USER GUIDE

ATWINC1500 Wi-Fi Network Controller - Software Design Guide

Atmel SmartConnect

Introduction

Atmel® SmartConnect ATWINC1500 is an IEEE® 802.11 b/g/n network controller SoC for applications in the Internet-Of-Things. It is an ideal add-on to existing MCU solutions bringing Wi-Fi and network capabilities through a UART-to-Wi-Fi or SPI-to-Wi-Fi interface. The WINC1500 connects to any Atmel AVR® or Atmel | SMART MCU with minimal resource requirements.

Features

- Wi-Fi IEEE 802.11 b/g/n STA, AP, and Wi-Fi Direct® modes
- Wi-Fi Protected Setup (WPS)
- Support of WEP, WPA/WPA2 personal, and WPA/WPA2 Enterprise security
- Embedded network stack protocols to offload work from the MCU (minimize the host CPU requirements). This allows the WINC to operate with a wide range of MCUs including low end MCUs.
- Embedded TCP/IP stack with BSD-style socket API
- Embedded network protocols
  - DHCP client/server
  - DNS resolver client
  - SNTP client for UTC time synchronization
- Embedded TLS security abstracted behind BSD-style socket API
- HTTP Server for provisioning over AP mode
- Ultra-low cost IEEE 802.11 b/g/n RF/PH/MAC SoC
- Fast boot from on-chip Boot ROM
- 4Mb internal Flash memory with OTA firmware upgrade
- Low power consumption with different power saving modes
- SPI, I^{2}C, and UART support
- Low footprint host driver with the following capabilities:
  - Can run on 8, 16, and 32 bit MCU
  - Little and Big endian support
  - Consumes about 8KB of code memory and 1KB of data memory on host MCU

Table of Contents

Version History **7**

Icon Key Identifiers **7**

Glossary **7**

References **7**

1 Host Driver Architecture **8**

1.2 WLAN API 8
1.3 Socket API 8
1.4 Host Interface (HIF) 9
1.5 Board Support Package (BSP) 9
1.6 Serial Bus Interface 9

2 WINC System Architecture **10**

2.1 Bus Interface 10
2.2 Non-Volatile Storage 10
2.3 CPU 11
2.4 IEEE 802.11 MAC Hardware 11
2.5 Program Memory 11
2.6 Data Memory 11
2.7 Shared Packet Memory 11
2.8 IEEE 802.11 MAC Firmware 11
2.9 Memory Manager 11
2.10 Power Management 11
2.11 WINC RTOS 11
2.12 WINC IoT Library 12

3 WINC Initialization and Simple Application **13**

3.1 BSP Initialization 13
3.2 WINC Host Driver Initialization 13
3.3 Socket Layer Initialization 13
3.4 WINC Event Handling 13
3.4.1 Asynchronous Events 14
3.4.2 Interrupt Handling 14
3.5 Code Example 15

4 WINC Configuration **16**

4.1 Device Parameters 16
4.1.1 System Time 16
4.1.2 Firmware and Driver Version 16
4.2 WINC Modes of Operation 16
4.2.1 Idle Mode 17
4.2.2 Wi-Fi Station Mode 17
4.2.3 Wi-Fi Direct (P2P) Mode 18
4.2.4 Wi-Fi Hotspot (AP) Mode 18
4.2.5 Sniffer Mode (monitoring mode) 18
4.3 Network Parameters 18
4.3.1 Device Name 18

4.3.2 Wi-Fi MAC Address 19
4.3.3 IP Address 19
4.4 Power Saving Parameters 19
4.4.1 Power Saving Modes 19
4.4.1.1 M2M_PS_MANUAL 20
4.4.1.2 M2M_PS_AUTOMATIC 20
4.4.1.3 M2M_PS_H_AUTOMATIC 21
4.4.1.4 M2M_PS_DEEP_AUTOMATIC 21
4.4.2 Configuring Listen Interval and DTIM Monitoring 21

5 Wi-Fi Station Mode 22

5.1 Scan Configuration Parameters 22
5.1.1 Scan Region 22
5.1.2 Scan Options 22
5.2 Wi-Fi Scan 22
5.3 On Demand Wi-Fi Connection 23
5.4 Default Connection 24
5.5 Wi-Fi Security 25
5.6 Example Code 26

6 WINC Socket Programming 27

6.1 Overview 27
6.1.1 WINC Socket Types 27
6.1.2 Socket Properties 27
6.1.3 Limitations 27
6.2 WINC Sockets API 27
6.2.1 API Prerequisites 27
6.2.2 Non-blocking Asynchronous Socket APIs 28
6.2.3 Socket API Functions 28
6.2.3.1 socketInit 28
6.2.3.2 registerSocketCallback 28
6.2.3.3 socket 28
6.2.3.4 connect 28
6.2.3.5 bind 29
6.2.3.6 listen 29
6.2.3.7 accept 29
6.2.3.8 send 30
6.2.3.9 sendto 30
6.2.3.10 recv / recvfrom 31
6.2.3.11 close 31
6.2.3.12 setsockopt 31
6.2.3.13 gethostbyname 31
6.2.4 Summary 32
6.3 Socket Connection Flow 33
6.3.1 TCP Client Operation 34
6.3.2 TCP Server Operation 35
6.3.3 UDP Client Operation 36
6.3.4 UDP Server Operation 37
6.3.5 DNS Host Name Resolution 38
6.4 Example Code 39
6.4.1 TCP Client Example Code 39

6.4.2 TCP Server Example Code 40
6.4.3 UDP Client Example Code 42
6.4.4 UDP Server Example Code 43

7 Transport Layer Security (TLS) 45

7.1 TLS Connection Establishment 45
7.2 Server Certificate Installation 46
7.2.1 Technical Background 46
7.2.1.1 Public Key Infrastructure 46
7.2.1.2 TLS Server Authentication 46
7.2.2 Adding a Certificate to the WINC Trusted Root Certificate Store 46
7.3 WINC TLS Limitations 46
7.3.1 Modes of Operation 46
7.3.2 Concurrent Connections 46
7.3.3 Supported Cipher Suites 46
7.3.4 Supported Hash Algorithms 46
7.4 SSL Client Code Example 47

8 Wi-Fi AP Mode 49

8.1 Overview 49
8.2 Setting WINC AP Mode 49
8.3 Limitations 49
8.4 Sequence Diagram 49
8.5 AP Mode Code Example 50

9 Wi-Fi Direct P2P Mode 51

9.1 Overview 51
9.2 WINC P2P Capabilities 51
9.3 WINC P2P Limitations 51
9.4 WINC P2P States 51
9.5 WINC P2P Listen State 51
9.6 WINC P2P Connection State 51
9.7 WINC P2P Disconnection State 52
9.8 P2P Mode Code Example 52

10 Provisioning 55

10.1 Limitations 55
10.2 HTTP Provisioning 55
10.2.1 Provisioning Control Flow 56
10.2.2 HTTP Redirect Feature 57
10.2.3 Provisioning Code Example 57
10.3 Wi-Fi Protected Setup (WPS) 58
10.3.1 WPS Configuration Methods 58
10.3.2 WPS Limitations 58
10.3.3 WPS Control Flow 59
10.3.4 WPS Code Example 60

11 Over-The-Air Upgrade 61

11.1 Overview 61
11.2 OTA Image Architecture 61
11.3 OTA Download Sequence Diagram 62
11.4 OTA Firmware Rollback 62

11.5 OTA Limitations 63
11.6 OTA Code Example 63

12 Multicast Sockets 64

12.1 Overview 64
12.2 How to use Filters 64
12.3 Multicast Socket Code Example 64

13 WINC Serial Flash Memory 69

13.1 Overview and Features 69
13.2 Accessing to Serial Flash 69
13.3 Read/Write/Erase Operations 69
13.4 Serial (SPI) Flash Map 70
13.5 Flash Read, Erase, Write Code Example 70

14 Wi-Fi Sniffer Mode 72

14.1 Overview 72
14.2 Sniffer (Monitoring) Mode APIs 72
14.3 Monitoring Parameters 72
14.4 Sequence Diagram 73
14.5 Code Example 74

15 Writing a Simple Networking Application 75

15.1 Prerequisites 75
15.2 Solution Overview 75
15.3 Project Creation 76
15.4 Wi-Fi Software API Files 76
15.5 Reading Temperature Sensor and Controlling LED Status 78
15.6 Step By Step Development 79

16 Host Interface Protocol 97

16.1 Transfer Sequence Between HIF Layer and WINC Firmware 98
  16.1.1 Frame Transmit 98
  16.1.2 Frame Receive 99
16.2 HIF Message Header Structure 100
16.3 HIF Layer APIs 100
16.4 Scan Code Example 101

17 WINC SPI Protocol 107

17.1 Introduction 107
  17.1.1 Command Format 108
  17.1.2 Response Format 112
  17.1.3 Data Packet Format 113
  17.1.4 Error Recovery Mechanism 114
  17.1.5 Clockless Registers Access 115
17.2 Message Flow for Basic Transactions 116
  17.2.1 Read Single Word 116
  17.2.2 Read Internal Register (for clockless registers) 117
  17.2.3 Read Block 118
  17.2.4 Write Single Word 119
  17.2.5 Write Internal Register (for clockless registers) 119
  17.2.6 Write Block 120

17.3 SPI Level Protocol Example 120
17.3.1 TX (Send Request) 121
17.3.2 RX (Receive Response) 132

Appendix A. How to Generate Certificates 147

A.1 Introduction 147
A.2 Steps 147

Appendix B. X.509 Certificate Format and Conversion 148

B.1 Introduction 148
B.2 Conversion Between Different Formats 148
B.2.1 Using Windows 148
B.2.2 Using OpenSSL 148
B.2.3 Online Conversion 148

Appendix C. How to Download the Certificate into WINC 149

C.1 Overview 149
C.2 Certificate Downloading 149
C.3 Adding New Certificate 149

Appendix D. Firmware Image Downloader 150

D.1 Preparing Environment 150
D.2 Download Firmware 151

Appendix E. Gain Settings Builder 153

E.1 Introduction 153
E.2 Preparing Environment 153
E.3 How to use 153
E.3.1 Method 1 153
E.3.2 Method 2 153

Appendix F. API Reference 155

F.1 WLAN Module 155
F.1.1 Defines 155
F.1.2 Enumeration/Typedef 157
F.1.3 Function 179
F.2 BSP 212
F.2.1 Defines 212
F.2.2 Data Types 212
F.2.3 Function 212
F.3 Socket 214
F.3.1 Defines 214
F.3.2 Error Codes 215
F.3.3 Enumeration/Typedef 217
F.3.4 Function 221
F.4 SPI Flash 236
F.4.1 Function 236

Appendix G. Revision History 239

Version History

Version	Date	Description
Version 0.0	12-2-2015	First Draft Version released
Version 0.1	14-2-2015	First review by Samer
Version 0.2	22-2-2015	First review by Jon Edney and EDC peer reviews
Version 0.3	26-2-2015	Review by Samer
Version 0.4	26-2-2015	Second review by Jon Edney and EDC peer reviews

Icon Key Identifiers

INFO	Delivers contextual information about a specific topic.
TIP	Highlights useful tips and techniques.
TO DO	Highlights objectives to be completed.
RESULT	Highlights the expected result of an assignment step.
WARNING	Indicates important information.

