refrain from introducing a large delay between the scan request and the scan result request, to prevent an errors occurring.

Returns:

The function returns **`M2M_SUCCESS`** for successful operations and a negative value otherwise.

Example:

The code snippet demonstrates an example of how the scan request is called from the application's main function and the handling of the events received in response.

```

1 #include "m2m_wifi.h"
2 #include "m2m_types.h"
3
4 void wifi_event_cb(uint8 u8WifiEvent, void * pvMsg)
5 {
6     static uint8    u8ScanResultIdx = 0;
7
8     switch(u8WifiEvent)
9     {

```

```

10     case M2M_WIFI_RESP_SCAN_DONE:
11     {
12         tstrM2mScanDone *pstrInfo = (tstrM2mScanDone*)pvMsg;
13
14         printf("Num of AP found %d\n",pstrInfo->u8NumofCh);
15         if(pstrInfo->s8ScanState == M2M_SUCCESS)
16         {
17             u8ScanResultIdx = 0;
18             if(pstrInfo->u8NumofCh >= 1)
19             {
20                 m2m_wifi_req_scan_result(u8ScanResultIdx);
21                 u8ScanResultIdx ++;
22             }
23             else
24             {
25                 printf("No AP Found Rescan\n");
26                 m2m_wifi_request_scan(M2M_WIFI_CH_ALL);
27             }
28         }
29         else
30         {
31             printf("(ERR) Scan fail with error <%d>\n",pstrInfo-
>s8ScanState);
32         }
33     }
34     break;
35
36     case M2M_WIFI_RESP_SCAN_RESULT:
37     {
38         tstrM2mWifiscanResult      *pstrScanResult
=(tstrM2mWifiscanResult*)pvMsg;
39         uint8                      u8NumFoundAPs =
m2m_wifi_get_num_ap_found();
40
41         printf(">>%02d RI %d SEC %s CH %02d BSSID
%02X:%02X:%02X:%02X:%02X:%02X SSID %s\n",
42             pstrScanResult->u8index,pstrScanResult->s8rssi,
43             pstrScanResult->u8AuthType,
44             pstrScanResult->u8ch,
45             pstrScanResult->au8BSSID[0], pstrScanResult->au8BSSID[1],
pstrScanResult->au8BSSID[2],
46             pstrScanResult->au8BSSID[3], pstrScanResult->au8BSSID[4],
pstrScanResult->au8BSSID[5],
47             pstrScanResult->au8SSID);
48
49         if(u8ScanResultIdx < u8NumFoundAPs)
50         {
51             // Read the next scan result
52             m2m_wifi_req_scan_result(index);
53             u8ScanResultIdx ++;
54         }
55     }
56     break;
57     default:
58     break;
59 }

```

```

60 }
61
62 int main()
63 {
64     tstrWifiInitParam    param;
65
66     param.pfAppWifiCb    = wifi_event_cb;
67     if(!m2m_wifi_init(&param))
68     {
69         // Scan all channels
70         m2m_wifi_request_scan(M2M_WIFI_CH_ALL);
71
72         while(1)
73         {
74             m2m_wifi_handle_events(NULL);
75         }
76     }

```

- m2m_wifi_req_curr_rssi
 - NMI_API sint8 m2m_wifi_req_curr_rssi (void)

Asynchronous request for the current RSSI of the connected AP. The response received in through the **M2M_WIFI_RESP_CURRENT_RSSI** event.

Precondition:

- A Wi-Fi notification callback of type tpfAppWifiCb MUST be implemented and registered before initialization. Registering the callback is done through passing it to the **m2m_wifi_init** through the **tstrWifiInitParam** initialization structure.
- The event **M2M_WIFI_RESP_CURRENT_RSSI** must be handled in the callback to receive the requested connection information

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Example:

The code snippet demonstrates how the RSSI request is called in the application's main function and the handling of event received in the callback.

```

1 #include "m2m_wifi.h"
2 #include "m2m_types.h"
3
4 void wifi_event_cb(uint8 u8WifiEvent, void * pvMsg)
5 {
6     static uint8    u8ScanResultIdx = 0;
7
8     switch(u8WifiEvent)
9     {
10     case M2M_WIFI_RESP_CURRENT_RSSI:
11         {
12             sint8    *rssi = (sint8*)pvMsg;
13             M2M_INFO("ch rssi %d\n",*rssi);
14         }
15         break;
16     default:
17         break;
18     }

