ATWILC1000/ATWILC3000

Baremetal Wi-Fi®/BLE Link Controller Software Design Guide

Introduction

The SmartConnect ATWILC* Baremetal is an IEEE® 802.11b/g/n link controller and Bluetooth® 5 controller SoC for Internet of Things (IoT) applications. This feature allows for the addition of Wi-Fi and Bluetooth to an MPU/MCU application via an SDIO/SPI to Wi-Fi and UART to Bluetooth. The ATWILC* family contains two flavors:

- ATWILC1000, which is a Wi-Fi only SoC
- ATWILC3000, which is Wi-Fi and Bluetooth/BLE SoC

Note: All references to the ATWILC* module include ATWILC1000 and ATWILC3000, unless otherwise noted.

Features

Wi-Fi

- Ultra Low Cost IEEE 802.11b/g/n RF/PH/MAC SoC
- Low-Power Consumption with Different Power-Saving Modes
- Wi-Fi IEEE 802.11 b/g/n with Station (STA), Access Point (AP), and Wi-Fi Direct® Client Modes
- Wi-Fi Protected Setup (WPS) for STA Mode
- Support of WEP, WPA/WPA2 Personal
- Support for WPA/WPA2 and Enterprise Security for STA Mode (supported only on ATWILC1000)
  - Supported cipher suites
    - TLS_RSA_WITH_AES_128_CBC_SHA
    - TLS_RSA_WITH_AES_128_CBC_SHA256
  - Supported authentication mode(s)
    - EAP-TTLS with MsChapv2.0
- Serial Peripheral Interface (SPI), Secure Digital Input/Output (SDIO), and Inter-Integrated Circuit (I^2C) support
- Ethernet Data Interface
- Low Footprint Host Driver with the Following Capabilities:
  - 8-, 16-, and 32-bit MCU support
  - Little- and big-endian support
  - Consumes about 8 KB of code memory and 1 KB of data memory on the host MCU
- Concurrency Support for the Following Modes:
  - Station-to-AP
  - Station-to-P2P Client

  – AP-to-P2P Client

Bluetooth (ATWILC3000)

- Bluetooth 5
- Host Control Interface (HCI) via High Speed Universal Asynchronous Receiver/Transmitter (UART)

Table of Contents

1. Overview

This section describes the host driver software architecture and Wi-Fi system architecture of the ATWILC*.

1.1 Host Driver Architecture

The ATWILC* host driver software is a C library, which provides the host MCU application with necessary APIs to perform WLAN and Ethernet operations.

The BT/BLE Stack communicates with the ATWILC* using standard HCI over UART. The ATWILC* host driver initializes the BT/BLE core and firmware, and the BT/BLE stack handles all BT/BLE data and controller paths.

The following figures show the architecture of the ATWILC* host driver software which runs on the host MCU. The components of the host driver are described in the following sub-sections.

Figure 1-1. Host Driver Software Architecture for Wi-Fi -Only Chipsets

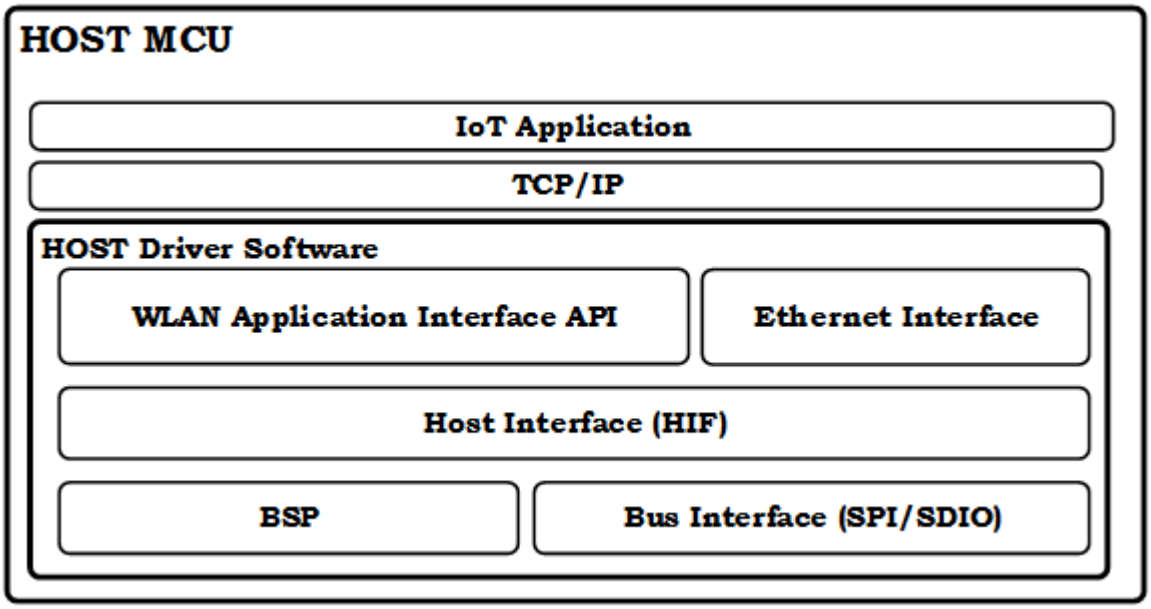
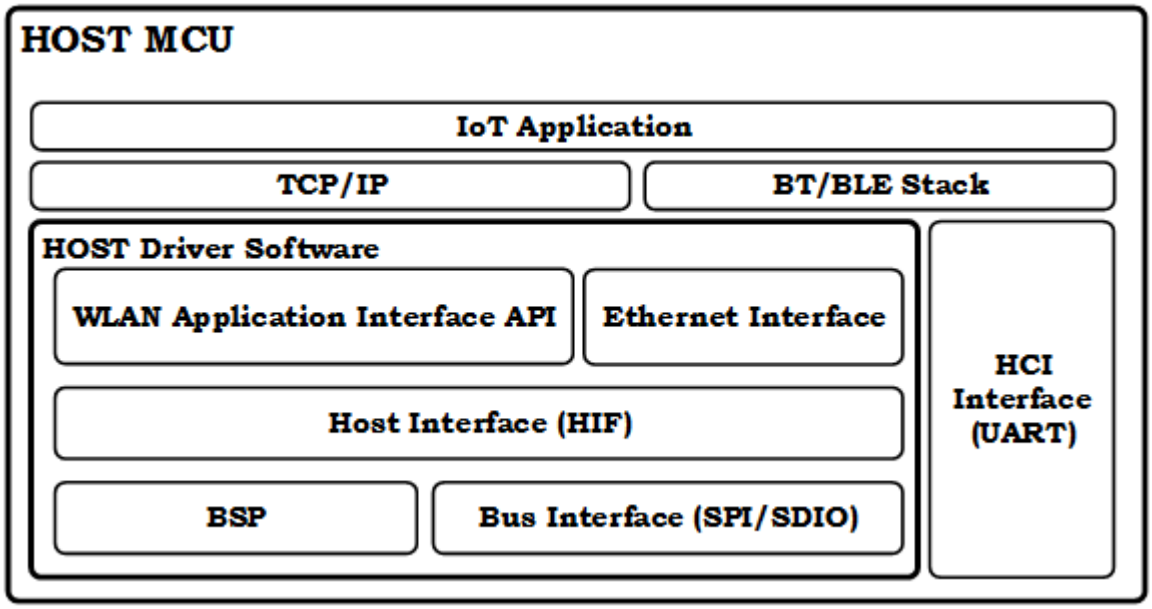


Figure 1-2. Host Driver Software Architecture for BT/BLE Capable Chipsets



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, and so on)
  - WPS activation/deactivation
- Wi-Fi AP enable/disable
- Wi-Fi power save control API

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

1.1.2 Host Interface

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

The interface is defined in the `m2m_hif.h` file. For detailed description of the HIF design, see the Host Interface Protocol chapter.

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 bus interface module abstracts the hardware associated with implementing the bus between the host and the ATWILC*. The serial bus interface abstracts SPI, SDIO, or I^2C buses. The basic bus access

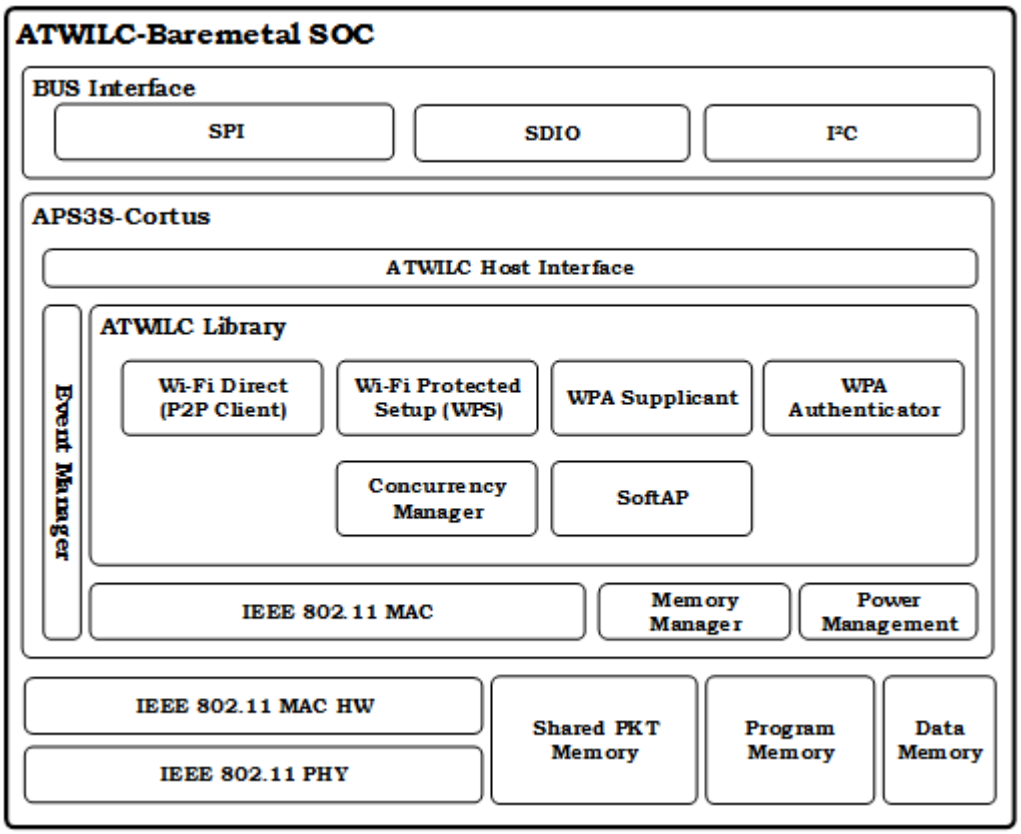
operations (read and write) are implemented in this module for the interface type and the specific hardware.

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

1.2 ATWILC* Wi-Fi System Architecture

The following figure shows the ATWILC* Wi-Fi system architecture. In addition to its built-in Wi-Fi IEEE-802.11 physical layer and RF front end, the ATWILC* ASIC contains an embedded APS3S-Cortus 32-bit CPU to run the ATWILC* 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 1-3. ATWILC* System Architecture



1.2.1 Bus Interface

The bus interface provides hardware logic for the supported bus types for the ATWILC* communications.

1.2.2 CPU

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

1.2.3 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 the 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.

1.2.4 Program Memory

160 KB Instruction RAM is provided for execution of the ATWILC3000 firmware code, and 128 KB for ATWILC1000.

1.2.5 Data Memory

The 128 KB data RAM is provided to execute the ATWILC* firmware data storage.

1.2.6 Shared Packet Memory

The 128 KB 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 software component.

1.2.7 IEEE 802.11 MAC Firmware

The system supports IEEE 802.11 b/g/n Wi-Fi MAC including a 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.

1.2.8 Memory Manager

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

1.2.9 Power Management

The power management module handles different power-saving modes supported by the ATWILC* and coordinating these modes with the Wi-Fi transceiver.

1.2.10 ATWILC Library

- **EAP-TTLS/MSCHAPV2.0** – This module implements the EAP-TTLS/MsChapv2.0 authentication protocol, used for establishing a Wi-Fi connection and AP with WPA-Enterprise security.
- **Wi-Fi Protected Setup** – For more details on WPS protocol implementation, see the Wi-Fi Protected Setup chapter.
- **Wi-Fi Direct** – For more details on Wi-Fi Direct protocol implementation, see the Wi-Fi Direct P2P Mode chapter.
- **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)
  - MsChapv2.0 (used as the EAP-TTLS inner authentication algorithm)
  - AES-128, and 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. ATWILC* Initialization and Simple Application

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

- BSP initialization
- ATWILC* host driver initialization
- Call ATWILC* driver entry point

Note: The initialization sequence must be completed to successfully operate the ATWILC* start-up procedure.

2.1 ATWILC* Device Selection

There are two ATWILC* devices available, and a corresponding Compilation flag must be defined to select and use the device:

- ATWILC1000B – CONF_WILC_USE_1000_REV_B
- ATWILC3000 – CONF_WILC_USE_3000_REV_A

2.2 BSP Initialization

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

- Resets the ATWILC* using the corresponding host MCU control GPIOs.
  Note: Refer to the *ATWILC1000 Datasheet* for more information about ATWILC* Power-Up/Down Sequence.
- Initializes the host MCU GPIO which connects to the ATWILC* interrupt line. It configures the GPIO as an interrupt source to the host MCU. During run time, the ATWILC* interrupts the host to notify the application of events and data that are pending with the ATWILC* firmware.
- Initializes the host MCU delay function used within `nm_bsp_sleep` implementation.

2.3 ATWILC* Host Driver Initialization

Initialize the ATWILC* host driver by calling the `m2m_wifi_init` API. The host driver initialization routine performs the following steps:

- Initializes the bus wrapper, I^2C, SPI, or SDIO, depending on the host driver software bus interface configuration compilation flag `CONF_WILC_USE_SPI`, or `CONF_WILC_USE_SDIO` respectively. The IRQ of SDIO can be used either with external GPIO, or muxed with SDIO DAT[1] terminal. To configure the bus for using the external GPIO IRQ, `CONF_WILC_USE_SDIO_EXT_IRQ` together with `CONF_WILC_USE_SDIO`.
- Configure the ATWILC* host driver HIF queue sizes `CONF_WILC_TXQ_BUF_SIZE` and `CONF_WILC_RXQ_BUF_SIZE`. The recommended HIF queue size is between 1600 bytes to 10kb. The bus performance depends on the size of MCU buffer allocation.
- Registers an application-defined Wi-Fi event handler.
- Initializes the driver and ensures that the current ATWILC* firmware matches the current driver version.

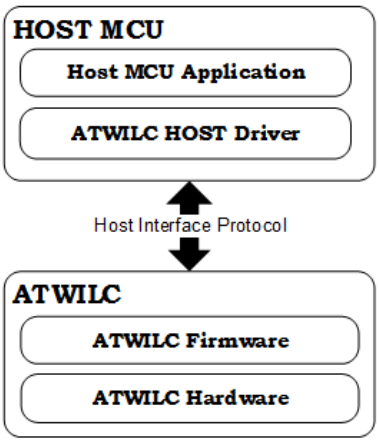
- Initializes the host interface and the Wi-Fi layer, and registers the BSP interrupt
- Downloads Bluetooth firmware, and starts the BT CPU, if applicable.

Note: A Wi-Fi event handler is required for the operation of any ATWILC* application.

2.4 ATWILC* Event Handling

The ATWILC* host driver API allows the host MCU application to interact with the ATWILC* firmware. The ATWILC* driver implements the Host Interface (HIF) Protocol as described in Host Interface Protocol chapter. The HIF protocol defines how to serialize and de-serialize the API requests and response callbacks over the bus interface (I^2C, SPI, or SDIO).

Figure 2-1. ATWILC* System Architecture



The ATWILC* host driver API provides services to the host MCU applications that are mainly divided in two major categories:

- Wi-Fi control services – Allow actions such as channel scanning, network identification, connection and disconnection.
- Ethernet interface services – Allows the application to transfer Ethernet frames when a Wi-Fi connection is established.

2.4.1 Asynchronous Events

Some of the ATWILC* host driver APIs are synchronous function calls, where the result is ready by the return of the function. However, most of the ATWILC* host driver API functions are asynchronous. This indicates 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 complete. When complete, a notification is provided as a HIF protocol message from the ATWILC* firmware to the host which is delivered to the application via a callback[1] function. Asynchronous operation is required when the requested service, such as Wi-Fi connection, takes significant time to complete. In general, the ATWILC* firmware uses asynchronous events to signal the host driver about a status change or pending data.

The HIF uses push architecture, where data and events are pushed from the ATWILC* firmware to the host MCU on a First-Come First-Served (FCFS) manner. For instance, suppose the ATWILC* receives two Ethernet packets, then HIF delivers the frame data in two HIF protocol messages in the order they are received. The HIF does not allow reading the packet two data before the packet one data.

[1] The callback is C function which contains an application-defined logic. The callback is registered using the ATWILC* host driver registration API to handle the result of the requested service.

2.4.2 Rx-Event Handling

The HIF notifies the host MCU when one or more events are pending in the ATWILC* firmware. The host MCU application is a big state machine which processes received data and events when the ATWILC* driver calls the event callback function(s). To receive event callbacks, the host MCU application is required to call the `m2m_wifi_handle_rx_events` API to let the host driver retrieve and process the pending events from the ATWILC* firmware. It is recommended to call this function if any of the following events occur:

- Host MCU application polls the API in main loop or a dedicated task.
- Host MCU receives event from the ATWILC* firmware.
  Note: All the application defined event callback functions that are registered with the ATWILC* driver run in the context `m2m_wifi_handle_rx_events` API.

The above HIF architecture allows the ATWILC* host driver to be flexible to run the following configurations:

- Host MCU with no operating system configuration – the MCU main loop handles deferred work from interrupt handler.
- Host MCU with operating system configuration – a separate task or thread is required to call `m2m_wifi_handle_rx_events` to handle deferred work from the interrupt handler.

Note: When the host MCU is polling `m2m_wifi_handle_rx_events`, the API checks for a pending unhandled interrupt from the ATWILC*. If no interrupt is pending, it returns immediately. If an interrupt is pending, `m2m_wifi_handle_rx_events` reads all the pending HIF messages 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.

2.4.3 Tx-Event Handling

The HIF notifies the host MCU when one or more events are pending in the Tx-Queue of the ATWILC* host driver. To send all host events to the ATWILC* firmware, the host MCU application is required to call the `m2m_wifi_handle_tx_events` API to let the host driver send the pending events from the ATWILC* host driver.

To enhance the bus performance, multiple outgoing packets are queued in the HIF buffer sent in group instead of single packet each time. It is recommended to call this function either:

- When host MCU application polls the API in main loop or a dedicated task;
- Or at least once when host MCU receives an notification from the ATWILC* host driver.
  Note: All the application-defined event callback functions registered with the ATWILC* driver run in the context `m2m_wifi_handle_tx_events` API.
- **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_tx_events` to handle deferred work from interrupt handler.
  Note: When the host MCU is polling `m2m_wifi_handle_tx_events`, the API checks for pending unhandled events from the ATWILC* host driver. If no event is pending, it returns immediately by putting task wait on next Tx-Event notification. If an event is pending, `m2m_wifi_handle_tx_events` sends all the pending the HIF messages in group to the ATWILC* firmware.

2.4.4 Code Example

The following code example is used for the initialization as described in the 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_rx_events(NULL) != M2M_SUCCESS) {
        }
        while(m2m_wifi_handle_tx_events(NULL) != M2M_SUCCESS) {
        }
    }
}
```

3. ATWILC* Configuration

The ATWILC* firmware offers 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 unset by the host MCU application use their default values assigned during the initialization of the ATWILC* firmware. A host MCU application configures the 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 ATWILC* firmware version.

Note: If the current driver version is less than the minimum driver version required by the ATWILC* firmware, the driver initialization fails.

The version parameters provided are as follows:

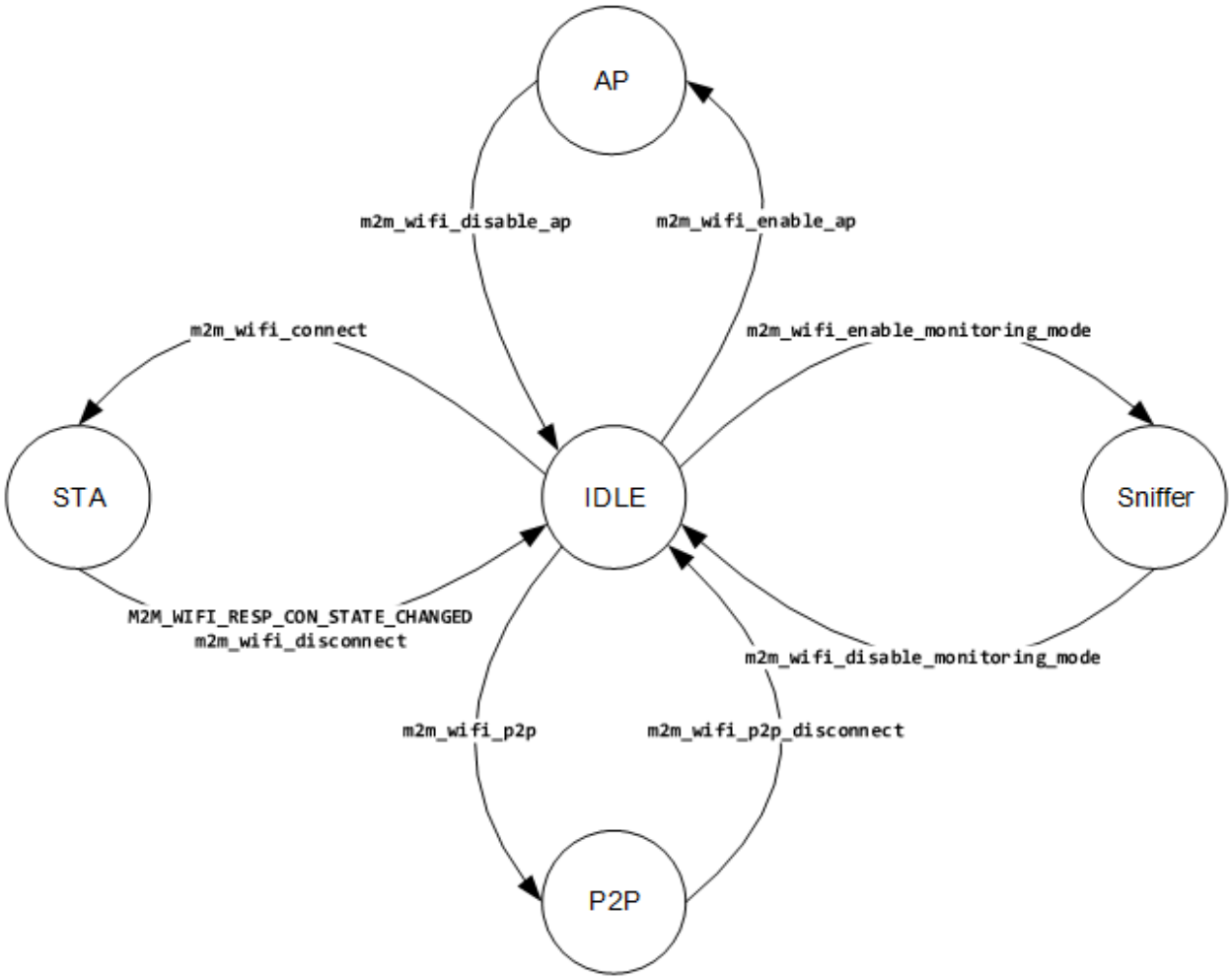
- `M2M_FIRMWARE_VERSION_MAJOR_NO` – firmware major release version number
- `M2M_FIRMWARE_VERSION_MINOR_NO` – firmware minor release version number
- `M2M_FIRMWARE_VERSION_PATCH_NO` – firmware patch release version number
- `M2M_DRIVER_VERSION_MAJOR_NO` – driver major release version number
- `M2M_DRIVER_VERSION_MINOR_NO` – driver minor release version number
- `M2M_DRIVER_VERSION_PATCH_NO` – driver patch release version number

3.2 ATWILC* Wi-Fi Modes of Operation

The ATWILC* firmware supports the following modes of operation:

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

Figure 3-1. ATWILC* Modes of Operation



3.2.1 Idle Mode

After the host MCU application calls the ATWILC* driver initialization `m2m_wifi_init` API, the ATWILC* remains in Idle mode waiting for any command to change the mode or to update the configuration parameters. In this mode, the ATWILC* enters the Power Save mode which disables the IEEE 802.11 radio and all unneeded peripherals and suspends the ATWILC* CPU. If the ATWILC* receives any configuration commands from the host MCU, the ATWILC* updates the configuration, sends back the response to the host MCU, and then returns to the Power Save mode.

3.2.2 Wi-Fi Station Mode

The ATWILC* enters the Station (STA) mode when the host MCU requests to connect to an AP using the `m2m_wifi_connect` API. The ATWILC* exits the 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. The ATWILC* firmware ignores mode change requests until the ATWILC* exits this mode.

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

For more information about this mode, refer to Wi-Fi Station Mode chapter.

3.2.3 Wi-Fi Direct P2P Mode

In Wi-Fi Direct mode, the ATWILC* can discover, negotiate and connect wirelessly to Wi-Fi Direct capable peer devices. To enter into the P2P mode, the host MCU application calls `m2m_wifi_p2p` API. To exit the P2P mode, the application calls `m2m_wifi_p2p_disconnect` API.

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

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

3.2.4 Wi-Fi Hotspot (AP) Mode

In AP mode, the ATWILC* allows Wi-Fi stations to connect to the ATWILC* device. To enter AP mode, the host MCU application calls `m2m_wifi_enable_ap` API. To exit the AP mode, the application calls `m2m_wifi_disable_ap` API.

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

For more information about this mode, refer to the Wi-Fi AP Mode chapter.

3.3 Network Parameters

3.3.1 Device Name

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

Note: If no device name is provided, the default device name is the `WILCP2P_DEVICE`.

3.3.2 Wi-Fi MAC Address

The ATWILC* firmware provides two methods to assign a MAC address to the ATWILC*:

- Assignment from host MCU – this method occurs when the host MCU application calls the `m2m_wifi_set_mac_address` API after initialization using the `m2m_wifi_init` API.
- Assignment from ATWILC OTP memory – the ATWILC* supports an internal MAC address assignment method through a built-in OTP memory. If the MAC address is programmed in the ATWILC* OTP memory, the ATWILC* working MAC address defaults to the OTP MAC address unless the host MCU application sets a different MAC address programmatically after initialization using the `m2m_wifi_set_mac_address` API.
  Note: The OTP MAC address is programmed in the ATWILC* OTP memory at manufacturing time.

For more details, refer to the description of the following APIs in Appendix A:

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

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

3.3.3 BT Address

The ATWILC* BT/BLE firmware provides two methods to assign the ATWILC* BT address:

- Assignment from the host MCU – The host MCU can use the HCI vendor-specific command to set the BT/BLE address. For more information on how to use the HCI vendor-specific commands, refer to the Vendor Specific HCI Commands section.
- Assignment from the ATWILC* OTP memory – Same as Wi-Fi, the BT address can be assigned from the OTP memory. In this case, the assigned BT address is the address in the OTP after incrementing the MSB by 1.

3.4 Wi-Fi Power-Saving Parameters

When a Wi-Fi station is idle, it disables the Wi-Fi radio and enters into the Power Save mode. The AP buffers data while stations are in the Power Save mode and transmits data later when the stations wake up. The AP transmits a beacon frame periodically to synchronize the network for every beacon period. A station which is in the Power Save mode 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 of the station and buffers inside the AP while the station sleeps. The beacon also provides information to the station when the AP is about to send broadcast/multicast data.

3.4.1 Wi-Fi Power-Saving Modes

The ATWILC* firmware supports multiple power-saving modes which provide flexibility to the host MCU application to change the system power consumption. The host MCU can configure the ATWILC* power-saving policy using the `m2m_wifi_set_sleep_mode` and `m2m_wifi_set_lsn_int` APIs. The ATWILC* supports the following Power Save modes:

- `M2M_NO_PS`
- `M2M_PS_DEEP_AUTOMATIC`

3.4.1.1 M2M_NO_PS

The Power Save mode is disabled and the chip is ON during this time.

3.4.1.2 M2M_PS_DEEP_AUTOMATIC

This mode implements the Wi-Fi standard power-saving method. However, when the ATWILC* enters Sleep state, the system clock is turned OFF.

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

While sleeping, the ATWILC* wakes up if one of the following events occur:

- Expiry of the hardware sleep timer. The ATWILC* wakes up to receive the upcoming beacon from AP.
- The ATWILC* wakes up when the host MCU application requests services from the ATWILC* by calling any host driver API function, for example, Wi-Fi or data operation.

Note: The wake up sequence is internally handled in the ATWILC* host driver `hif_chip_wake` API. For more information, see Host Interface Protocol chapter.

3.4.2 Configuring Listen Interval and DTIM Monitoring

The ATWILC* allows the host MCU application to change the system power consumption by configuring beacon monitoring parameters. The AP periodically sends beacons every beacon period (for example, 100 ms). The beacon contains a Traffic Indication Message (TIM) element which informs the station

about the presence of unicast data for the station buffer in the AP. The station negotiates with the AP with a listen interval, which is how many beacon periods the station can sleep before it wakes-up to receive data buffered in AP. The AP beacon also contains the DTIM which contains information for the station about the presence of broadcast/multicast data. The AP is ready to transmit after this beacon and normal channel access rules (CSMA/CA).

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

- **Configure DTIM monitoring** – that is to 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

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

3.5 Bluetooth Low Energy Power Saving

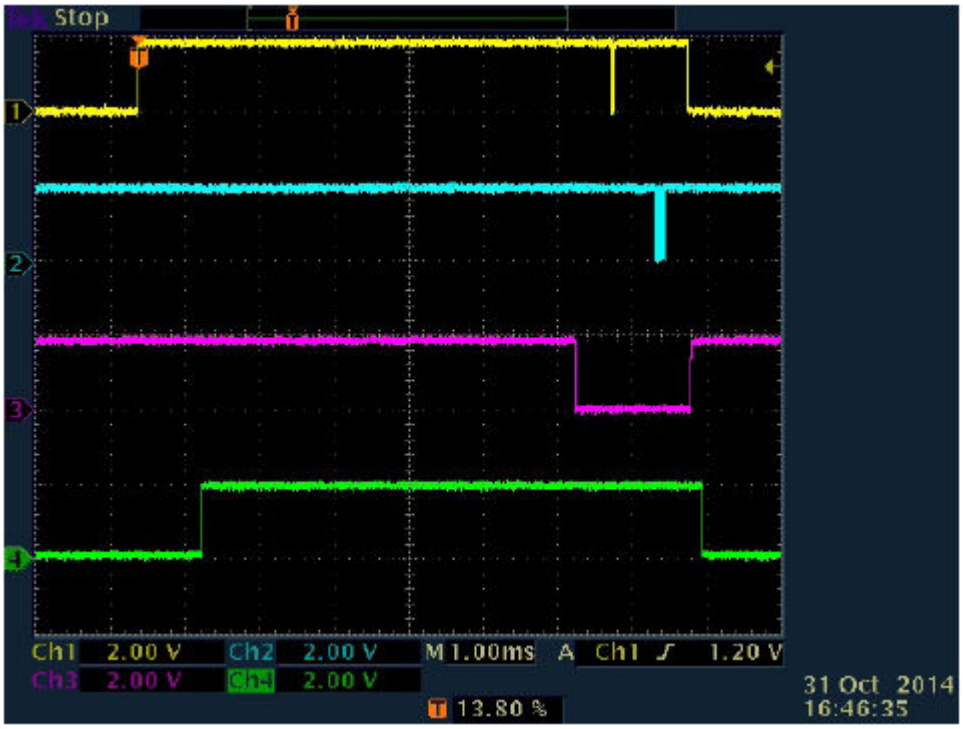
The Bluetooth Low Energy communication between the host and ATWILC3000 Bluetooth Low Energy controller uses HCI commands over UART. When the host application does not need to perform any Bluetooth Low Energy activity, the application sends the ATWILC3000 Bluetooth Low Energy controller into the Power Save mode. The Bluetooth Low Energy firmware running in the ATWILC3000 is equipped with a power save feature; it is the HOST application that can trigger/send the power save commands using UART.

To use the Bluetooth Low Energy power save feature, UART flow control must be enabled to hold the host from sending new commands to the ATWILC3000 Bluetooth Low Energy controller when it is in the Sleep mode. This can be done using the Update UART Parameters Command to enable flow control on the ATWILC3000, then update the host's UART configuration to enable flow control. Also, the host application must allow the ATWILC3000 Bluetooth Low Energy controller to enter Power Save mode, by setting the host's UART Tx line low, entering the Break mode. Before starting any HCI communication, the application must ensure that host UART is not in the Break mode and sends the HCI commands to the ATWILC3000.

When the ATWILC3000 is in the Power Save mode, it sets the UART RTS line high to restrain the host from sending HCI commands. Once the host UART Tx line is high, the ATWILC3000 switches the Power Save mode, but will not be fully active. After the ATWILC3000 is high and ready to receive more HCI commands, it sets the UART RTS line low, and the host can send more HCI commands.

This sequence is illustrated in the following figure:

Figure 3-2. Bluetooth Low Energy Power Saving Sequence



1. Yellow – UART Rx (ATWILC3000 perspective)
2. Blue – UART Tx
3. Purple – UART RTS
4. Green – ATWILC3000 Ready

Note: The user must disable the AP mode of operation to observe the power save functionality because the AP mode prevents the firmware from going into the Power Save mode.

3.5.1 Code Example

The following code example creates a task that reads the user input to enable or disable the Power Save mode by controlling the host's USART0 RX line break condition.

Task Creation:

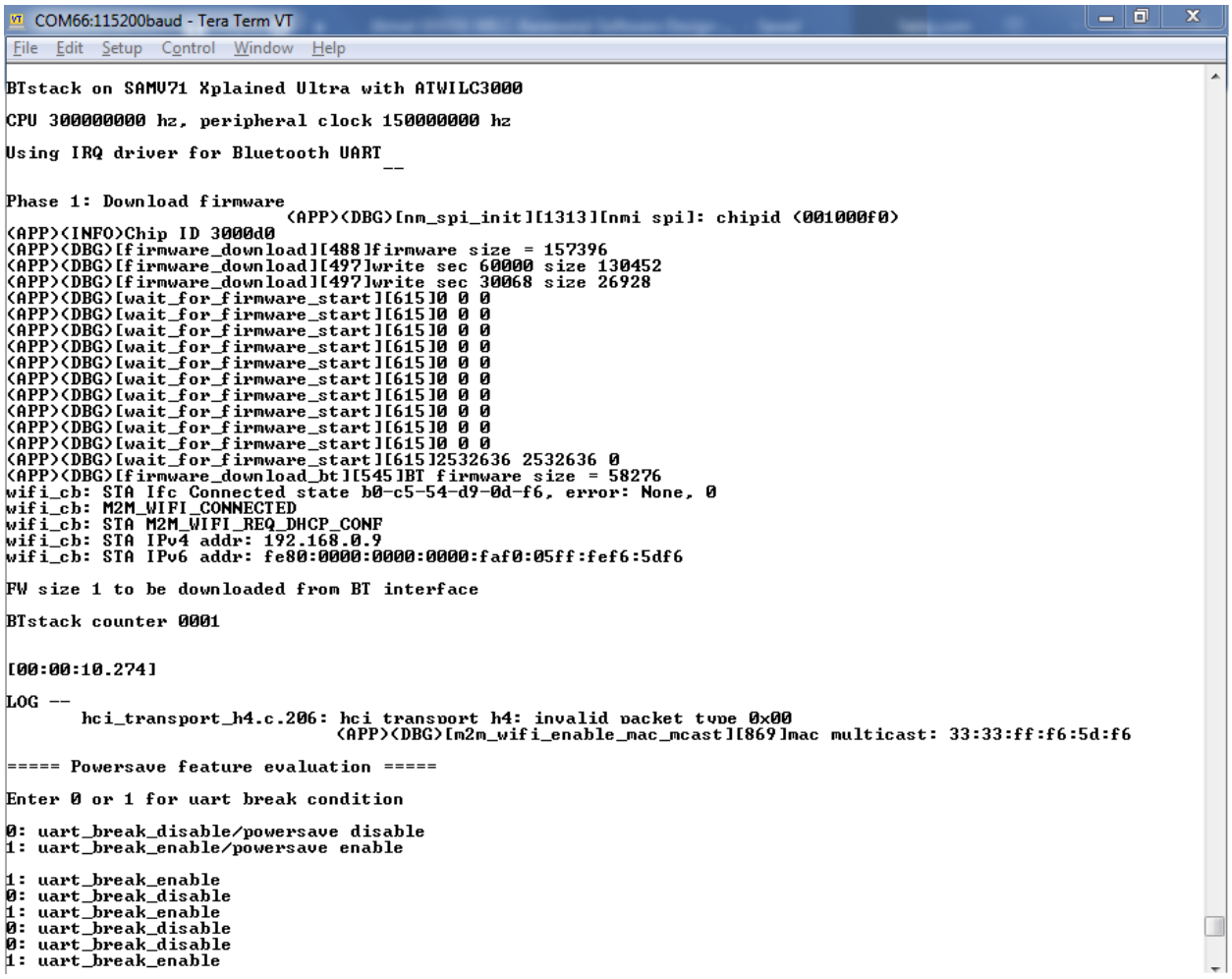
```
/* Create task to take user input for UART break */
xTaskCreate(uart_break, "uart_break", TASK_UART_BREAK_STACK_SIZE, NULL,
TASK_UART_BREAK_STACK_PRIORITY, 0);
```

Task Function:

```
/**
 *  \brief below piece of code is to read user input and put UART break condition accordingly.
 *
 *  \return void.
 */
#include "os/include/m2m_wifi_ex.h"
#define TASK_UART_BREAK_STACK_SIZE                ((1024*2)/sizeof(portSTACK_TYPE))
#define TASK_UART_BREAK_STACK_PRIORITY            (tskIDLE_PRIORITY + 1)

static void uart_break(void *arg)
{
      vTaskDelay(6000); // approximate wait time for wifi and ble f/w download
      printf("\r\n===== Powersave feature evaluation =====\r\n");
      printf("\r\nEnter 0 or 1 for uart break condition\r\n");
      printf("\r\n0: uart_break/powersave disable\r\n1: uart_break/powersave enable\r\n\n");
```

```
        uint8_t data;
        while(1) {
          vTaskDelay(500);
          if(uart_is_rx_ready((Uart *)USART1))
          {
              uart_read((Uart *)USART1, &data); // sometimes it's called CONF_UART
              if (data == '1') {
                  printf(""%c: uart_break_enable\r\n", data);
                  usart_start_tx_break(USART0); // also known as BOARD_USART
                  os_m2m_wifi_set_sleep_mode(M2M_PS_DEEP_AUTOMATIC, true);

              }
              else if (data == '0') {
                  printf("%c: uart_break_disable\r\n", data);
                  usart_stop_tx_break(USART0);
                  os_m2m_wifi_set_sleep_mode(M2M_NO_PS, true);
              }
          }
      }
  }
```

Figure 3-3. Sample User Console Input and Output



4. Wi-Fi Station Mode

This chapter provides detailed information about the ATWILC* Wi-Fi station (STA) mode. The Wi-Fi Station mode involves scan operation, association to an AP using parameters (SSID and credentials) provided by the host MCU or using the AP parameters stored in the ATWILC* nonvolatile storage (default connection). This 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 ATWILC* initial region configuration is equal to 11 channels (North America), 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, the ATWILC* sends probe request Wi-Fi frames and waits for 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 creates 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. The ATWILC* firmware default scan wait time is optimized to provide the tradeoff between power consumption and scan accuracy. The ATWILC* 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.

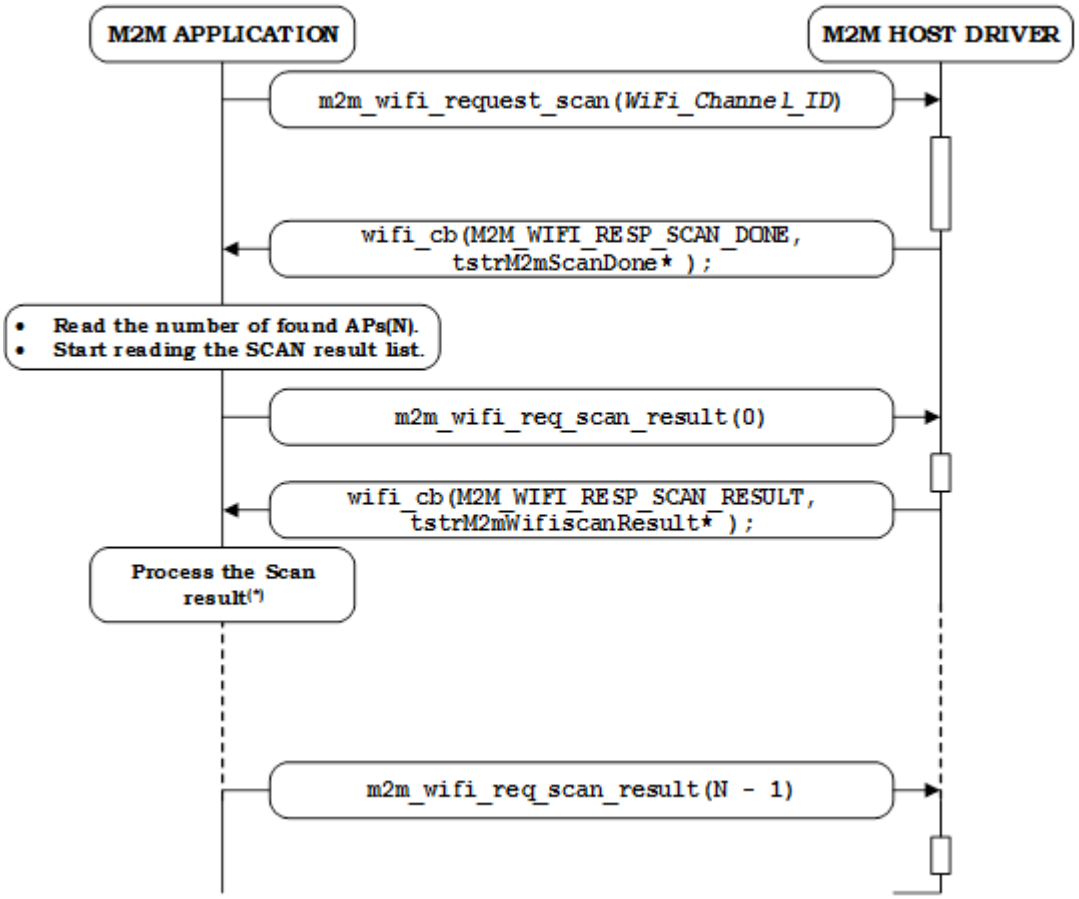
4.2 Wi-Fi Scan

A Wi-Fi scan operation can be initiated by calling the `m2m_wifi_request_scan` API. The scan can be performed on all 2.4 GHz Wi-Fi channels or on a specifically-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`. For more information on how to customize the scan parameters, see Appendix A.

The scan operation is illustrated in the following figure.

Figure 4-1. Wi-Fi Scan Operation



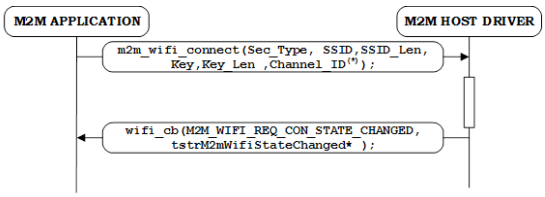
4.3 On Demand Wi-Fi Connection

The host MCU application establishes 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 calls the `m2m_wifi_connect` API.

Note: Using `m2m_wifi_connect` implies that the host MCU application is aware of the connection parameters. For instance, connection parameters can be stored on nonvolatile storage attached to the host MCU.

The Wi-Fi on demand connection operation is described in the following figure.

Figure 4-2. On Demand Wi-Fi Connector



4.4 Wi-Fi Security

The following types of security are supported in the ATWILC* Wi-Fi STA mode.

- `M2M_WIFI_SEC_OPEN`
- `M2M_WIFI_SEC_WEP`
- `M2M_WIFI_SEC_WPA_PSK` (WPA/WPA2-Personal Security Mode that is Passphrase)
- `M2M_WIFI_SEC_802_1X` (WPA-Enterprise security)

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

4.5 Generate `CertOut.h` Client Certificate

The following are the steps to convert the enterprise certificate image file to `CertOut.h` header file to build with RTOS host applications.

1. Run the script `root_certificate_generator -n` *`num_cer cer1.cer cer2.cer …..cer_num_cer.cer`*
   (Where, `cer1.cer, cer2.cer ...cerN.cer` are the certificate image files in “DER encoded binary X.509 (CER)” format).
2. Copy the output header file `CertOut.h` to `src\ASF\common\components\wifi\wilc\driver\include`.
3. Include the `CertOut.h` file where the driver API `m2m_wifi_download_cert()` is to be invoked.
   Note: The `root_certificate_generator` tool is located at ATWILC*’s page on Microchip’s website.
   - ATWILC1000
   - ATWILC3000

4.6 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
    tuniM2MWifiAuth sta_auth_param;
    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 <%d>\n",ret);
        while(1);
    }

    /* Connect to a WPA-Enterprise AP
*/
            /* Request firmware to download Root Certificate */
            m2m_wifi_download_cert(cert_image,sizeof(cert_image));
```

```
            strncpy((char*)sta_auth_param.strCred1x.au8UserName,
MAIN_WLAN_802_1X_USR_NAME,M2M_1X_USR_NAME_MAX-1);
            sta_auth_param.strCred1x.au8UserName[M2M_1X_USR_NAME_MAX-1] = '\0';

            strncpy((char*)sta_auth_param.strCred1x.au8Passwd, MAIN_WLAN_802_1X_PWD,
M2M_1X_PWD_MAX-1);
            sta_auth_param.strCred1x.au8Passwd[M2M_1X_PWD_MAX-1]='\0';

            m2m_wifi_connect((char *)MAIN_WLAN_SSID, sizeof(MAIN_WLAN_SSID),
            M2M_WIFI_SEC_802_1X, &sta_auth_param, M2M_WIFI_CH_ALL);
    while(1)
    {

        /*======================================================================*/
        /* Handle the app state machine plus the WILC event handler             */
        /*======================================================================*/
        while(m2m_wifi_handle_rx_events(NULL) != M2M_SUCCESS)
        {
        }
        while(m2m_wifi_handle_tx_events(NULL) != M2M_SUCCESS)
        {
        }

    }
}
```

5. Wi-Fi AP Mode

This chapter provides an overview of the ATWILC* Access Point (AP) mode and describes how to set up this mode and configure its parameters.

5.1 Setting ATWILC* AP Mode

Set the ATWILC* AP mode configuration parameters using the `tstrM2MAPConfig` structure.

There are two functions to enable/disable the ATWILC* AP mode:

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

For more information about structure and APIs, see Appendix A.

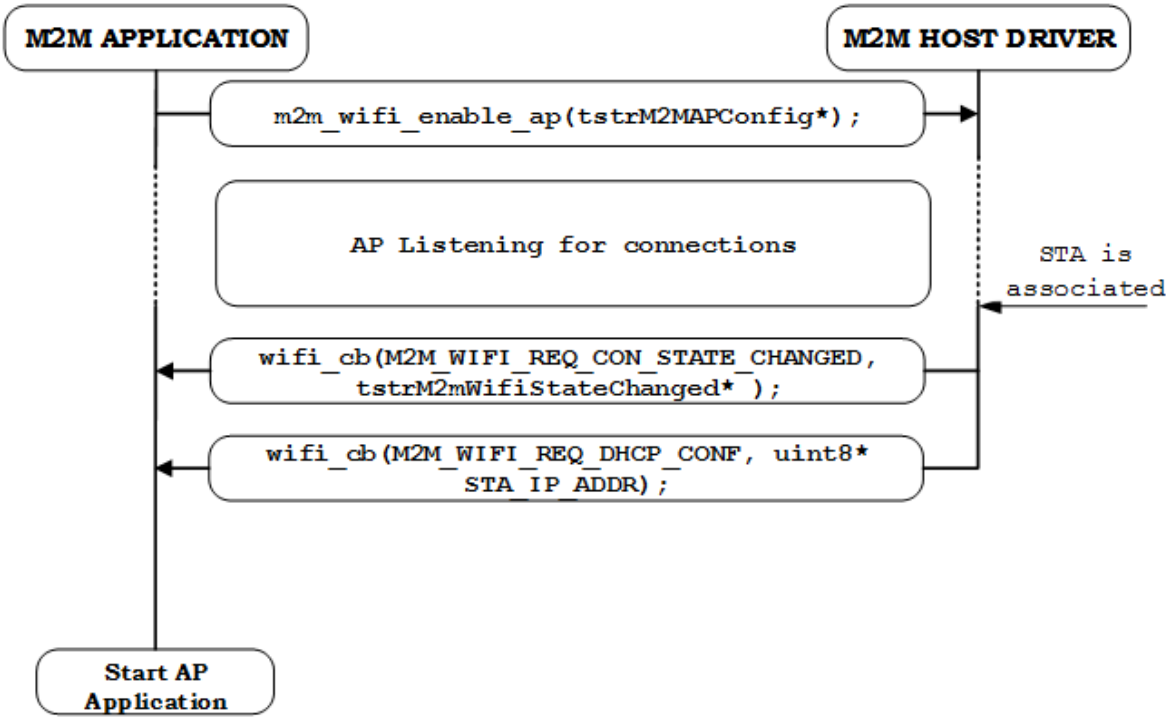
5.2 Capabilities

- The AP supports up to eight associated stations.
- Supports all modes of security (Open, WEP, and WPA/WPA2)
- Works concurrently with a station P2P interface.
  Note: In case of using WEP security on both STA and AP interfaces, use the same password for both interfaces. In addition to that, use the same WEP authentication type (OPEN, SHARED, or ANY) for both STA and AP interfaces.

5.3 Sequence Diagram

Once the AP mode is established, data interface does not exist until 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 the following figure.

Figure 5-1. ATWILC* AP Mode Establishment



5.4 AP Mode Code Example

The following example shows how to configure the ATWILC* AP mode with WILC_SSID as the broadcasted SSID on channel one with open security, and with an IP address of 192.168.1.1.

```
#include "m2m_wifi.h"
#include "m2m_types.h"
void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    switch(u8WiFiEvent)
    {
        case M2M_WIFI_RESP_CON_STATE_CHANGED : {
        }
        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 = M2M_WIFI_CH_11;    // Set Channel

        apConfig.u8SecType = M2M_WIFI_SEC_WEP;    // Set Security to WEP
        apConfig.uniAuth.strWepInfo.u8KeyIndx = 0;    // Set WEP Key Index
        apConfig.uniAuth.strWepInfo.u8KeySz = WEP_40_KEY_STRING_SIZE;    // Set WEP Key Size
        strcpy(apConfig.uniAuth.strWepInfo.au8WepKey, "1234567890");    // Set WEP Key
        // Start AP mode
```

```
        m2m_wifi_enable_ap(&apConfig);
        while(1)
        {
            m2m_wifi_handle_tx_events();
            while (!m2m_wifi_handle_rx_events());
        }
    }
}
```

6. Wi-Fi Direct (P2P) Mode

The Wi-Fi Direct, or Peer to Peer (P2P) mode allows two wireless devices to discover each other, decides on which device acts as a group owner, forms a group including WPS key generation and makes a connection. The ATWILC* supports a subset of this functionality that allows the ATWILC* firmware to connect to other P2P capable devices that are prepared to become the group owner.

6.1 ATWILC* Capabilities

- Supports P2P Client mode
- P2P device discovery
- P2P Listen state

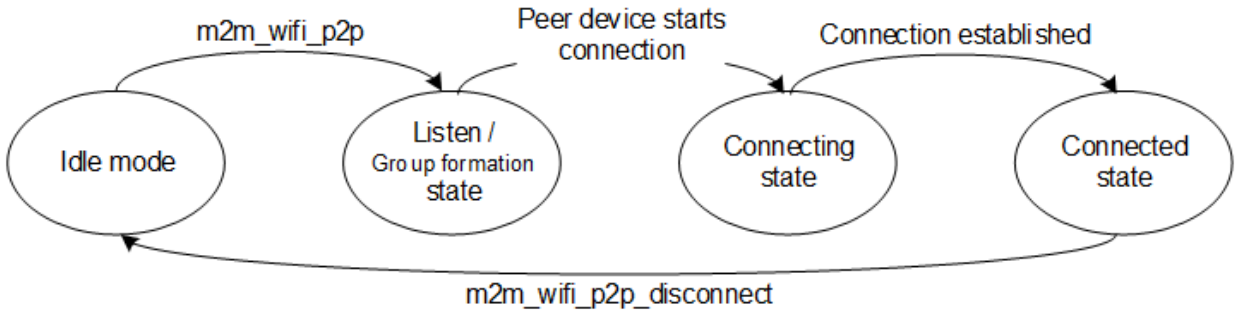
6.2 ATWILC* Limitations

- GO mode is not supported (P2P negotiation with GO intent set to 1)
- GO-NOA (Notice-Of-Absence) is not supported
- Power Save mode is disabled during the P2P mode
- The ATWILC* cannot initiate the P2P connection; instead, the other device must be an initiator

6.3 ATWILC* P2P States

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

Figure 6-1. P2P Mode State Diagram



6.4 ATWILC* P2P Listen State

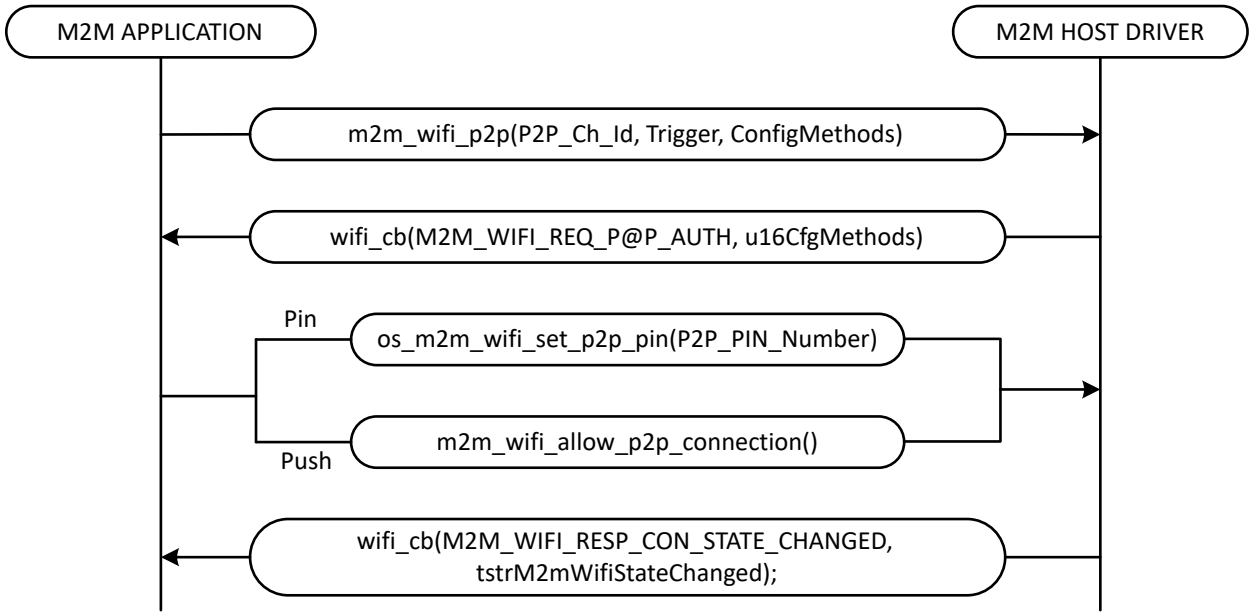
The ATWILC* device becomes discoverable to other P2P devices on a predefined listen channel, ready to accept any connection initiations. To enter the Listen state, the user must call the `m2m_wifi_p2p` function to set the ATWILC* firmware in the Listening state at a certain listen channel defined through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

6.5 ATWILC* P2P Connection State

The peer P2P device initiates Group Owner (GO) negotiation and the ATWILC* device always declines to become the GO. Assuming the peer device takes the GO role, the ATWILC* performs a WPS exchange

to establish a mutual shared key. The following sequence diagram shows the connection flow described above for the ATWILC* P2P device.

Figure 6-2. P2P Connection Flow



6.6 ATWILC* P2P Disconnection State

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

6.7 P2P Mode Code Example

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

#define MAIN_WLAN_DEVICE_NAME    "WILC_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;
        }
        case M2M_WIFI_REQ_P2P_AUTH:
        {
            tstrM2MP2pDevInfo *pstrP2PDevInfo = (tstrM2MP2pDevInfo *)pvMsg;
            if (pstrP2PDevInfo->u16CfgMethods & CONF_METHOD_KEYPAD) {
                osprintf("\r\nPlease enter P2P pin\r\n(Usage: P2P_PIN <pin-number displayed
on phone>\r\n");
            }
            else if (pstrP2PDevInfo->u16CfgMethods & CONF_MEHTOD_DISPLAY){
```

```
                osprintf("\r\nPlease enter P2P pin on phone <12345678>\r\n");
                    /* Set device name. */
                    os_m2m_wifi_set_p2p_pin((uint8_t *)"12345678", 8);
            }
            else {
                osprintf("\r\nAllow pushbutton method\r\n");
                m2m_wifi_allow_p2p_connection();
            }
            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, P2P_PBC, CONF_METHOD_PHYSICAL_PBC|
CONF_METHOD_LABEL);
    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_rx_events(NULL) != M2M_SUCCESS) {
        }
    }
    m2m_wifi_handle_tx_events();
    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 a normal static PSK configuration.

7.1 WPS Configuration Methods

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

- **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 ATWILC* by calling `m2m_wifi_wps` with input parameter `WPS_PBC_TRIGGER`.
- **PIN method** – The AP is available for WPS initiation, however, requires proof that the user knows an 8-digit PIN, usually printed on the body of the AP. Since ATWILC* 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 ATWILC* device. Some APs allow the PIN to change through configuration. WPS is initiated on the ATWILC1000 by calling `m2m_wifi_wps` with input parameter `WPS_PIN_TRIGGER`. Due to the difficulty of this approach, it is not recommend for most of the applications.

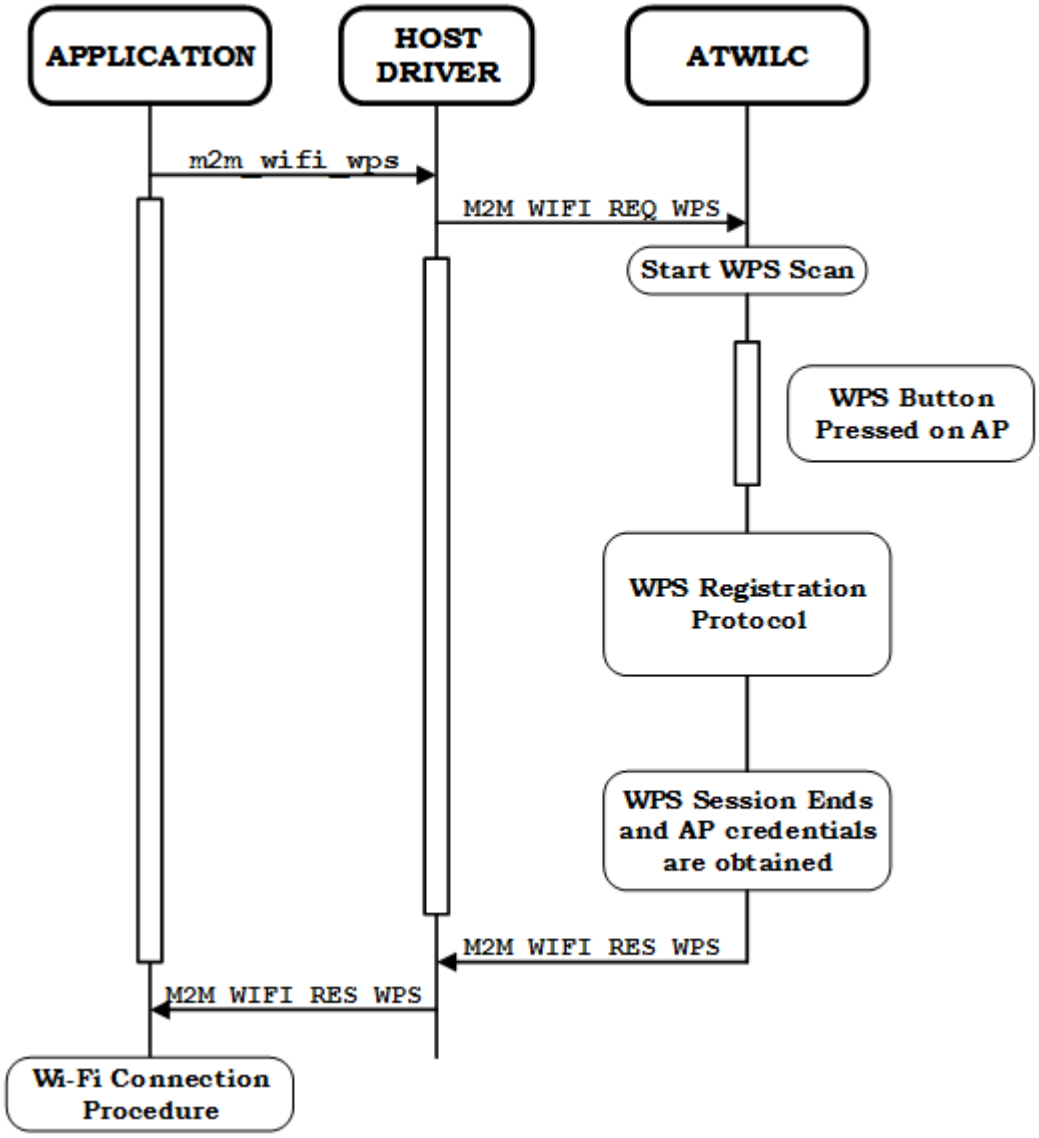
The flow of messages and actions for WPS operation is shown in WPS Operation for Push Button Trigger figure.

7.2 WPS Limitations

- The WPS is used to transfer the WPA/WPA2 key only; other security types are not supported.
- The WPS standard rejects the session (WPS response fail) if the WPS button is pressed on more than one AP in the same proximity, and the application must try after couple of minutes
- If WPS button is not pressed on the AP, the WPS scan expires two minutes from the initial WPS trigger.
- The WPS is responsible for delivering the connection parameters to the application. The connection procedure and the connection parameters' validity is the application's responsibility.
- WPS is supported on the STA interface only.

7.3 WPS Control Flow

Figure 7-1. WPS Operation for Push Button Trigger



7.4 WPS Code Example

```
void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    if(u8WiFiEvent == M2M_WIFI_RES_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_rx_events();
            m2m_wifi_handle_tx_events();
        }
    }
}
```

8. Concurrency

The firmware of the ATWILC* device supports the following modes of concurrent operations:

- Station-AP Concurrency
- Station-P2P Client Concurrency
- AP-P2P Client Concurrency

8.1 Concurrency Limitations

Single channel concurrency, that is, the two logical interfaces must operate on the same channel.

8.2 Controlling Second Interface

When using P2P concurrency, the application must select the required Concurrency mode, that is, STA-P2P or AP-P2P. This is controlled using `m2m_wifi_set_p2p_control_ifc()` API.

8.3 Station-AP Concurrency

In this mode of concurrency, the driver can connect to one AP using one interface and start an AP on the second interface regardless of first interface. However, it has the following facts and limitations:

1. If the AP is started first then the station interface connects to an AP on a different channel, the AP sends a de-authentication frame to all the associated stations and moves immediately to the same channel of the station interface, so that the previously associated station can connect on the new channel.
2. If the station interface is connected to an AP then the AP mode starts, the AP starts on the same channel of the station mode, regardless of the channel number passed in the AP start request. The following is the demo application code for Station - AP concurrency.

```
#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.uniAuth.strWepInfo.au8WepKey,"1234567890");
                strM2MAPConfig.uniAuth.strWepInfo.u8KeySz = WEP_40_KEY_STRING_SIZE;
                strM2MAPConfig.uniAuth.strWepInfo.u8KeyIndx = 0;
                strcpy(strM2MAPConfig.au8SSID,"WILC_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;
    tuniM2MWifiAuth sta_auth_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
    strcpy(sta_auth_param.au8PSK, DEMO_WLAN_PSK);
    ret = m2m_wifi_connect((char *)DEMO_WLAN_SSID, sizeof(DEMO_WLAN_SSID),
    DEMO_WLAN_AUTH, &sta_auth_param, 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_rx_events(NULL) != M2M_SUCCESS);
        m2m_wifi_handle_rx_events());
    }
    return 0;
}
```

8.4 Station-P2P Client Concurrency

In this mode of concurrency, the driver can connect to an AP using one interface and start P2P connection on the second interface regardless of first interface. However, it has the following facts and limitations:

- If the station interface is connected to an AP and then to the P2P connection, the ATWILC* firmware enforces the GO channel number of the station interface during the group negotiation frames.
- If the P2P connection happens first, the station interface must connect on the same channel of the P2P group, otherwise the P2P connection is dropped. The following is the demo application code for Station - P2P client concurrency.

```
#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_p2p_control_ifc(P2P_STA_CONCURRENCY_INTERFACE);
                        m2m_wifi_p2p(M2M_WIFI_CH_11, P2P_PBC, CONF_METHOD_PHYSICAL_PBC|
CONF_METHOD_LABEL);
                                interfaceNo = 2;
            }
            break;
        }
        case M2M_WIFI_REQ_P2P_AUTH:
        {
            tstrM2MP2pDevInfo *pstrP2PDevInfo = (tstrM2MP2pDevInfo *)pvMsg;
            if (pstrP2PDevInfo->u16CfgMethods & CONF_METHOD_KEYPAD) {
                osprintf("\r\nPlease enter P2P pin\r\n(Usage: P2P_PIN <pin-number displayed
on phone>\r\n");
            }
            else if (pstrP2PDevInfo->u16CfgMethods & CONF_MEHTOD_DISPLAY){
                osprintf("\r\nPlease enter P2P pin on phone <12345678>\r\n");
                    /* Set device name. */
                    os_m2m_wifi_set_p2p_pin((uint8_t *)"12345678", 8);
            }
            else {
                osprintf("\r\nAllow pushbutton method\r\n");
                m2m_wifi_allow_p2p_connection();
            }
            break;
        }

        default:
        {
            break;
        }
    }
}


int main(void)
{
    tstrWifiInitParam param;
    tuniM2MWifiAuth sta_auth_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
    strcpy(sta_auth_param.au8PSK, DEMO_WLAN_PSK);
    ret = m2m_wifi_connect((char *)DEMO_WLAN_SSID, sizeof(DEMO_WLAN_SSID),
    DEMO_WLAN_AUTH, &sta_auth_param, M2M_WIFI_CH_ALL);
    if (M2M_SUCCESS != ret) {
        printf("main: m2m_wifi_connect call error!\r\n");
        while (1) {
        }
    }
```

```
    while (1) {
        /* Handle pending events from network controller. */
        while (m2m_wifi_handle_rx_events(NULL) != M2M_SUCCESS);
        m2m_wifi_handle_tx_events();
    }
    return 0;
}
```

9. Data Send/Receive

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

9.1 Send Ethernet Frame

The `m2m_wifi_send_ethernet_pkt` API is used to transmit the Ethernet frame over the air.

Note: If the Wi-Fi is not connected to an AP, the frame is dropped by the firmware and does not transmit over the air.

The `m2m_wifi_send_ethernet_pkt` function is a synchronous function when it returns with successful code. This indicates that the frame is transferred from the host driver to the firmware, however, the frame is not transmitted over the air. Also, there is no way to make sure that the frame is delivered to its final target successfully, or if lost over the air. This must be handled by the upper layer protocol, for example, the TCP layer.

If the function returns `M2M_ERR_MEM_ALLOC` error code, this means the chip is temporarily out of buffers and the frame is not transferred to the chip memory. The application waits and will retry sending until the function returns a success code.

Note: Frame allocation and freeing is the application responsibility once the `m2m_wifi_send_ethernet_pkt` function returns, the application can free the frame or reuse the buffer.