Glossary

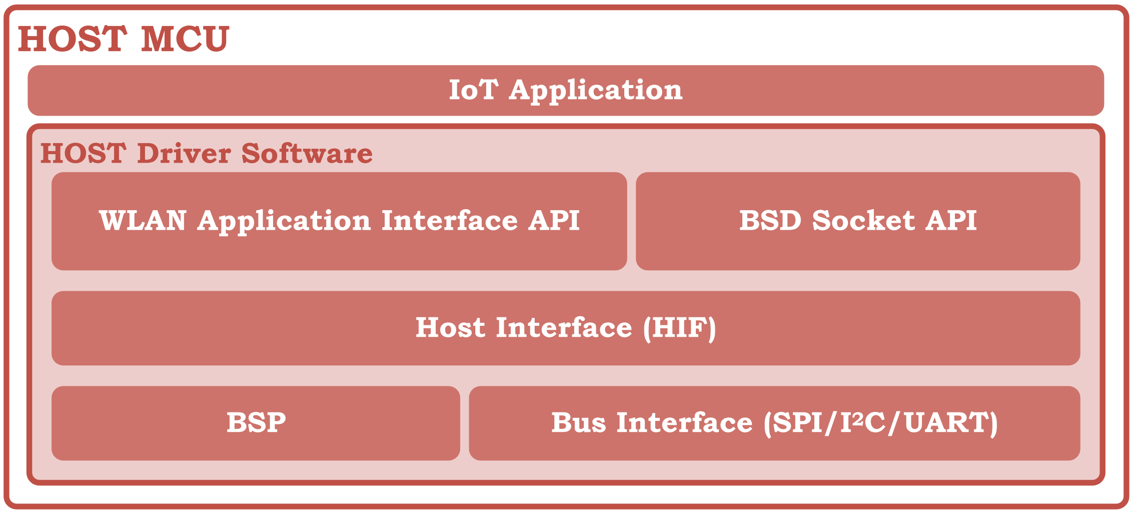
BSD	Berkeley Software Distribution
BSP	Board Support Package
HIF	Host Interface Layer
IoT	Internet of Things
OTA	Over The Air
OTP	One Time Programmable
TLS	Transport Layer Security
WINC	Wi-Fi Network Controller

References

[R01] Atmel-42417-SAMD21-ATWINC1500-Platform_Getting_Started_Guide
[R02] Atmel-42418-SAM-D21-ATWINC1500-Software_Programming_Guide
[R03] ATWINC1500-MR210P Datasheet

1 Host Driver Architecture

Figure 1-1. Host Driver Software Architecture



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

1.2 WLAN API

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

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

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

1.3 Socket API

This module provides the socket communication APIs that are mostly compliant with the well-known BSD sockets to enable rapid application development. To comply with the nature of MCU application environment, there are differences in API prototypes and in usage of some APIs between WINC sockets and BSD sockets. Appendix F in this document (the API reference manual) along with the provided socket code examples will guide you to understand similarities and differences between WINC and BSD sockets.

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

The detailed description of the socket operations is provided in Chapter 6: WINC Socket Programming.

1.4 Host Interface (HIF)

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

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

The detailed description of the HIF design is provided in Chapter 16: Host Interface Protocol.

1.5 Board Support Package (BSP)

The Board Support Package 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`.

1.6 Serial Bus Interface

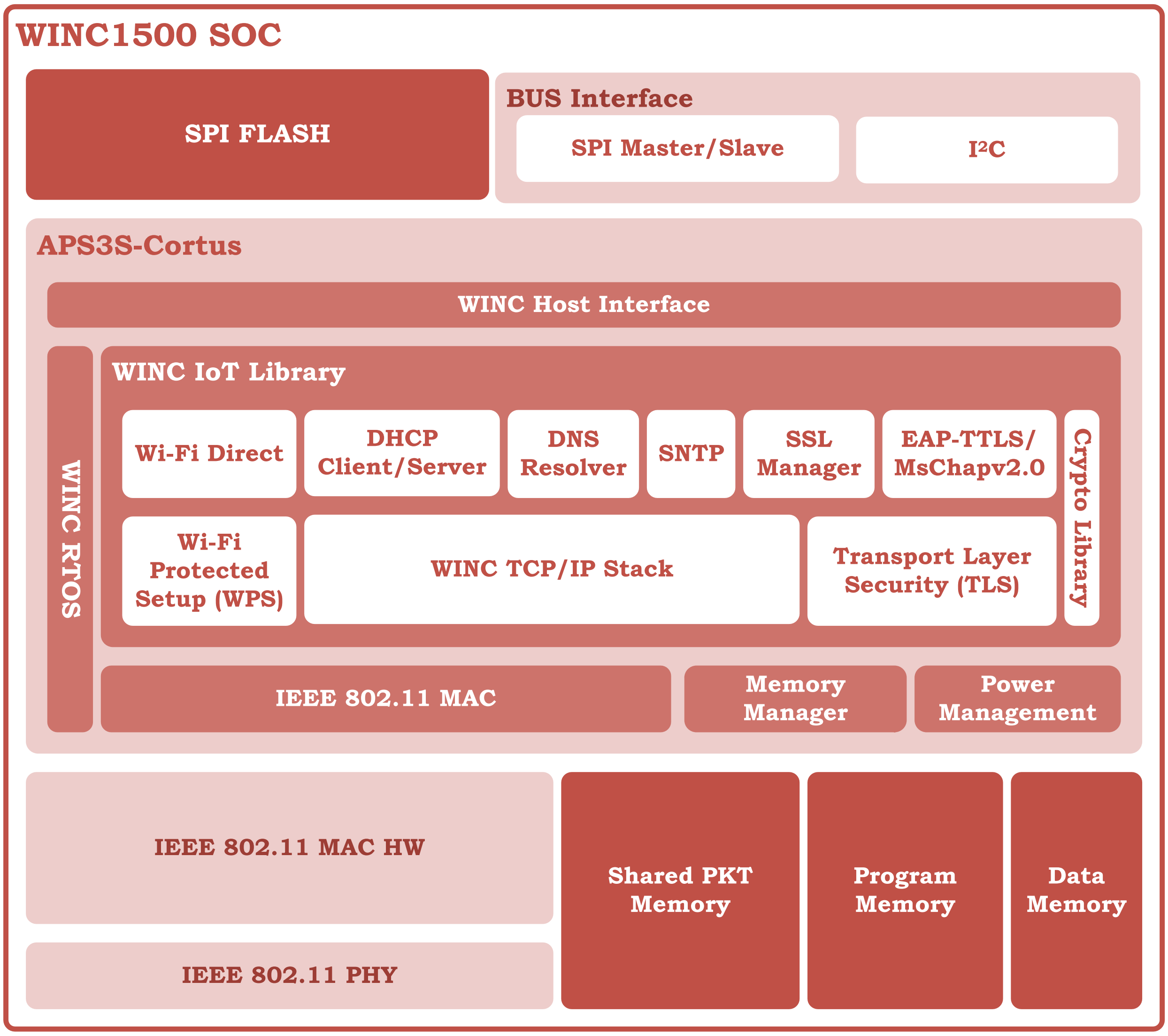
The Serial Bus Interface module abstracts the hardware associated with implementing the bus between the Host and the WINC. The serial bus interface abstracts I^{2}C, SPI, or UART bus interface. The basic bus access operations (Read and Write) are implemented in this module as appropriate for the interface type and the specific hardware.

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

2 WINC System Architecture

Figure 2-1 shows the WINC system architecture. In addition to its built-in Wi-Fi IEEE-802.11 physical layer and RF front end, the WINC ASIC has an embedded APS3S-Cortus 32-bit CPU to run the WINC 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 2-1. WINC System Architecture



2.1 Bus Interface

Hardware logic for the supported bus types for WINC communications.

2.2 Non-Volatile Storage

The SoC has an integrated 4Mb serial flash inside the WINC package (SIP). This stores the WINC firmware image and has room to store a second image to support OTA. It also stores information used by WINC firmware in the run-time.

The detailed description of the serial flash is provided in Chapter 13: WINC Serial Flash Memory.

2.3 CPU

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

2.4 IEEE 802.11 MAC Hardware

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

2.5 Program Memory

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

2.6 Data Memory

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

2.7 Shared Packet Memory

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

2.8 IEEE 802.11 MAC Firmware

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

2.9 Memory Manager

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

2.10 Power Management

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

2.11 WINC RTOS

The firmware includes a low-footprint real-time scheduler which allows concurrent multi-tasking on WINC CPU. The WINC RTOS provides semaphores and timer functionality.

2.12 WINC IoT Library

The WINC IoT library provides a set of networking protocols in WINC firmware. It offloads the host MCU from networking and transport layer protocols. The following sections describe the components of WINC IoT library.

- **WINC TCP/IP STACK**

The WINC TCP/IP is an IPv4.0 stack based on the uIP TCP/IP stack (pronounced micro IP).

uIP is a low footprint TCP/IP stack which has the ability to run on a memory-constrained microcontroller platform. It was originally developed by Adam Dunkels, licensed under a BSD style license, and further developed by a wide group of developers. The WINC TCP/IP stack adds to the original uIP implementation several enhancements to boost TCP and UDP throughput.

- **DHCP CLIENT/SERVER**

A DHCP client is embedded in WINC firmware that can obtain an IP configuration automatically after connecting to a Wi-Fi network.

WINC firmware provides an instance of a DHCP server that starts automatically when WINC AP mode is enabled. When the host MCU application activates the AP mode, it is allowed to configure the DHCP Server IP address pool range within the AP configuration parameters.

- **DNS RESOLVER**

WINC firmware contains an instance of an embedded DNS resolver. This module can return an IP address by resolving the host domain names supplied with the socket API call gethostbyname.

- **SNTP**

The SNTP (Simple Network Time Protocol) module implements an SNTP client used to synchronize the WINC internal clock to the UTC clock.

- **EAP-TTLS/MSCHAPV2.0**

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

- **TRANSPORT LAYER SECURITY**

For TLS implementation, see Chapter 7: Transport Layer Security (TLS) for details.

- **WI-FI PROTECTED SETUP**

For WPS protocol implementation, see Section 10.3: Wi-Fi Protected Setup (WPS) for details.

- **WI-FI DIRECT**

For Wi-Fi Direct protocol implementation, see Chapter 9: Wi-Fi Direct P2P Mode for details.

- **CRYPTO LIBRARY**

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

- MD4 Hash algorithm (Used only for MsChapv2.0 digest calculation)
- MD5 Hash algorithm
- SHA-1 Hash algorithm
- SHA-256 Hash algorithm
- DES Encryption (Used only for MsChapv2.0 digest calculation)
- MS-CHAPv2.0 (Used as the EAP-TTLS inner authentication algorithm)
- AES-128, AES-256 Encryption (Used for securing WPS and TLS traffic)
- BigInt module for large integer arithmetic (for Public Key Cryptographic computations)
- RSA Public Key cryptography algorithms (includes RSA Signature and RSA Encryption algorithms)

3 WINC Initialization and Simple Application

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

- BSP Initialization
- WINC Host Driver Initialization
- Socket Layer Initialization
- Call WINC driver entry point

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

3.1 BSP Initialization

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

- Resets the WINC[1] using corresponding host MCU control GPIOs
- Initializes the host MCU GPIO which connects to WINC interrupt line. It configures the GPIO as an interrupt source to the host MCU. During runtime, WINC interrupts the host to notify the application of events and data pending inside WINC firmware.
- Initializes the host MCU delay function used within `nm_bsp_sleep` implementation

3.2 WINC Host Driver Initialization

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

- Initializes the bus wrapper, I^{2}C, SPI, or UART, depending on the host driver software bus interface configuration compilation flag `USE_I2C`, `USE_SPI` or `USE_UART` respectively
- Registers an application-defined Wi-Fi event handler
- Initializes the driver and ensures that the current WINC firmware matches the current driver version
- Initializes the host interface and the Wi-Fi layer and registers the BSP Interrupt

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

3.3 Socket Layer Initialization

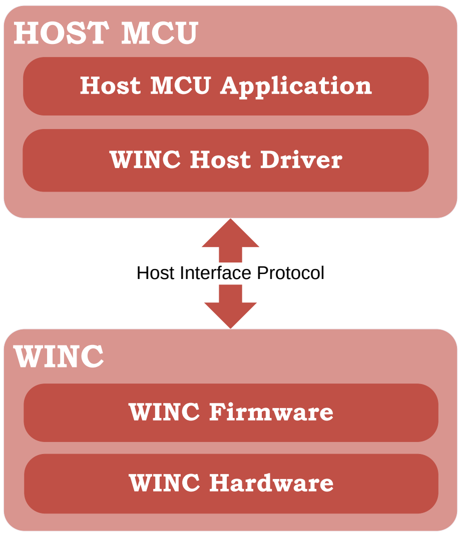
Socket layer initialization is carried out by calling the `socketInit` API. It must be called prior to any socket activity. Refer to Section 6.2.1 for more information about socket initialization and programming.