```

```

19 }
20
21 int main()
22 {
23     tstrWifiInitParam  param;
24
25     param.pfAppWifiCb   = wifi_event_cb;
26     if(!m2m_wifi_init(&param))
27     {
28         // Scan all channels
29         m2m_wifi_req_curr_rssi();
30
31         while(1)
32         {
33             m2m_wifi_handle_events(NULL);
34         }
35     }

```

- `m2m_wifi_get_otp_mac_address`
 - NMI_API sint8 `m2m_wifi_get_otp_mac_address` (uint8 *pu8MacAddr, uint8 *pu8IsValid)

Request the MAC address stored on the OTP (one time programmable) memory of the device. The function is blocking until the response is received.

Parameters:

out	<i>pu8MacAddr</i>	Output MAC address buffer of 6 bytes size. Valid only if *pu8Valid=1.
out	<i>pu8IsValid</i>	An output boolean value to indicate the validity of pu8MacAddr in OTP. Output zero if the OTP memory is not programmed, non-zero otherwise.

Precondition:

`m2m_wifi_init` required to call any WIFI function

See also:

- `m2m_wifi_get_mac_address`

Returns:

The function returns **M2M_SUCCESS** for success and a negative value otherwise.

- `m2m_wifi_get_mac_address`
 - NMI_API sint8 `m2m_wifi_get_mac_address` (uint8 *pu8MacAddr)

Function to retrieve the current MAC address. The function is blocking until the response is received.

Parameters:

out	<i>pu8MacAddr</i>	Output MAC address buffer of 6 bytes size
-----	-------------------	---

Precondition:

`m2m_wifi_init` required to be called before any WIFI function.

See also:

- `m2m_wifi_get_otp_mac_address`

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- `m2m_wifi_set_sleep_mode`
 - NMI_API sint8 `m2m_wifi_set_sleep_mode` (uint8 `PsTyp`, uint8 `BcastEn`)

Synchronous power-save mode setting function for the NMC1000.

Parameters:

in	PsTyp	Desired power saving mode. Supported types are defined in tenuPowerSaveModes .
in	BcastEn	Broadcast reception enable flag. If it is 1, the ATWILC1000 must be awake each DTIM beacon for receiving broadcast traffic. If it is 0, the ATWILC1000 will not wakeup at the DTIM beacon, but its wakeup depends only on the configured Listen Interval.

Warning:

The function called once after initialization.

See also:

- **tenuPowerSaveModes**
- **m2m_wifi_get_sleep_mode**

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- `m2m_wifi_request_sleep`
 - NMI_API sint8 `m2m_wifi_request_sleep` (uint32 `u32SlpReqTime`)

Synchronous power save request function, which requests from the NMC1000 device to sleep in the mode previously set for a specific time. This function should be used in the M2M_PS_MANUAL Power save mode (only).

Parameters:

in	u32SlpReqTime	Request Sleep in ms
----	---------------	---------------------

Warning:

The function should be called in M2M_PS_MANUAL power save only.

See also:

- **tenuPowerSaveModes**
- **m2m_wifi_set_sleep_mode**

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- `m2m_wifi_get_sleep_mode`
 - NMI_API uint8 `m2m_wifi_get_sleep_mode` (void)

Synchronous power save mode retrieval function.

See also:

- **tenuPowerSaveModes**
- **m2m_wifi_set_sleep_mode**

Returns:

The current operating power saving mode.

- `m2m_wifi_set_device_name`
 - NMI_API sint8 `m2m_wifi_set_device_name` (uint8 *`pu8DeviceName`, uint8 `u8DeviceNameLength`)

Set the ATWILC1000 device name which is to be used as a P2P device name.

Parameters:

In	<i>pu8DeviceName</i>	Buffer holding the device name
In	<i>u8DeviceNameLength</i>	Length of the device name. Should not exceed the maximum device name's length M2M_DEVICE_NAME_MAX.