9.2 Receive Ethernet Frame

During initialization, an Ethernet callback function must be registered and a receive buffer must be allocated to use as a receive buffer with the HIF. The registered callback function must add handling to the `M2M_WIFI_RESP_ETHERNET_RX_PACKET` notification to receive Ethernet frames; see the following 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;
        tstrM2MDataBufCtrl *PstrM2mIpCtrlBuf =( tstrM2MDataBufCtrl *)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;
    ret = m2m_wifi_init(&param);
    return ret;
}
```

After the return of the callback function, the HIF reuses the registered buffer and overwrites the data inside so the application must either move the frame from the buffer or update the buffer details using the `m2m_wifi_set_receive_buffer` API before the return of the callback function.

The receive buffer contains a header at its start that is used internally for communication between the host and the firmware. This header contains a size of `M2M_ETHERNET_HDR_OFFSET + M2M_ETH_PAD_SIZE`. The pad size is used to align the Ethernet payload on word boundary.

9.3 Concurrency Send

If the concurrency is used, the application can send frames on the second interface using the (`m2m_wifi_send_ethernet_pkt_ifc1`) API. This API has the same characteristics as of `m2m_wifi_send_ethernet_pkt()` with an exception, as it sends the frame on interface 2.

9.4 Concurrency Receive

If the concurrency is used, the application can distinguish between the frames received on each interface parameter `u8IfcId` parameter, included in the `tstrM2MDataBufCtrl` structure.

9.5 Bluetooth Packets

All Bluetooth packets are carried over a standard HCI interface with no interaction from the host driver. The BT/BLE stack handles the UART interface directly after initializing the host driver.

10. Host Interface Protocol

Communication between the user application and the ATWILC* device is facilitated by the driver software. This driver implements the Host Interface Protocol (HIF) and exposes an API to the application with various services.

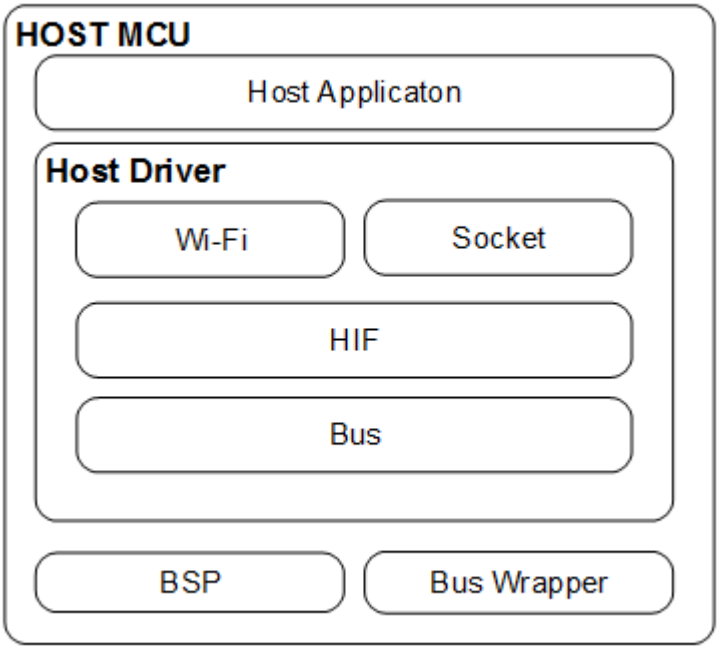
The services are broadly in two categories:

- Wi-Fi device control
  - The Wi-Fi device control services allow actions such as channel scanning, network identification, connection and disconnection.
- Ethernet data
  - The data services allow data transfer once a connection is 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. Once the action is complete, notification is provided in a subsequent message from the ATWILC* device to the host which is delivered to the application via a callback function. In general, the ATWILC* firmware uses asynchronous events to signal the host driver of certain status changes. Asynchronous operation is essential where functions (such as Wi-Fi connection) take significant time.

After the application sends a request, the host driver (Wi-Fi) formats the request and sends it to the HIF layer which then interrupts the ATWILC* device announcing that a new request is posted. Upon receipt, the ATWILC* firmware parses the request and starts the required operation.
Note: Dealing with HIF messages in the host MCU application is an advanced topic. For most of the applications, it is recommended to use Wi-Fi. This layer hides the complexity of the HIF APIs.

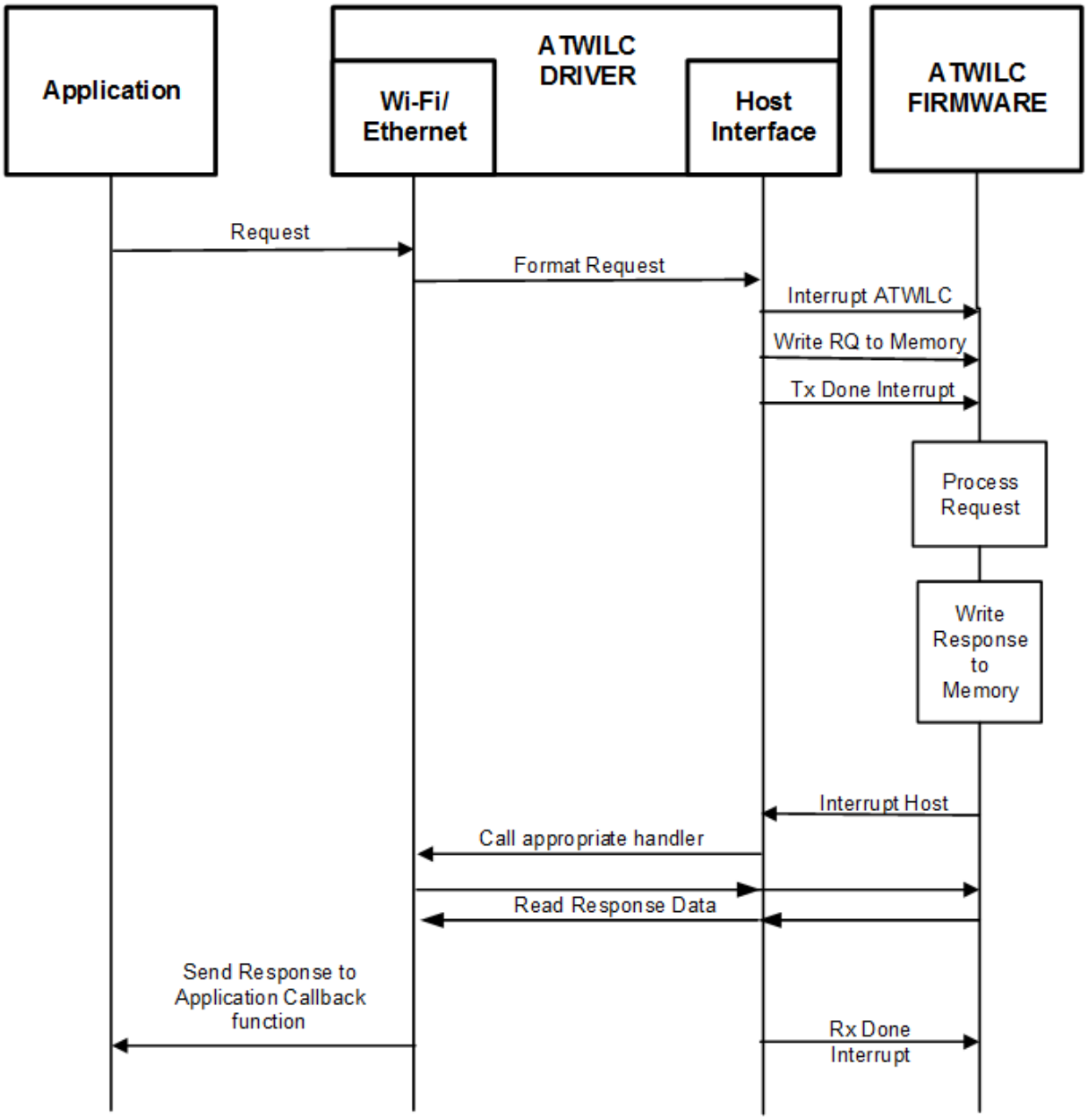
Figure 10-1. ATWILC* Driver Layers



The host interface layer is responsible for handling communication between the host MCU and the ATWILC* device. This includes interrupt handling, DMA control and management of communication logic between firmware driver at host and the ATWILC* firmware.

The request/response sequence between the host and the ATWILC* chip is shown in the following figure.

Figure 10-2. Request/Response Sequence Diagram



10.1 Chip Initialization Sequence

The chip initialization sequence depends on the ATWILC* chipset as described in the following sections.

10.1.1 ATWILC1000

The following table shows the sequence and the registers required to initialize the ATWILC1000 hardware.

Table 10-1. Registers to Initialize ATWILC1000

Step	Description
Step (1) Read the chip ID to ensure the bus and the chip functions	Read the chip ID from the `0x1000` register. This must return `0x1002xx` value
Step (2) Download the firmware into the chip memory	For more details, see the ATWILC Firmware Download chapter
Step (3) Disable the boot ROM	In the `0xC0000` register write the value as `0x71`
Step (4) Reset the state register	In the `NMI_STATE_REG` register write the value as `0`
Step (5) Set MUX to enable CPU reset from the GLOBAL RESET register	In the `0x1118` register write the value as `1`
Step (6) Set `NMI_VMM_CORE_CFG` to the SPI bus	In the `NMI_VMM_CORE_CFG` register write the value as `1`
Step (7) Reset the Wi-Fi CPU	In the `NMI_GLB_RESET_0` register change the value of bit `[10]` from `0` to `1`
Step (8) Poll on state register to make sure firmware is started successfully	Read the `NMI_STATE_REG` register and compare the value with `M2M_FINISH_INIT_STATE`
Step (9) Set MUX to enable IRQN pin output	Set the bit 8 in `NMI_PIN_MUX_0` register
Step (10) Enable IRQ on IRQN pin	Set the bit 16 in `NMI_INTR_ENABLE` register

10.1.2 ATWILC3000

The following table shows the sequence and registers needed to initialize the ATWILC3000 hardware.

Table 10-2. Registers to Initialize ATWILC3000

Step	Description
Step (1) Read chip ID to make sure the bus and the chip are working fine	Read the `0x3b0000` register. This must return `0x3000xx` value.
Step (2) Set MUX to enable BT CPU reset from the `GLOBAL RESET` register	In the Write in register 0x1118 as value 1
Step (3) Disable Wi-Fi CPU	In the `NMI_GLB_RESET_0` register reset the bit 10
Step (4) Disable the Wi-Fi boot ROM	In the `0xC0000` register write the value as `0x71`
Step (5) Reset the state register	In the `NMI_STATE_REG` register write the value as `0`
Step (6) Download the Wi-Fi firmware into the chip memory	For more details, see the ATWILC Firmware Download chapter
Step (7) Set `NMI_VMM_CORE_CFG` to SPI bus	In the `NMI_VMM_CORE_CFG` register write the value as `1`

.....continued

Step	Description
Step (8) Rest the chip Wi-Fi CPU	In the `NMI_GLB_RESET_0` register change the bit `[10]` from `0` to `1`
Step (9) Poll on state register to make sure firmware is started successfully	Read the `NMI_STATE_REG` register and compare the value with `M2M_FINISH_INIT_STATE`
Step (10) Set MUX to enable BT CPU reset from the GLOBAL RESET register	In the `0x3b0090` register write the value `1`
Step (11) Disable BT CPU	In the `0x3b0400` register reset the bit 2
Step (12) Disable the BT boot ROM	In the `0x4f0000` register write the value as `0x71`
Step (13) Download the BT Firmware into the chip memory	For more details, see the ATWILC Firmware Download chapter
Step (14) Rest the chip Wi-Fi CPU	In the `0x3b0400` register change the value of bit `[2]` from `0` to `1`
Step (15) Set MUX to enable IRQN pin output	Set the bit 8 in `NMI_PIN_MUX_0` register
Step (16) Enable IRQ on IRQN pin	Set the bit 16 in `NMI_INTR_ENABLE` register

10.2 Transfer Sequence Between HIF Layer and ATWILC* Firmware

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

10.2.1 Frame Transmit

The following diagram shows the steps and states involved in sending a message from the host to the ATWILC* device.

Figure 10-3. HIF Frame Transmit to ATWILC*

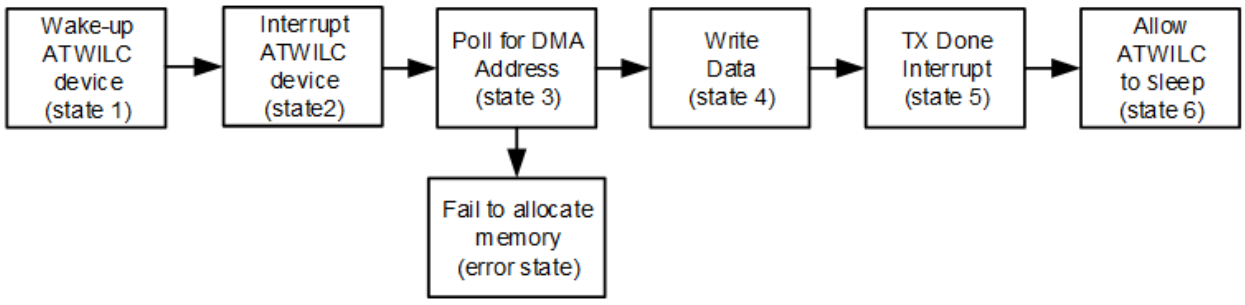


Table 10-3. HIF Frame Transmit to ATWILC*

Step	Description
Step (1) Wake up the ATWILC* device	Wake up the device to receive host requests

.....continued	
Step	**Description**
Step (2) Interrupt the ATWILC* device	• Prepare and set the HIF layer header to `NMI_STATE_REG` register (4 bytes header describing the sent packet) • Set the bit '`1`' of `WIFI_HOST_RCV_CTRL_2` register to raise an interrupt to the ATWILC* chip
Step (3) Poll for DMA address	Wait until the ATWILC* 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 buffer containing the HIF header, the Control buffer (if any), and the Data buffer (if any)
Step (5) TX Done interrupt	Broadcast after writing the data by setting bit '`1`' of `WIFI_HOST_RCV_CTRL_3` register. The ATWILC3000 device is set bit '`1`' of `INTERRUPT_CORTUS_2_3000D0`
Step (6) Allow the ATWILC* device to sleep	Allow the ATWILC* device to enter the Sleep mode again (if it wishes)

10.2.2 Frame Receive

The following figure shows the steps and states involved in sending a message from the ATWILC* device to the host.

Figure 10-4. HIF Frame Receive from ATWILC* to Host

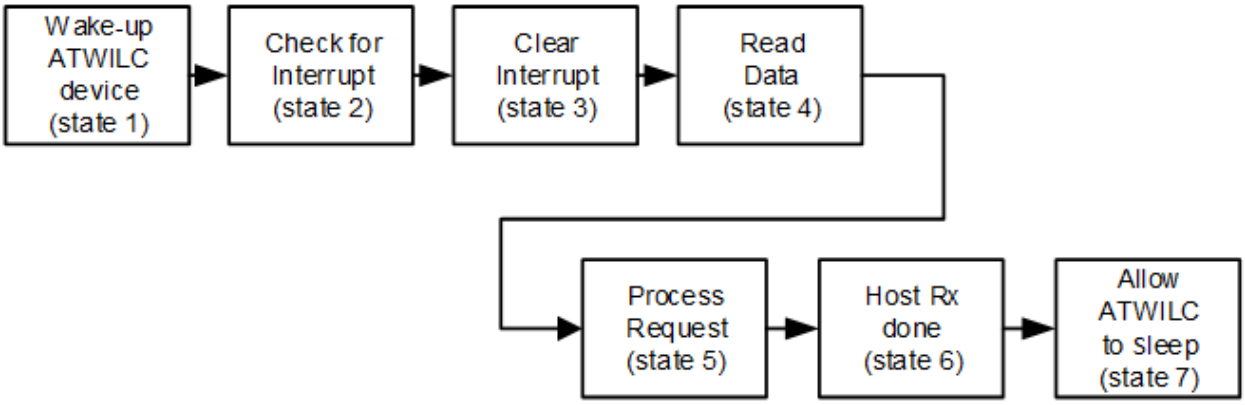


Table 10-4. HIF Frame Receive from ATWILC* to Host

Step	Description
Step (1) Wake-up the ATWILC* device	Wake-up the device to receive Host requests.
Step (2) Check for interrupt	Monitor BIT[IRG_FLAGS_OFFSET] of `WILC_INT_STATUS_REG` register. Disable the host from receiving interrupts (until this is processed).
Step (3) Clear interrupt	Clear the interrupt by writing 1 to `WILC_INT_CLEAR_REG` register.

.....continued	
Step	**Description**
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: ATWILC1000 – setting bit '`1`' of `WIFI_HOST_RCV_CTRL_0` register. ATWILC3000 – Setting bit '`0`' of `INTERRUPT_CORTUS_0_3000D0` Enable Host interrupt reception again.
Step (7) Allow the ATWILC* device to sleep	Allow the ATWILC* 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 the ATWILC* 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. The 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 `m2m_types.h` header file.
- **Payload Length (16-bits)** – the payload length in shown in bytes (does not include header).

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Group ID								**Op Code**							
Payload Length															
Payload **...** **...** **...**															

10.4 HIF Layer APIs

The interface between the application and the driver is performed 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`
- `hif_set_receive_buffer`

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 ATWILC* 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 the Sleep mode for the ATWILC* 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 sets the callback function for different components (e.g. `M2M_WIFI`, `M2M_HIF`, `M2M_OTA`, and so on). 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 ATWILC* 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 ATWILC* chip. The location and length of the data must be known in advance and specified. This is typically extracted from an earlier part of a transaction.
- `sint8 hif_send(uint8 u8Gid,uint8 u8Opcode,uint8 *pu8CtrlBuf,uint16 u16CtrlBufSize,uint8 *pu8DataBuf,uint16 u16DataSize, uint16 16DataOffset)` - This function causes the host driver to send data to the ATWILC* chip. The ATWILC* chip is 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.

1. The application requests a Wi-Fi scan using the following API.

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

2. The host driver Wi-Fi layer formats the request and forward it to HIF (Host Interface) layer as shown in the following code.

```
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;
}
```

3. The HIF layer sends the request to the ATWILC* chip as shown in the following code.

```
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");
            ret =  M2M_ERR_MEM_ALLOC;
            goto ERR1;
        }
    }
    else
    {
        M2M_ERR("(HIF)Fail to wakup the chip\n");
        goto ERR1;
    }
            /* TX STEP (6) */
    ret = hif_chip_sleep();
ERR1:
    return ret;}
```

4. The ATWILC* chip processes the request and interrupts the host after finishing the operation.
5. 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;
}
```

6. The appropriate handler is the Wi-Fi layer (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…
}
```

7. 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. ATWILC* SPI Protocol

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

The ATWILC* SPI protocol is implemented as a command-response transaction and assumes one 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 for checking the data integrity.

The format of Command and Response and Data frames are described in the following sections. The following are the behavior of the commands.

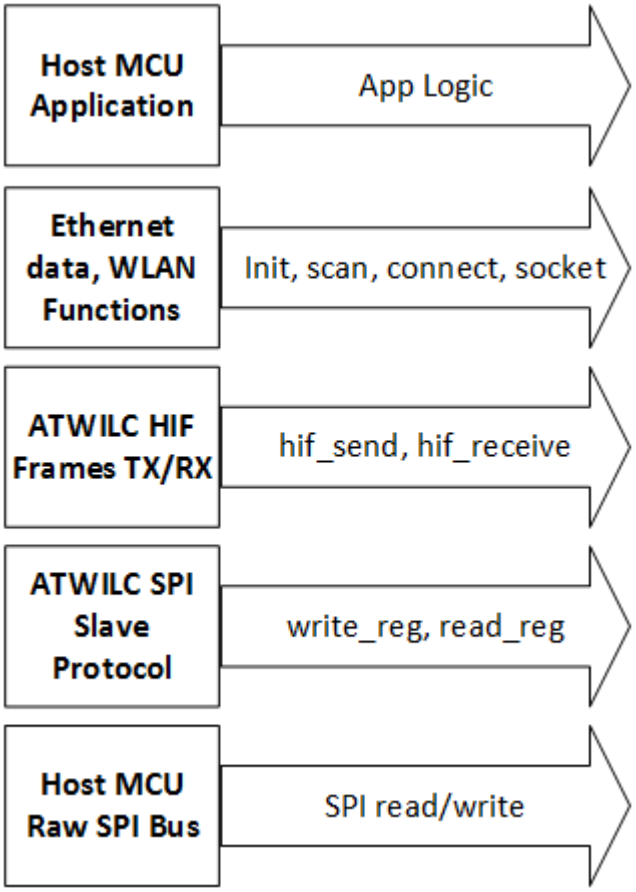
- There is a response for each command
- Transmitted/received data is divided into packets with fixed size
- For a write transaction (slave is receiving data packets), the slave must reply by a response for each data packet
- For a Read (RD) transaction (master is receiving data packets), the master does not send a response. If there is an error, the master requests a retransmission on the lost data packet
- Protection of commands and data packets by Cyclic Redundancy Check (CRC) is optional

11.1 SPI Protocol Layers

The ATWILC* SPI protocol consists of three layers:

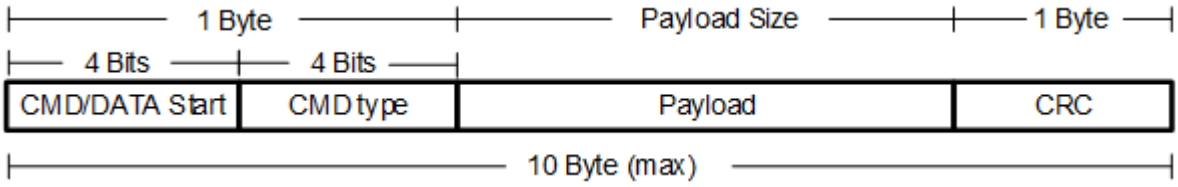
- **Layer 1** – the ATWILC* SPI slave protocol, allows the host MCU application to perform register/memory read and write operation in the ATWILC* device using raw SPI data exchange.
- **Layer 2** – the host MCU application uses the register and memory read and write capabilities to exchange host interface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

Figure 11-1. ATWILC* SPI Protocol Layers



11.1.1 Command Format

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



The first byte contains two fields:

- The Command (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

The following table details the frame format fields.

Table 11-1. Frame Format Fields

Field	Size	Description
CMD Start	4 bits	Command Start : 4’b1100

.....continued

Field	Size	Description
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	A: 3 B: 5 C: 6 D: 7	The Payload field may be of Type A,B,C or D **Type A (length 3)** **1- DMA single-word RD** Param: Read Address: Payload bytes: • B0: ADDRESS[23:16] • B1: ADDRESS[15:8] • B2: ADDRESS[7:0] **2- internal register RD** Param: Offset address (2 bytes): Payload bytes: • B0: OFFSET-ADDR[15:8] • B1: OFFSET-ADDR[7:0] • B2: 0 **3- Transaction termination command** Param: none Payload bytes: • B0: 0 • B1: 0 • B2: 0 **4- Repeat Data PKT command**

.....continued

Field	Size	Description
		Param: none Payload bytes: • B0: 0 • B1: 0 • B2: 0 **5- Soft reset command** Param: none Payload bytes: • B0: 0xFF • B1: 0xFF • B2: 0xFF **Type B (length 5)** ***1- DMA RD Transaction*** Params: • DMA Start Address : 3 bytes • DMA count : 2 bytes Payload bytes: • B0: ADDRESS[23:16] • B1: ADDRESS[15:8] • B2: ADDRESS[7:0] • B3: COUNT[15:8] • B4: COUNT[7:0] **2- DMA WR Transaction** Params: • DMA Start Address : 3 bytes • DMA count : 2 bytes Payload bytes: • B0: ADDRESS[23:16] • B1: ADDRESS[15:8] • B2: ADDRESS[7:0] • B3: COUNT[15:8] • B4: COUNT[7:0] **Type C (length 6)** **1- DMA Extended RD transaction** Params:

.....continued

Field	Size	Description
		• DMA Start Address : 3 bytes • DMA extended count: 3 bytes Payload bytes: • B0: ADDRESS[23:16] • B1: ADDRESS[15:8] • B2: ADDRESS[7:0] • B3: COUNT[23:16] • B4: COUNT[15:8] • B5: COUNT[7:0] **2- DMA Extended WR transaction** Params: • DMA Start Address : 3 bytes • DMA extended count: 3 bytes Payload bytes: • B0: ADDRESS[23:16] • B1: ADDRESS[15:8] • B2: ADDRESS[7:0] • B3: COUNT[23:16] • B4: COUNT[15:8] • B5: COUNT[7:0] **3- Internal register WR*** Params: • Offset address: 3 bytes • Write Data: 3 bytes * “clocked or clockless registers” Payload bytes: • 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] **Type D (length 7)** **1- DMA single-word WR** Params:

.....continued		
Field	**Size**	**Description**
		• Address: 3 bytes • DMA Data: 4 bytes Payload bytes: • 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]
CRC7	1 byte	Optional data integrity field comprising of 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

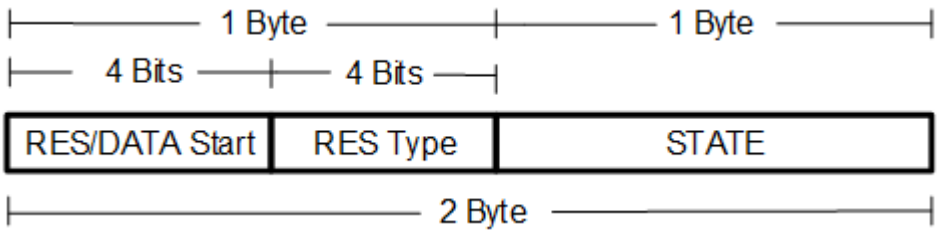
The following table summarizes the different commands according to the payload type (DMA address = 3-bytes).

Table 11-2. Commands in Payload

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 ATWILC* 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 4-bit fields which identify the response message and the response type.

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

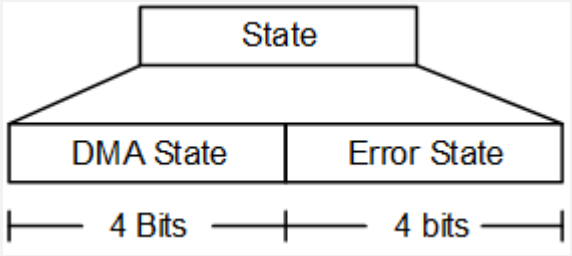
- B0-B3: Error state
- B4-B7: DMA state

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

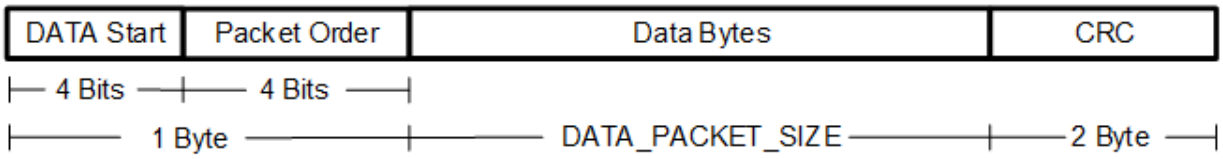
Field	Size	Description
Res Start	4 bits	Response Start: 4’b1100
Response Type	4 bits	If the response packet is for Command: • Contains of copy of the Command Type field in the Command If the response packet is for received Data Packet: • 4’b0001: First data packet is received • 4’b0010: Receiving data packets • 4’b0011: Last data packet is received • 4’b1111: Reserved value

.....continued

Field	Size	Description
State	1 byte	This field is divided into two sub fields:  DMA State: • 4'b0000: DMA ready for any transaction • 4'b0001: DMA engine is busy Error State: • 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 send 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 in the following image.



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 256 B, 512 B, 1 KB (default), 2 KB, 4 KB, or 8 KB. If a transfer has a length of **m**, which exceeds DATA_PACKET_SIZE, the sender must split it into multiple DATA_PACKET_SIZE as shown in Equation 1:

(m – (n-1) DATA_PACKET_SIZE)----------Equation 1*

Where,

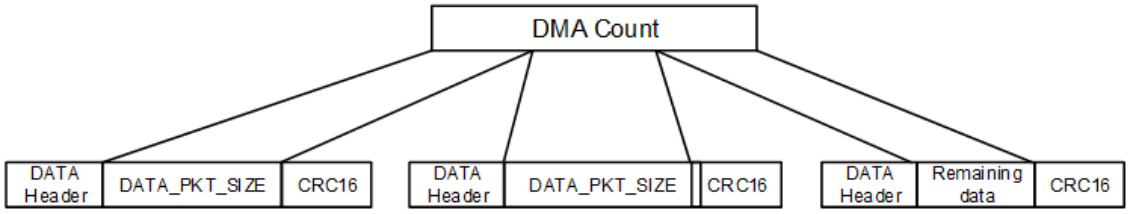
1.. n-1 = length of the DATA_PACKET_SIZE

n = frame length

This is illustrated below:

- If DMA count <= DATA_PACKET_SIZE:
  The data packet is *DATA_Header + DMA count +optional CRC16*, that is, 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, that is no padding.

The frame fields are described in the following table.

Table 11-3. Frame Field

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

Table 11-4. Error Recovery Mechanism

Error Type	Recovery Mechanism
Master	

.....continued

Error Type	Recovery Mechanism
CRC error in command	• Error response received from slave. • Retransmit the command.
CRC error in received data	• Issue a repeat command for the data packet that has a CRC error. • Slave sends a response to the previous command. • Slave keeps the start DMA address of the previous data packet, so it can retransmit it. • Receive the data packet again.
No response is received from slave	• Synchronization is lost between the master and slave. • The worst case is when slave is in receiving data state. • Solution: The master must 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	• Transmit all ones until master receives a response of all ones from the slave. • Then deactivate the output data line.
Slave	
Unsupported command	• Send response with error. • Returns to command monitor state.
Receive command CRC error	• Send response with error. • Wait for command retransmission.
Received data CRC error	• Send response with error. • Wait for retransmission of the data packet.
Internal general error	• The Master must do a Soft Reset on the Slave.
TX/RX Data count error	• Only the master can detect this error. • Slave operates with the data count received until the count finishes or the master terminates the transaction. • In both cases, the master can retry the command from the start.

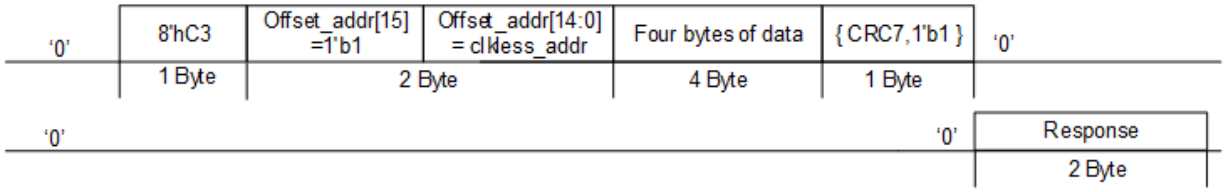
.....continued

Error Type	Recovery Mechanism
No response to Soft Reset command	• 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 must 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 Notes	• The slave must monitor the received line for command reception at any time. • When a CMD start is detected, the slave receives 8 bytes then return again to the command reception state. • When the slave is transmitting data, it must also monitor for command reception. • When the slave is receiving data, it monitors for command reception between the data packets. • Issuing a Soft Reset command is detected in all cases.

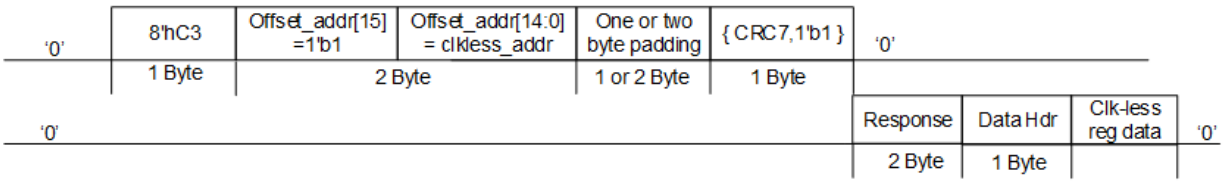
11.1.5 Clockless Registers Access

Clockless register access allows a host device to access registers on the ATWILC* 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 must be set '1' to differentiate between clockless and clocked Access mode.

For clockless register write – the protocol master must wait for the response as shown here.



For clockless register read – according to the interface, the protocol slave must not send CRC16. One or two byte padding depends on three or four byte DMA addresses.

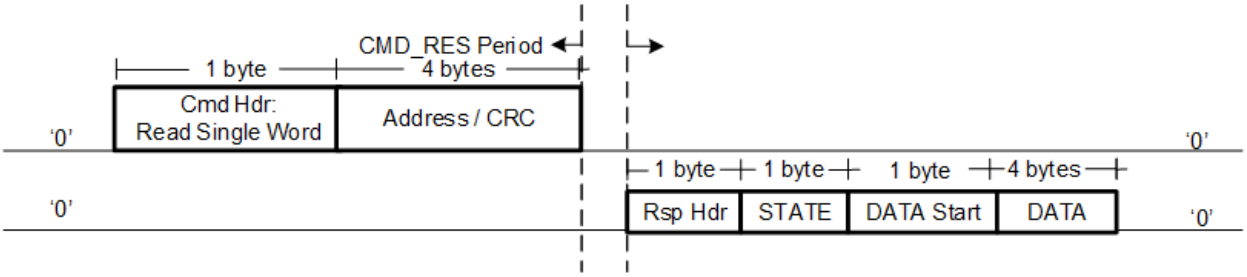


11.2 Message Flow for Basic Transactions

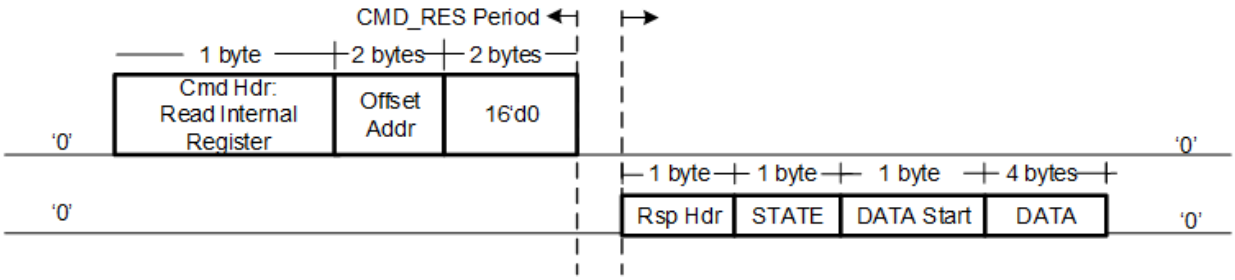
This section shows the essential message exchanges and timings associated with the following commands:

- 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.

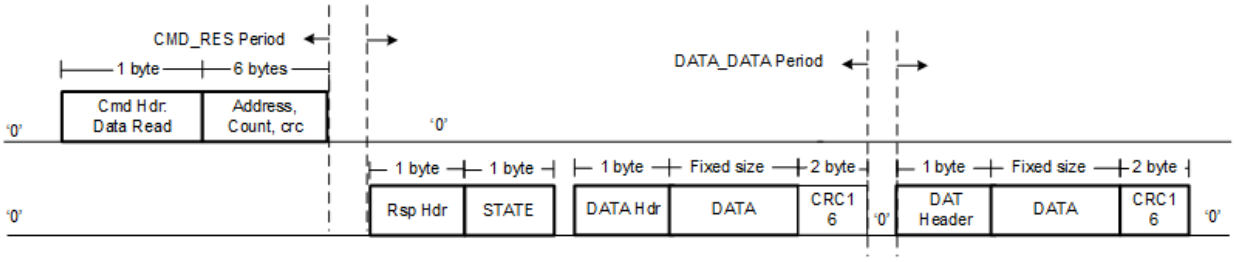
Slave – sends the data packets, separated by DATA_DATA_PERIOD (see note below) where DATA_DATA_PERIOD is controlled by software and has one of these values:

- NO_DELAY (default)
- 4_BYTE_PERIOD
- 8_BYTE_PERIOD
- 16_BYTE_PERIOD

Slave – continues sending until the count ends.

Master – receive data packets. No response is sent for data packets but a termination/re-transmit command may be sent if there is an error.

The message sequence for this case is shown below:



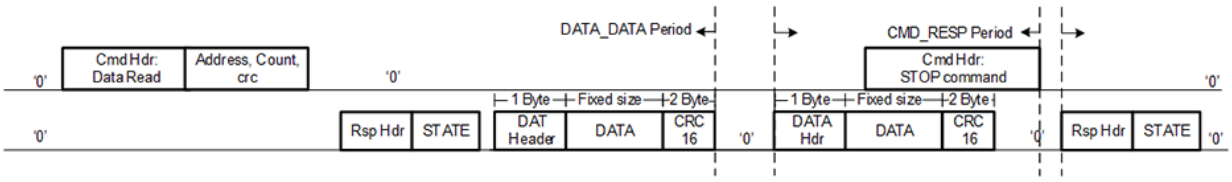
Termination Command Is Issued

Master – may issue a termination command at any time during the transaction.

Master – must monitor for RES_START after CMD_RESP_PERIOD.

Slave – must cut off the current running data packet if any.

Slave – must respond to the termination command after CMD_RESP_PERIOD from the end of the termination command packet.



Repeat Command Is Issued

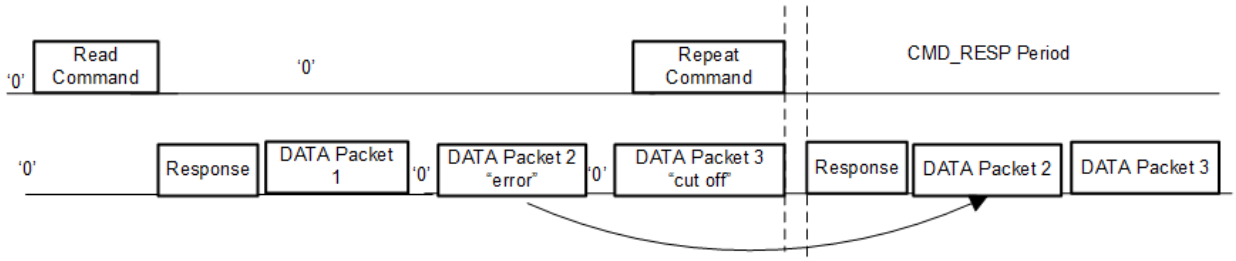
Master – may issue a repeat command at any time during the transaction.

Master – must monitor for RES_START after CMD_RESP_PERIOD.

Slave – must cut off the current running data packet, if any.

Slave – must respond to the repeat command after CMD_RESP_PERIOD from the end of the repeat command packet.

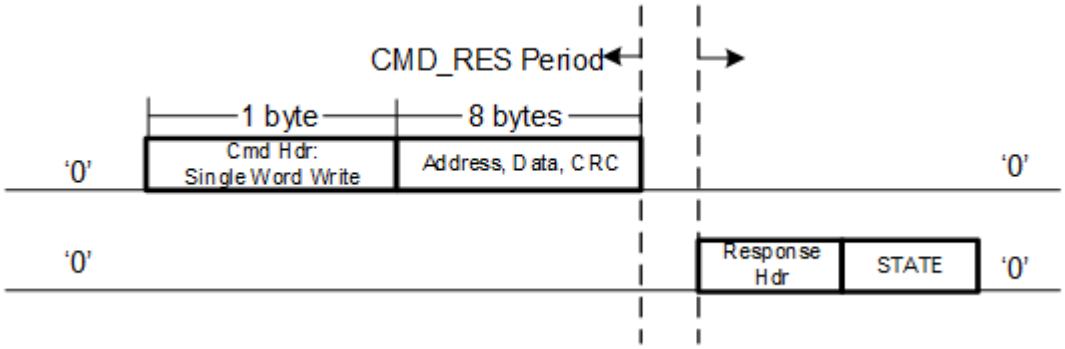
Slave – re-sends the data packet that has an error then continues the transaction as normal.



11.2.4 Write Single Word

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

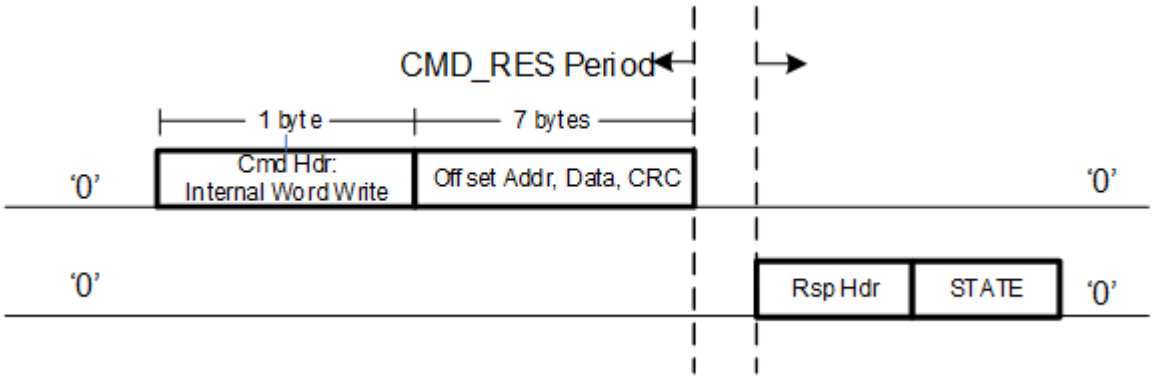
Slave - takes the data and sends a command response.



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

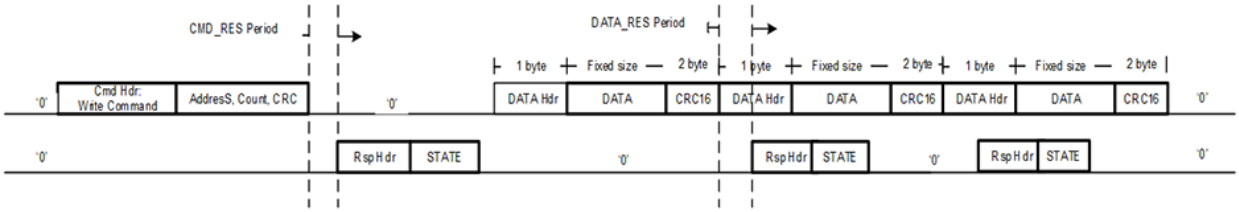
Note: CMD_RES_PERIOD is controlled by SW taking one of the values:

- NO_DELAY (default)
- 1_BYTE_PERIOD
- 2_BYTE_PERIOD
- 3_BYTE_PERIOD

The master must monitor for RES_START after CMD_RES_PERIOD.

Note: DATA_RES_PERIOD is controlled by software taking one of these values:

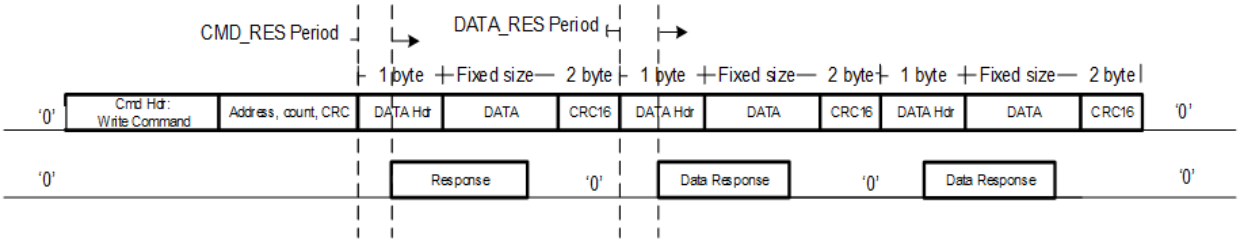
- NO_DELAY (default)
- 1_BYTE_PERIOD
- 2_BYTE_PERIOD
- 3_BYTE_PERIOD



Case 2 - Master does not wait for a command response

Master - sends the data packets directly after the command and 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

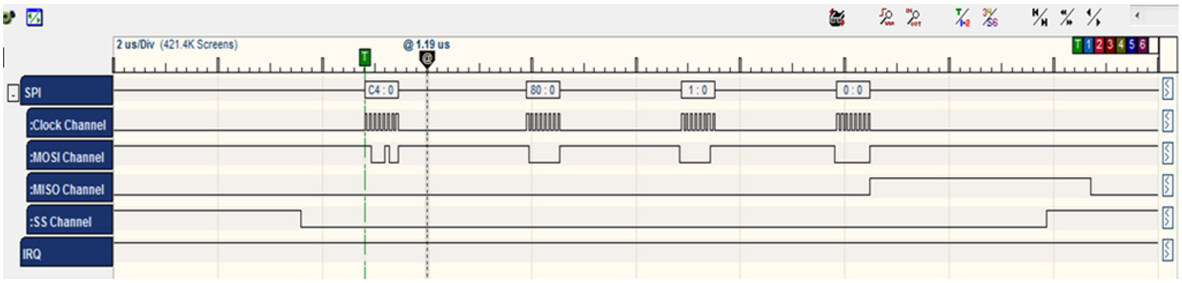
To illustrate how the ATWILC* SPI protocol works, the SPI bytes from the scan request example are dumped and the sequence is described in the following sections.

11.3.1 TX (Send Request)

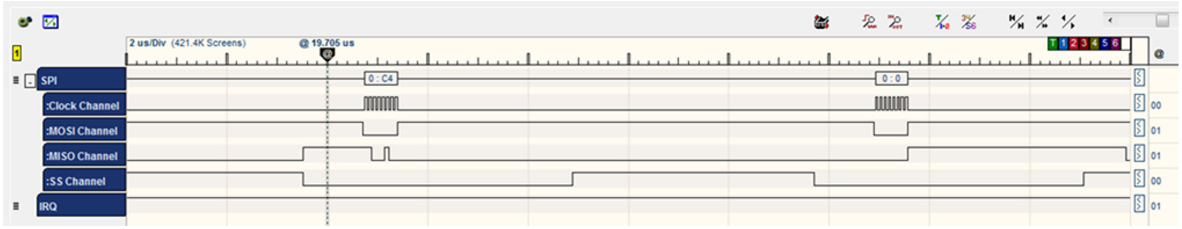
1. First step in `hif_send() API` is to wake up the chip.