3.4 WINC Event Handling

The WINC host driver API allows the host MCU application to interact with the WINC firmware. To facilitate interaction, the WINC driver implements the *Host Interface (HIF) Protocol* described in Chapter 16: Host Interface Protocol. The HIF protocol defines how to serialize and de-serializes API requests and response callbacks over the serial bus interface: I^{2}C, UART, or SPI.

[1] Refer to WINC1500 datasheet [R03] for more information about WINC hardware reset sequence.

Figure 3-1. WINC System Architecture



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

3.4.1 Asynchronous Events

Some WINC host driver APIs are synchronous function calls, where the result is ready by the return of the function. However, most WINC host driver API functions are asynchronous. This means that when the application calls an API to request a service, the call is non-blocking and returns immediately, most often before the requested action is completed. When completed, a notification is provided in the form of a HIF protocol message from the WINC firmware to the host which, in turn, is delivered to the application via a callback[2] function. Asynchronous operation is essential when the requested service such as Wi-Fi connection may take significant time to complete. In general, the WINC firmware uses asynchronous events to signal the host driver about status change or pending data.

The HIF uses “push” architecture, where data and events are pushed from WINC firmware to the host MCU in FCFS manner. For instance, suppose that host MCU application has two open sockets: socket 1 and socket 2. If WINC receives socket 1 data followed by socket 2 data, then HIF shall deliver socket data in two HIF protocol messages in the order they were received. HIF does not allow reading socket 2 data before socket 1 data.

3.4.2 Interrupt Handling

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

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

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

INFO All the application-defined event callback functions registered with WINC driver run in the context **m2m_wifi_handle_events** API.