Warning:

The function should be called once after initialization.

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- `m2m_wifi_set_lsn_int`
 - NMI_API sint8 `m2m_wifi_set_lsn_int` (tstrM2mLsnInt *`pstrM2mLsnInt`)

Synchronous function for setting the Wi-Fi listen interval for power save operation. It is represented in units of AP Beacon periods. Function

Parameters:

In	<i>pstrM2mLsnInt</i>	Structure holding the listen interval configurations
----	----------------------	--

Precondition:

Function `m2m_wifi_set_sleep_mode` shall be called first.

Warning:

The function should be called once after initialization.

See also:

- `tstrM2mLsnInt`
- `m2m_wifi_set_sleep_mode`

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- `m2m_wifi_send_ethernet_pkt`
 - NMI_API sint8 `m2m_wifi_send_ethernet_pkt` (uint8 *`pu8Packet`, uint16 `u16PacketSize`)

Synchronous function to transmit an Ethernet packet. Transmit a packet directly in bypass mode where the TCP/IP stack is disabled and the implementation of this packet is left to the application developer. The Ethernet packet composition is left to the application developer.

Parameters:

In	<i>pu8Packet</i>	Pointer to a buffer holding the whole Ethernet frame
In	<i>u16PacketSize</i>	The size of the whole bytes in packet

Note:

Packets are the user's responsibility.

Returns:

The function returns M2M_SUCCESS for successful operations and a negative value otherwise.

- m2m_wifi_set_cust_InfoElement
 - NMI_API sint8 m2m_wifi_set_cust_InfoElement (uint8 *pau8M2mCustInfoElement)

Synchronous function to Add/Remove user-defined Information Element to the Wi-Fi beacon and Probe Response frames while chip mode is Access Point Mode.

According to the information element layout shown below, if it is required to set new data for the information elements, pass in the buffer with the information according to the sizes and ordering defined below. However, if it's required to delete these IEs, fill the buffer with zeros.

Parameters:

In	<i>pau8M2mCustInfoElement</i>	Pointer to Buffer containing the IE to be sent. It is the application developer's responsibility to ensure on the correctness of the information element's ordering passed in.
----	-------------------------------	--

Note:

IEs Format will be follow the following layout:

Byte[0]	Byte[1]	Byte[2]	Byte[3:length1+2]	Byte[n]	Byte[n+1]	Byte[n+2:lengthx+2]
#of all Bytes	IE1 ID	Length1	Data1(Hex Coded)	IEx ID	Lengthx	Datax(Hex Coded)

Warning:

Size of All elements combined must not exceed 255 byte.

- Used in Access Point Mode

See also:

- m2m_wifi_enable_sntp
- tstrSystemTime

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Example:

The example demonstrates how the information elements are set using this function.

```
1 char elementData[21];
2 static char state = 0; // To Add, Append, and Delete
3 if(0 == state) { //Add 3 IEs
4     state = 1;
5     //Total Number of Bytes
6     elementData[0]=12;
7     //First IE
8     elementData[1]=200; elementData[2]=1; elementData[3]='A';
9     //Second IE
10    elementData[4]=201; elementData[5]=2; elementData[6]='B';
11    elementData[7]='C';
12    //Third IE
13    elementData[8]=202; elementData[9]=3; elementData[10]='D';
14    elementData[11]=0; elementData[12]='F';
15 } else if(1 == state) {
```

```

14    //Append 2 IEs to others, Notice that we keep old data in array starting
with\n
15    //element 13 and total number of bytes increased to 20
16    state = 2;
17    //Total Number of Bytes
18    elementData[0]=20;
19    //Fourth IE
20    elementData[13]=203; elementData[14]=1; elementData[15]='G';
21    //Fifth IE
22    elementData[16]=204; elementData[17]=3; elementData[18]='X';
elementData[19]=5; elementData[20]='Z';
23 } else if(2 == state) { //Delete All IEs
24     state = 0;
25     //Total Number of Bytes
26     elementData[0]=0;
27 }
28 m2m_wifi_set_cust_InfoElement(elementData);

```

- m2m_wifi_enable_mac_mcast
 - NMI_API sint8 m2m_wifi_enable_mac_mcast (uint8 *pu8MulticastMacAddress, uint8 u8AddRemove)

Synchronous function to Add/Remove MAC addresses in the multicast filter to receive multicast packets in bypass mode.