```
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;
}
```

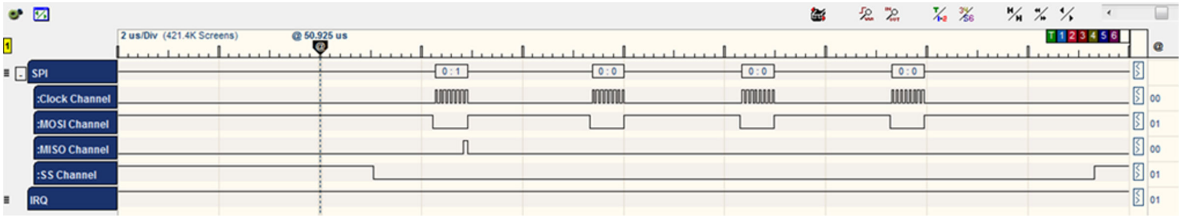
```
Command        CMD_INTERNAL_READ:    0xC4    /* internal register read */
            BYTE [0] = CMD_INTERNAL_READ
               BYTE [1] = address >> 8;            /* address = 0x01 */
            BYTE [1] |= (1 << 7);               /* clockless register */
            BYTE [2] = address;
    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



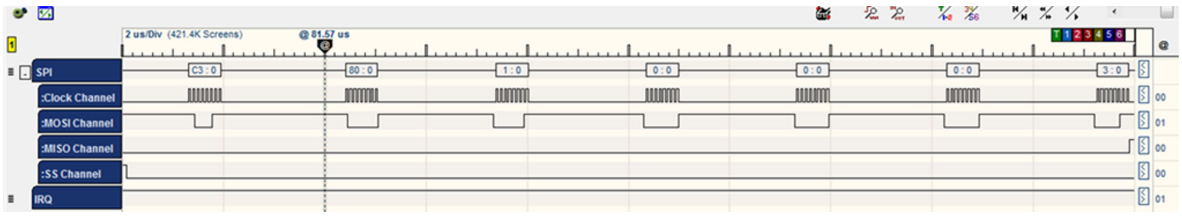
2. The ATWILC* acknowledges the command by sending three bytes [C4] [0] [F3].



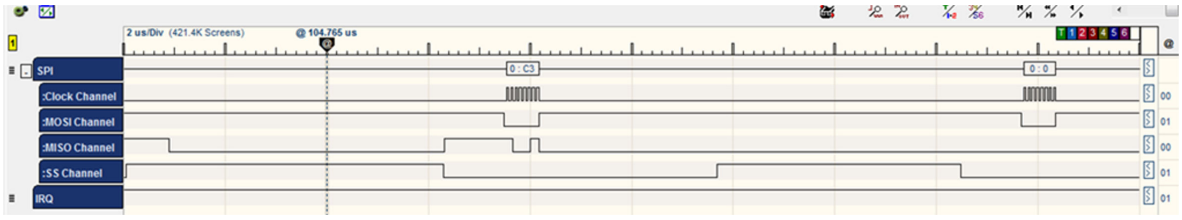
3. The ATWILC* chip sends the value of the register 0x01 which equals 0x01.



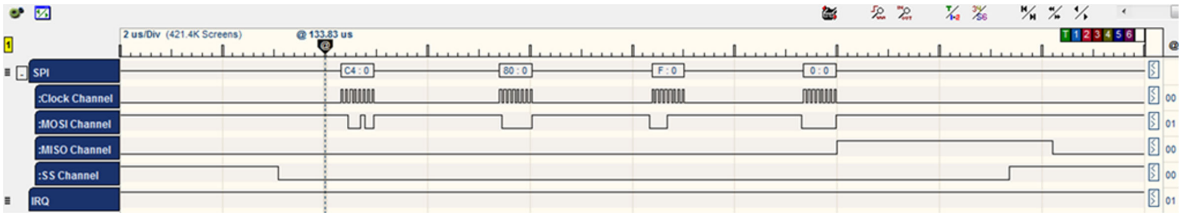

```
Command    CMD_INTERNAL_WRITE:    C3          /*     internal register write */
        BYTE [0] = CMD_INTERNAL_WRITE
        BYTE [1] = address >> 8;               /*     address = 0x01          */
        BYTE [1] |= (1 << 7);              /*     clockless register     */
        BYTE [2] = address;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



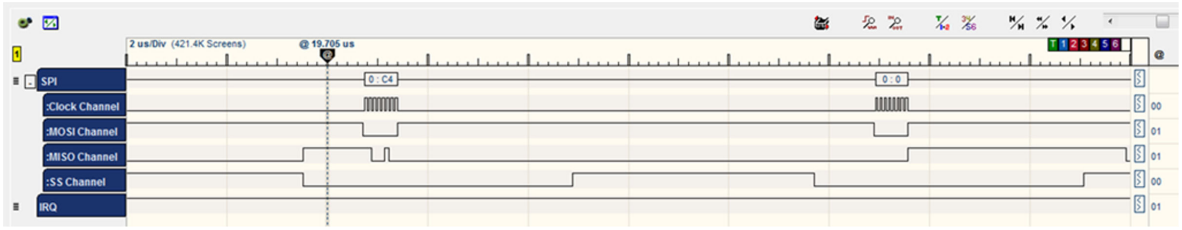
4. The ATWILC* acknowledges the command by sending two bytes [C3] [0].




```
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;
```

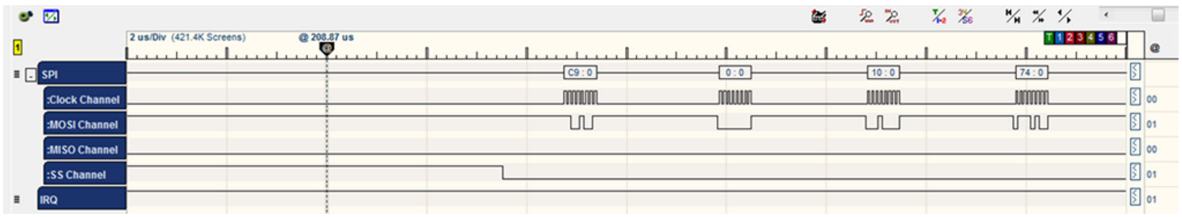


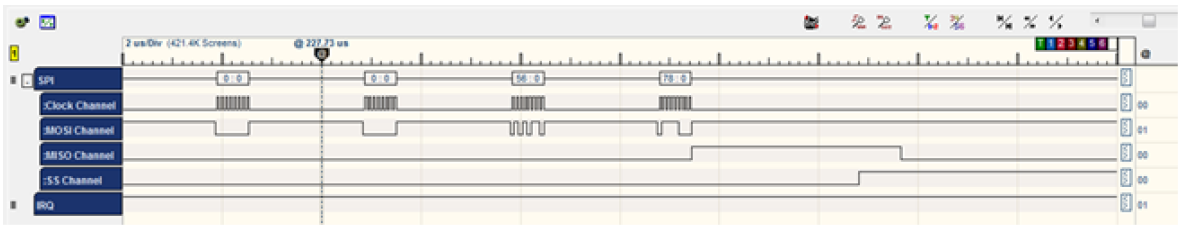
5. The ATWILC* acknowledges the command by sending three bytes [C4] [0] [F3].



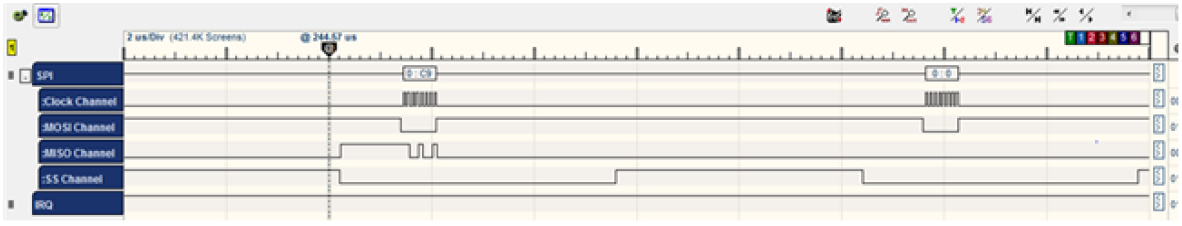
6. Then ATWILC* chip sends the value of the register 0x01 which equals 0x07.

```
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;
```





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

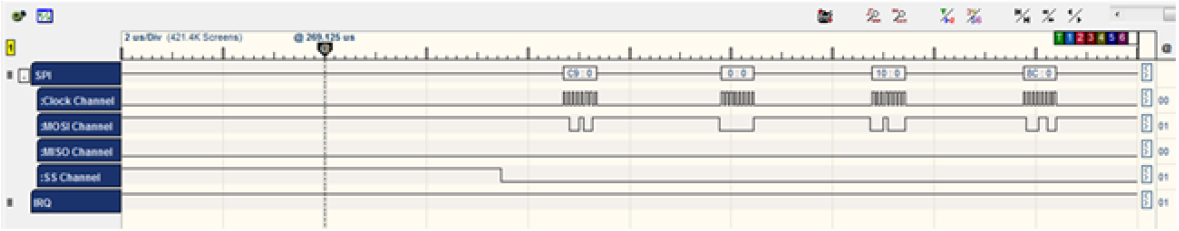


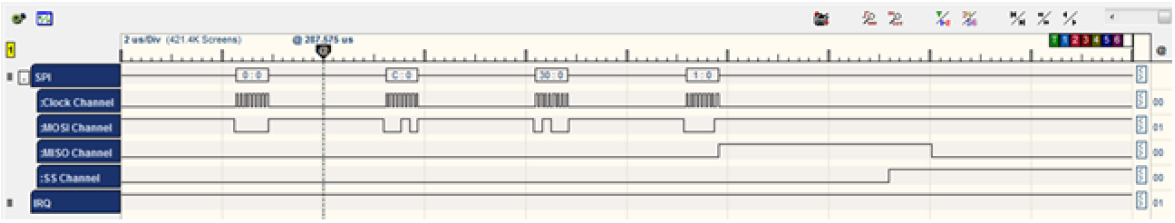
8. At this point, HIF finishes executing the clockless wake up of the ATWILC* chip.
9. The HIF layer prepares and sets the HIF layer header to NMI_STATE_REG register (4 byte or 8 byte header describing the packet to be sent).

10. Set bit '1' of WIFI_HOST_RCV_CTRL_2 register to raise an interrupt to the chip.

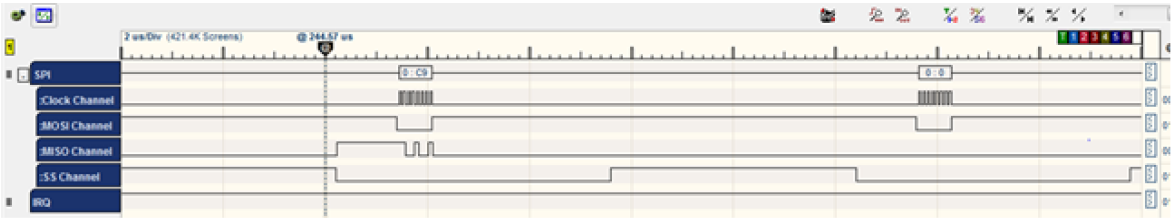
```
sint8 hif_send(uint8 u8Gid,uint8 u8Opcode,uint8 *pu8CtrlBuf,uint16 u16CtrlBufSize,
               uint8 *pu8DataBuf,uint16 u16DataSize, uint16 u16DataOffset)
{
    volatile tstrHifHdr    strHif;
    volatile uint32 reg;
    strHif.u8Opcode    = u8Opcode&(~NBIT7);
    strHif.u8Gid        = u8Gid;
    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 = 0x108c */
        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  */
```



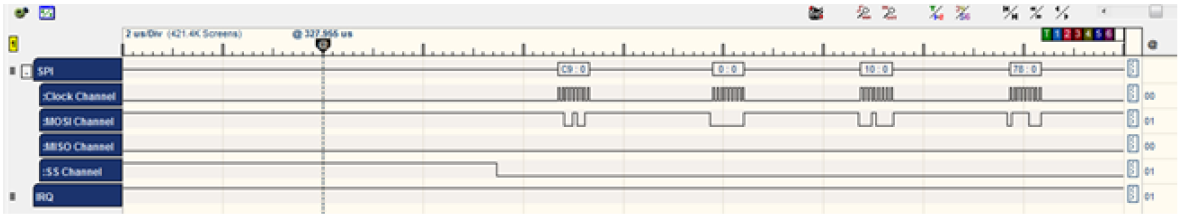


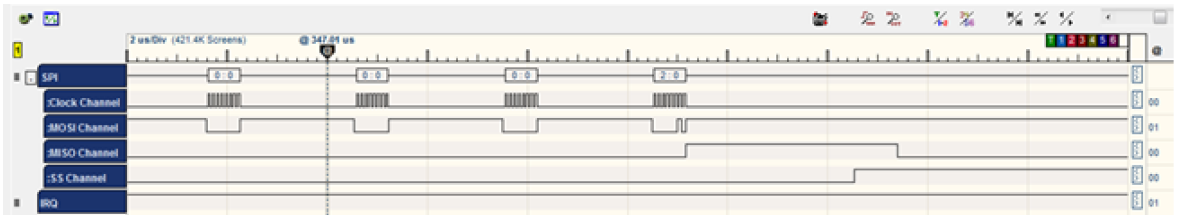
11. The ATWILC* acknowledges the command by sending two bytes [C9] [0].



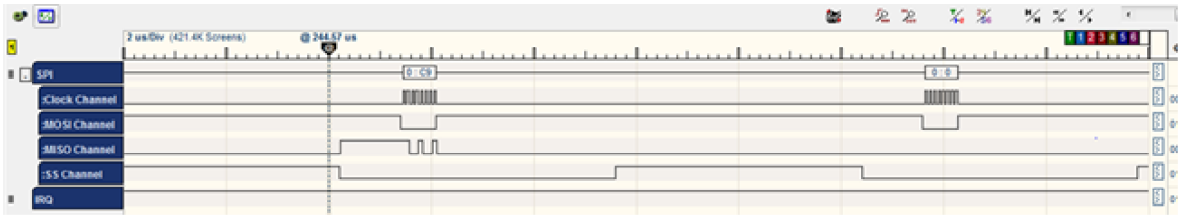

```
Command    CMD_SINGLE_WRITE:0XC9        /*     single word write */
        BYTE [0] = CMD_SINGLE_WRITE
        BYTE [1] = address >> 16;       /*     WIFI_HOST_RCV_CTRL_2address = 0x1078*/
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = u32data >> 24;        /*     Data = 0x02 */
        BYTE [5] = u32data >> 16;
```

```
        BYTE [6] = u32data >> 8;
        BYTE [7] = u32data;
```





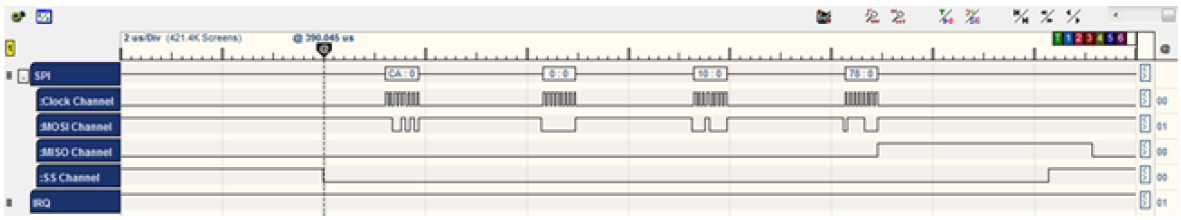
12. The ATWILC* acknowledges the command by sending two bytes [C9] [0].



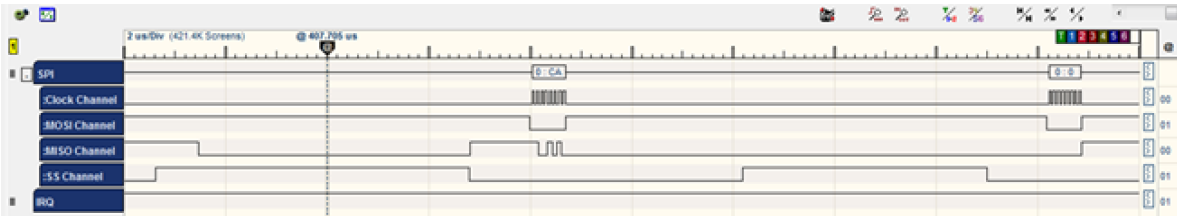
13. Then HIF polls for DMA address.

```
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;
    }
}
```

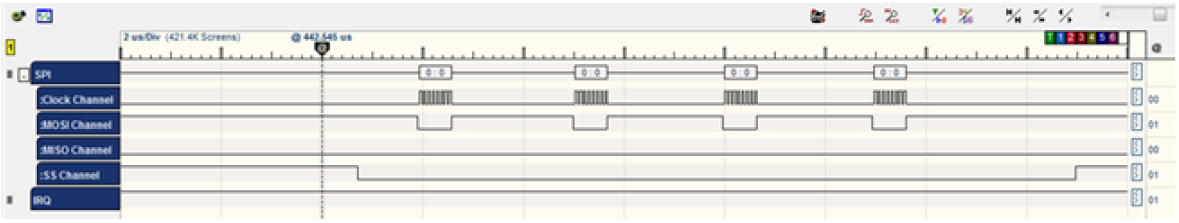
```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_2 address = 0x1078 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



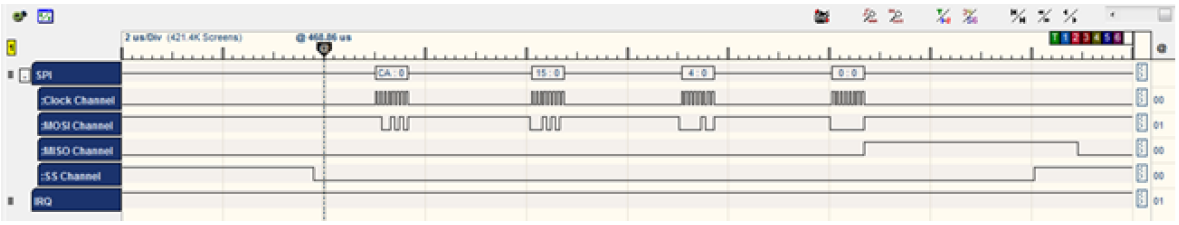
14. The ATWILC* acknowledges the command by sending three bytes [CA] [0] [F3].



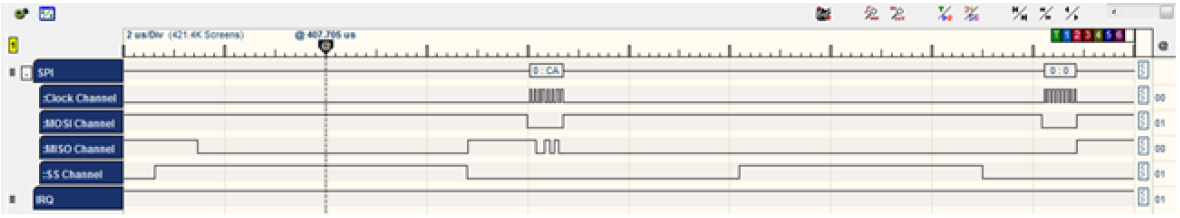
15. The ATWILC* chip sends the value of the register 0x1078, which equals 0x00.



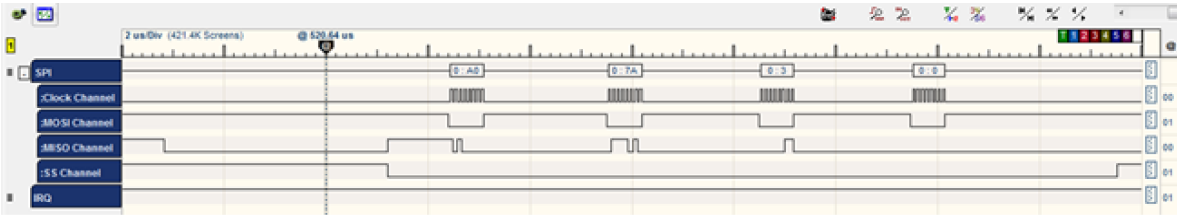
```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;          /* address = 0x1504 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



16. The ATWILC* acknowledges the command by sending three bytes [CA] [0] [F3].



17. The ATWILC* chip sends the value of the register 0x1504, which equals 0x037AA0.

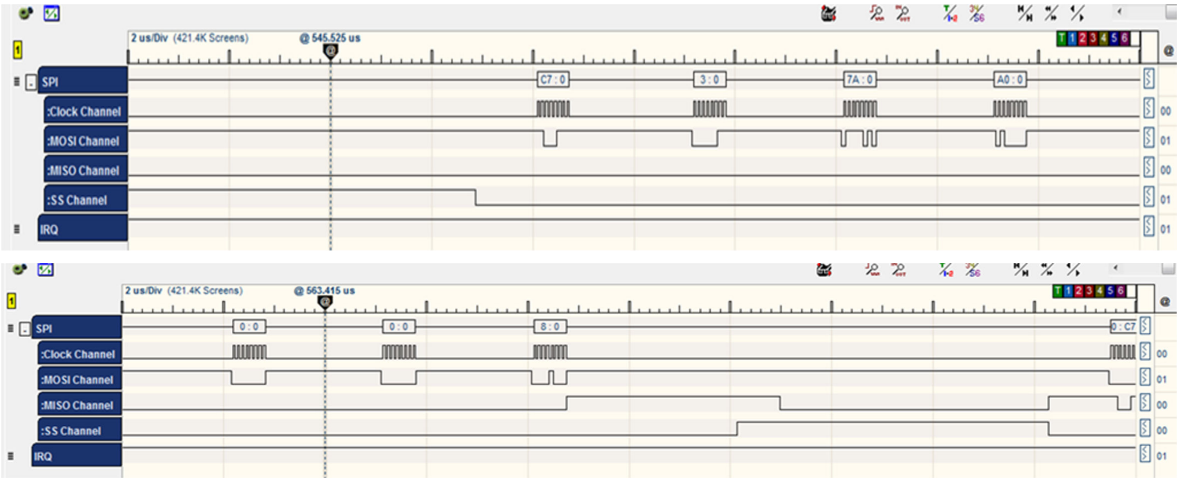


18. The ATWILC* writes the HIF header to the DMA memory address.

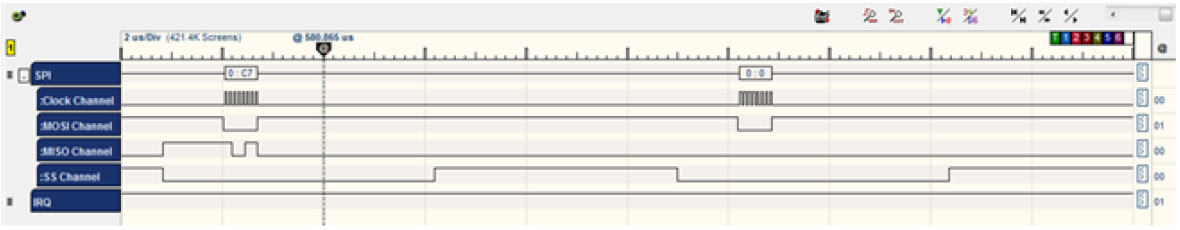
```
u32CurrAddr = dma_addr;
strHif.u16Length=NM_BSP_B_L_16(strHif.u16Length);
ret = nm_write_block(u32CurrAddr, (uint8*)&strHif, M2M_HIF_HDR_OFFSET);
```