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

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

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

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

3.5 Code Example

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

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

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

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

        /*intilize the WINC 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 WINC event handler */
                while(m2m_wifi_handle_events(NULL) != M2M_SUCCESS) {
                }
        }
}
```

4 WINC Configuration

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

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

4.1 Device Parameters

4.1.1 System Time

It is important to set WINC system to UTC time to ensure proper validity check of the X509 certificate expiration date. Since WINC does not contain a built-in real-time clock (RTC), there are two ways to obtain UTC time:

- **Using the internal SNTP client**: which is enabled by default in the WINC firmware at start-up. The SNTP client synchronizes the WINC system clock to the UTC time from well-known time servers, e.g. "*time-c.nist.gov*". The SNTP client uses a default update cycle of one day.
- **From host MCU RTC**: If the host MCU has a RTC, the application may disable the SNTP client by calling `m2m_wifi_disable_sntp` after WINC initialization. The application shall provision the WINC system time by calling `m2m_wifi_set_system_time` API.

4.1.2 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 WINC firmware version.

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

The version parameters provided are:

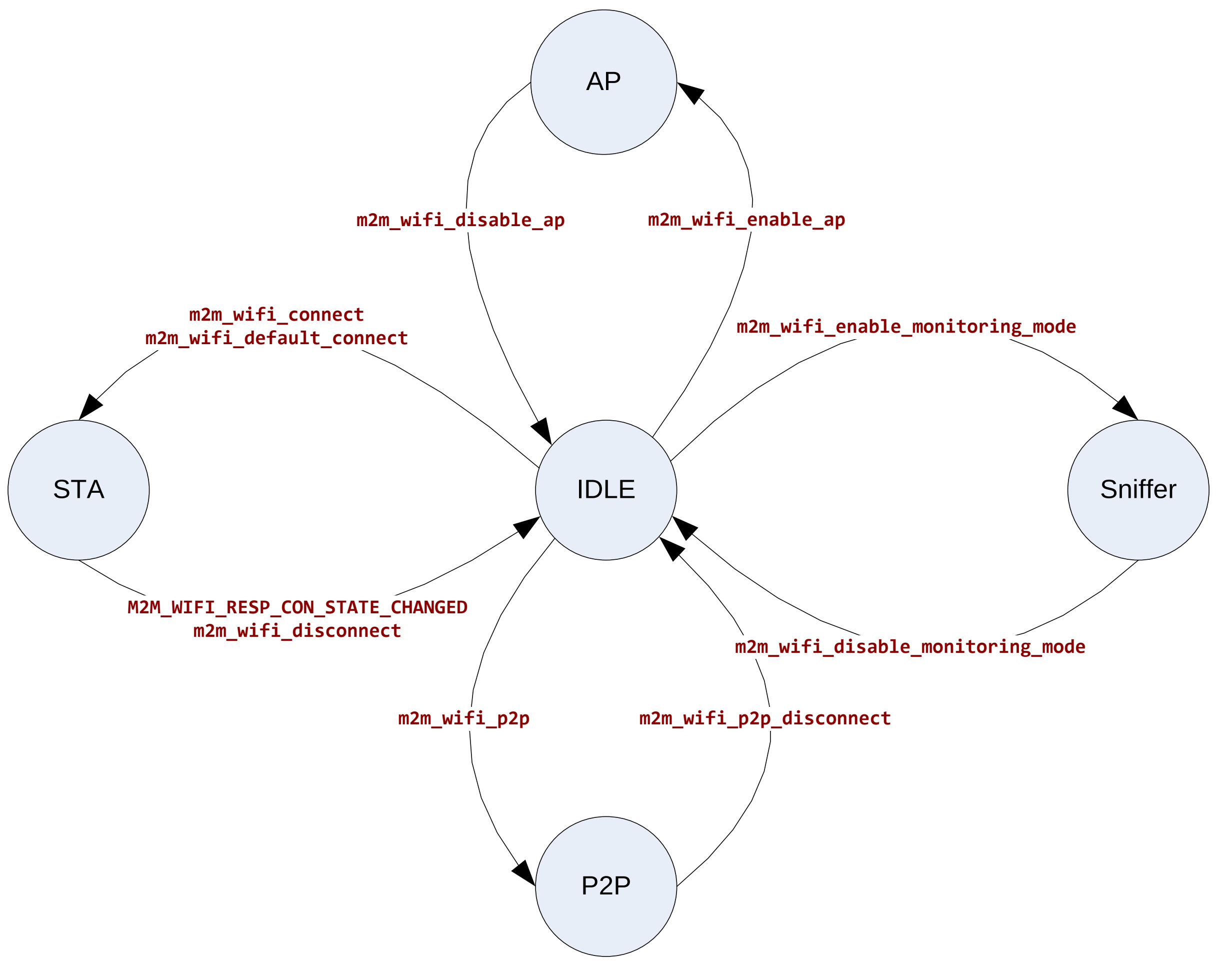
• `M2M_FIRMWARE_VERSION_MAJOR_NO`:	Firmware Major release version number
• `M2M_FIRMWARE_VERSION_MINOR_NO`:	Firmware Minor release version number
• `M2M_FIRMWARE_VERSION_PATCH_NO`:	Firmware patch release version number
• `M2M_DRIVER_VERSION_MAJOR_NO`:	Driver Major release version number
• `M2M_DRIVER_VERSION_MINOR_NO`: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

4.2 WINC Modes of Operation

The WINC firmware supports the following modes of operation:

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

Figure 4-1. WINC Modes of Operation



4.2.1 Idle Mode

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

4.2.2 Wi-Fi Station Mode

WINC enters station (STA) mode when the host MCU requests connection to an AP using the **m2m_wifi_connect** or **m2m_wifi_default_connect** APIs. WINC exits STA mode when it receives a disconnect request from the Wi-Fi AP conveyed to the host MCU application via the event callback **M2M_WIFI_RESP_CON_STATE_CHANGED** or when the host MCU application decides to terminate the connection via **m2m_wifi_disconnect** API. WINC firmware ignores mode change requests while in this mode until WINC exits the mode.

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

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

4.2.3 Wi-Fi Direct (P2P) Mode

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

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

For more information about this mode, refer to Chapter 9: Wi-Fi Direct P2P Mode.

4.2.4 Wi-Fi Hotspot (AP) Mode

In AP mode, WINC allows Wi-Fi stations to connect to WINC and obtain IP address from WINC DHCP server. To enter AP mode, host MCU application calls `m2m_wifi_enable_ap` API. To exit AP mode, the application calls `m2m_wifi_disable_ap` API. WINC firmware ignores mode change requests while in this mode until WINC exits the mode.

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

For more information about this mode, refer to Chapter 8: Wi-Fi AP Mode.

4.2.5 Sniffer Mode (monitoring mode)

In this mode, WINC is in promiscuous mode in which it passes all received packets on the current Wi-Fi channel from other non-associated stations and other BSSs. WINC supports a programmable packet filter to selectively receive packets which match the filter criteria. To enter sniffer mode, the host MCU application calls `m2m_wifi_enable_monitoring_mode` API. To exit sniffer mode, the host MCU application calls `m2m_wifi_disable_monitoring_mode` API. WINC firmware ignores mode change requests while in this mode until WINC exits the mode.

TIPS The supported API functions in this mode use the HIF command type **tenuM2mConfigCmd**. See the full list of commands in the header file **m2m_types.h**.

For more information about this mode, refer to Chapter 14: Wi-Fi Sniffer Mode.

4.3 Network Parameters

4.3.1 Device Name

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

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

4.3.2 Wi-Fi MAC Address

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

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

INFO OTP MAC address is programmed in WINC OTP memory at manufacturing time.

For more details, refer to description of the following APIs in Appendix F: API Reference.

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

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

4.3.3 IP Address

WINC firmware uses the embedded DHCP client to obtain an IP configuration automatically after a successful Wi-Fi connection. After the IP configuration is obtained, the host MCU application is notified by the asynchronous event `M2M_WIFI_RESP_IP_CONFIGURED`.

Alternatively, the host MCU application can set a static IP configuration by calling `m2m_wifi_set_static_ip` API. Setting a static IP address will cancel any pending DHCP request and disable the DHCP client until the next Wi-Fi connection attempt to the same AP or any other AP.

4.4 Power Saving Parameters

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

4.4.1 Power Saving Modes

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

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

TIPS `M2M_PS_DEEP_AUTOMATIC` mode recommended for most applications.

4.4.1.1 M2M_PS_MANUAL

This is a fully host-driven power saving mode.

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

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

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

This mode is useful for applications which send notifications very rarely due to a certain trigger. It fits also applications which send notifications periodically with a very long spacing between notifications. Careful power planning is required when using this mode. If the host MCU decides to sleep for very long period, it may use `M2M_PS_MANUAL` or may power off WINC[3] completely. The advantage of this mode compared to powering off WINC is that `M2M_PS_MANUAL` saves the time required for WINC firmware to boot since the firmware is always loaded in WINC memory. The real pros and cons depend on the nature of the application. In some applications, the sleep duration could be long enough to be a power-efficient decision to power off WINC and power it on again and reconnect to the AP when host MCU wakes up. In other situations, a latency-sensitive application may choose to use `M2M_PS_MANUAL` to avoid WINC firmware boot latency on the expense of slightly increased power consumption.

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