Parameters:

in	<i>pu8MulticastMacAddress</i>	Pointer to MAC address
in	<i>u8AddRemove</i>	A flag to add or remove the MAC ADDRESS, based on the following values: 0: remove MAC address 1: add MAC address

Note:

Maximum number of MAC addresses that could be added is 8.

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- m2m_wifi_set_receive_buffer
 - NMI_API sint8 m2m_wifi_set_receive_buffer (void *pvBuffer, uint16 u16BufferLen)

Synchronous function for setting or changing the receiver buffer's length. Changes are made according to the developer option in bypass mode and this function should be called in the receive callback handling.

Parameters:

In	<i>pvBuffer</i>	Pointer to Buffer to receive data. NULL pointer causes a negative error M2M_ERR_FAIL.
In	<i>u16BufferLen</i>	Length of data to be received

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- m2m_wifi_set_control_ifc
 - NMI_API sint8 m2m_wifi_set_control_ifc(uint8 u8IfcId)

Synchronous function for setting the interface that will be under control, i.e. all the coming control functions will apply on that interface, this API is used in case of using the concurrency.

Parameters:

In	<i>u8IfcId</i>	Interface ID either 1 or 2
----	----------------	----------------------------

Warning:

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

A.2 BSP

This module contains NMC1000 BSP APIs declarations.

A.2.1 Defines

Defines	Definition	Value
#define NMI_API	Attribute used to define memory section to map Functions in host memory	
#define CONST	Used for code portability	const
#define NULL	Void Pointer to '0' in case of NULL is not defined	((void*)0)
#define BSP_MIN	Computes the minimum of x and y	(x, y) ((x)>(y)?(y):(x))

```
n typedef void(* tpfNmBsplsr) (void)
```

Pointer to function. Used as a data type of ISR function registered by **nm_bsp_register_isr**.

A.2.2 Data Types

Define	Definition
unsigned char uint8	Range of values between 0 to 255
unsigned short uint16	Range of values between 0 to 65535
unsigned long uint32	Range of values between 0 to 4294967295
signed char sint	Range of values between -128 to 127
signed short sint16	Range of values between -32768 to 32767
signed long sint32	Range of values between -2147483648 to 2147483647

A.2.3 Function

- nm_bsp_init
 - sint8 nm_bsp_init (void)

Initialization for BSP such as Reset and Chip Enable Pins for WILC, delays, register ISR, enable/disable IRQ for WILC, etc. You must use this function in the head of your application to enable WILC and Host Driver communicate each other.

Note:

Implementation of this function is host dependent.

Warning:

Missing use will lead to failure in driver initialization.

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- nm_bsp_deinit
 - sint8 nm_bsp_deinit (void)

De-initialization for BSP (Board Support Package).

Precondition:

Initialize **nm_bsp_init** first.

Note:

Implementation of this function is host dependent.

Warning:

Missing use may lead to unknown behavior in case of soft reset.

See also:

- **nm_bsp_init**

Returns:

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

- **nm_bsp_reset**
 - void nm_bsp_reset (void)

Resetting NMC1000 SoC by setting CHIP_EN and RESET_N signals low, then after specific delay the function will put CHIP_EN high then RESET_N high, for the timing between signals, review the WILC datasheet.

Precondition:

Initialize **nm_bsp_init** first.

Note:

Implementation of this function is host dependent and called by HIF layer.

See also:

- **nm_bsp_init**

Returns:

None

- **nm_bsp_sleep**
 - void nm_bsp_sleep (uint32 u32TimeMsec)

Sleep in units of milliseconds. This function used by HIF Layer according to different situations.

Parameters:

in	<i>u32TimeMsec</i>	Time unit in milliseconds
----	--------------------	---------------------------

Precondition:

Initialize **nm_bsp_init** first.