```
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;              /* size = 0x08     */
```

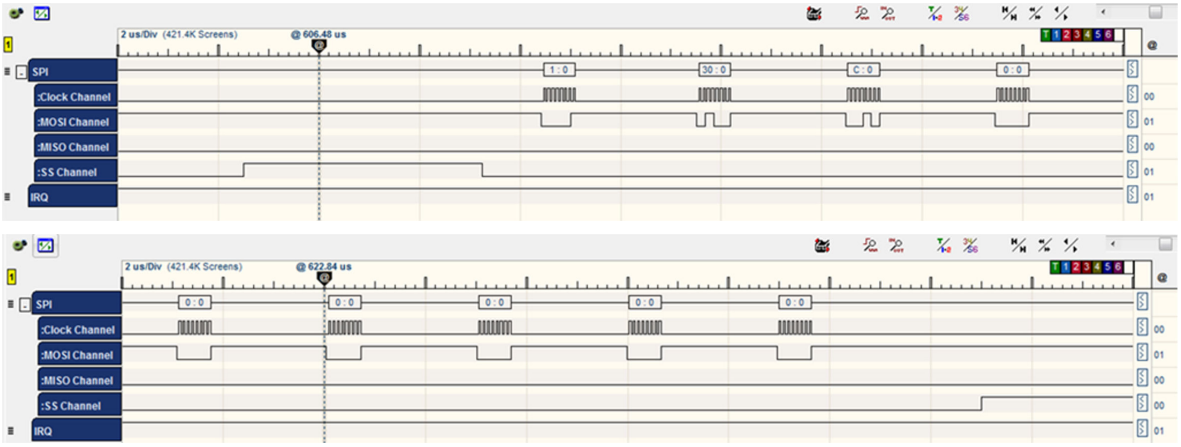
```
        BYTE [5] = size >> 8;
        BYTE [6] = size;
```



19. The ATWILC* acknowledges the command by sending three bytes [C7] [0] [F3].



20. The HIF layer writes the data.

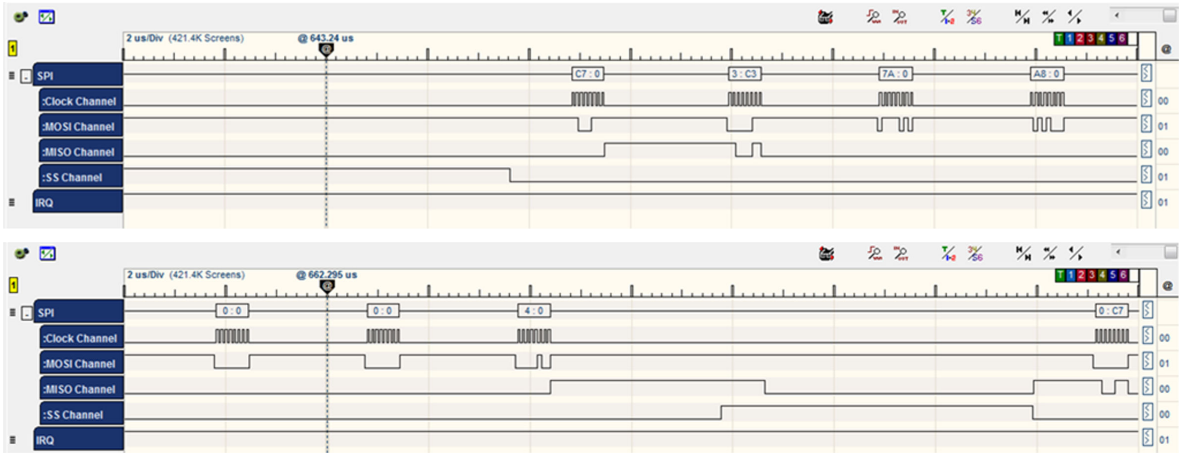


21. The HIF writes the Control Buffer data (part of the framing of the request).

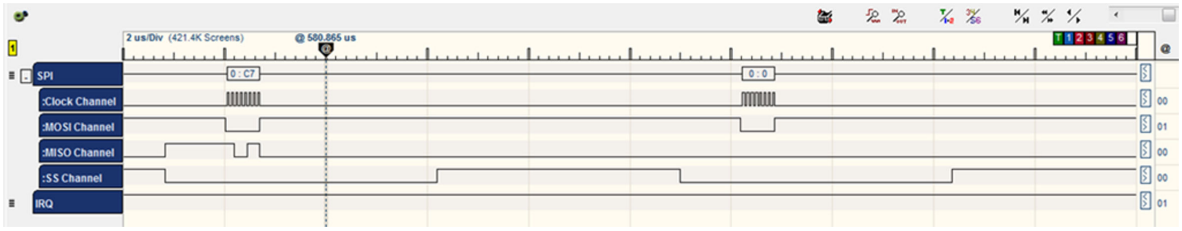
```
if (pu8CtrlBuf != NULL)
{
    ret = nm_write_block(u32CurrAddr, pu8CtrlBuf, u16CtrlBufSize);
    if(M2M_SUCCESS != ret) goto ERR1;
    u32CurrAddr += u16CtrlBufSize;
}
```

```
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     */
```

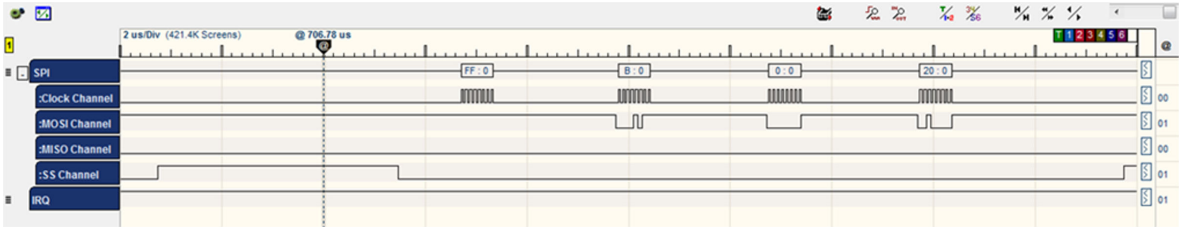
```
        BYTE [5] = size >> 8;
        BYTE [6] = size;
```



22. The ATWILC* acknowledges the command by sending three bytes [C7] [0] [F3].



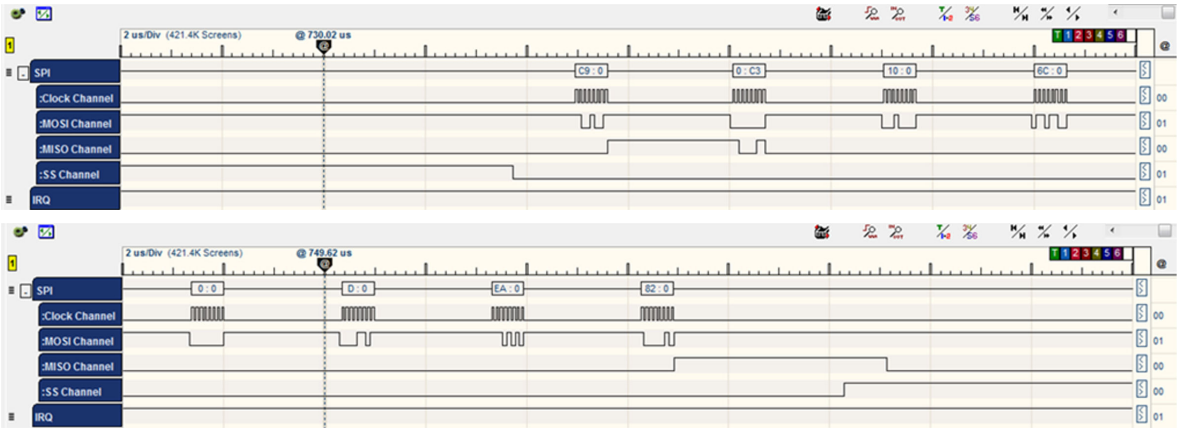
23. The HIF layer writes the data.



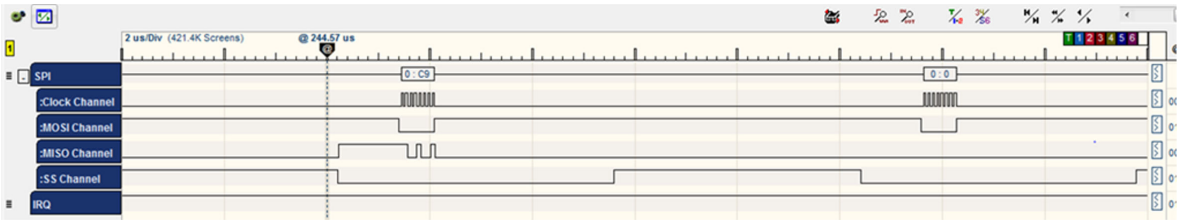
24. The HIF finished writing the request data to memory and is going to interrupt the chip notifying that host TX is done.

```
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;
```



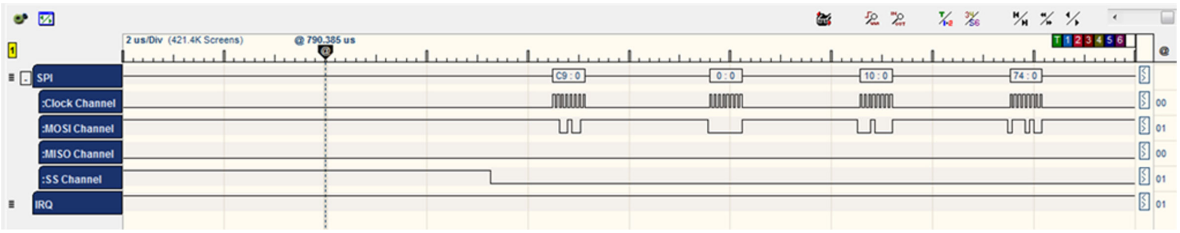
25. The ATWILC* acknowledges the command by sending two bytes [C9] [0].

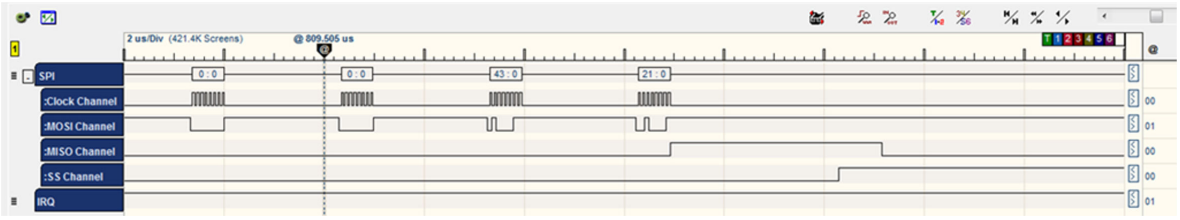


26. The HIF layer allows the chip to enter Sleep mode again.

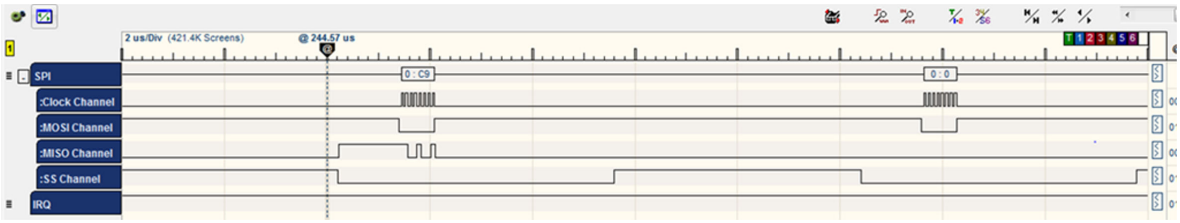
```
sint8 hif_chip_sleep(void)
{
    sint8 ret = M2M_SUCCESS;
    uint32 reg = 0;
    ret = nm_write_reg(WAKE_REG, SLEEP_VALUE);
    /* Clear bit 1 */
    ret = nm_read_reg_with_ret(0x1, &reg);
    if(reg&0x2)
    {
        reg &=~(1 << 1);
        ret = nm_write_reg(0x1, reg);
    }
}
```

```
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;
```

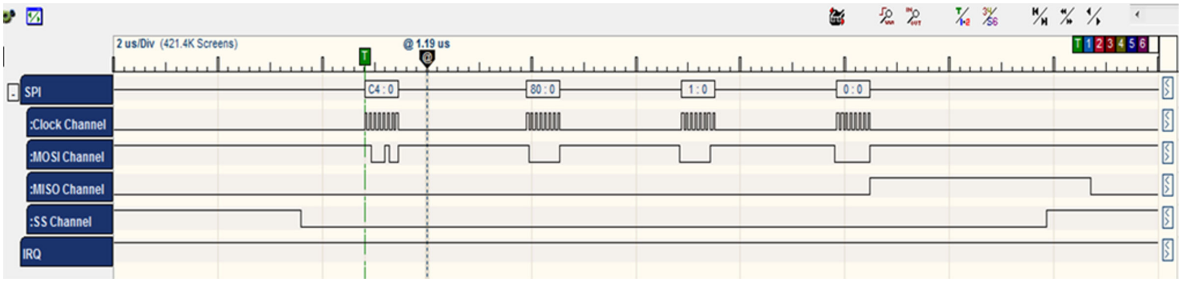




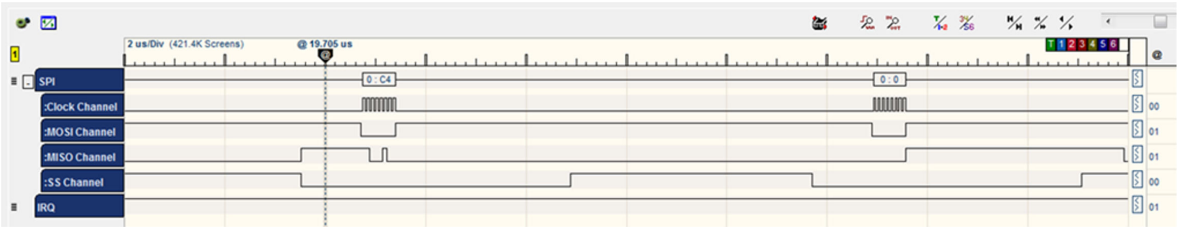
27. The ATWILC* acknowledges the command by sending two bytes [C9] [0].



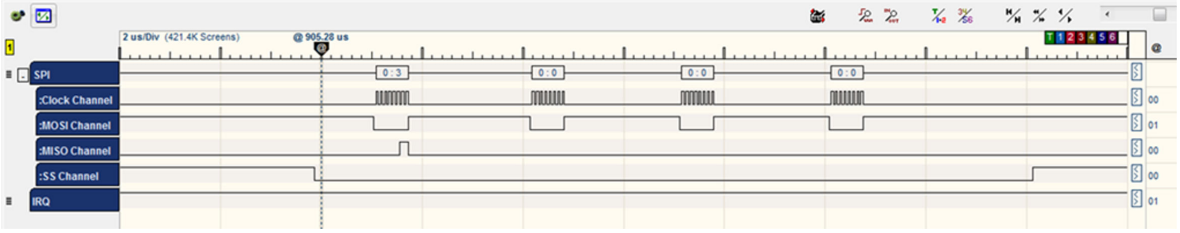

```
Command     CMD_INTERNAL_READ:    0xC4     /* internal register read     */
        BYTE [0] = CMD_INTERNAL_READ
        BYTE [1] = address >> 8;          /* address = 0x01     */
        BYTE [1] |= (1 << 7);             /* clockless register     */
        BYTE [2] = address;
        BYTE [3] =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



28. The ATWILC* acknowledges the command by sending three bytes [C4] [0] [F3].

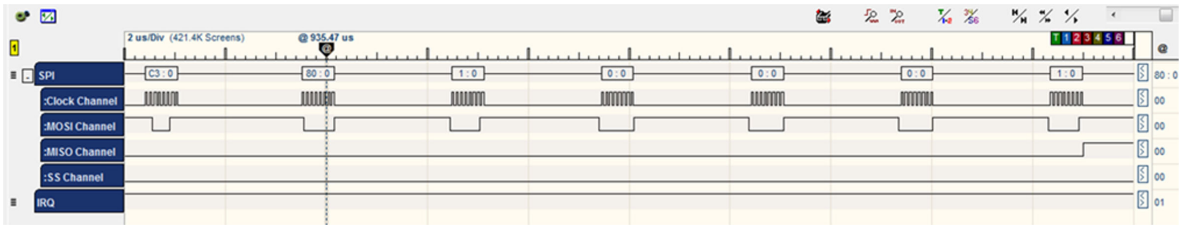


29. The ATWILC* chip sends the value of the register 0x01 which equals 0x03.

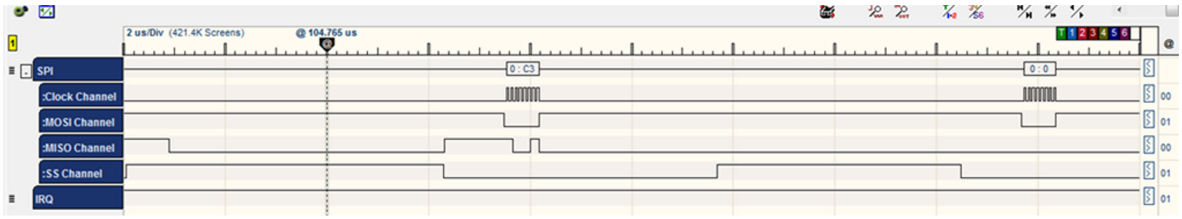



```
Command    CMD_INTERNAL_WRITE:    C3        /* internal register write      */
        BYTE [0] = CMD_INTERNAL_WRITE
        BYTE [1] = address >> 8;          /* address = 0x01      */
        BYTE [1] |= (1 << 7);                   /* clockless register     */
        BYTE [2] = address;
      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

```
        BYTE [5] = u32data >> 8;
        BYTE [6] = u32data;
```



30. The ATWILC* chip acknowledges the command by sending two bytes [C3] [0].



31. At this point, the HIF layer has completed posting the scan Wi-Fi request to the ATWILC* chip for processing.

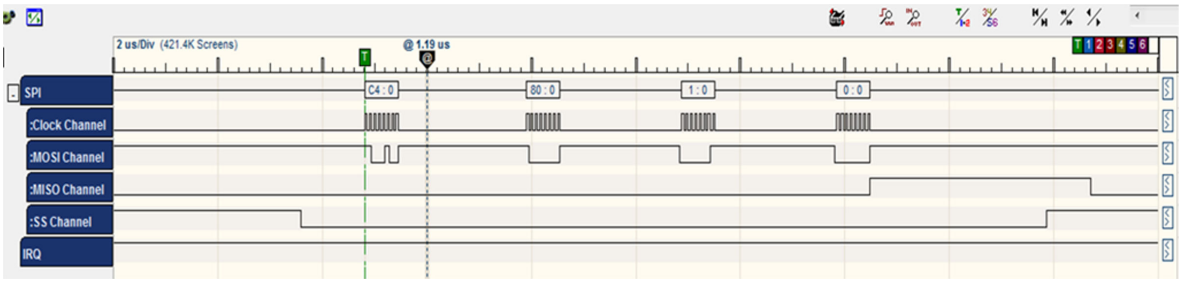
11.3.2 RX (Receive Response)

After finishing the required operation (scan Wi-Fi), the ATWILC* interrupts the host to notify of the processing of the request. The host handles this interrupt to receive the response.

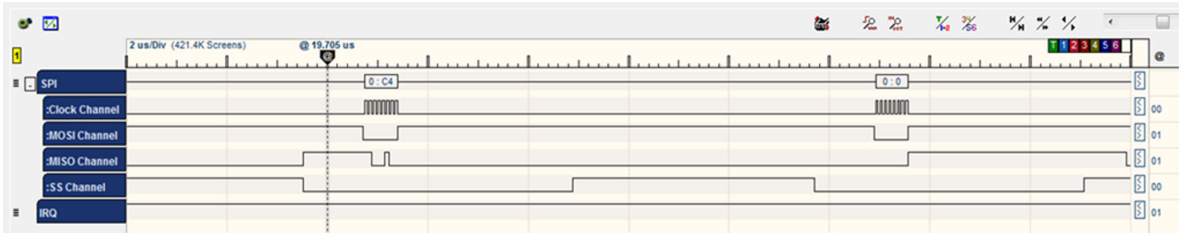
1. First step in `hif_isr ( )` is to wake up the ATWILC* chip.

```
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;
}
```

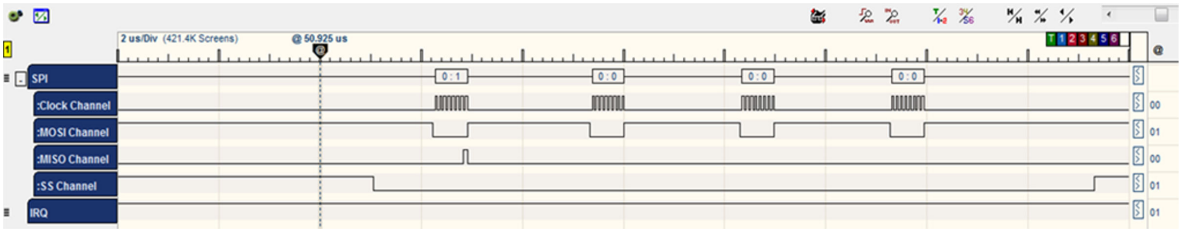
```
Command    CMD_INTERNAL_READ:    0xC4    /* internal register read */
        BYTE [0] = CMD_INTERNAL_READ
        BYTE [1] = address >> 8;          /* address = 0x01 */
        BYTE [1] |= (1 << 7);          /* clockless register */
        BYTE [2] = address;
        BYTE [3] = 0x00;
```



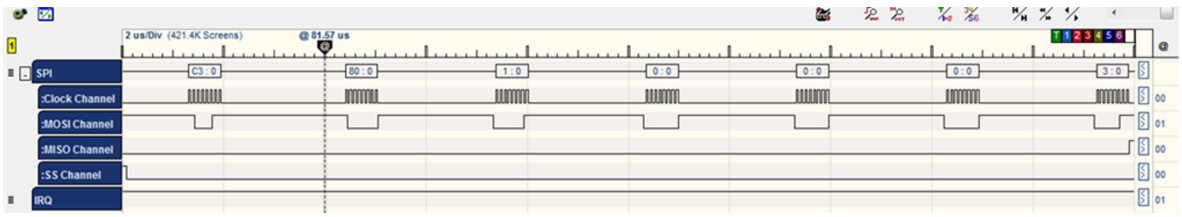
2. The ATWILC* acknowledges the command by sending three bytes [C4] [0] [F3].



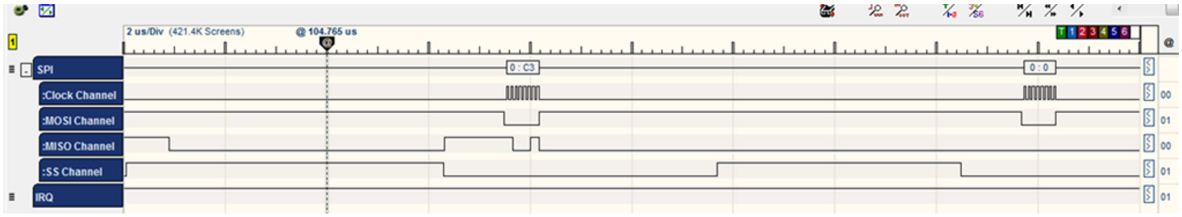
3. The ATWILC* chip sends the value of the register 0x01 which equals 0x01.



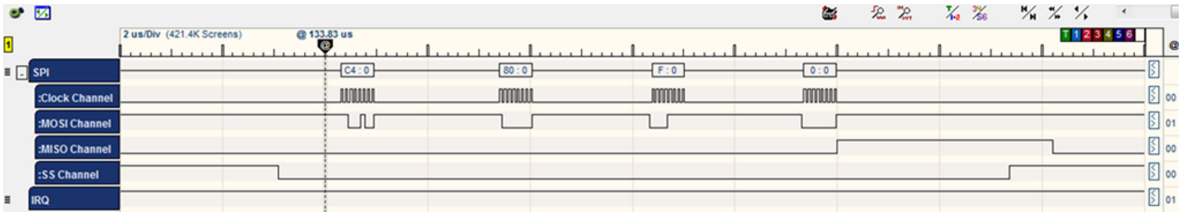

```
Command    CMD_INTERNAL_WRITE:    C3          /*     internal register write */
        BYTE [0] = CMD_INTERNAL_WRITE
        BYTE [1] = address >> 8;               /*     address = 0x01          */
        BYTE [1] |= (1 << 7);                /*     clockless register     */
        BYTE [2] = address;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



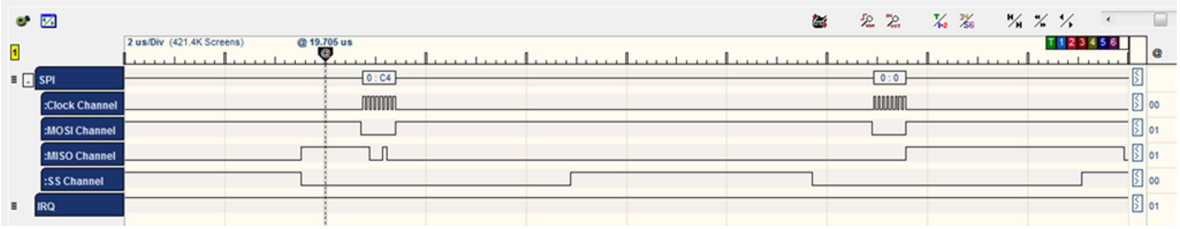
4. The ATWILC* acknowledges the command by sending two bytes [C3] [0].



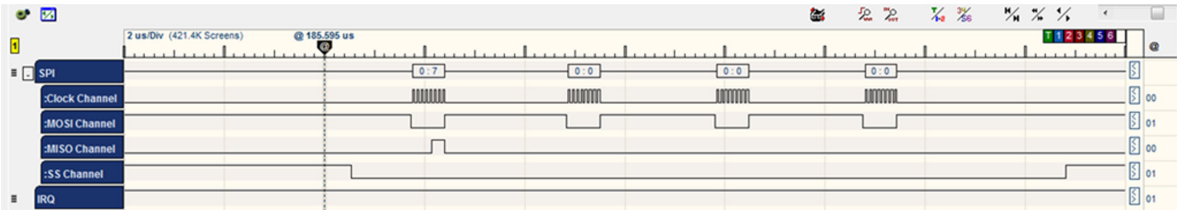

```
Command    CMD_INTERNAL_READ:    0xC4          /*     internal register read    */
        BYTE [0] = CMD_INTERNAL_READ
        BYTE [1] = address >> 8;             /*     address = 0x0F          */
        BYTE [1] |= (1 << 7);                /*     clockless register     */
        BYTE [2] = address;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



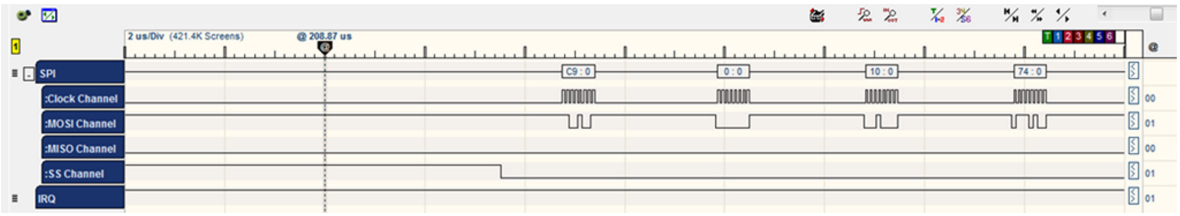
5. The ATWILC* acknowledges the command by sending three bytes [C4] [0] [F3].

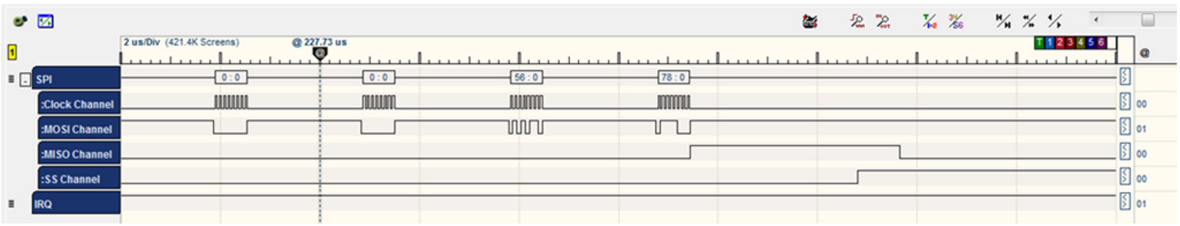


6. Then ATWILC* chip sends the value of the register 0x01 which equals 0x07.

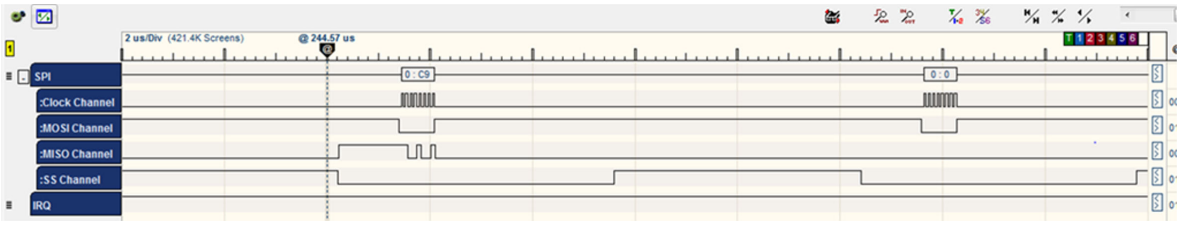



```
Command    CMD_SINGLE_WRITE:0XC9            /* single word write */
        BYTE [0] = CMD_SINGLE_WRITE
        BYTE [1] = address >> 16;        /* WAKE_REG address = 0x1074 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = u32data >> 24;        /* WAKE_VALUE Data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





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

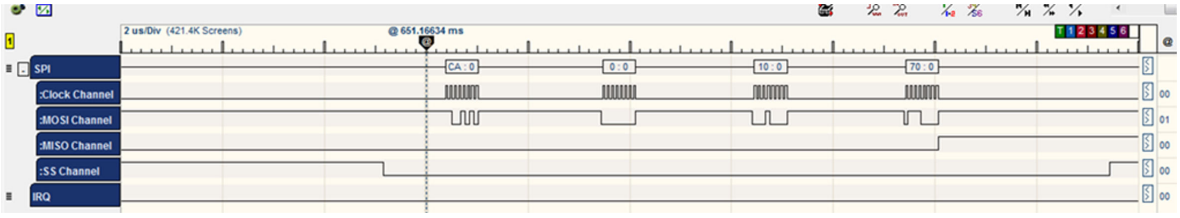


8. Read register `WIFI_HOST_RCV_CTRL_0` to check if there is a new interrupt, and clear it.

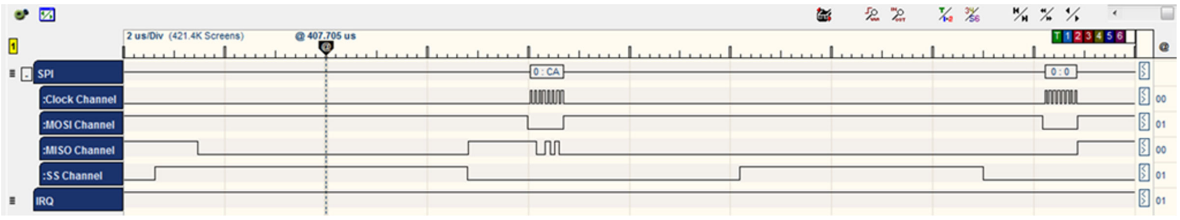
```
static sint8 hif_isr(void)
{
    sint8 ret ;
    uint32 reg;
    volatile tstrHifHdr strHif;

    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);
```

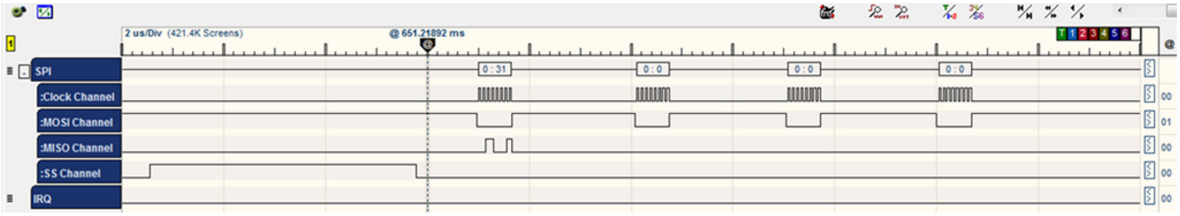
```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read         */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



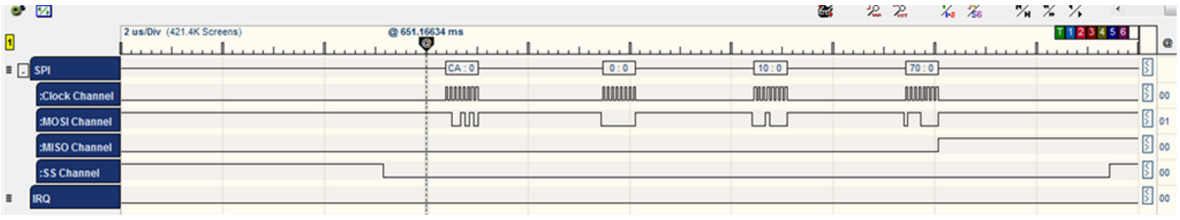
9. The ATWILC* acknowledges the command by sending three bytes [CA] [0] [F3].



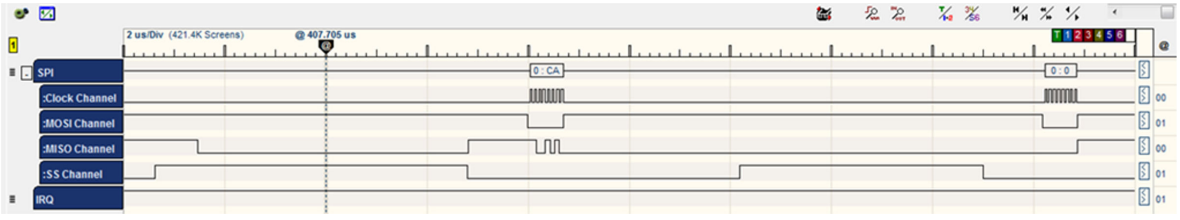
10. The ATWILC* chip sends the value of the register 0x1070 which equals 0x31.



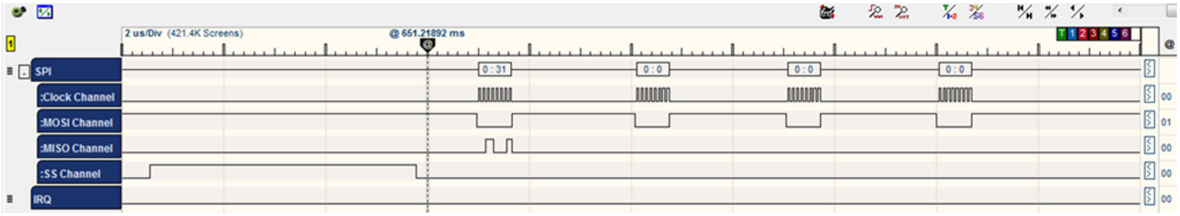

```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read         */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



11. The ATWILC* acknowledges the command by sending three bytes [CA] [0] [F3].

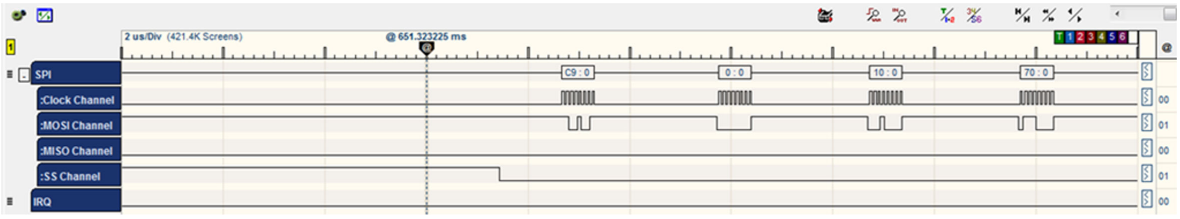


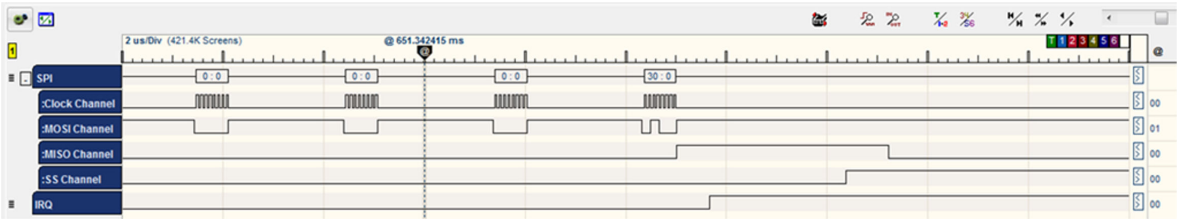
12. The ATWILC* chip sends the value of the register 0x1070 which equals 0x31.