4.4.1.2 M2M_PS_AUTOMATIC

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

[3] Refer to WINC datasheet in [R03] for hardware power off sequence.

4.4.1.3 M2M_PS_H_AUTOMATIC

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

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

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

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

4.4.1.4 M2M_PS_DEEP_AUTOMATIC

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

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

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

- Expiry of the hardware sleep timer. WINC wakes up to receive the upcoming beacon from AP.
- WINC wakes up[4] when the host MCU application requests services from WINC by calling any host driver API function, e.g. Wi-Fi or socket operation.

4.4.2 Configuring Listen Interval and DTIM Monitoring

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

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

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

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

[4] The wakeup sequence is handled internally in the WINC host driver in the `hif_chip_wake` API. Refer to the reference Chapter 16 for more information.

5 Wi-Fi Station Mode

This chapter provides information about WINC Wi-Fi station (STA) mode described in Section 4.2.2. Wi-Fi station mode involves scan operation; association to an AP using parameters (SSID and credentials) provided by host MCU or using AP parameters stored in WINC non-volatile storage (default connection). The chapter also provides information about supported security modes along with code examples.

5.1 Scan Configuration Parameters

5.1.1 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 WINC initial region configuration is equal to 14 channels (Asia), but this can be changed by setting the scan region using: the `m2m_wifi_set_scan_region` API.

5.1.2 Scan Options

During Wi-Fi scan operation, WINC sends probe request Wi-Fi frames and waits for some time on the current Wi-Fi channel to receive probe response frames from nearby APs before it switches to the next channel. Increasing the scan wait time has a positive effect on the number of access pointed detected during scan. However, it has a negative effect on the power consumption and overall scan duration. WINC firmware default scan wait time is optimized to provide the tradeoff between power consumption and scan accuracy. WINC 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.

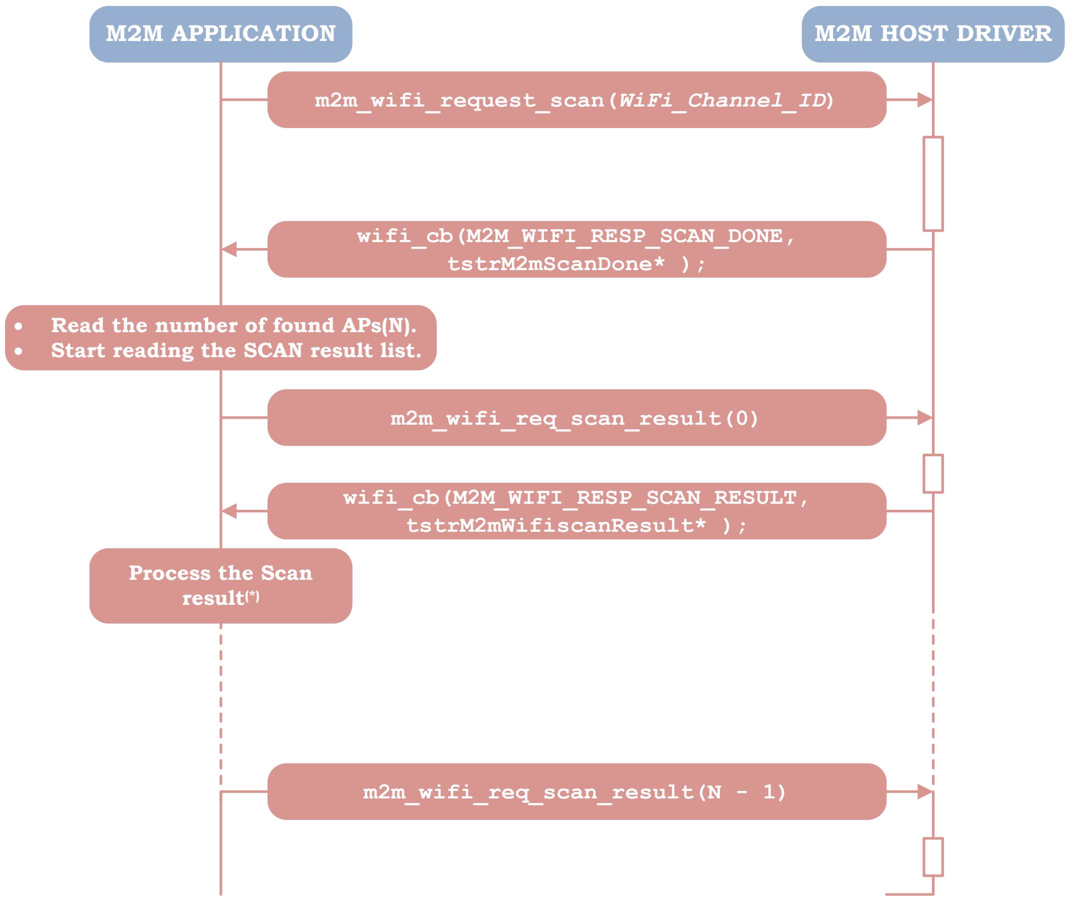
5.2 Wi-Fi Scan

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

The scan response time depends on the scan options. For instance, if the host MCU application requests to scan all channels, the scan time will be equal to N`oOfChannels (14) * M2M_SCAN_MIN_NUM_SLOTS* M2M_SCAN_MIN_SLOT_TIME` (Refer to the Appendix F API Reference chapter for how to customize the scan parameters).

The scan operation is illustrated in Figure 5-1.

Figure 5-1. Wi-Fi Scan Operation


M2M APPLICATION
M2M HOST DRIVER
m2m_wifi_request_scan(WiFi_Channel_ID)
wifi_cb(M2M_WIFI_RESP_SCAN_DONE, tstrM2mScanDone*);
• Read the number of found APs(N).
• Start reading the SCAN result list.
m2m_wifi_req_scan_result(0)
wifi_cb(M2M_WIFI_RESP_SCAN_RESULT, tstrM2mWifiscanResult*);
Process the Scan result(*)
m2m_wifi_req_scan_result(N - 1)

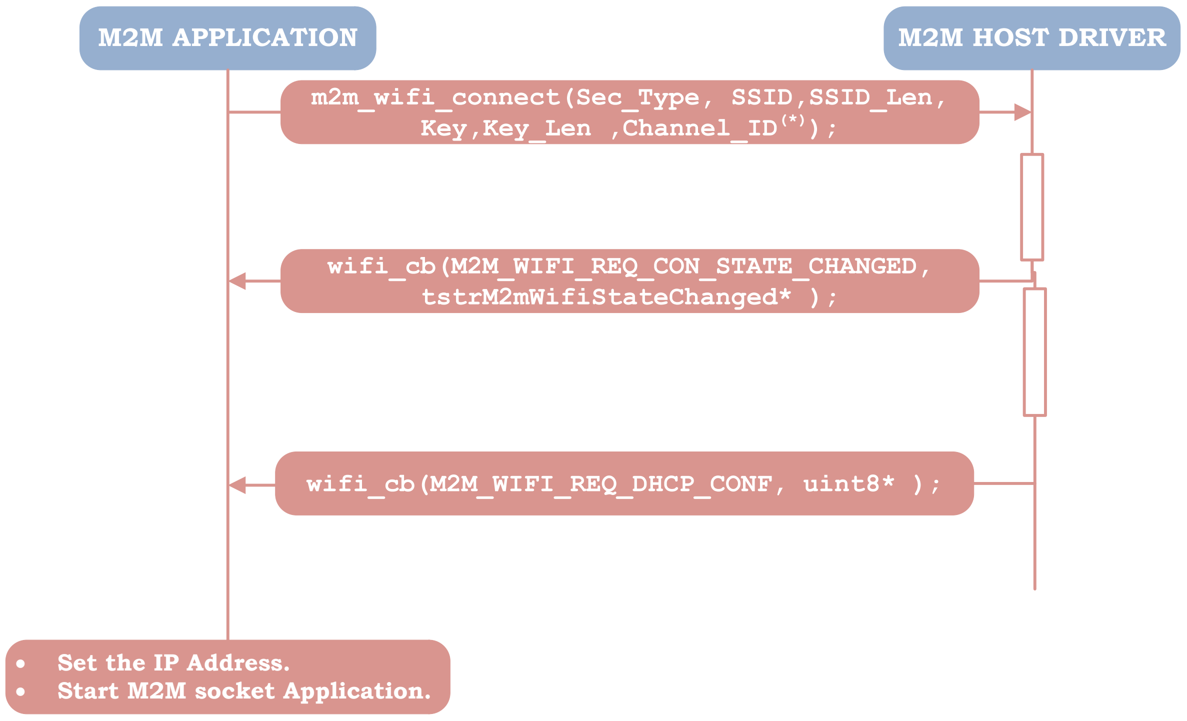
5.3 On Demand Wi-Fi Connection

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

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

The Wi-Fi on demand connection operation is described in Figure 5-2.

Figure 5-2. On-demand Wi-Fi Connection


M2M APPLICATION
M2M HOST DRIVER
m2m_wifi_connect(Sec_Type, SSID,SSID_Len, Key,Key_Len ,Channel_ID(*));
wifi_cb(M2M_WIFI_REQ_CON_STATE_CHANGED, tstrM2mWifiStateChanged*);
wifi_cb(M2M_WIFI_REQ_DHCP_CONF, uint8*);
Set the IP Address.
Start M2M socket Application.

5.4 Default Connection

The host MCU application may establish a Wi-Fi connection without prior knowledge to the AP information by calling the API `m2m_wifi_default_connect`.

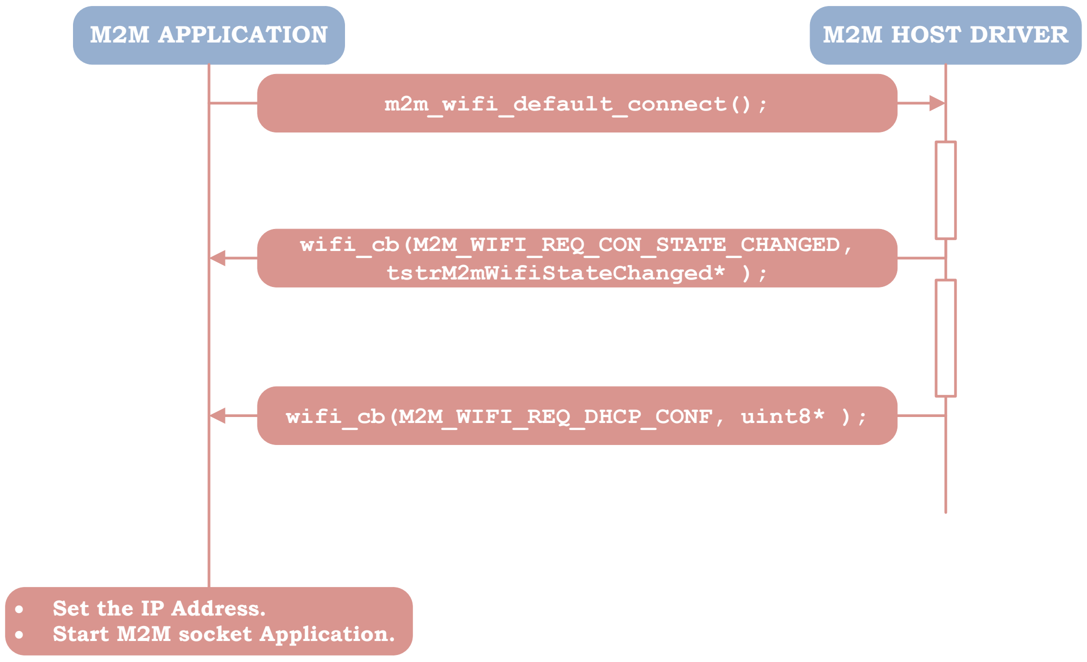
Default connection relies on the connection profiles provisioned into WINC serial flash via the provisioning method described in Chapter 10: Provisioning. Alternatively, connection profiles are created and stored in WINC serial flash when the host MCU application successfully connects once to an AP using the `m2m_wifi_connect` API described in Section 5.3. If there are no cached profiles or if a connection cannot be established with any of the cached profiles, an event of type `M2M_WIFI_RESP_DEFAULT_CONNECT` is delivered to the host driver indicating failure.

Upon successful default connection, the host application can read the current Wi-Fi connection status information by calling `m2m_wifi_get_connection_info` API. The `m2m_wifi_get_connection_info` is an asynchronous API. The actual connection information is provided in the asynchronous event `M2M_WIFI_RESP_CONN_INFO` in Wi-Fi callback. The callback parameter of type `tstrM2MConnInfo` provides information about AP SSID, RSSI (AP received power level), security type, IP address obtained by DHCP.

TIPS A connection profile is cached in the serial flash if and only if the connection is successfully established with the target AP.

The Wi-Fi default connection operation is described in Figure 5-3.

Figure 5-3. Wi-Fi Default Connection


M2M APPLICATION
M2M HOST DRIVER
m2m_wifi_default_connect();
wifi_cb(M2M_WIFI_REQ_CON_STATE_CHANGED, tstrM2mWifiStateChanged*);
wifi_cb(M2M_WIFI_REQ_DHCP_CONF, uint8*);
• Set the IP Address.
• Start M2M socket Application.

5.5 Wi-Fi Security

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

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

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

5.6 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 = AUTH_CREDENTIALS;

     nm_bsp_init();

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

     /* intilize the WINC 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
        */
        m2m_wifi_connect("DEMO_AP", sizeof("DEMO_AP"), M2M_WIFI_SEC_802_1X,
          (uint8*)&gstrCred1x, M2M_WIFI_CH_ALL);

     while(1)
     {

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

6 WINC Socket Programming

6.1 Overview

The WINC socket application programming interface (API) provides a method that allows host MCU application to interact with intranet and remote internet hosts. The WINC sockets API is based on the BSD (Berkeley) sockets. This chapter explains the WINC socket programming and how it differs from regular BSD sockets.

Info: This chapter assumes the reader to have a basic understanding of the BSD sockets, TCP, UDP, and the Internet protocols. Follow the online references provided in link in the name of each topic for more information.

6.1.1 WINC Socket Types

The WINC sockets API provides two types of sockets:

- **Datagram sockets** (connection-less sockets): which use the **UDP** protocol
- **Stream sockets** (connection oriented sockets): which use the **TCP** protocol

6.1.2 Socket Properties

Each WINC socket is identified by a unique combination of:

- **Socket ID**: It is a unique identifier for each socket. This is the return value of the "`socket`" API.
- **Local socket address**: a combination of WINC IP address and port number assigned by the WINC firmware for the socket
- **Protocol**: This is the transport layer protocol, either TCP or UDP

TIPS Note that TCP port 53 and UDP port 53 represent two different sockets.

- **Remote socket address**: applicable only for TCP stream sockets. This is necessary since TCP is connection oriented. Each connection is made to a specific IP address and port number requires a separate socket. The remote socket address can be obtained in the socket event callback as discussed later.

6.1.3 Limitations

- The WINC sockets API support a maximum of 7 TCP sockets and 4 for UDP sockets
- The WINC sockets API support only IPv4. It does not support IPv6

6.2 WINC Sockets API

6.2.1 API Prerequisites

- **The C header file** "`socket.h`": includes all the necessary socket API function declarations. When using any WINC sockets API described in the following sections, the host MCU application should to include the socket.h header file.
- **Initialization**: The WINC socket API shall be initialized once before calling any sockets API function. This is done using the "`socketInit`" API described in the Section 6.2.3.

6.2.2 Non-blocking Asynchronous Socket APIs

Most WINC socket APIs are asynchronous function calls that do not block the host MCU application. The behavior of WINC asynchronous APIs is described in Section 3.4.1.

For example, the host MCU application can register an application-defined socket event callback function using the WINC socket API `registerSocketCallback`. When the host MCU application calls the socket API `connect`, the API returns a zero value (`SUCCESS`) immediately indicating that the request is accepted. The host MCU application must then wait for the WINC socket API to call the registered socket callback when the connection is established or if there was a connection timeout. The socket callback function provides the necessary information to determine if the connection was successful or not.

6.2.3 Socket API Functions

WINC sockets API provide the following functions (see the subsections below).

6.2.3.1 socketInit

The host MCU application must call the API `socketInit` once during initialization. The API is a synchronous API.

6.2.3.2 registerSocketCallback

The `registerSocketCallback` function allows the host MCU application to provide the WINC sockets with application-defined event callbacks for socket operations. The API is a synchronous API. The API registers the following callbacks:

- The socket event callback
- The DNS resolve callback

The socket event callback is an application-defined function that is called by the WINC socket API whenever a socket event occurs. Within this handler, the host MCU application should provide an application-defined logic that handles the events of interest.

The DNS resolve event handler is the application-defined function that is called by the WINC socket API to return the results of `gethostbyname`. By implication, this will only occur after the host MCU application has called the `gethostbyname` function. If successful, the callback provides the IP address for the desired domain name.

6.2.3.3 socket

The `socket` function creates a new socket of a specified type and returns the corresponding socket ID. The API is a synchronous API.

The socket ID is required by most other socket functions and is also passed as an argument to the socket event callback function to identify which socket generated the event.

6.2.3.4 connect

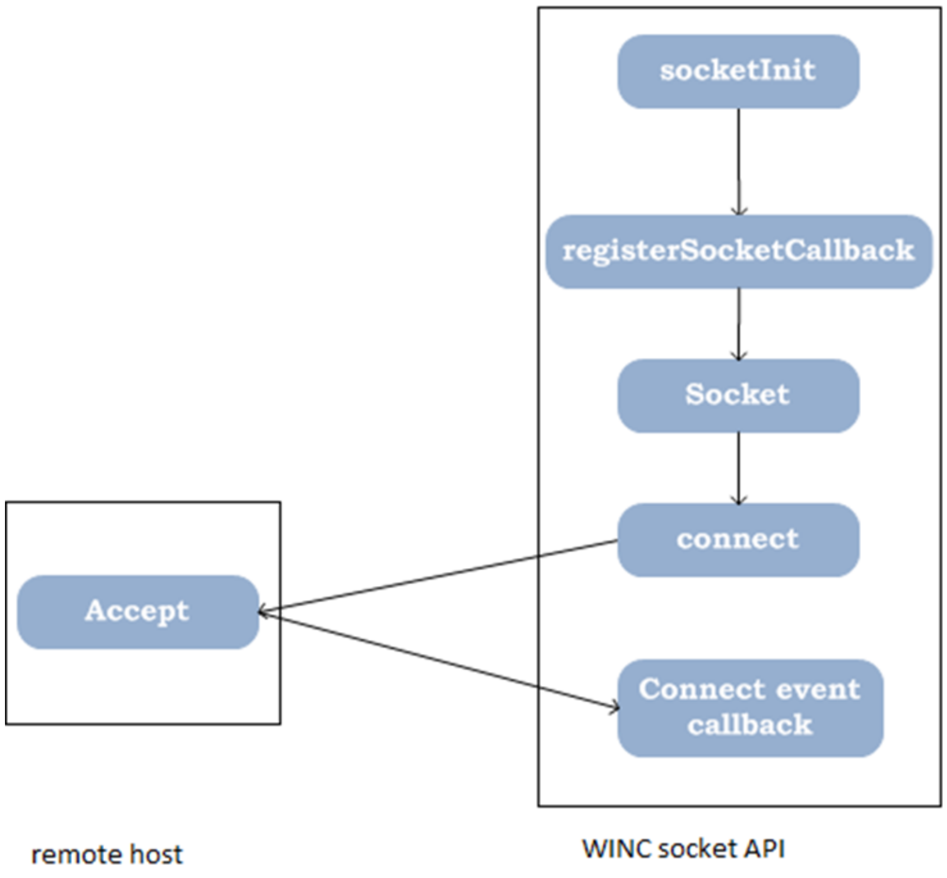
The `connect` function is used with TCP sockets to establish a new connection to a TCP server.

The `connect` function will result in a `SOCKET_MSG_CONNECT` sent to the socket event handler callback upon completion. The connect event will be sent when the TCP server accepts the connection or, if no remote host response is received, after a timeout interval of approximately 30 seconds.

TIPS The `SOCKET_MSG_CONNECT` event callback provides a `tstrSocketConnectMsg` containing an error code. The error code is 0 if the connection was successful or a negative value to indicate an error due to a timeout condition or if `connect` is used with UDP socket.

Figure 6-1 shows the WINC socket API connect to remote server host.

Figure 6-1. TCP Client API Call Sequence



6.2.3.5 bind

The `bind` function can be used for server operation for both UDP and TCP sockets. Its purpose is to associate a socket with an address structure (port number and IP address).

The bind function call will result a `SOCKET_MSG_BIND` event sent to the socket callback handler with the bind status. Calls to `listen`, `send`, `sendto`, `recv`, and `recvfrom` functions should be not be issued until the bind callback is received.

6.2.3.6 listen

The `listen` function is used for server operations with TCP stream sockets. After calling the `listen` API the socket will accept a connection request from a remote host. The `listen` function causes a `SOCKET_MSG_LISTEN` event notification to be sent to the host after the socket port is ready to indicate listen operation success or failure.

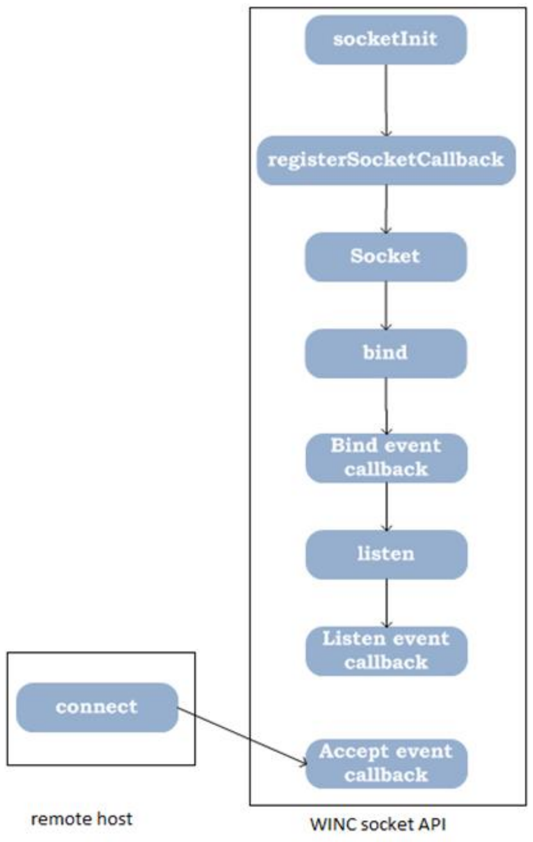
When a remote peer establishes a connection, a `SOCKET_MSG_ACCEPT` event notification is sent to the application.

6.2.3.7 accept

The `accept` function is deprecated and calling this API has no effect. It is kept only for backward compatibility.

TIPS The listen API will implicitly accepts accept connections from a TCP remote peer.

Figure 6-2. TCP Server API Call Sequence



Although the `accept` function is deprecated, the `SOCKET_MSG_ACCEPT` event occurs whenever a remote host connects to the WINC TCP server. The event message will contain the IP address and port number of the connected remote host.

6.2.3.8 send

The `send` function is used by the application to send data to a remote host. The `send` function can be used to send either UDP or TCP data depending on the type of socket. For a TCP socket a connection must be established first. For a UDP socket, the target remote host must be specified as part of the address structure during the `bind` function.

The `send` function will generate a `SOCKET_MSG_SEND` event callback after the data is transmitted to the remote host. For TCP sockets, this event guarantees that the data has been delivered to the remote host TCP/IP stack (the remote application must use the `recv` function to be able to read the data though). For UDP sockets it means that the data has been transmitted but there are no guarantees that the data has arrived to the remote host as per UDP protocol nature. The application is responsible to guarantee data delivery in the UDP sockets case.

The `SOCKET_MSG_SEND` event callback will return the size of the data transmitted if the transmission in the success case and zero or negative value in case of an error.

6.2.3.9 sendto

The `sendto` function is used by the application to send UDP data to a remote host. It can only be used with UDP sockets. The IP address and port of the destination remote host is included as a parameter to the `sendto` function.

The `SOCKET_MSG_SENDTO` event callback returns the size of the data transmitted in the success case and zero or negative value in case of an error.