Note:

Implementation of this function is host dependent.

Warning:

Maximum value must not exceed 4294967295 milliseconds which is equal to 4294967.295 seconds.

See also:

- **nm_bsp_init**

Returns:

None

- **nm_bsp_register_isr**

- void nm_bsp_register_isr (tpfNmBsplsr pflsr)

Register ISR (Interrupt Service Routine) in the initialization of HIF (Host Interface) Layer.

When the interrupt trigger the **BSP** layer should call the **pfISR** function once inside the interrupt.

in	<i>pflsr</i>	Pointer to ISR handler in HIF
----	--------------	-------------------------------

Warning:

Make sure that ISR for IRQ pin for WILC is enabled by default in your implementation.

Note:

Implementation of this function is host dependent and called by HIF layer.

See also:

- tpfNmBsplsr

Returns:

None

- void nm_bsp_interrupt_ctrl (uint8 u8Enable)
 - void nm_bsp_interrupt_ctrl (uint8 u8Enable)

Synchronous enable/disable the MCU interrupts.

Parameters:

in	<i>u8Enable</i>	'0' disable interrupts. '1' enable interrupts
----	-----------------	---

Note:

Implementation of this function is host dependent and called by HIF layer.

See also:

- tpfNmBsplsr

Returns:

None

A.2.4 Enumeration/Typedef

A.2.4.1 Asynchronous Events

Specific enumeration used for asynchronous operations

ATMEL EVALUATION BOARD/KIT IMPORTANT NOTICE AND DISCLAIMER

This evaluation board/kit is intended for user's internal development and evaluation purposes only. It is not a finished product and may not comply with technical or legal requirements that are applicable to finished products, including, without limitation, directives or regulations relating to electromagnetic compatibility, recycling (WEEE), FCC, CE or UL. Atmel is providing this evaluation board/kit "AS IS" without any warranties or indemnities. The user assumes all responsibility and liability for handling and use of the evaluation board/kit including, without limitation, the responsibility to take any and all appropriate precautions with regard to electrostatic discharge and other technical issues. User indemnifies Atmel from any claim arising from user's handling or use of this evaluation board/kit. Except for the limited purpose of internal development and evaluation as specified above, no license, express or implied, by estoppel or otherwise, to any Atmel intellectual property right is granted hereunder. ATMEL SHALL NOT BE LIABLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RELATING TO USE OF THIS EVALUATION BOARD/KIT.

ATMEL CORPORATION
1600 Technology Drive
San Jose, CA 95110
USA

13 Document Revision History

Doc Rev.	Date	Comments
42504A	10/2015	Initial document release.



Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA T: (+1)(408) 441.0311 F: (+1)(408) 436.4200 | www.atmel.com

© 2015 Atmel Corporation. / Rev.: Atmel-42504A-ATWILC1000-SPI-Wi-Fi-Link-Controller_UserGuide_102015.

Atmel®, Atmel logo and combinations thereof, Enabling Unlimited Possibilities®, and others are registered trademarks or trademarks of Atmel Corporation in U.S. and other countries. ARM®, ARM Connected® logo, and others are the registered trademarks or trademarks of ARM Ltd. Other terms and product names may be trademarks of others.

DISCLAIMER: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. EXCEPT AS SET FORTH IN THE ATMEL TERMS AND CONDITIONS OF SALES LOCATED ON THE ATMEL WEBSITE, ATMEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS AND PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and products descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

SAFETY-CRITICAL, MILITARY, AND AUTOMOTIVE APPLICATIONS DISCLAIMER: Atmel products are not designed for and will not be used in connection with any applications where the failure of such products would reasonably be expected to result in significant personal injury or death ("Safety-Critical Applications") without an Atmel officer's specific written consent. Safety-Critical Applications include, without limitation, life support devices and systems, equipment or systems for the operation of nuclear facilities and weapons systems. Atmel products are not designed nor intended for use in military or aerospace applications or environments unless specifically designated by Atmel as military-grade. Atmel products are not designed nor intended for use in automotive applications unless specifically designated by Atmel as automotive-grade.