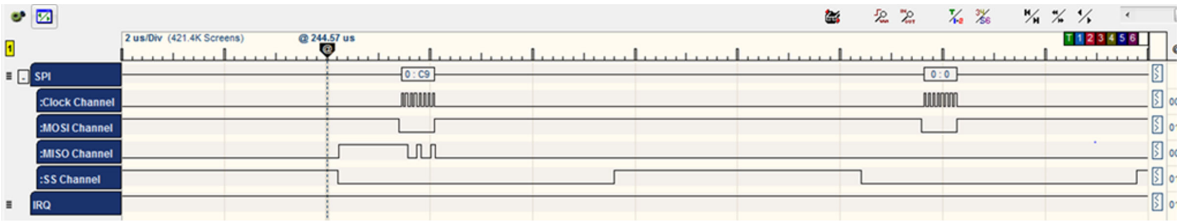
13. Clear the ATWILC* Interrupt.

```
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;
```





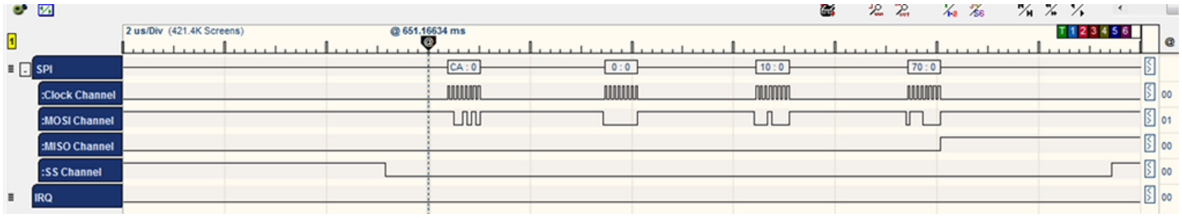
14. The chip acknowledges the command by sending two bytes [C9] [0].



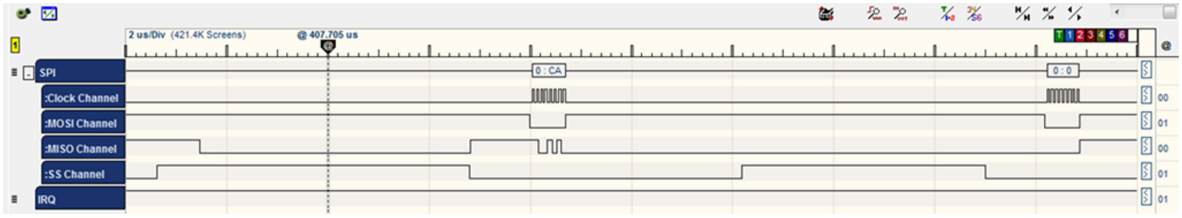
15. The HIF reads the data size.

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

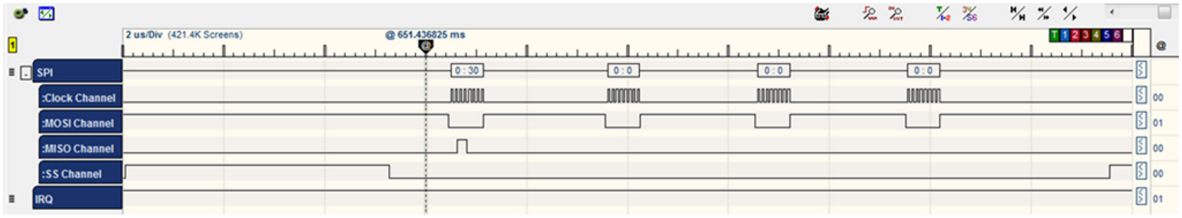
```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read         */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;           /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



16. The ATWILC* acknowledges the command by sending three bytes [CA] [0] [F3].



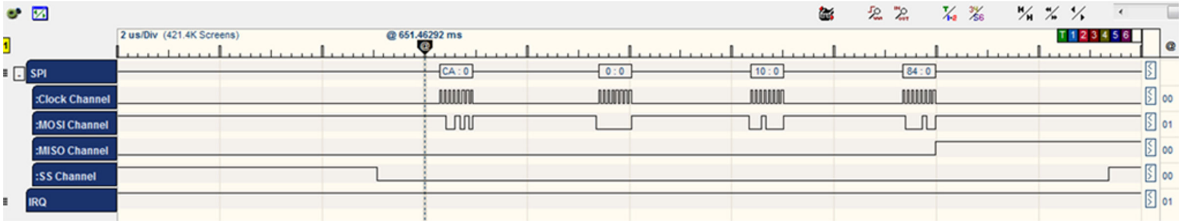
17. The ATWILC* chip sends the value of the register 0x1070 which equals 0x30.



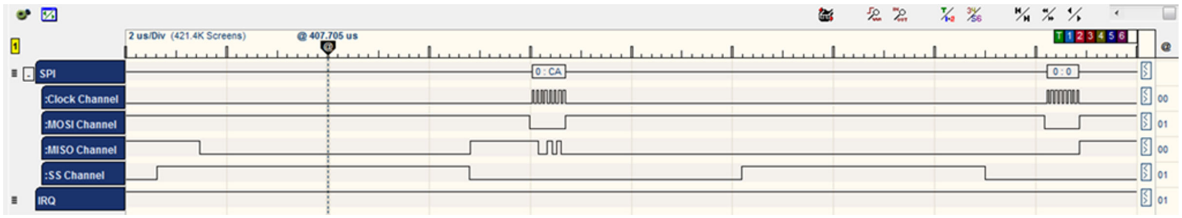
18. The HIF reads hif header address.

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

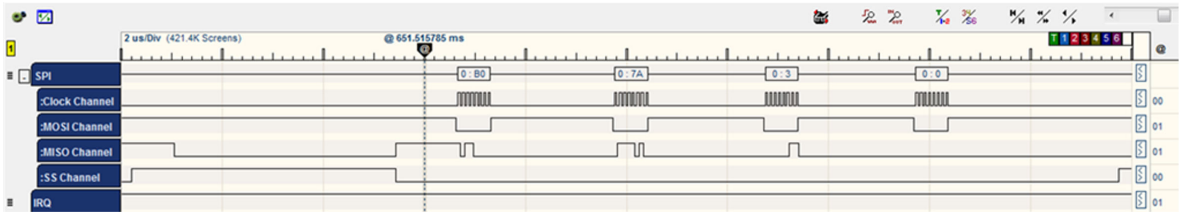
```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_1 address = 0x1084 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



19. The ATWILC* acknowledges the command by sending three bytes [CA] [0] [F3].



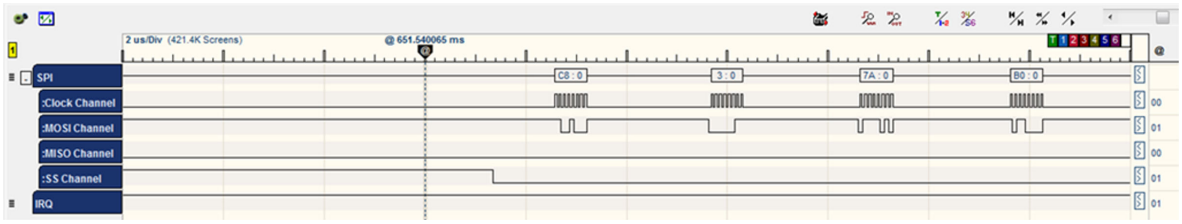
20. The ATWILC* chip sends the value of the register 0x1078 which equals 0x037AB0.

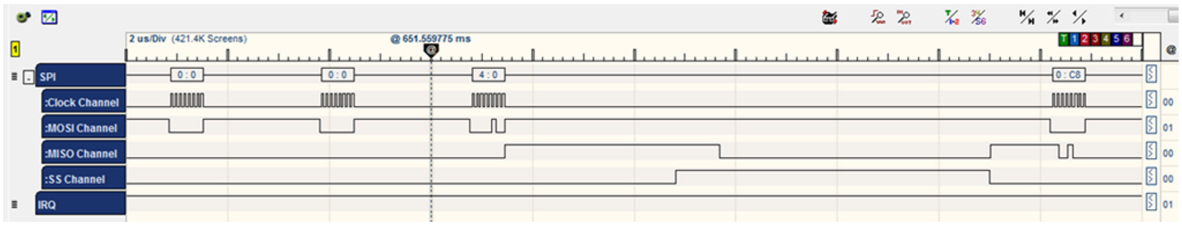


21. The HIF reads the hif header data (as a block).

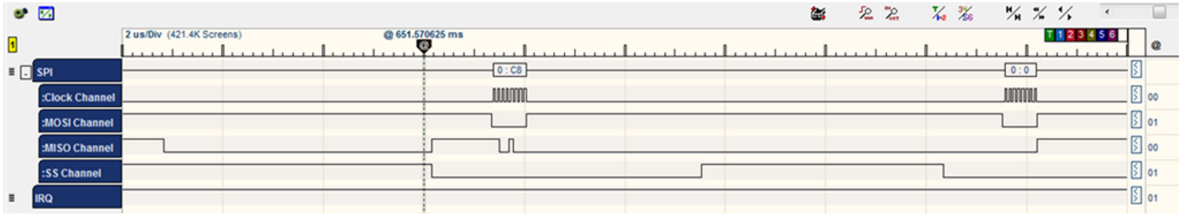
```
ret = nm_read_block(address, (uint8*)&strHif, sizeof(tstrHifHdr));
```

```
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 >>;
        BYTE [6] = size;
```

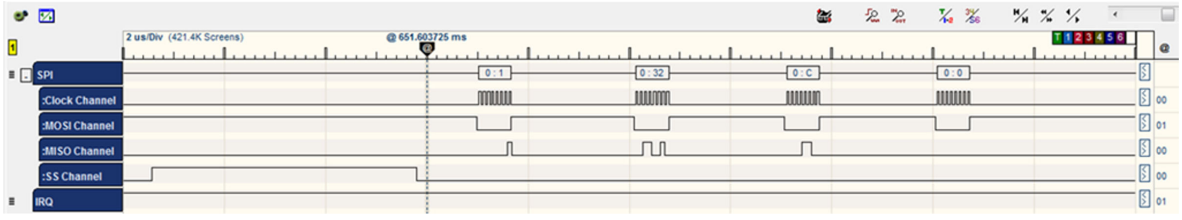




22. The ATWILC* acknowledges the command by sending three bytes [C8] [0] [F3].



23. The ATWILC* sends the data block (four bytes).

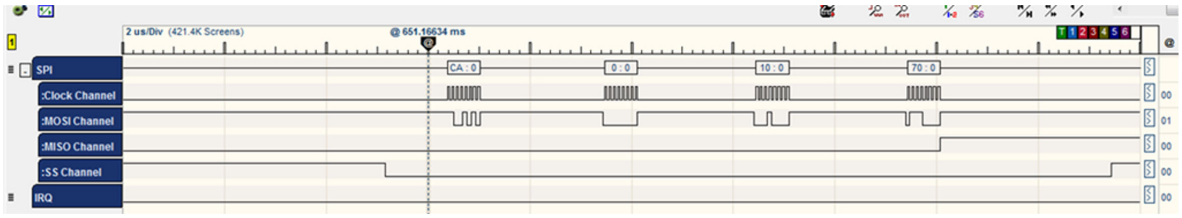


24. The HIF calls the appropriate handler according to the hif header received which tries to receive the Response data payload.

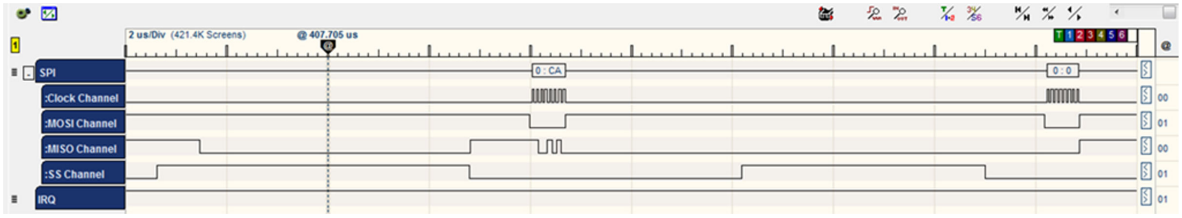
Note: `hif_receive ( )` obtains additional data.

```
sint8 hif_receive(uint32 u32Addr, uint8 *pu8Buf, uint16 u16Sz, uint8 isDone)
{
    uint32 address, reg;
    uint16 size;
    sint8 ret = M2M_SUCCESS;

    ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_0,&reg);
    size = (uint16)((reg >> 2) & 0xfff);
    ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_1,&address);
    /* Receive the payload */
    ret = nm_read_block(u32Addr, pu8Buf, u16Sz);

}
```

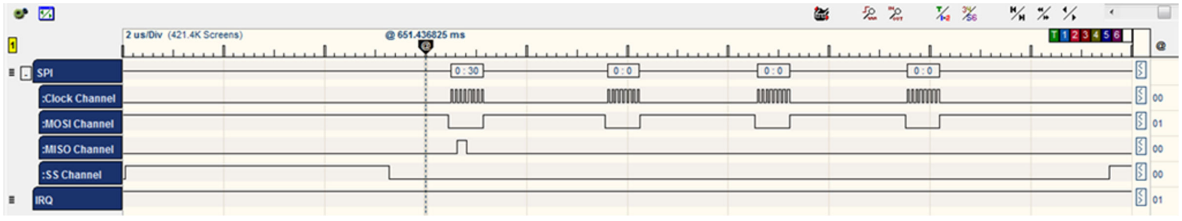
```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read         */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;        /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



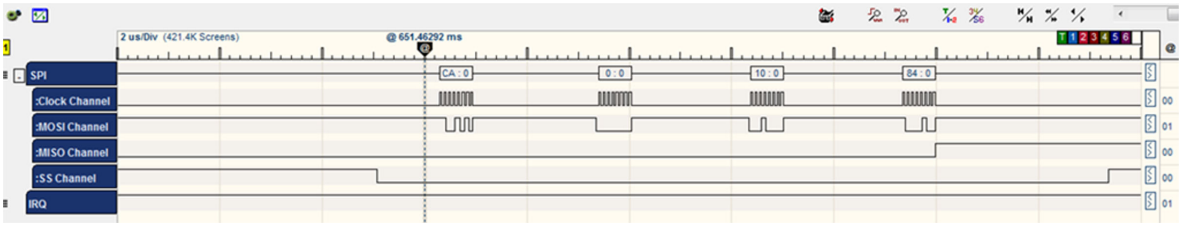
25. The ATWILC* acknowledges the command by sending three bytes [CA] [0] [F3].



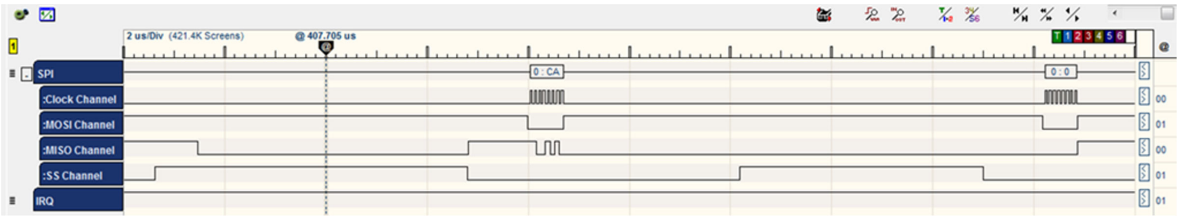
26. The ATWILC* chip sends the value of the register 0x1070 which equals 0x30.



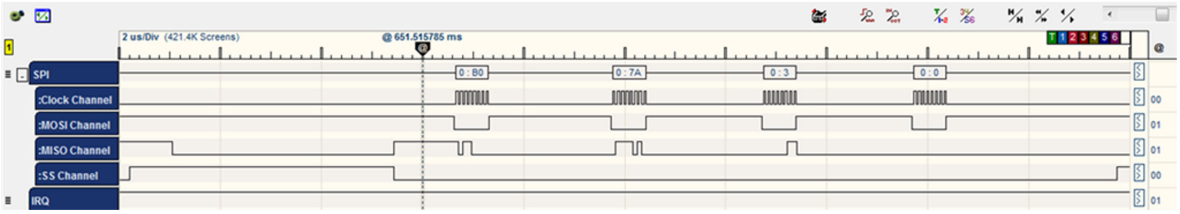

```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read         */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_1 address = 0x1084 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



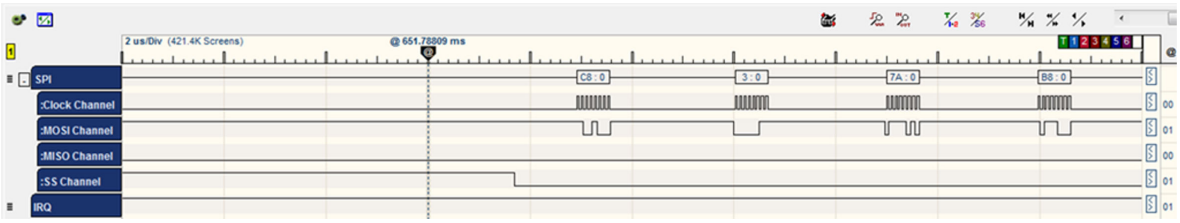
27. The ATWILC* acknowledges the command by sending three bytes [CA] [0] [F3].

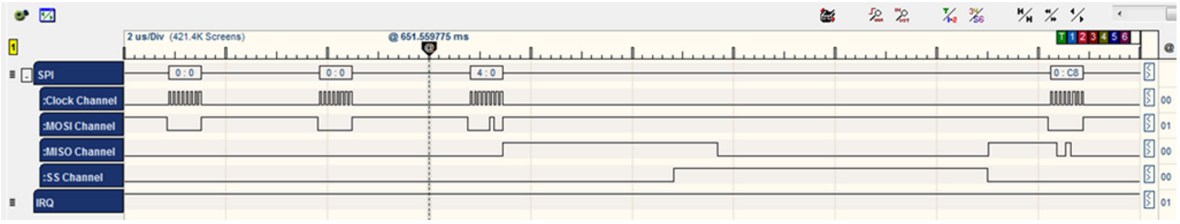


28. The ATWILC* chip sends the value of the register 0x1078 which equals 0x037AB0.

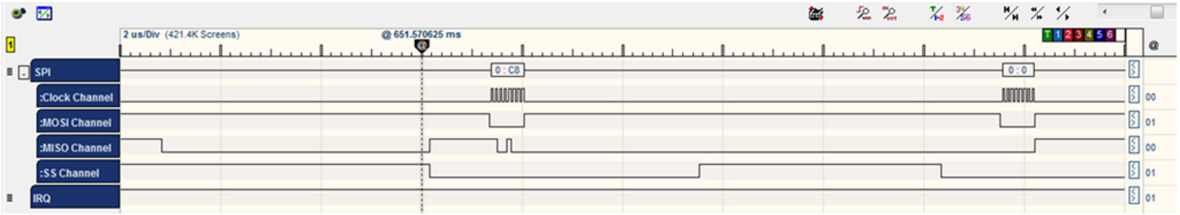



```
Command    CMD_DMA_EXT_READ:    C8          /* dma extended read */
        BYTE [0] = CMD_DMA_EXT_READ
        BYTE [1] = address >> 16;           /* address = 0x037AB8*/
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = size >> 16;
        BYTE [5] = size >>;
    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

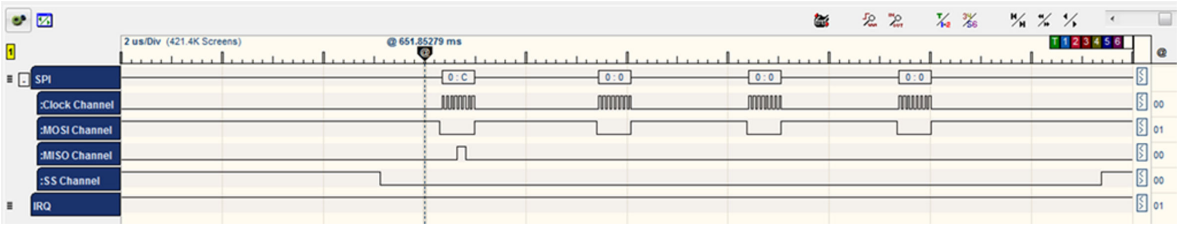




29. The ATWILC* acknowledges the command by sending three bytes [C8] [0] [F3].



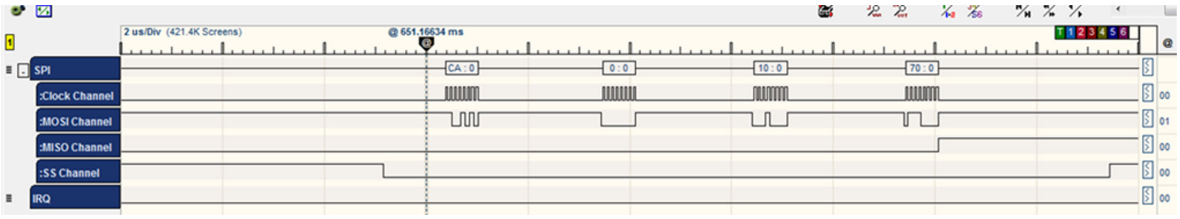
30. The ATWILC* sends the data block (four bytes).



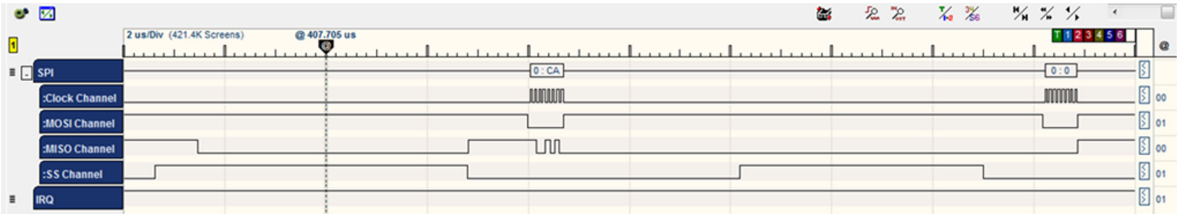
31. After the HIF layer received the response, it interrupts the chip to send the notification that the host RX is done.

```
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);
}
```

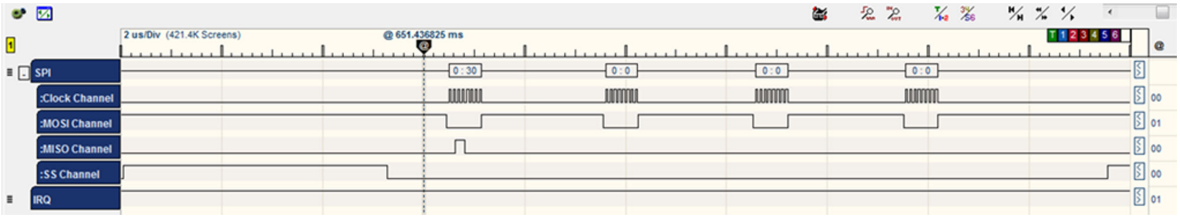
```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read         */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



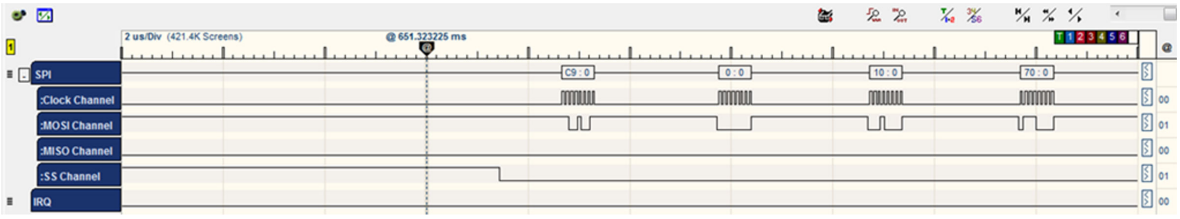
32. The ATWILC* acknowledges the command by sending three bytes [CA] [0] [F3].

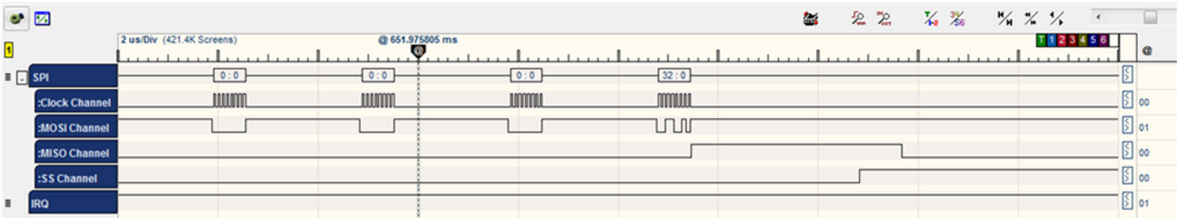


33. The ATWILC* chip sends the value of the register 0x1070 which equals 0x30.

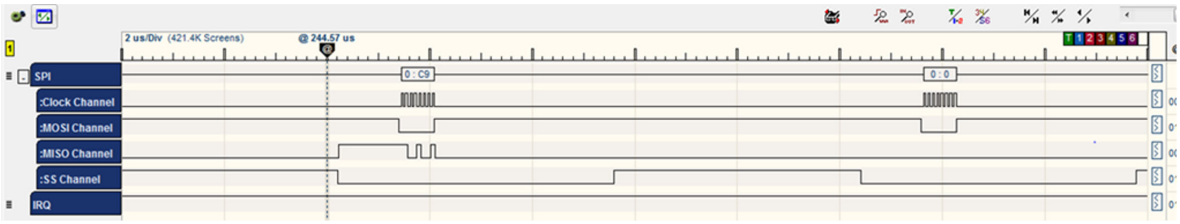



```
Command    CMD_SINGLE_WRITE:0XC9            /* single word write */
        BYTE [0] = CMD_SINGLE_WRITE
        BYTE [1] = address >> 16;        /* WIFI_HOST_RCV_CTRL_0 address = 0x1070  */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = u32data >> 24;        /* Data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





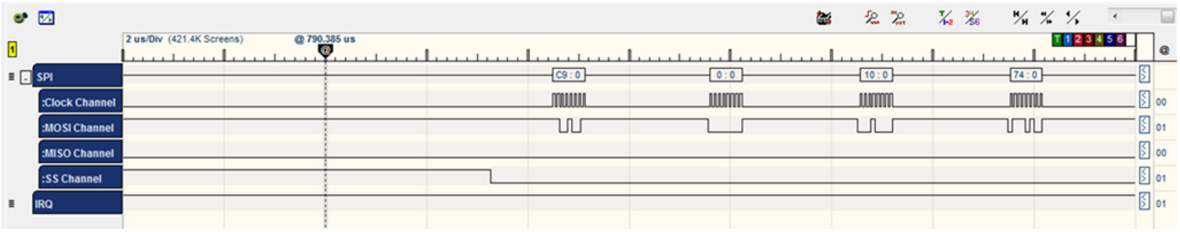
34. The chip acknowledges the command by sending two bytes [C9] [0].

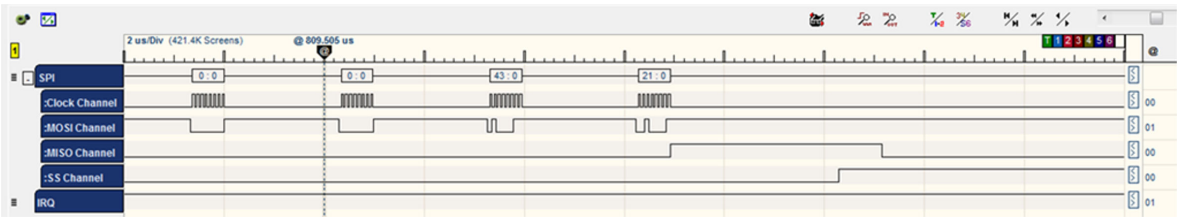


35. The HIF layer allows the chip to enter Sleep mode again.

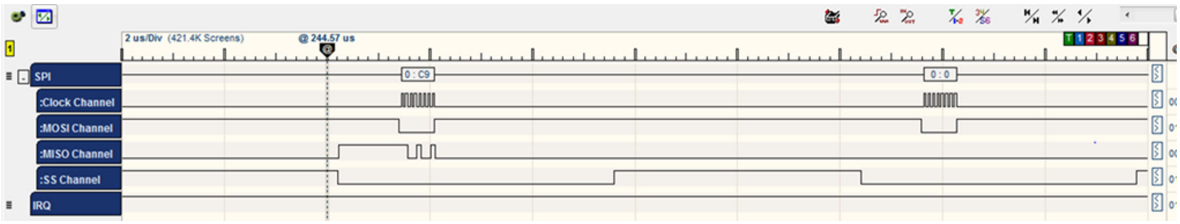
```
sint8 hif_chip_sleep(void)
{
    sint8 ret = M2M_SUCCESS;
    uint32 reg = 0;
    ret = nm_write_reg(WAKE_REG, SLEEP_VALUE);
    /* Clear bit 1 */
    ret = nm_read_reg_with_ret(0x1, &reg);
    if(reg&0x2)
    {
        reg &=~(1 << 1);
        ret = nm_write_reg(0x1, reg);
    }
}
```

```
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;
```



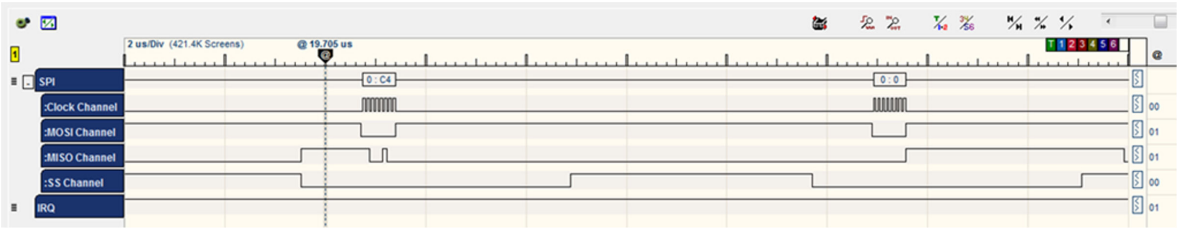


36. The ATWILC* acknowledges the command by sending two bytes [C9] [0].

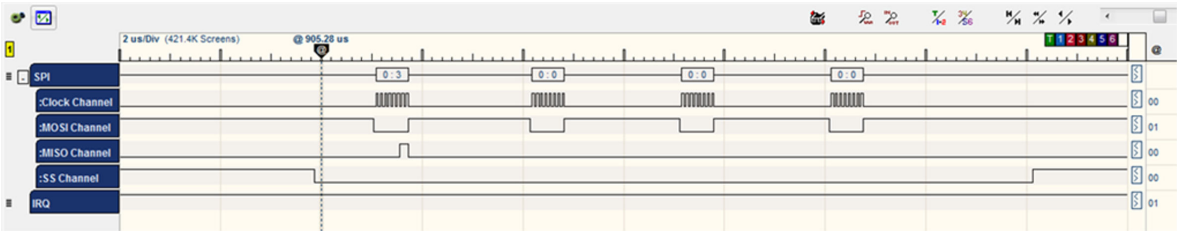



```
Command    CMD_INTERNAL_READ:    0xC4        /* internal register read     */
        BYTE [0] = CMD_INTERNAL_READ
        BYTE [1] = address >> 8;             /* address = 0x01         */
        BYTE [1] |= (1 << 7);                /* clockless register     */
        BYTE [2] = address;
     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

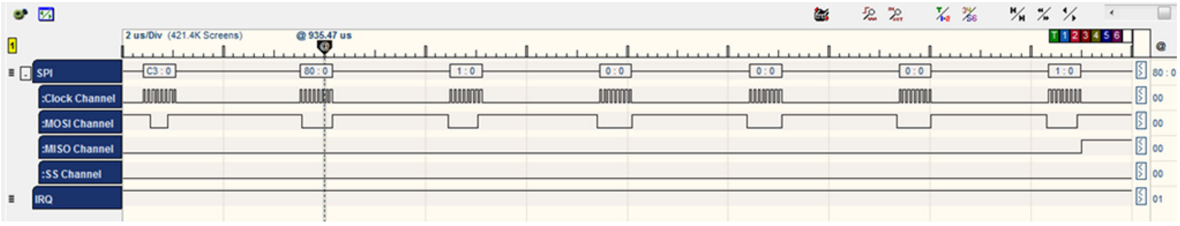
37. The ATWILC* acknowledges the command by sending three bytes [C4] [0] [F3].



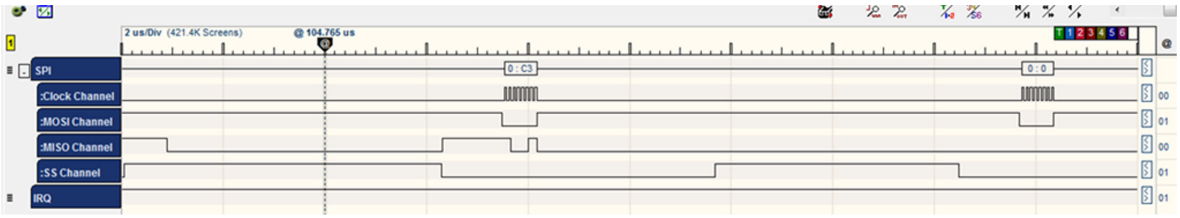
38. Then ATWILC* chip sends the value of the register 0x01 which equals 0x03.




```
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;
```



39. The ATWILC* chip acknowledges the command by sending two bytes [C3] [0].



40. A scan Wi-Fi request is sent to the ATWILC* chip and the response is successfully sent to the host.

12. ATWILC* Firmware Download

The ATWILC* hardware does not have an internal Flash to store the firmware, therefore, the firmware must be stored on the host Flash and download to the ATWILC* RAM after the driver initialization.