6.2.3.10 recv / recvfrom

The **recv** and **recvfrom** functions are used to read data from TCP and UDP sockets respectively. Their operation is otherwise identical.

The host MCU application calls the **recv** or **recvfrom** function with a pre allocated buffer. When the **SOCKET_MSG_RECV** or **SOCKET_MSG_RECVFROM** event callback arrives this buffer will contain the received data.

The received data size indicates the status:

- Positive: data received
- Zero: socket connection is terminated
- Negative value: This indicates an error

In the case of TCP sockets, it's recommended to call the **recv** function after each successful socket connection (client or server). Otherwise, received data will be buffered in the WINC firmware wasting the systems resources until the socket is explicitly closed using a **close** function call.

6.2.3.11 close

The **close** function is used to release the resources allocated to the socket and, for a TCP stream socket, also terminate an open connection.

Each call to the **socket** function should be match with a call to the **close** function. In addition, sockets that have been accepted on a server socket port should also be closed using this function.

6.2.3.12 setsockopt

The **setsockopt** function may be used to set socket options to control the socket behavior.

The options supported are:

- **SO_SET_UDP_SEND_CALLBACK**: enables or disables the **send** /**sendto** event callbacks. The user might want to disable the **sendto** event callback for UDP sockets to enhance the socket connection throughput.
- **IP_ADD_MEMBERSHIP**: Used to subscribe to IP Multicast addresses
- **IP_DROP_MEMBERSHIP**: Used to unsubscribe to IP Multicast addresses

TIPS Disabling send/sendto callbacks using **setsockopts** is recommended in high through-put applications.

6.2.3.13 gethostbyname

The **gethostbyname** function is used to resolve a host name (e.g. URL) to a host IP address via the Domain Name System (**DNS**). This is limited for IPv4 addresses only. Operation depends on having configured a DNS server IP address and having access to the DNS hierarchy through the internet.

After **gethostbyname** has been called, a callback to the DNS resolver handler will be made. If the IP address has been determined it will be returned. If it cannot be determined or if the DNS server is not accessible (30 second timeout) an IP address value of zero will be indicated.

WARNING A return IP value of zero indicates an error (e.g. the internet connection is down or DNS is unavailable) and the host MCU application may try the function call **gethostbyname** again later.

6.2.4 Summary

The following table summarizes the WINC socket API and shows its compatibility with BSD socket APIs:

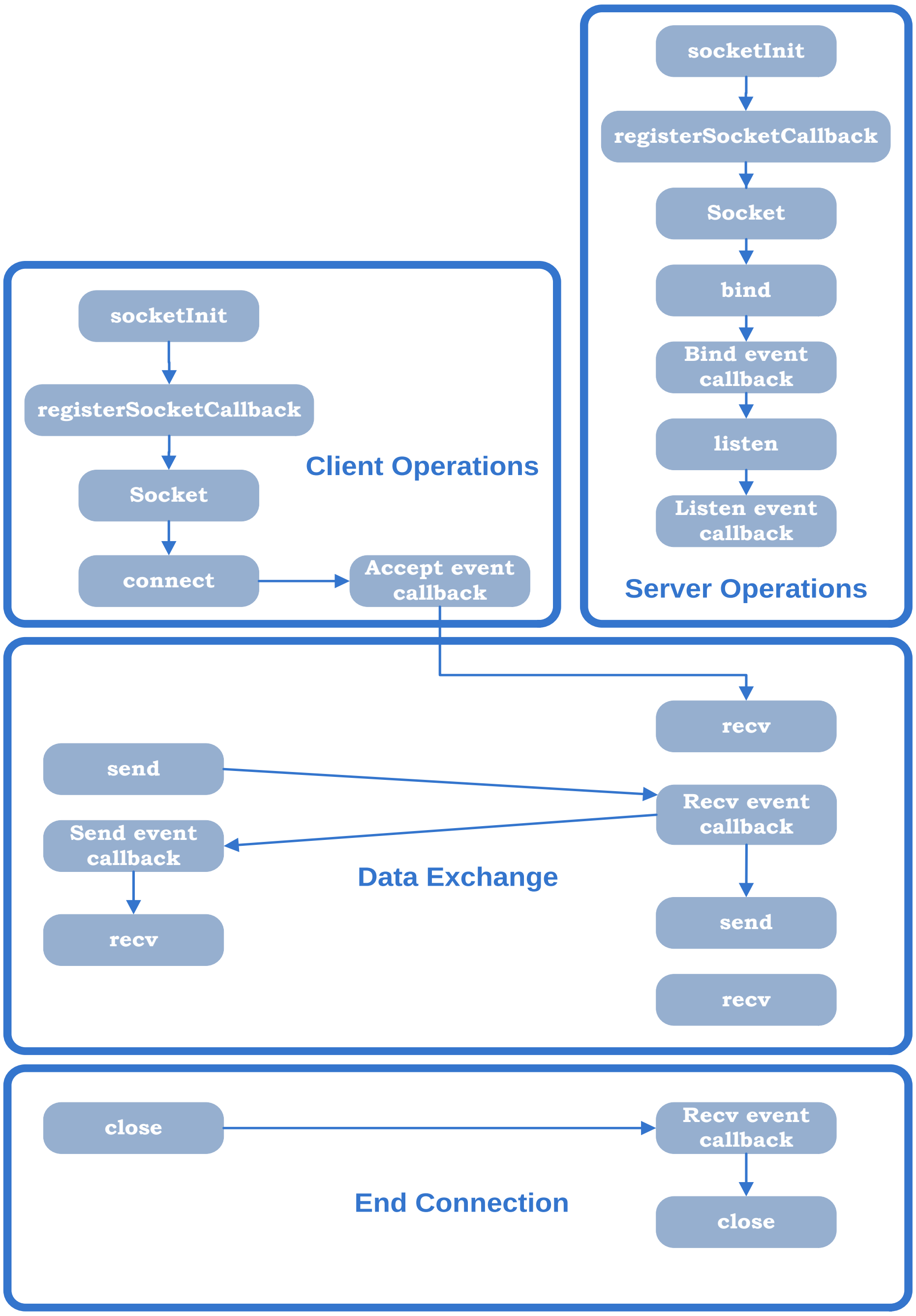
Table 6-1. WINC Socket API Summary

BSD API	WINC API	WINC API Type	Server/Client	TCP/UDP	Brief
socket	socket	Synchronous	Both	both	Creates a new socket.
connect	connect	Asynchronous	Client	TCP	Initialize a TCP connection request to a remote server.
bind	bind	Asynchronous	Server	both	Binds a socket to an address (ad-dress/port).
listen	listen	Asynchronous	Server	TCP	Allow a bound socket to listen to remote connections for its local port.
accept	accept		Depreciated, Implicit accept in listen.		
send	send	Asynchronous	Both	Both	Sends packet.
sendto	sendto	Asynchronous	Both	UDP	Sends packet over UDP sockets.
write		Not supported			
recv	recv	Asynchronous	Both	Both	Receive packet.
recvfrom	recvfrom	Asynchronous	Both	Both	Receive packet.
read		Not supported			
close	close	Synchronous	Both	Both	Terminate TCP connection and re-lease system resources.
gethostbyname	gethostbyname	Asynchronous	Both	Both	Get IP address of certain host name
gethostbyaddr		Not supported			
select		Not supported			
poll		Not supported			
setsockopt	setsockopt	Synchronous	Both	Both	Sets socket option.
getsockopt		Not supported			
htons/ntohs	_htons/_ntohs	Synchronous	Both	Both	Convert a 2-byte integer from the host representation to the Network byte order representation (and vice versa).
htonl/ntohl	_htonl/_ntohl	Synchronous	Both	Both	Convert a 4-byte integer from the host representation to the Network byte order representation (and vice versa).

6.3 Socket Connection Flow

In the following sub-sections the TCP and UDP (client and server) operations are described in details.

Figure 6-3. Typical Socket Connection Flow



6.3.1 TCP Client Operation

Figure 6-4 shows the message flow for transferring data with a TCP client.

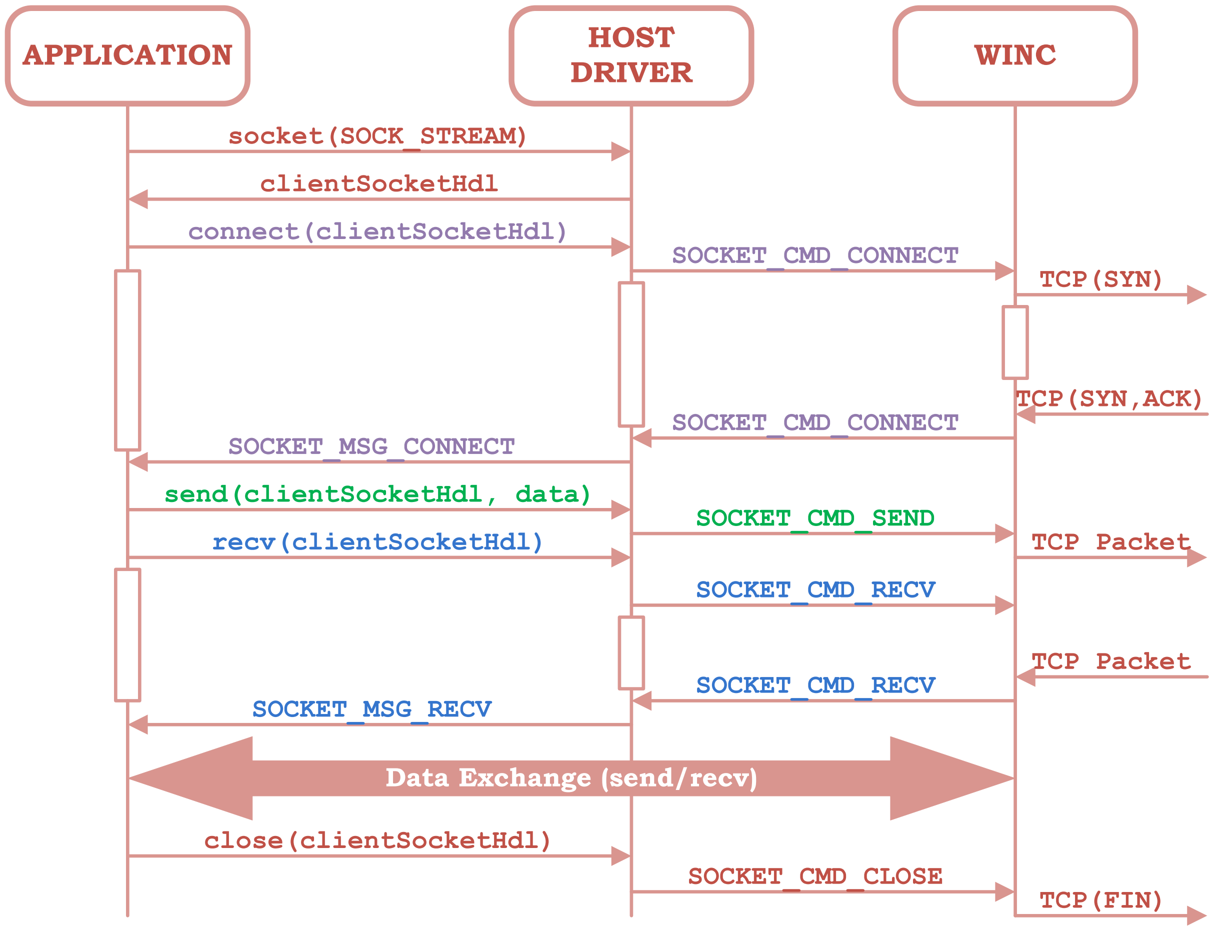


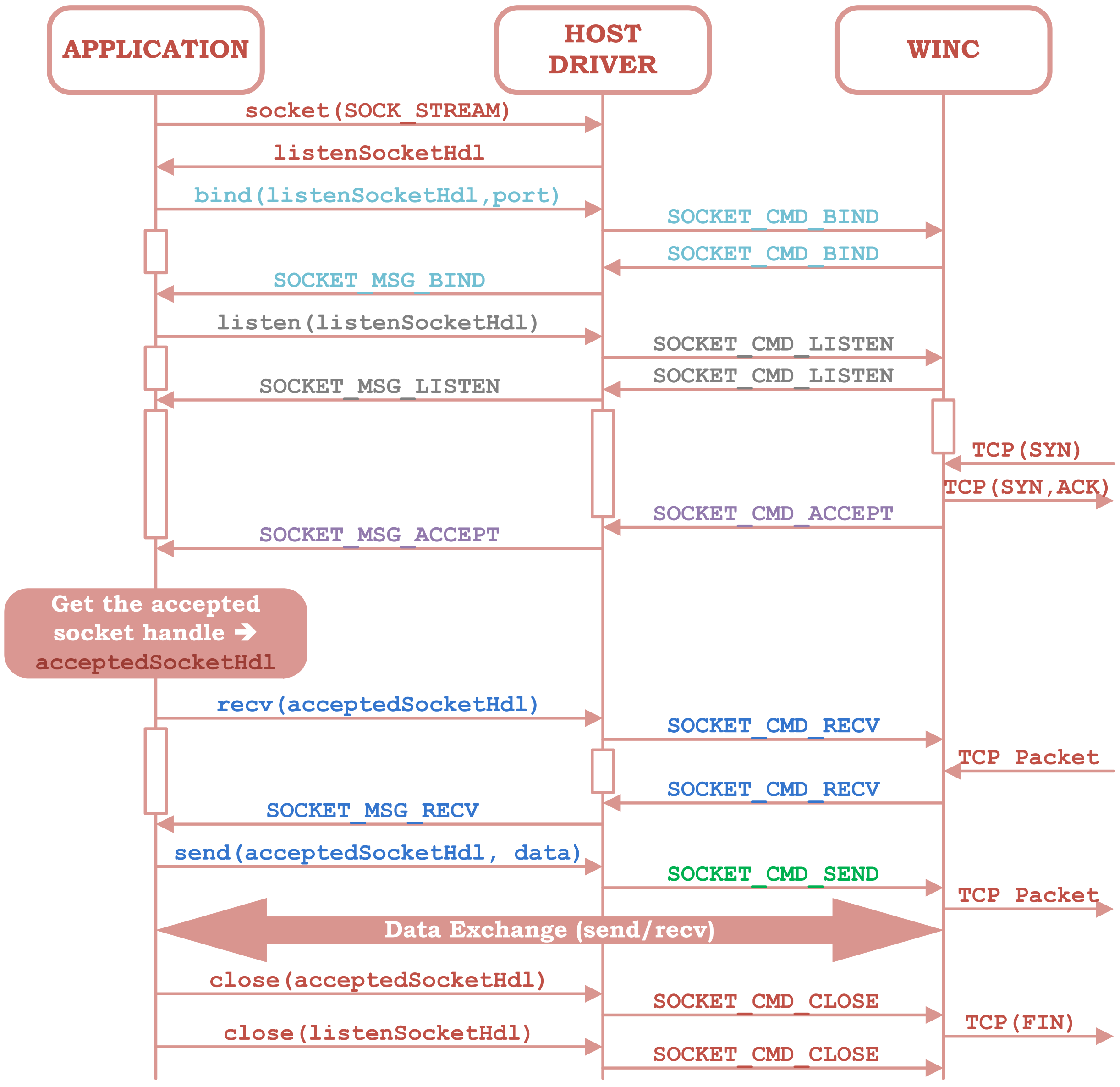
Figure 6-4. TCP Client Sequence Diagram

Notes: 1. The host application must register a socket notification callback function. The function must be of type: `tpfAppSocketCb` and must handle socket event notifications appropriately.

2. If the client knows the IP of the server, it may call `connect` directly as shown in Figure 6-4. If only the server URL is known, then the application should resolve the server URL first calling the `gethostbyname` API.

6.3.2 TCP Server Operation

Figure 6-5. TCP Server Sequence Diagram

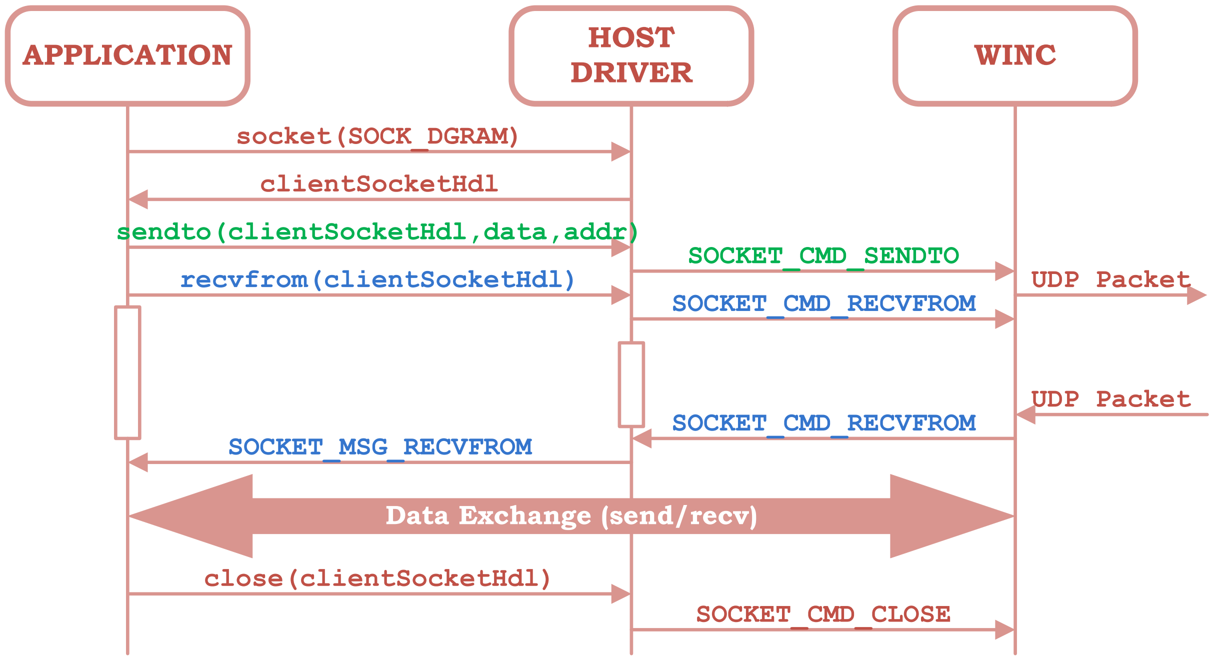


Note: The host application must register a socket notification callback function. The function must be of type: `tpfAppSocketCb` and must handle socket event notifications appropriately.

6.3.3 UDP Client Operation

Figure 6-6 shows the message flow for transferring data with a UDP client.

Figure 6-6. UDP Client Sequence Diagram



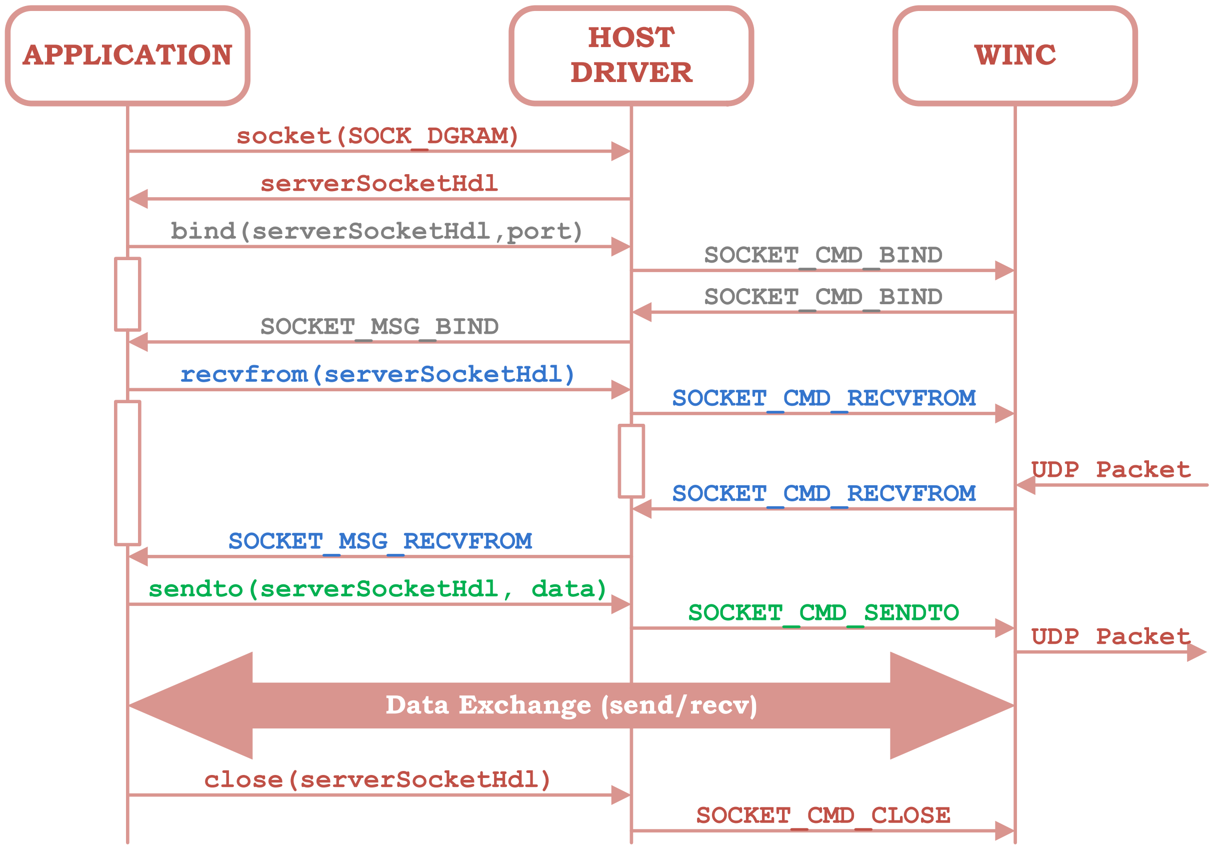
Notes: 1. The first send message must be performed with the `sendto` API with the destination address specified.

2. If further messages will be sent to the same address, the `send` API can be used.
3. `recv` can be used instead of `recvfrom`.

6.3.4 UDP Server Operation

Figure 6-7 shows the message flow for transferring data after establishing a UDP server.

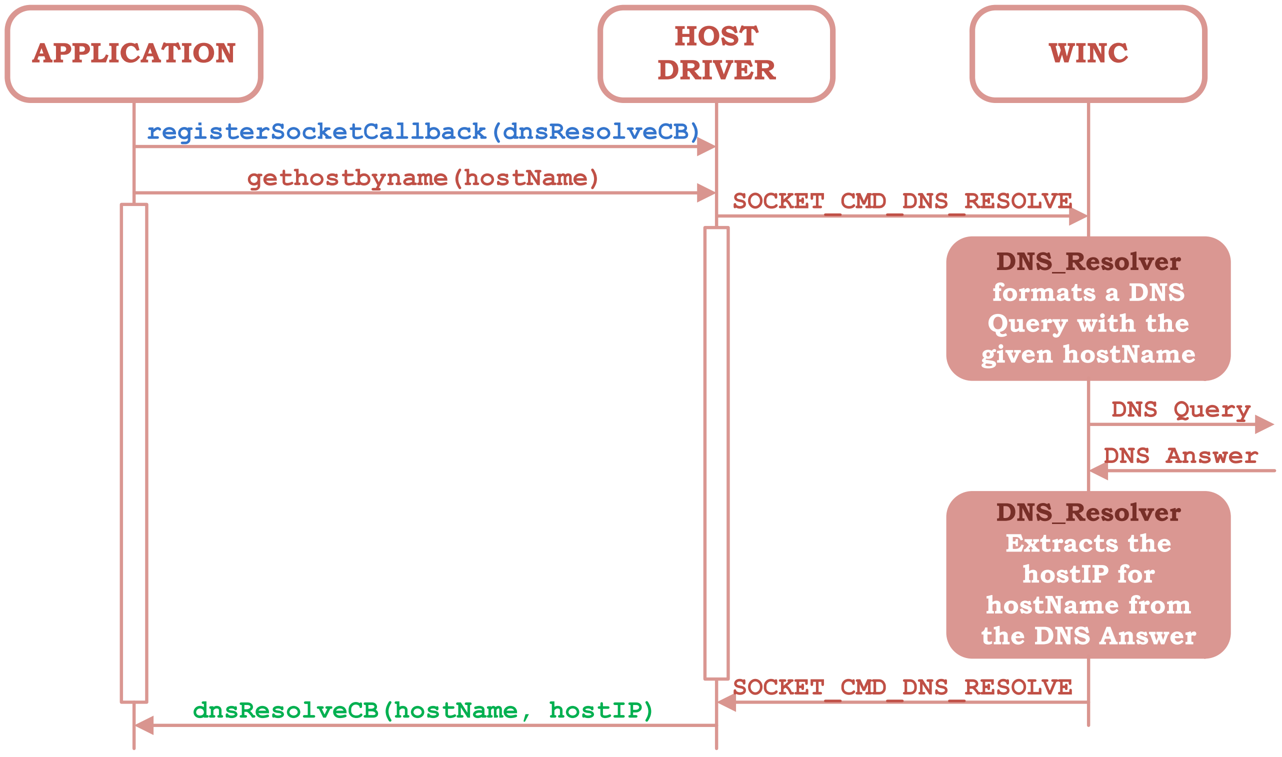
Figure 6-7. UDP Server Sequence Diagram



6.3.5 DNS Host Name Resolution

Figure 6-8 shows the message flow for resolving a URL to and IP address.

Figure 6-8. DNS Resolution Sequence


APPLICATION
HOST DRIVER
WINC
registerSocketCallback(dnsResolveCB)
gethostbyname(hostName)
SOCKET_CMD_DNS_RESOLVE
DNS_Resolver formats a DNS Query with the given hostName
DNS Query
DNS Answer
DNS_Resolver Extracts the hostIP for hostName from the DNS Answer
SOCKET_CMD_DNS_RESOLVE
dnsResolveCB(hostName, hostIP)

Notes: 1. The host application requests to resolve hostname (e.g. www.foobar.com), by calling the function **`gethostbyname`**.

2. Before calling the **`gethostbyname`**, the application must register a DNS response callback function using the function **`registerSocketCallback`**.

3. After the WINC DNS_Resolver module obtains the IP Address (hostIP) corresponding to the given HostName, the **`dnsResolveCB`** will be called with the hostIP.

4. If an error occurs or if the DNS request encounters a timeout, the **`dnsResolveCB`** is called with IP Address value zero indicating a failure to resolve the domain name.

6.4 Example Code

This section provides code samples for different socket applications. For additional socket code example, refer to [R02] WINC1500 Software Programming Guide.

6.4.1 TCP Client Example Code