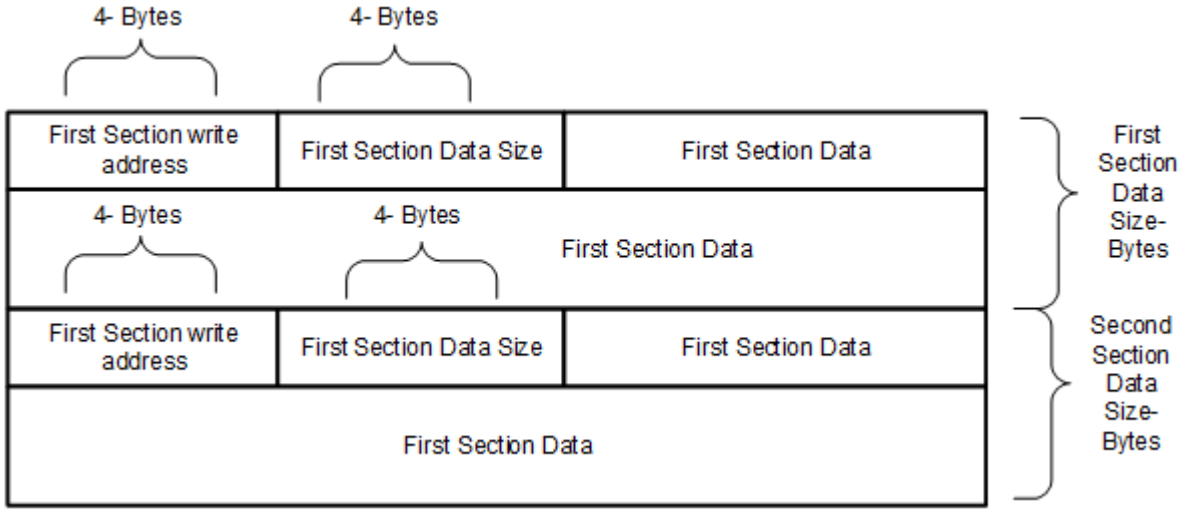
12.1 Wi-Fi Firmware Download

The firmware binary is delivered as a part of the ATWILC* software release at the `firmware.h` file, 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 ATWILC* chip memory
2. 4-Bytes the current section data size
3. Section data with length equals to the section size

Figure 12-1. Wi-Fi Firmware Download



Note: The driver must repeat the previous pattern until the end of the array and must not assume a predefined number of sections, as the number of sections changes from release to release.

Note: On some platforms Flash memory is not directly connected to the DMA engine, in such case the driver must copy the firmware in chunks to the host RAM and write those chunks to the bus interface.

Writing the firmware on the bus must follow the same sequence at Data Packet format.

The following is a code sample to download the firmware with RAM chunks of 32 bytes, this can 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;
}
```

Note: There is only one firmware for all modes of operation of the ATWILC* (Station, AP, and P2P).

12.2 BLE Firmware Download

The BLE firmware binary is delivered as a part of the ATWILC* device software release at the file `ble_firmware.h`, where the firmware binary array is defined as `const char firmware_ble[]`.

The firmware binary is composed of a number of sections back-to-back that must be written on the ATWILC* device IRAM starting at address 0x400000. The downloaded firmware is responsible for copying each section to the corresponding location when it is initializing.

13. Antenna Switching

The ATWILC* device supports antenna diversity where dual antennas are connected to the chip using an external antenna switch. The antenna switches are controlled using two input signals to select the operation of an antenna, and the user can use the following two different configurations with respect to the control GPIOs:

1. Dual GPIO – two different ATWILC* device GPIOs are used to control each of the control lines of the antenna switch.
2. Single GPIO – single ATWILC* device GPIO is used to control one of the control lines of the switch, and its inverse is connected to the other control line. This configuration requires an external inverter.

The antenna selection algorithm evaluates the average RSSI every second, and based on the average the ATWILC* determines if it needs to switch the antenna. The average RSSI is calculated based on the RSSI read while receiving each packet of the data. If the average RSSI is below the threshold value, the ATWILC* switches to the other antenna, and sets a new threshold to the average RSSI of the abandoned antenna. To avoid unnecessary switching, the antenna switching occurs only when the RSSI is below -30 dBm, with a margin of 1 dBm to avoid hysteresis.

`m2m_wifi_set_antenna_mode()` API can be used to configure the ATWILC* device driver for the Antenna Diversity mode, and the GPIOs to control the antenna switch at run time.

13.1 Antenna Switch GPIO Control

The following API is used to configure the Antenna Switch mode, GPIO Control modes, and GPIOs that are connected to the antenna switch.

```
m2m_wifi_set_antenna_mode (antenna_mode, gpio_ctrl_mode, GPIO_NUM#1, GPIO_NUM#2)
```

where,

- `gpio_ctrl_mode` is `ANT_SWTCH_GPIO_SINLGE` or `ANT_SWTCH_GPIO_DUAL`
- `antenna_mode`
  - `ANTENNA1=0`, Manual mode fixed to antenna 1
  - `ANTENNA2=1`, Manual mode fixed to antenna 2
  - `DIVERSITY=2`, Automatic switching based on the antenna performance
- `GPIO_NUM` is any valid GPIO for antenna diversity
  - Valid GPIOs for the ATWILC1000 are 0, 1, 3, 4, and 6.
  - Valid GPIOs for the ATWILC3000 are 3, 4, 6, 17, 18, 19, and 20.

In the Manual modes, the GPIOs are set according to the following table.

Table 13-1. Single Mode

Antenna Selected	GPIO1 Value
Antenna 1	1
Antenna 2	0

In the Dual mode, the GPIOs are set according to the following table.

Table 13-2. Dual Mode

Antenna Selected	GPIO1 Value	GPIO2 Value
Antenna 1	1	0
Antenna 2	0	1

14. Appendix A - API Reference

14.1 WLAN Module

14.1.1 Configuration Switches

Configuration Switch	Definition
COMPUT_PMK_IN_HOST	Enable computing PMK in host rather than in ATWILC*. ATWILC1000 has an HW accelerator that speeds up the computation on the device. ATWILC3000 does not have an HW accelerator, and with this option not defined, PMK will be computed in the FW, which takes up to 5 secs.
M2M_WIFI_RESP_PACKET_SENT	Sends notification to the application when last tx packet is sent, and FW is ready to receive new packets. In this mode, the application should not attempt sending a new packet unless the notification is received.

14.1.2 Defines

Define	Definition	Value
#defineM2M_FIRMWARE_VERSION_MAJOR_NO	Firmware major release version number	4
#define M2M_FIRMWARE_VERSION_MINOR_NO	Firmware minor release version number	3
#define M2M_FIRMWARE_VERSION_PATCH_NO	Firmware patch release version number	0
#define M2M_DRIVER_VERSION_MAJOR_NO	Driver major release version number	4
#define M2M_DRIVER_VERSION_MINOR_NO	Driver minor release version number	3
#define M2M_DRIVER_VERSION_PATCH_NO	Driver patch release version number	0
#define M2M_BUFFER_MAX_SIZE	Maximum size for the shared packet buffer	1600UL- 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	36
#define M2M_ETH_PAD_SIZE	Pad size before the Ethernet header to keep the Ethernet data aligned	2
#define M2M_ETHERNET_HDR_LEN	Length of the Ethernet header in bytes	14

.....continued

Define	Definition	Value
#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_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 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 ATWILC* firmware	2

.....continued

Define	Definition	Value
#define M2M_SCAN_MIN_SLOT_TIME	The min. duration in milliseconds of a scan slots performed by the ATWILC* firmware	(20)
#define M2M_SCAN_ERR_WIFI	Currently not used	((sint8)-2)
#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_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
#define P2P_MAX_PIN_SIZE	The maximum pin size for P2P PIN config method	8
#define CONF_METHOD_LABEL	P2P-WSC Label Display config method	0x0004
#define CONF_METHOD_PBC	P2P-WSC Push button config method	0x0080
#define CONF_METHOD_VIRTUAL_DISPLAY	P2P-WSC Virtual display config method	0x2008
#define CONF_METHOD_PHYSICAL_PBC	P2P-WSC Physical push button config method	0x0480
#define CONF_METHOD_VIRTUAL_PBC	P2P-WSC Virtual push button config method	0x0280
#define CONF_METHOD_KEYPAD	P2P-WSC Key config method	0x0100
#define CONF_METHOD_DISPLAY	P2P-WSC Display config method	0x0008
#define CONF_METHOD_PHYSICAL_DISPLAY	P2P-WSC Physical display config method	0x4008
#define P2P_DEFAULT_PIN	Device Password ID - Default PIN	0x00
#define P2P_USER_SPECIFIED	Device Password ID - User specified	0x01
#define P2P_PUSHBUTTON	Device Password ID - Push button	0x04
#define P2P_REGISTER_SPECIFIED	Device Password ID registrar specified	0x05

14.1.3 Enumeration-Typedef

enum tenuM2mConfigCmd

This enum contains all the host commands used to configure the ATWILC* firmware.

Enumerator Values	Description
M2M_WIFI_REQ_RESTART	Reserved for firmware use not allowed from host driver
M2M_WIFI_REQ_SET_MAC_ADDRESS	Sets the ATWILC* mac address (will overwrite production eFused boards)
M2M_WIFI_REQ_CURRENT_RSSI	Requests 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	Sets the ATWILC* device name property
M2M_WIFI_REQ_CUST_INFO_ELEMENT	Adds custom element to beacon management frame
M2M_WIFI_RESP_FIRMWARE_STRTED	Response message to indicate the firmware successfully started
M2M_WIFI_REQ_SET_TX_POWER	Sets TX Power
M2M_WIFI_REQ_SET_MAX_TX_RATE	Limits the Tx rate to a user-specific rate
M2M_WIFI_REQ_ENABLE_MCAST_FILTER	Requests to enable multi-cast filter
M2M_WIFI_REQ_DISABLE_MCAST_FILTER	Requests to disable multi-cast filter
M2M_WIFI_REQ_SET_ANT_SWITCH_MODE	Sets the Antenna Switch mode
M2M_WIFI_P2P_AUTH_RES	Sets the P2P PIN number and allows connection

enum tenuM2mStaCmd

This enum contains all the ATWILC* commands while in the Station mode.

Enumerator Values	Description
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_CHANGED	Connection state changed response
M2M_WIFI_REQ_SLEEP	Set PS mode command
M2M_WIFI_REQ_SCAN	Request 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_START_WPS	Request WPS start command
M2M_WIFI_REQ_DISABLE_WPS	Request to disable WPS command
M2M_WIFI_REQ_LSN_INT	Set Wi-Fi listen interval

.....continued

Enumerator Values	Description
M2M_WIFI_REQ_SEND_ETHERNET_PACKET	Send Ethernet packet in Bypass mode
M2M_WIFI_RESP_ETHERNET_RX_PACKET	Receive Ethernet packet in Bypass mode
M2M_WIFI_REQ_SET_SCAN_OPTION	Set Scan options: slot time, slot number, and so on
M2M_WIFI_REQ_SET_SCAN_REGION	Set scan region
M2M_WIFI_REQ_SET_MAC_MCAST	Set the ATWILC* multi-cast filters
M2M_WIFI_REQ_SET_P2P_IFC_ID	Set P2P control Interface
M2M_WIFI_RESP_PACKET_SENT	Notification that the last packet sent to the firmware successfully and the firmware is ready to receive the next packet. Requires INT_BASED_TX to be defined
M2M_WIFI_REQ_CERT_ADD_CHUNK	Download one chunk of the certification
M2M_WIFI_REQ_CERT_DOWNLOAD_DONE	Certificate download done
M2M_WIFI_REQ_CHG_MONITORING_CHNL	Change monitoring channel
M2M_WIFI_RESP_ANT_SWITCH_MODE	Antenna Switch mode response
M2M_WIFI_REQ_P2P_AUTH	Application event to allow P2P connection

enum tenuM2mP2pCmd

This enum contains all the ATWILC* commands while in the P2P mode.

Enumerator Values	Description
M2M_WIFI_REQ_ENABLE_P2P	Enable P2P mode command
M2M_WIFI_REQ_DISABLE_P2P	Disable P2P mode command

enum tenuM2mApCmd

This enum contains all the ATWILC* commands while in the AP mode.

Enumerator Values	Description
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 information about the associated stations
M2M_WIFI_RESP_AP_ASSOC_INFO	Response to get information about the associated stations

enum tenuM2mConnState

Wi-Fi Connection State.

Enumerator Values	Description
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	Description
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 tenuM2mWepKeyIndex

Wi-Fi WEP Key Index

Enumerator Values	Description
M2M_WIFI_WEP_KEY_INDEX_1	WEP Key Index (0)
M2M_WIFI_WEP_KEY_INDEX_2	WEP Key Index (1)
M2M_WIFI_WEP_KEY_INDEX_3	WEP Key Index (2)
M2M_WIFI_WEP_KEY_INDEX_4	WEP Key Index (3)

enum tenuM2mWepAuthType

Wi-Fi WEP Authentication type

Enumerator Values	Description
WEP_OPEN_SYSTEM	Open system
WEP_SHARED_KEY	Shared key
WEP_ANY	Both

enum tenuM2mSsidMode

Wi-Fi Supported SSID types

Enumerator Values	Description
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
EUROPE
ASIA

enum tenuPowerSaveModes

Power save modes

Enumerator Values	Description
M2M_NO_PS	Power save is disabled
M2M_PS_DEEP_AUTOMATIC	Power save is done automatically by the ATWILC*. Achieve the highest possible power save.

enum tenuWPSTrigger

WPS triggering methods.

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

enum tenuP2PTrigger

P2P triggering methods.

Enumerator Values	Description
P2P_PIN	P2P is triggered in PIN method
P2P_PBC	P2P is triggered via push button

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. The 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	Description
MANAGEMENT	Wi-Fi Management frame (Probe Req/Resp, Beacon, Association Req/Resp, and so on.)
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). The 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
ACK
CFEND
CFEND_ACK
BLOCKACK_REQ
BLOCKACK
Sub-Types related to Data
DATA
DATA_ACK

.....continued
Enumerator Values
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 tenuP2pControlInterface

P2P Control Interface

Enumerator Values	Description
P2P_STA_CONCURRENCY_INTERFACE	This interface is used for P2P-Station Concurrency mode. Both Station and P2P must be on the same social channels (M2M_WIFI_CH_1, M2M_WIFI_CH_6 or M2M_WIFI_CH_11) in order to create P2P-Station Concurrency mode.
P2P_AP_CONCURRENCY_INTERFACE	This interface is used for P2P-AP Concurrency mode. Both AP and P2P must be on the same social channels (M2M_WIFI_CH_1, M2M_WIFI_CH_6 or M2M_WIFI_CH_11) in order to create P2P-AP Concurrency mode.

enum tenuM2mConnChangedErrcode

Error codes passed structure tstrM2mWifiStateChanged to indicate errors that caused state change.

See also:

- **tpfAppWifiCb**
- **M2M_WIFI_RESP_CON_STATE_CHANGED**
- **tTE_CHANGED tstrM2mWifiStateChanged**

Enumerator Values	Description
M2M_ERR_NONE	Indicates no error
M2M_ERR_AP_NOT_FOUND	Indicates that the ATWILC* didn't find the requested AP
M2M_ERR_AUTH_FAIL	Indicates that the ATWILC* board has failed to authenticate with the AP
M2M_ERR_ASSOC_FAIL	Indicates that the ATWILC* board has failed to associate with the AP
M2M_ERR_LINK_LOSS	Indicates that the AP/STA is out of range
M2M_ERR_SEC_CNTRMSR	Indicates a disconnection because of security countermeasure
M2M_ERR_STATION_IS_LEAVING	Indicates that the station connected to ATWILC1000 is leaving
M2M_ERR_AP_OVERLOAD	Indicates that the maximum number of STAs were connected
M2M_ERR_UNKNOWN_FAIL	Indicates the other generic failures

enum tenuAntSwitchMode

Antenna Selection Modes

Data Field	Definition
ANTENNA1	Used to configure Antenna mode in the ATWILC to select the antenna1
ANTENNA2	Used to configure Antenna mode in the ATWILC to select the antenna2
DIVERSITY	Used to configure Antenna Diversity mode or Dual Antenna mode

See also:

- **tenuAntSelGpio**
- **tstrM2mAntDivParams**

enum tenuAntSelGpio

Antenna Selection Modes

Data Field	Definition
ANT_SWTCH_GPIO_NONE	Antenna GPIO switch is turned OFF
ANT_SWTCH_GPIO_SINLGE	Antenna switch is ON with single GPIO
ANT_SWTCH_GPIO_DUAL	Antenna switch is ON with dual GPIO

See also:

- **tenuAntSwitchMode**

- **tstrM2mAntDivParams**

typedef struct tstrM2mAntDivParams

Structure is used to hold antenna switch mode configurations.

Data Field	Definition
uint8 mode	Antenna Selection Mode
uint8 antenna1	Antenna GPIO controller 1
uint8 antenna2	Antenna GPIO controller 2
uint8 gpio_mode	Antenna GPIO switch mode

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 initialize the Wi-Fi operations, set the structure attributes and pass it as a parameter (part of the Wi-Fi configuration structure **tstrWifiInitParam**) to the **m2m_wifi_init** function.

Data Field	Definition
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**

typedef struct tstrWifiInitParam

Structure, holding the Wi-Fi configuration attributes such as the Wi-Fi callback, Monitoring mode callback and Ethernet parameter initialization structure. Such configuration parameters are required to be set before calling the Wi-Fi initialization function **m2m_wifi_init**.

Enumerator Values	Description
tpfAppWifiCb pfAppWifiCb	Callback for Wi-Fi notifications
tpfAppMonCb pfAppMonCb	Callback for monitor interface
tstrEthInitParam strEthInitParam	Structure to hold Ethernet interface parameters

See also:

- **m2m_wifi_init**

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 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 u8WepAuthType	WEP security authentication type

typedef union tuniM2MWifiAuth

WEP security key parameters for all supported Security modes.

Data Field	Definition
uint8 au8PSK[M2M_MAX_PSK_LEN]	Pre-Shared key of the AP in case of WPA-Personal security
uint8 au8PMK[M2M_MAX_PMK_LEN]	PMK key if it is calculated in the host
tstr1xAuthCredentials strCred1x	Credentials for RADIUS server authentication in case of WPAEnterprise security
tstrM2mWifiWepParams	WEP key parameters in case of WEP security
uint8 u8KeySz	Wep key Size WEP_40_KEY_STRING_SIZE or WEP_104_KEY_STRING_SIZE
uint8 au8WepKey[WEP_104_KEY_STRING_SIZE +1]	Wep key null terminated

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

.....continued	
Data Field	**Definition**
Uint16 u16BeaconInterval	Time between two consecutive beacons in TUs (1024 usec). A value of 0 would use the FW default
uint8 u8ListenChanel	Wi-Fi RF Channel on which the AP will operate
uint8 u8IsPMKUsed	For internal use by the driver, should not be set by the application
uint8 u8SecType	Security type: Open or WEP only in the current implementation
uint8 u8SsidHide	SSID Status "Hidden(1)/Visible(0)"
tuniM2MWifiAuth uniAuth	Union holding all possible authentication parameters corresponding the current security types

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
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.

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

typedef struct tstrM2MPinInfo

P2P Pin number

The PIN number is assigned by application in response to the authorization request in P2P connection.

Data Field	Definition
uint8 au8Pin[P2P_MAX_PIN_SIZE+1]	NULL terminated device name

typedef struct tstrM2MAssocEntryInfo

Holds the association information 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 tstrM2MDataBufCtrl

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 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**

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
uint8 u8DataOffset	Offset of the Ethernet packet inside the receive buffer
uint8 u8IfcId	The logical interface to which the frame belongs as defined in tenuControlInterface

typedef struct tstrM2MMulticastMac

M2M add/remove multi-cast mac address.

Data Field	Definition
uint8 au8macaddress[M2M_MAC_ADDRES_LEN]	Mac address needed to be added or removed from filter

.....continued	
Data Field	**Definition**
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.

See also:

- **tenuP2PTrigger**

Data Field	Definition
uint8 u8ListenChannel	P2P Listen Channel (1, 6, or 11)
tenuP2PTrigger enuTrigger	P2P trigger type
uint16 u16WPS_CfgMethods	P2P config methods
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 au8SSID[M2M_MAX_SSID_LEN]	SSID of the favorite AP. If this AP is found after scan is completed, ATWILC* will connect to it
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 two 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 increase 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 ATWILC* load the mac address from the effuse by default to the ATWILC* configuration memory, but that function is used to let the application overwrite the configuration memory with the mac address from the host.

- 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

.....continued

Data Field	Definition
uint8 au8Mac1[6]	MAC address 2 array

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 au8DeviceName[M2M_DEVICE_NAME_MAX]	P2P Device Name
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
uint8 u8SecType	Wi-Fi network security type. See tenuM2mSecType for supported security types.
uint8 u8IsPMKUsed	Set to true if the PMK is calculated on the host
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**
- **tenuControlInterface**

Data Field	Definition
uint8 u8CurrState	Current Wi-Fi connection state
uint8 u8ErrCode	Error type
int8 u8IfcId	Interface on which state was changed
uint8 u8Ch	Channel number on state change
uint8 u8MAcAddr[6]	Mac Address of the STA/AP involved in state change
uint8 __PAD16__[2]	Padding bytes for forcing 4-byte alignment

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

.....continued	
Data Field	**Definition**
uint8 au8SSID[M2M_MAX_SSID_LEN]	SSID obtained from WPS
uint8 au8PSK[M2M_MAX_PSK_LEN]	PSK for the network obtained from WPS

typedef struct tstrM2MP2pDevInfo

The bitmaps of the WPS config methods shall contain the method opted by a P2P peer device.

See also:

- **tenuP2PTrigger**

Data Field	Definition
uint16 u16CfgMethods	WSC config methods requested by peer methods

14.1.4 Function

1. m2m_wifi_init
   – NMI_API sint8 m2m_wifi_init (tstrWifiInitParam *pWifiInitParam)

   Synchronous initialization function for the ATWILC* driver. This function initializes the driver by registering the callback function for M2M_WIFI layer, and 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 callback 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_ETHERNET_RX_PACKET
   If Monitoring mode is used:
   – M2M_WIFI_RESP_WIFI_RX_PACKET

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

   Parameters:

In	*pWifiInitParam*	This is a pointer to the *tstrWifiInitParam* structure which holds the pointer to the application Wi-Fi layer callback function, Monitoring mode callback and *tstrEthInitParam* structure containing Bypass mode parameters

Precondition:

Prior to this function call, application developers must provide a callback 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.

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

Synchronous de-initialization function to the ATWILC1000 driver which de-initializes 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.

3. m2m_wifi_handle_rx_events
   - NMI_API sint8 m2m_wifi_handle_rx_events()

Asynchronous M2M event handler function, responsible for handling the RX events received from the ATWILC* firmware. Application developers should call this function periodically either in main loop or dedicated task, in order to receive the events that are to be handled by the callback functions implemented by the application.

4. m2m_wifi_handle_tx_events
   - NMI_API sint8 m2m_wifi_handle_tx_events (void * arg)

Asynchronous M2M event handler function, responsible for handling the TX events to the ATWILC* firmware. Application developers should call this function periodically either in main loop or dedicated task, inorder to send the events to the ATWILC* firmware.

Precondition:

Prior to receiving Wi-Fi interrupts, the ATWILC* driver must be successfully initialized by calling the **m2m_wifi_init** function.

Returns:

The function returns **M2M_SUCCESS** for successful RX event dispatch and a negative value otherwise.

See also:

- **m2m_wifi_handle_events**

5. m2m_wifi_handle_tx_events
   - NMI_API sint8 m2m_wifi_handle_tx_events()

   Asynchronous M2M event handler function, responsible for handling the TX events to the ATWILC* firmware. Application developers should call this function periodically either in main loop or dedicated task, in order to send the events to the ATWILC* firmware.

   Precondition:

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

   Returns:

   The function returns **M2M_SUCCESS** for successful TX event dispatch and a negative value otherwise.
6. m2m_wifi_disconnect
   - NMI_API sint8 m2m_wifi_disconnect (void)

   Precondition:

   Disconnection must be made to a successfully connected AP. If the ATWILC* 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**

   Returns:

   The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.
7. m2m_wifi_get_connection_info
   - NMI_API sint8 m2m_wifi_get_connection_info (void)

   Asynchronous connection status retrieval function that 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 start-up. 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 information.

Warning:

Calling this function is valid ONLY in the STA CONNECTED state. Otherwise, the ATWILC* software 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.

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


void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    switch(u8WiFiEvent)
    {
    case M2M_WIFI_RESP_CONN_INFO:
        {
            tstrM2MConnInfo     *pstrConnInfo = (tstrM2MConnInfo*)pvMsg;

            printf("CONNECTED AP INFO\n");
            printf("SSID                : %s\n",pstrConnInfo->acSSID);
            printf("SEC TYPE            : %d\n",pstrConnInfo->u8SecType);
            printf("Signal Strength     : %d\n", pstrConnInfo->s8RSSI);
            printf("Local IP Address    : %d.%d.%d.%d\n",
                pstrConnInfo->au8IPAddr[0] , pstrConnInfo->au8IPAddr[1],
pstrConnInfo->au8IPAddr[2], pstrConnInfo->au8IPAddr[3]);
        }
        break;


    default:
        break;
    }
}

int main()
{
    tstrWifiInitParam   param;
    tuniM2MWifiAuth   sta_auth_param;

    param.pfAppWifiCb   = wifi_event_cb;

    if(!m2m_wifi_init(&param))
    {
```

```
47          // connect to the AP
48          strcpy(sta_auth_param.au8PSK, DEMO_WLAN_PSK);
49          m2m_wifi_connect((char *)DEMO_WLAN_SSID, sizeof(DEMO_WLAN_SSID),
50                  M2M_WIFI_SEC_OPEN, &sta_auth_param, M2M_WIFI_CH_ALL);
51          while(1)
52          {
53              m2m_wifi_handle_rx_events();
54              m2m_wifi_handle_tx_events();
56          }
57      }
58  }
```

8. m2m_wifi_set_mac_address
   – NMI_API sint8 m2m_wifi_set_mac_address (uint8 au8MacAddress0[6] uint8 au8MacAddress1[6])

   Synchronous MAC address assigning to the ATWILC*, used for non-production software. Assign MAC address to the ATWILC* device.

   Parameters:

in	*au8MacAddress0;*	MAC Address to be provisioned to the ATWILC* for STA interface
in	*au8MacAddress1;*	MAC Address to be provisioned to the ATWILC* for AP interface

   Returns:

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

9. m2m_wifi_wps
   – NMI_API sint8 m2m_wifi_wps (uint8 u8TriggerType, const char * pcPinNumber)

   Asynchronous WPS triggering function. This function is called for the ATWILC* 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	*u8TriggerType*	WPS Trigger method. Could be: • **WPS_PIN_TRIGGER Push button method** • **WPS_PBC_TRIGGER Pin method**
In	*pcPinNumber*	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 start-up. 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 ATWILC* 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.

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

void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    switch(u8WiFiEvent)
    {
    case 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");
            }
        }
        break;

    default:
        break;
    }
}

int main()
{
    tstrWifiInitParam   param;

    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_rx_events();
            m2m_wifi_handle_tx_events();
        }
    }
}
```

10. 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.

11. m2m_wifi_p2p
    – NMI_API sint8 m2m_wifi_p2p(uint8 u8Channel, tenuP2PTrigger enuTrigger, uint16 u16WPS_CfgMehods);

Asynchronous Wi-Fi Direct (P2P) Enabling mode function. The ATWILC* supports P2P in device Listening mode ONLY (intent is ZERO). The ATWILC* P2P implementation does not support P2P GO (Group Owner) mode. Active P2P devices (e.g. phones) can find the ATWILC* in the search list.

When the P2P GO Peer device initiates provisional discovery request, a Wi-Fi notification event **M2M_WIFI_REQ_P2P_AUTH** is triggered with config methods opted by the peer device. In response to callback event **M2M_WIFI_REQ_P2P_AUTH**, the Application must send response to ATWILC with either **m2m_wifi_allow_p2p_connection()** API in case of CONF_METHOD_PBC, or m2m_wifi_set_p2p_pin() for CONF_METHOD_DISPLAY and CONF_METHOD_KEYPAD (pin request) to allow P2P connection. However, the application can ignore the event to reject the P2P connection request.

When a device is connected to ATWILC*, a Wi-Fi notification event **M2M_WIFI_RESP_CON_STATE_CHANGED** is triggered.

Parameters:

in	*u8Channel*	P2P Listen RF channel. According to the P2P standard It must hold only one of the following values: 1, 6, or 11.
in	enuTrigger	P2P Trigger method can be the following: • [P2P_PIN] Push button method • [P2P_PBC] Pin method
in	u16WPS_CfgMehods	WSC Config methods supported by P2P device

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_REQ_P2P_AUTH** must be handled in the callback.
– 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**
- **M2M_WIFI_REQ_P2P_AUTH**
- **tstrM2MP2pDevInfo**
- **tenuP2PTrigger**
- **m2m_wifi_set_p2p_pin**
- **m2m_wifi_allow_p2p_connection**

Returns:

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

Example:

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

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

void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    switch(u8WiFiEvent)
    {
    case M2M_WIFI_RESP_CON_STATE_CHANGED:
        {
            tstrM2mWifiStateChanged *pstrWifiState = (tstrM2mWifiStateChanged*)pvMsg;
            M2M_INFO("Wifi State :: %s :: ErrCode %d\n", pstrWifiState->u8CurrState?
"CONNECTED":"DISCONNECTED",pstrWifiState->u8ErrCode);

            // Do something
        }
        break;
    case M2M_WIFI_REQ_P2P_AUTH:
    {
        tstrM2MP2pDevInfo *pstrP2PDevInfo = (tstrM2MP2pDevInfo *)pvMsg;
        if (pstrP2PDevInfo->u16CfgMethods & CONF_METHOD_KEYPAD) {
            osprintf("\r\nPlease enter P2P pin\r\n(Usage: P2P_PIN <pin-number
displayed on phone>\r\n");
        }
        else if (pstrP2PDevInfo->u16CfgMethods & CONF_MEHTOD_DISPLAY){
            osprintf("\r\nPlease enter P2P pin on phone <12345678>\r\n");
                os_m2m_wifi_set_p2p_pin((uint8_t *)"12345678", 8);
        }
        else {
            m2m_wifi_allow_p2p_connection();
        }
        break;
    }
    default:
        break;
    }
}

int main()
{
    tstrWifiInitParam    param;

    param.pfAppWifiCb    = wifi_event_cb;
    if(!m2m_wifi_init(&param))
    {
        // Trigger P2P
        m2m_wifi_p2p(1, P2P_PBC, CONF_METHOD_PHYSICAL_PBC|CONF_METHOD_LABEL);

        while(1)
        {
            m2m_wifi_handle_rx_events();
```

```
49                      m2m_wifi_handle_tx_events();
50              }
51      }
```

12. 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 be enabled and active before disconnect can be called.

    See also:

    - **m2m_wifi_p2p**

    Returns:

    The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.
13. m2m_wifi_set_p2p_pin
    - NMI_API sint8 m2m_wifi_set_p2p_pin(uint8 *pu8PinNumber, uint8 u8PinLength)

    Set P2P PIN number and allow Wi-Fi direct (P2P) connection request.

    Parameters:

in	*pu8PinNumber*	PIN number for P2P PIN method. It must follow the rules stated by the WPS Standard.
in	*u8PinLength*	Length of the PIN number. Should not exceed the maximum P2P_MAX_PIN_SIZE.

    Precondition:

    The p2p mode must be enabled and active.

    See also:

    - **m2m_wifi_p2p**
    - **M2M_WIFI_REQ_P2P_AUTH**
    - **m2m_wifi_allow_p2p_connection**
    - **tenuP2PTrigger**

    Returns:

    The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.
14. m2m_wifi_allow_p2p_connection
    - NMI_API sint8 m2m_wifi_allow_p2p_connection(void)

    Allow P2P connection for P2P_PBC method on request by WILC driver.

    Precondition:

    The p2p mode must be enabled and active.

    See also:

    - **m2m_wifi_p2p**
    - **M2M_WIFI_REQ_P2P_AUTH**

    Returns:

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

15. m2m_wifi_enable_ap
    - NMI_API sint8 m2m_wifi_enable_ap (CONST tstrM2MAPConfig *pstrM2MAPConfig)

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

- Up to eight STA could be associated at a time
- Open and WEP and WPA2 security types are supported

Parameters:

in	*pstrM2MAPConfig*	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 main function of application.

```
 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             printf("STA has Associated\n”);
11         }
12         break;
13
14        default:
15         break;
16     }
17 }
18
19 int main()
20 {
```

```
21      tstrWifiInitParam   param;
22
23      param.pfAppWifiCb   = wifi_event_cb;
24      if(!m2m_wifi_init(&param))
25      {
26          tstrM2MAPConfig     apConfig;
27
28          strcpy(apConfig.au8SSID, "WILC_SSID");
29          strcpy((char *)cfg.uniAuth.au8PSK, "12345678");
30          apConfig.u8ListenChannel    = 1;
31          apConfig.u8SecType          = M2M_WIFI_SEC_WPA_PSK;
32          apConfig.u8SsidHide         = SSID_MODE_VISIBLE;
33
34          // Trigger AP
35          m2m_wifi_enable_ap(&apConfig);
36
37          while(1)
38          {
39              m2m_wifi_handle_rx_events();
46              m2m_wifi_handle_tx_events();
47          }
48       }
49
```

16. 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 that 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 start-up. 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 ATWILC* software ignores the request.

See also:

- **tpfAppWifiCb**
- **m2m_wifi_init**
- **M2M_WIFI_RESP_AP_ASSOC_INFO**
- **tstrM2MAPAssocInfo**

Returns:

The function returns **M2M_SUCCES**S 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->u8NoConnSta);
                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],pstrAssocInfo-
>astrM2MAssocEntryInfo[i].BSSID[2],
                    pstrAssocInfo->astrM2MAssocEntryInfo[i].BSSID[3],pstrAssocInfo-
>astrM2MAssocEntryInfo[i].BSSID[4],
                    pstrAssocInfo->astrM2MAssocEntryInfo[i].BSSID[5],pstrAssocInfo-
>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,"WILC_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_rx_events();
                                  m2m_wifi_handle_tx_events();
          }
     }
 }
```

17. 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	*u8NumOfSlot;*	The minimum number of slots is two for every channel. For every slot the SoC will send Probe Req on air, and wait/listen for PROBE RESP/BEACONS for the u8slotTime in ms.
in	*u8SlotTime;*	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.

18. 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.

Parameters:

in	*ScanRegion;*	Scan region as defined in tenuM2mScanRegion

See also:

– **tenuM2mScanCh**
– **m2m_wifi_request_scan**
– **tenuM2mScanRegion**

Returns:

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

19. 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 tenuM2mScanCh. With a value of M2M_WIFI_CH_ALL(255)), means to scan all channels.

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**
- **m2m_wifi_request_scan_ssid**

Returns:

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

Example:

The code snippet is 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                 }
```

```
                else
                {
                    printf("No AP Found Rescan\n");
                    m2m_wifi_request_scan(M2M_WIFI_CH_ALL);
                }
            }
            else
            {
                printf("(ERR) Scan fail with error <%d>\n",pstrInfo->s8ScanState);
            }
        }
        break;

    case M2M_WIFI_RESP_SCAN_RESULT:
        {
            tstrM2mWifiscanResult       *pstrScanResult
=(tstrM2mWifiscanResult*)pvMsg;
            uint8                       u8NumFoundAPs =
m2m_wifi_get_num_ap_found();

            printf(">>%02d RI %d SEC %s CH %02d BSSID %02X:%02X:%02X:%02X:%02X:
%02X SSID %s\n",
                pstrScanResult->u8index,pstrScanResult->s8rssi,
                pstrScanResult->u8AuthType,
                pstrScanResult->u8ch,
                pstrScanResult->au8BSSID[0], pstrScanResult->au8BSSID[1],
pstrScanResult->au8BSSID[2],
                pstrScanResult->au8BSSID[3], pstrScanResult->au8BSSID[4],
pstrScanResult->au8BSSID[5],
                pstrScanResult->au8SSID);

            if(u8ScanResultIdx < u8NumFoundAPs)
            {
                // Read the next scan result
                m2m_wifi_req_scan_result(index);
                u8ScanResultIdx ++;
            }
        }
        break;
    default:
        break;
    }
}

int main()
{
    tstrWifiInitParam   param;

    param.pfAppWifiCb   = wifi_event_cb;

    if(!m2m_wifi_init(&param))
    {
        // Scan all channels
        m2m_wifi_request_scan(M2M_WIFI_CH_ALL);

        while(1)
        {
            m2m_wifi_handle_rx_events();
            m2m_wifi_handle_tx_events();
        }
    }
}
```

20. m2m_wifi_request_scan_ssid
    - NMI_API sint8 m2m_wifi_request_scan_ssid (uint8 ch, char*pcssid)

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

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 event **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.

21. 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 in STA mode only.

Precondition:

- m2m_wifi_request_scan needs 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:

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

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

void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    static uint8    u8ScanResultIdx = 0;

    switch(u8WiFiEvent)
    {
    case M2M_WIFI_RESP_SCAN_DONE:
        {
            tstrM2mScanDone *pstrInfo = (tstrM2mScanDone*)pvMsg;

            printf("Num of AP found %d\n",pstrInfo->u8NumofCh);
            if(pstrInfo->s8ScanState == M2M_SUCCESS)
            {
                u8ScanResultIdx = 0;
                if(pstrInfo->u8NumofCh >= 1)
                {
                    m2m_wifi_req_scan_result(u8ScanResultIdx);
                    u8ScanResultIdx ++;
                }
                else
                {
                    printf("No AP Found Rescan\n");
                    m2m_wifi_request_scan(M2M_WIFI_CH_ALL);
                }
            }
            else
            {
                printf("(ERR) Scan fail with error <%d>\n",pstrInfo->s8ScanState);
            }
        }
        break;

    case M2M_WIFI_RESP_SCAN_RESULT:
        {
            tstrM2mWifiscanResult       *pstrScanResult
=(tstrM2mWifiscanResult*)pvMsg;
            uint8                       u8NumFoundAPs =
m2m_wifi_get_num_ap_found();

            printf(">>%02d RI %d SEC %s CH %02d BSSID %02X:%02X:%02X:%02X:%02X:
%02X SSID %s\n",
                pstrScanResult->u8index,pstrScanResult->s8rssi,
                pstrScanResult->u8AuthType,
                pstrScanResult->u8ch,
                pstrScanResult->au8BSSID[0], pstrScanResult->au8BSSID[1],
pstrScanResult->au8BSSID[2],
                pstrScanResult->au8BSSID[3], pstrScanResult->au8BSSID[4],
pstrScanResult->au8BSSID[5],
                pstrScanResult->au8SSID);

            if(u8ScanResultIdx < u8NumFoundAPs)
            {
                // Read the next scan result
                m2m_wifi_req_scan_result(index);
                u8ScanResultIdx ++;
            }
        }
        break;
    default:
        break;
    }
}

int main()
{
    tstrWifiInitParam   param;

    param.pfAppWifiCb   = wifi_event_cb;
```

```
73      if(!m2m_wifi_init(&param))
74      {
75          // Scan all channels
76          m2m_wifi_request_scan(M2M_WIFI_CH_ALL);
77
78          while(1)
79          {
80              m2m_wifi_handle_rx_events();
80              m2m_wifi_handle_tx_events();

82          }
83      }
84  }
```

22. 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 starts 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 start-up. 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:

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.

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;
72     if(!m2m_wifi_init(&param))
73     {
74         // Scan all channels
75         m2m_wifi_request_scan(M2M_WIFI_CH_ALL);
76
77         while(1)
78         {
79             m2m_wifi_handle_rx_events();
```

```
80              m2m_wifi_handle_tx_events();
81          }
82      }
83}
```

23. 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 is 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;
31
32     if(!m2m_wifi_init(&param))
33     {
34         // Scan all channels
35         m2m_wifi_req_curr_rssi();
36
37         while(1)
38         {
39             m2m_wifi_handle_rx_events(NULL);
40             m2m_wifi_handle_tx_events();

42         }
43     }
44 }
```

24. 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	*pu8MacAddr*	Output MAC address buffer of 6 bytes size. Valid only if *pu8Valid=1.
out	*pu8IsValid*	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.

25. m2m_wifi_get_mac_address
    – NMI_API sint8 m2m_wifi_get_mac_address (uint8 *pu8MacAddr0, uint8 *pu8MacAddr1)

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

Parameters:

out	*pu8MacAddr0*	Output MAC address buffer of 6 bytes size for AP interface
out	*pu8MacAddr1*	Output MAC address buffer of 6 bytes size for STA interface

Precondition:

m2m_wifi_init is required to be called before any Wi-Fi function.

See also:

– **m2m_wifi_get_otp_mac_address**

Returns:

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

26. 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 ATWILC*.

Parameters:

in	*PsTyp*	Desired Power-Saving mode. Supported types are defined in **tenuPowerSaveModes**.
in	*BcastEn*	Broadcast Reception Enable flag. If it is 1, the ATWILC* must be awake each DTIM beacon for receiving broadcast traffic. If it is 0, the ATWILC* will not wake up at the DTIM beacon, but its wake up 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.

27. m2m_wifi_request_sleep
    - NMI_API sint8 m2m_wifi_request_sleep (uint32 u32SlpReqTime)

Synchronous power-save request function, which requests from the ATWILC* 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.

28. m2m_wifi_get_sleep_mode
    - NMI_API uint8 m2m_wifi_get_sleep_mode (void)

See also:

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

Returns:

The current operating Power-Saving mode.

29. m2m_wifi_set_device_name
    - NMI_API sint8 m2m_wifi_set_device_name (uint8 *pu8DeviceName, uint8 u8DeviceNameLength)

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

Parameters:

In	*pu8DeviceName*	Buffer holding the device name
In	*u8DeviceNameLength*	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.

30. 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.

Parameters:

In	*pstrM2mLsnInt*	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.

31. 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 must be verified by the application developer.

Parameters:

In	*pu8Packet*	Pointer to a buffer holding the whole Ethernet frame at an offset of M2M_ETHERNET_HDR_OFFSET + M2M_ETH_PAD_SIZE bytes. This buffer has to be aligned appropriately for DMA operations according to the host's constraints (word aligned, cache line aligned, and so on).
In	*u16PacketSize*	The size of the Ethernet packet, not including the M2M Header (M2M_ETHERNET_HDR_OFFSET + M2M_ETH_PAD_SIZE)
In	*U8IfcId*	The interface selected to send Ethernet packet (AP_INTERFACE, STATION_INTERFACE or P2P_INTERFACE)

Note:

Packets are the user's responsibility.

Returns:

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

32. 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	*pau8M2mCustInfoElement*	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 bytes.

    - Used in Access Point mode

    See also:

    - **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';
```

```
elementData[7]='C';
11 //Third IE
12 elementData[8]=202; elementData[9]=3; elementData[10]='D';
elementData[11]=0; elementData[12]='F';
13 } 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);
```

33. 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	*pu8MulticastMacAddress*	Pointer to MAC address
in	*u8AddRemove*	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 eight.

    Returns:

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

34. m2m_wifi_enable_mcast_filter
    – NMI_API sint8 m2m_wifi_enable_mcast_filter(void)

    Enable multicast filtering. If multicast filter is enabled, only multicast packets sent to the multicast groups set using m2m_wifi_enable_mac_mcast will be passed to the host.

    Parameters:

    None

    Warning:

    Returns:

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

35. m2m_wifi_disable_mcast_filter

– NMI_API sint8 m2m_wifi_disable_mcast_filter(void)

Disable multicast filtering. If multicast filter is disabled, all multicast packets received will be passed to the host.

Parameters:

None

Warning:

Returns:

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

36. m2m_wifi_set_p2p_control_ifc
    – NMI_API sint8 m2m_wifi_set_p2p_control_ifc(uint8 u8IfcId)

Synchronous function to set on which interface should be used for P2P mode.

Parameters:

In	*u8IfcId*	Interface ID as defined in tenuP2pControlInterface

Returns:

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

37. m2m_wifi_download_cert
    – sint8 m2m_wifi_download_cert(uint8* pCertData,uint32 u32CertSize)

Download WPA Enterprise certificate to be used for authentication.

Parameters:

In	*pCertData*	Pointer to the certificate data
In	*u32CertSize*	Size of the certificate

Returns:

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

38. m2m_wifi_set_tx_power
    – NMI_API sint8 m2m_wifi_set_tx_power(uint8 u8TxPwrLevel)

Set the TX power level.

Parameters:

In	*u8TxPwrLevel*	Set TX power ppa to any one of the following values: 0 dBm, 3 dBm, 6 dBm, 9 dBm, 12 dBm, 15 dBm, or 18 dBm. Any other values set between the above mentioned values are floored to the nearest supported value. A maximum level is enforced by the firmware to comply to RF regulations.

Returns:

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

39. m2m_wifi_set_max_tx_rate
    - NMI_API sint8 m2m_wifi_set_max_tx_rate(tenuTxDataRate enuMaxTxDataRate)

Synchronous function for setting the max data rate to be used for transmission.

Parameters:

In	*enuMaxTxDataRate*	Maximum Tx data rate to be used while transmitting

See also:

- **tenuTxDataRate**

Returns:

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

40. m2m_wifi_set_antenna_mode
    - NMI_API sint8 m2m_wifi_set_antenna_mode(uint8 ant_mode, uint8 gpio_mode, uint8 ant_gpio1, uint8 ant_gpio2)

Set the antenna selection and GPIO Switch mode.

Parameters:

In	*ant_mode*	Antenna Selection Mode
In	*ant_gpio1*	Antenna GPIO controller 1
In	*ant_gpio2*	Antenna GPIO controller 2
In	*gpio_mode*	Antenna GPIO Switch mode - Single=1, Dual=2 or None=0

See also:

- **tenuAntSelGpio**

Returns:

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

14.2 BSP

This module contains the ATWILC BSP API declarations.

14.2.1 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 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**.

14.2.2 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

14.2.3 Functions

- 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 of this function 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 of this function 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	**u32TimeMsec**	**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 (tpfNmBspIsr pfIsr)

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	pfIsr	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:

- **tpfNmBspIsr**

Returns:

None

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

Synchronous enable/disable the MCU interrupts.

Parameters:

in	u8Enable	'0' disable interrupts. '1' enable interrupts

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

See also:

- tpfNmBspIsr

Returns:

None

- void nm_bsp_os_lock(void)
  void nm_bsp_os_lock(void)

Synchronous Enter Critical Section

Note:

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

See also:

nm_bsp_os_unlock

Returns:

None

- void nm_bsp_os_unlock(void)
  void nm_bsp_os_unlock(void)

Synchronous Exit Critical Section

Note:

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

See also:

nm_bsp_os_lock

Returns:

None

- void nm_bsp_os_disable_intr (void)
  void nm_bsp_os_disable_intr(void)

Synchronous disable the MCU interrupts

Note:

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

See also:

nm_bsp_os_enable_intr

Returns:

None

- void nm_bsp_os_enable_intr (void)
  void nm_bsp_os_enable_intr(void)

Synchronous enable the MCU interrupts

Note:

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

See also:

nm_bsp_os_disable_intr

Returns:

None

14.2.4 Enumeration/Typedef

14.2.4.1 Asynchronous Events

Specific enumeration used for asynchronous operations.

14.3 Bus Wrapper

This module contains ATWILC* Bus wrapper API declarations. Bus wrapper APIs are platform specific, and each platform must be implemented accordingly.

14.3.1 Defines

None.

14.3.2 Data Types

Define	Definition	Value
#define NM_BUS_TYPE_I2C	I^2C Bus Type	((uint8)0)
#define NM_BUS_TYPE_SPI	SPI Bus Type	((uint8)1)
#define NM_BUS_TYPE_UART	UART Bus Type	((uint8)2)
#define NM_BUS_TYPE_SDIO	SDIO Bus Type	((uint8)3)
#define NM_BUS_IOCTL_R	I^2C IOCTL Command read. Parameter- tstrNmI2cDefault/ tstrNm UartDefault	((uint8)0)
#define NM_BUS_IOCTL_W	I^2C IOCTL Command write. Parameter- tstrNmI2cDefault/ tstrNm UartDefault	((uint8)1)
#define NM_BUS_IOCTL_W_SPECIAL	I^2C IOCTL Command to write two buffers within the same transaction. Parameter-tstrNmI2cSpe cial	((uint8)2)
#define NM_BUS_IOCTL_RW	I^2C IOCTL Command to read and write in the same transaction. Parameter-tstrNmSpiRw	((uint8)3)
#define NM_BUS_IOCTL_WR_RESTAR T	I^2C IOCTL Command to write buffer and restart condition then read. Parameter-tstrNmI2cSpe cial	((uint8)4)
#define NM_BUS_IOCTL_CMD_52	Issue SDIO Command 52. Parameter-tstrNmSdioC md52	((uint8)5)
#define NM_BUS_IOCTL_CMD_53	Issue SDIO Command 53. Parameter-tstrNmSdioC md53	((uint8)6)

14.3.3 Enumeration/Typedef

typedef struct tstrNmSdioCmd52

The structure to hold SDIO command 52's parameters are:

Data Field	Definition
uint32_t read_write	R/W operation: 0 to read, 1 to write

.....continued	
Data Field	**Definition**
uint32_t function	SDIO function
uint32_t raw	In write operation, set raw to 1 to read the response after writing the data
uint32_t address	Address for command 52
uint32_t data	Data for command 52

typedef struct tstrNmSdioCmd53

Structure to hold SDIO command 53's parameters are:

Data Field	Definition
uint32_t read_write:1	R/W operation: 0 to read, 1 to write
uint32_t function:3	SDIO function
uint32_t block_mode:1	In write operation, set raw to 1 to read the response after writing the data
uint32_t increment:1	-
uint32_t address:17	Address for command 53
uint32_t count:9	If block_mode = 1, count is the number of blocks, otherwise it is the number of bytes.
uint32_t buffer	Data for command 53
uint32_t block_size	-

14.3.4 Function

1. nm_bus_init
   – sint8 nm_bus_init(void *)
   Initialize the bus wrapper

   Parameters

in	*pvInitVal*	Pointer to private data to be passed to the platform specific initialization

   Returns

   The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.
2. nm_bus_ioctl
   – sint8 nm_bus_ioctl(uint8 u8Cmd, void* pvParameter)
   Send/receive from the bus

   Parameters

in	*u8Cmd*	IOCTL command for the operation
in	*pvParameter*	Arbitrary parameter depending on IOCTL

Note: For SPI, it is important to send/receive at the same time.

Returns

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

3. nm_bus_deinit
   - sint8 nm_bus_deinit(void)
     De-initialize the bus wrapper

Parameters

None

Returns

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

4. nm_bus_reinit
   - sint8 nm_bus_reinit(void* config)
     De-initialize the bus wrapper

Parameters

in	*config*	Re-initialize the configuration data

Returns

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

15. Appendix B - BT HCI Interface

15.1 Standard HCI Commands

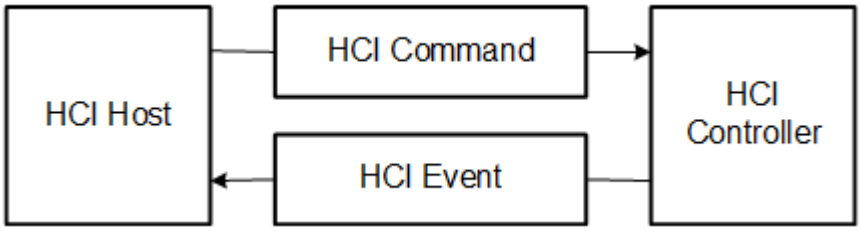
The ATWILC* devices that are BT capable use standard HCI commands to communicate between the host and BT controller. The HCI commands are defined in the Bluetooth Specification.

15.1.1 Introduction

There are four kinds of HCI packets that can be sent via the UART transport layer:

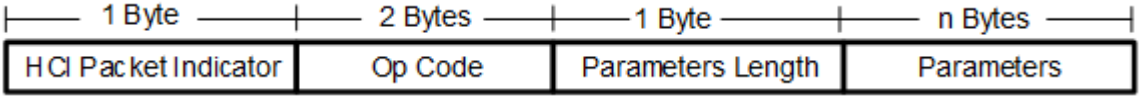
- HCI Command Packet
- HCI Event Packet
- HCI ACL Data Packet
- HCI Synchronous Data Packet

Vendor-specific commands utilize the HCI Command packet and the HCI Event packet only as per the following block diagram.



15.1.2 HCI Command Packet

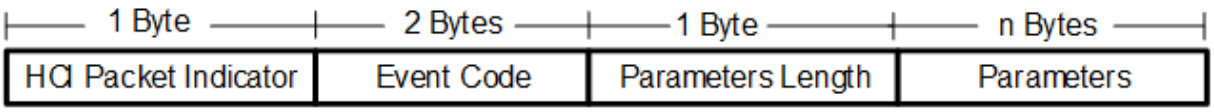
The HCI command packet is used to send commands to the controller from the host. It has the following structure:



HCI packet indicator	Used to differentiate among the four HCI packet types. The value “1” represents “HCI command packet”.
OP code	Used to determine the OP code of the HCI command.
Parameters length	Length of the command required for parameters, in bytes.
Parameters	The required parameters for the HCI command.

15.1.3 HCI Event Packet

The HCI event packet is used by the controller to notify the host when events occur. The host must accept the HCI event packets up to 255 bytes of data, excluding the HCI event packet header.



HCI packet indicator	This field is used to differentiate among the four HCI packet types. The value “4” represents “HCI event packet”.
OP code	Each event is assigned a 1-byte event code used to uniquely identify different types of events.
Parameters length	Length of the event parameters, in bytes.
Parameters	The required parameters for the HCI event.

15.2 Vendor Specific HCI Commands

Some non-standard HCI commands can be used by the host to get extended services and options from the ATWILC* device’s BT core.

15.2.1 Update UART Parameters Command

If the host requires a change to the UART settings of the BT controller, use this command to change the baud rate and flow control options.

HCI Packet Indicator	Op Code	Parameter Length	Parameters Payload	
1	0xFC53	5	Baud rate (4 bytes)	Flow control (1 byte)

15.2.2 Change BD Address

If the BD address is not stored on the BT controller’s side, the host can use this command to update the BD address.

HCI Packet Indicator	Op Code	Parameter Length	Parameters Payload
1	0xFC54	6	BD address (6 bytes)

Note: After issuing this command, a standard Reset Command (Op Code 0x0c03) should be issued for the new address to apply

15.2.3 Write Memory

This command is used when the boot ROM is running on the host controller only to write a block of memory to the BT controller. The main function is to download the firmware to the controller.

HCI Packet Indicator	Op Code	Parameter Length	Parameters Payload		Data Block
1	0xFC52	8	Address (6 bytes)	Size (4 bytes)	Data block to write to the memory

Note: After issuing this command, a vendor-specific reset command (Op Code: 0xFC55) must be issued for the new firmware to execute. This option cannot be used to write blocks of memory until the boot ROM executes again, which happens when a chip reset is done.

15.2.4 Vendor Specific Reset

This command is used when the boot ROM is running on the host controller only. It must be issued after a new firmware is downloaded to the BT controller memory to start it

HCI Packet Indicator	Op Code	Parameter Length	Parameters Payload
1	0xFC55	0	N/A

15.2.5 Read Register

This command is used to read registers from the BT controller using UART.

HCI Packet Indicator	Op Code	Parameter Length	Parameters Payload		
1	0xFC01	6	Register Address (4 bytes)	0x20	0x01

15.2.6 Set TX Power

This command is used to set the transmit power with specific level.

HCI Packet Indicator	Op Code	Parameter Length	Parameters Payload	
1	0xFC3B	3	Reserved (2 bytes)	TX Level (1 byte)

The TX level can be set to any one of the following values:

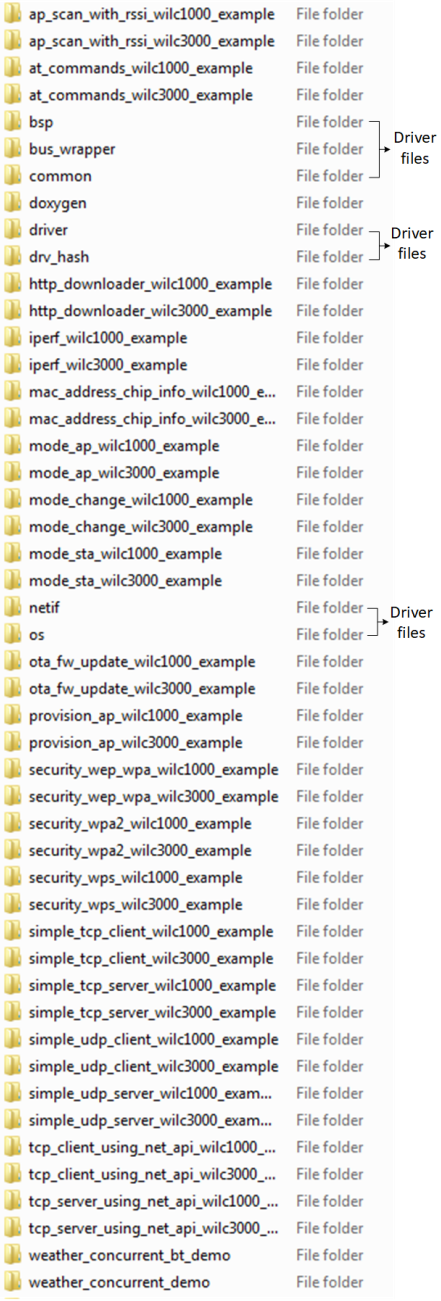
0 dBm, 3 dBm, 6 dBm, 9 dBm, 12 dBm, 15 dBm, or 18 dBm

Any other values set between the above mentioned values are floored to the nearest supported value. A maximum level is enforced by the firmware to comply to RF regulations.

16. Appendix C - Compiling the ASF Application in Linux or Makefile Environment

Perform the following to compile the ASF application in Linux or makefile environment for ATWILC1000/ ATWILC3000.

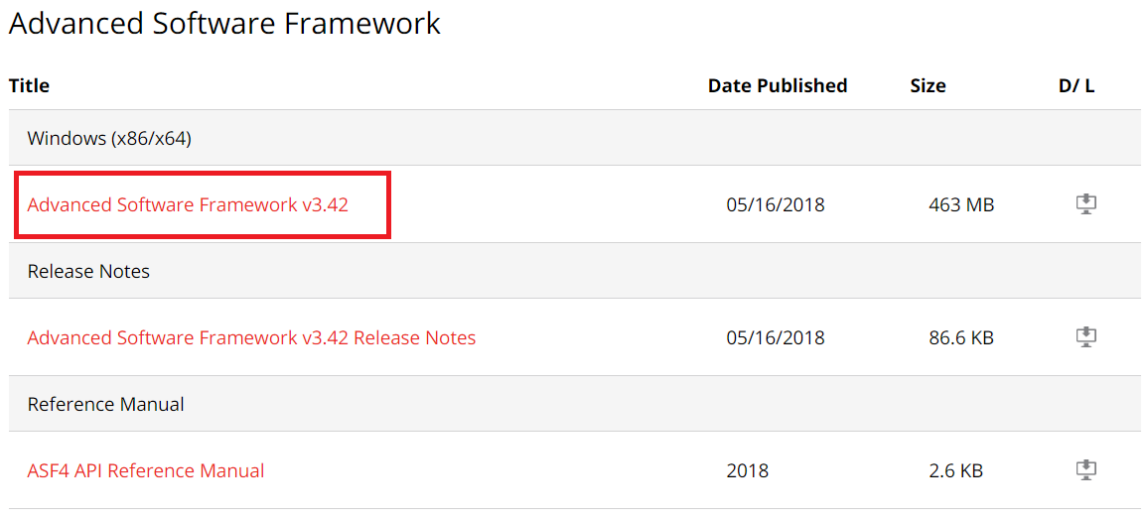
Figure 16-1. Driver and Application Files



In Figure 16-1, the list of files other than driver files are the application files.

1. Download the ASF standalone package from the ASF release from the product page.

Figure 16-2. ASF Standalone Package



2. Choose an application (example: `mode_sta_bypass_example`) from the following path.

```
\xdk-asf-3.xx.x\common\components\wifi\wilc\mode_sta_bypass_example
\samg55j19_samg_xplained_pro\gcc
```

Note: User can choose any application as required for the supported MCU.

3. From the `gcc` folder, enter the following command to compile in the Linux machine.

```
make clean all
```

The `make clean all` command compiles using the ARM GCC compiler installed in the Linux machine or any host machine, and create the images to load in to the Host MCU. For the Bypass mode it generates the SAMG55 host MCU image.

To compile the ATWILC1000/ATWILC3000 applications for other MCU's, navigate to the respective MCU GCC folder and perform the compilation command from Linux machine or any host machine.
Note: The ASF extensions are available in the Microchip Gallery.

17. Document Revision History

Revision	Date	Section	Description
D	06/2019	Program Memory	Updated the section
		ATWILCInitialization and Simple Application - Tx-Event Handling, Code Example	Added new section
		Wi-Fi Station Mode - Example Code	Updated the section
		Wi-Fi AP Mode - AP Mode Code Example	Updated the section
		Wi-Fi Direct P2P Mode - P2P Mode Code Example	Updated the section
		Wi-Fi Protected Setup - WPS Code Example	Updated the section
		Concurrency - Station-AP Concurrency, Station-P2P Client Concurrency	Updated the section
		Data Send/Receive - Receive Ethernet Frame	Updated the section
		WLAN Module - Configuration Switches	Added new section
		WLAN Module - Functions	Updated the section
		BSP - Functions	Updated the section

.....continued

Revision	Date	Section	Description
C	12/2018	BLE Power Saving and Code Example	Added new section
		Generate CertOut.h Client Certificate	Updated the section
		ATWILC P2P Connection State	Updated the P2P connection flow diagram
		P2P Mode Code Example	Updated the demo application code
		Station-P2P client Concurrency	Updated the demo application code
		Antenna Switching	Added new section
		Antenna Switch GPIO Control	Added new section
		WLAN Module	Added and updated the following APIs: • m2m_wifi_p2p • m2m_wifi_set_p2p_pin • m2m_wifi_allow_p2p_connection • m2m_wifi_set_antenna_mode
B	08/2018	• Generate CertOut.h Client Certificate • Example Code for Wi-Fi Station Mode • WLAN Module - Function	• Added steps on how to generate CertOut.h Client certificate. • Updated the example code for Wi-Fi Station mode. • Updated `m2m_wifi_set_tx_power` API to have a finer tx power granularity.

.....continued

Revision	Date	Section	Description
A	01/2018	Document	• Updated from Atmel to Microchip template. • Assigned a new Microchip document number. Previous version is Atmel 42504 revision A.

The Microchip Website

Microchip provides online support via our website at http://www.microchip.com/. This website is used to make files and information easily available to customers. Some of the content available includes:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip design partner program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Product Change Notification Service

Microchip's product change notification service helps keep customers current on Microchip products. Subscribers will receive email notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, go to http://www.microchip.com/pcn and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Embedded Solutions Engineer (ESE)
- Technical Support

Customers should contact their distributor, representative or ESE for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in this document.

Technical support is available through the web site at: http://www.microchip.com/support

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Legal Notice

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Trademarks

The Microchip name and logo, the Microchip logo, Adaptec, AnyRate, AVR, AVR logo, AVR Freaks, BesTime, BitCloud, chipKIT, chipKIT logo, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, HELDO, IGLOO, JukeBlox, KeeLoq, Kleer, LANCheck, LinkMD, maXStylus, maXTouch, MediaLB, megaAVR, Microsemi, Microsemi logo, MOST, MOST logo, MPLAB, OptoLyzer, PackeTime, PIC, picoPower, PICSTART, PIC32 logo, PolarFire, Prochip Designer, QTouch, SAM-BA, SenGenuity, SpyNIC, SST, SST Logo, SuperFlash, Symmetricom, SyncServer, Tachyon, TempTrackr, TimeSource, tinyAVR, UNI/O, Vectron, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

APT, ClockWorks, The Embedded Control Solutions Company, EtherSynch, FlashTec, Hyper Speed Control, HyperLight Load, IntelliMOS, Libero, motorBench, mTouch, Powermite 3, Precision Edge, ProASIC, ProASIC Plus, ProASIC Plus logo, Quiet-Wire, SmartFusion, SyncWorld, Temux, TimeCesium, TimeHub, TimePictra, TimeProvider, Vite, WinPath, and ZL are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, BlueSky, BodyCom, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, EtherGREEN, In-Circuit Serial Programming, ICSP, INICnet, Inter-Chip Connectivity, JitterBlocker, KleerNet, KleerNet logo, memBrain, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, SAM-ICE, Serial Quad I/O, SMART-I.S., SQI, SuperSwitcher, SuperSwitcher II, Total Endurance, TSHARC, USBCheck, VariSense, ViewSpan, WiperLock, Wireless DNA, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

The Adaptec logo, Frequency on Demand, Silicon Storage Technology, and Symmcom are registered trademarks of Microchip Technology Inc. in other countries.

ISBN: 978-1-5224-4719-1

Quality Management System

For information regarding Microchip's Quality Management Systems, please visit http://www.microchip.com/quality.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Fax: 480-792-7277 Technical Support: http://www.microchip.com/support Web Address: http://www.microchip.com **Atlanta** Duluth, GA Tel: 678-957-9614 Fax: 678-957-1455 **Austin, TX** Tel: 512-257-3370 **Boston** Westborough, MA Tel: 774-760-0087 Fax: 774-760-0088 **Chicago** Itasca, IL Tel: 630-285-0071 Fax: 630-285-0075 **Dallas** Addison, TX Tel: 972-818-7423 Fax: 972-818-2924 **Detroit** Novi, MI Tel: 248-848-4000 **Houston, TX** Tel: 281-894-5983 **Indianapolis** Noblesville, IN Tel: 317-773-8323 Fax: 317-773-5453 Tel: 317-536-2380 **Los Angeles** Mission Viejo, CA Tel: 949-462-9523 Fax: 949-462-9608 Tel: 951-273-7800 **Raleigh, NC** Tel: 919-844-7510 **New York, NY** Tel: 631-435-6000 **San Jose, CA** Tel: 408-735-9110 Tel: 408-436-4270 **Canada - Toronto** Tel: 905-695-1980 Fax: 905-695-2078	**Australia - Sydney** Tel: 61-2-9868-6733 **China - Beijing** Tel: 86-10-8569-7000 **China - Chengdu** Tel: 86-28-8665-5511 **China - Chongqing** Tel: 86-23-8980-9588 **China - Dongguan** Tel: 86-769-8702-9880 **China - Guangzhou** Tel: 86-20-8755-8029 **China - Hangzhou** Tel: 86-571-8792-8115 **China - Hong Kong SAR** Tel: 852-2943-5100 **China - Nanjing** Tel: 86-25-8473-2460 **China - Qingdao** Tel: 86-532-8502-7355 **China - Shanghai** Tel: 86-21-3326-8000 **China - Shenyang** Tel: 86-24-2334-2829 **China - Shenzhen** Tel: 86-755-8864-2200 **China - Suzhou** Tel: 86-186-6233-1526 **China - Wuhan** Tel: 86-27-5980-5300 **China - Xian** Tel: 86-29-8833-7252 **China - Xiamen** Tel: 86-592-2388138 **China - Zhuhai** Tel: 86-756-3210040	**India - Bangalore** Tel: 91-80-3090-4444 **India - New Delhi** Tel: 91-11-4160-8631 **India - Pune** Tel: 91-20-4121-0141 **Japan - Osaka** Tel: 81-6-6152-7160 **Japan - Tokyo** Tel: 81-3-6880- 3770 **Korea - Daegu** Tel: 82-53-744-4301 **Korea - Seoul** Tel: 82-2-554-7200 **Malaysia - Kuala Lumpur** Tel: 60-3-7651-7906 **Malaysia - Penang** Tel: 60-4-227-8870 **Philippines - Manila** Tel: 63-2-634-9065 **Singapore** Tel: 65-6334-8870 **Taiwan - Hsin Chu** Tel: 886-3-577-8366 **Taiwan - Kaohsiung** Tel: 886-7-213-7830 **Taiwan - Taipei** Tel: 886-2-2508-8600 **Thailand - Bangkok** Tel: 66-2-694-1351 **Vietnam - Ho Chi Minh** Tel: 84-28-5448-2100	**Austria - Wels** Tel: 43-7242-2244-39 Fax: 43-7242-2244-393 **Denmark - Copenhagen** Tel: 45-4450-2828 Fax: 45-4485-2829 **Finland - Espoo** Tel: 358-9-4520-820 **France - Paris** Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79 **Germany - Garching** Tel: 49-8931-9700 **Germany - Haan** Tel: 49-2129-3766400 **Germany - Heilbronn** Tel: 49-7131-72400 **Germany - Karlsruhe** Tel: 49-721-625370 **Germany - Munich** Tel: 49-89-627-144-0 Fax: 49-89-627-144-44 **Germany - Rosenheim** Tel: 49-8031-354-560 **Israel - Ra'anana** Tel: 972-9-744-7705 **Italy - Milan** Tel: 39-0331-742611 Fax: 39-0331-466781 **Italy - Padova** Tel: 39-049-7625286 **Netherlands - Drunen** Tel: 31-416-690399 Fax: 31-416-690340 **Norway - Trondheim** Tel: 47-72884388 **Poland - Warsaw** Tel: 48-22-3325737 **Romania - Bucharest** Tel: 40-21-407-87-50 **Spain - Madrid** Tel: 34-91-708-08-90 Fax: 34-91-708-08-91 **Sweden - Gothenberg** Tel: 46-31-704-60-40 **Sweden - Stockholm** Tel: 46-8-5090-4654 **UK - Wokingham** Tel: 44-118-921-5800 Fax: 44-118-921-5820