```
SOCKET         clientSocketHdl;
uint8          rxBuffer[256];


/* Socket event handler.
*/
void tcpClientSocketEventHandler(SOCKET sock, uint8 u8Msg, void * pvMsg)
{
     if(sock == clientSocketHdl)
     {
          if(u8Msg == SOCKET_MSG_CONNECT)
          {
               // Connect Event Handler.
               tstrSocketConnectMsg *pstrConnect = (tstrSocketConnectMsg*)pvMsg;
               if(pstrConnect->s8Error == 0)
               {
                    // Perform data exchange.
                    uint8acSendBuffer[256];
                    uint16 u16MsgSize;

                    // Fill in the acSendBuffer with some data here

                    // send data
                    send(clientSocketHdl, acSendBuffer, u16MsgSize, 0);
                    // Recv response from server.
                    recv(clientSocketHdl, rxBuffer, sizeof(rxBuffer), 0);
               }
               else
               {
                    printf("TCP Connection Failed\n");
               }
          }
          else if(u8Msg == SOCKET_MSG_RECV)
          {
               tstrSocketRecvMsg *pstrRecvMsg = (tstrSocketRecvMsg*)pvMsg;
               if((pstrRecvMsg->pu8Buffer != NULL) && (pstrRecvMsg->s16BufferSize > 0))
               {
                    // Process the received message.

                    // Close the socket.
                    close(clientSocketHdl);
               }
          }
     }
}

// This is the DNS callback. The response of gethostbyname is here.
void dnsResolveCallback(uint8* pu8HostName, uint32 u32ServerIP)
{
     struct sockaddr_in strAddr;

     if(u32ServerIP != 0)
     {
          clientSocketHdl = socket(AF_INET,SOCK_STREAM,u8Flags);
          if(clientSocketHdl >= 0)
          {
               strAddr.sin_family        = AF_INET;
               strAddr.sin_port          = _htons(443);
               strAddr.sin_addr.s_addr   = u32ServerIP;

               connect(clientSocketHdl, (struct sockaddr*)&strAddr, sizeof(struct sockaddr_in));
          }
     }
     else
     {
          printf("DNS Resolution Failed\n");
     }
}

/* This function needs to be called from main function. For the callbacks to be invoked correctly, the API
m2m_wifi_handle_events should be called continuously from main. */
void tcpConnect(char *pcServerURL)
{
     // Initialize the socket layer.
     socketInit();

     // Register socket application callbacks.
     registerSocketCallback(tcpClientSocketEventHandler, dnsResolveCallback);


     // Resolve Server URL.
     gethostbyname((uint8*)pcServerURL);
}
```

6.4.2 TCP Server Example Code

```
SOCKET    listenSocketHdl, acceptedSocketHdl;
uint8     rxBuffer[256];
uint8     bIsfinished = 0;

/* Socket event handler.
*/
void tcpServerSocketEventHandler(SOCKET sock, uint8 u8Msg, void * pvMsg)
{
     if(u8Msg == SOCKET_MSG_BIND)
     {
          tstrSocketBindMsg *pstrBind = (tstrSocketBindMsg*)pvMsg;
          if(pstrBind->status == 0)
          {
               listen(listenSocketHdl, 0);
          }
          else
          {
               printf("Bind Failed\n");
          }
     }
     else if(u8Msg == SOCKET_MSG_LISTEN)
     {
          tstrSocketListenMsg *pstrListen = (tstrSocketListenMsg*)pvMsg;
          if(pstrListen->status != 0)
          {
               printf("listen Failed\n");
          }
     }
     else if(u8Msg == SOCKET_MSG_ACCEPT)
     {
          // New Socket is accepted.
          tstrSocketAcceptMsg *pstrAccept = (tstrSocketAcceptMsg *)pvMsg;
          if(pstrAccept->sock >= 0)
          {
               // Get the accepted socket.
               acceptedSocketHdl = pstrAccept->sock;
```

```
                recv(acceptedSocketHdl, rxBuffer, sizeof(rxBuffer), 0);
            }
            else
            {
                printf("Accept Failed\n");
            }
        }
        else if(u8Msg == SOCKET_MSG_RECV)
        {
            tstrSocketRecvMsg      *pstrRecvMsg = (tstrSocketRecvMsg*)pvMsg;
            if((pstrRecvMsg->pu8Buffer != NULL) && (pstrRecvMsg->s16BufferSize > 0))
            {
                // Process the received message
                // Perform data exchange

                uint8     acSendBuffer[256];
                uint16    u16MsgSize;

                // Fill in the acSendBuffer with some data here

                // Send some data.
                send(acceptedSocketHdl, acSendBuffer, u16MsgSize, 0);

                // Recv response from client.
                recv(acceptedSocketHdl, rxBuffer, sizeof(rxBuffer), 0);

                // Close the socket when finished.
                if(bIsfinished)
                {
                    close(acceptedSocketHdl);
                    close(listenSocketHdl);
                }
            }
        }
}

/* This function needs to be called from main function. For the callbacks to be invoked correctly, the API
m2m_wifi_handle_events should be called continuously from main. */
void tcpStartServer(uint16 u16ServerPort)
{
    struct sockaddr_in  strAddr;

    // Initialize the socket layer.
    socketInit();

    // Register socket application callbacks.
    registerSocketCallback(tcpServerSocketEventHandler, NULL);

    // Create the server listen socket.
    listenSocketHdl = socket(AF_INET, SOCK_STREAM, 0);
    if(listenSocketHdl >= 0)
    {
        strAddr.sin_family              = AF_INET;
        strAddr.sin_port                = _htons(u16ServerPort);
        strAddr.sin_addr.s_addr         = 0; //INADDR_ANY
        bind(listenSocketHdl, (struct sockaddr*)&strAddr, sizeof(struct sockaddr_in));
    }
}
```

6.4.3 UDP Client Example Code

```
SOCKET    clientSocketHdl;
uint8     rxBuffer[256], acSendBuffer[256];

/* Socket event handler */
void udpClientSocketEventHandler(SOCKET sock, uint8 u8Msg, void * pvMsg)
{
    if((u8Msg == SOCKET_MSG_RECV) || (u8Msg == SOCKET_MSG_RECVFROM))
    {
        tstrSocketRecvMsg *pstrRecvMsg = (tstrSocketRecvMsg*)pvMsg;
        if((pstrRecvMsg->pu8Buffer != NULL) && (pstrRecvMsg->s16BufferSize > 0))
        {
            uint16 len;
            // Format a message in the acSendBuffer and put its length in len
            sendto(clientSocketHdl, acSendBuffer, len, 0,
                (struct sockaddr*)&strAddr, sizeof(struct sockaddr_in));

            recvfrom(clientSocketHdl, rxBuffer, sizeof(rxBuffer), 0);
            // Close the socket after finished
            close(clientSocketHdl);
        }
    }
}

/* This function needs to be called from main function. For the callbacks to be invoked correctly, the API
m2m_wifi_handle_events should be called continuously from main.
*/
void udpClientStart(char *pcServerIP)
{
    struct sockaddr_in strAddr;
    // Initialize the socket layer.
    socketInit();

    // Register socket application callbacks.
    registerSocketCallback(udpClientSocketEventHandler, NULL);

    clientSocketHdl = socket(AF_INET,SOCK_STREAM,u8Flags);
    if(clientSocketHdl >= 0)
    {
        uint16 len;
        strAddr.sin_family      = AF_INET;
        strAddr.sin_port        = _htons(1234);
        strAddr.sin_addr.s_addr = nmi_inet_addr(pcServerIP);

        // Format some message in the acSendBuffer and put its length in len
        sendto(clientSocketHdl, acSendBuffer, len, 0, (struct sockaddr*)&strAddr,
                  sizeof(struct sockaddr_in));

        recvfrom(clientSocketHdl, rxBuffer, sizeof(rxBuffer), 0);
    }
}
```

6.4.4 UDP Server Example Code

```
SOCKET    serverSocketHdl;
uint8     rxBuffer[256];

/* Socket event handler.
*/
void udpServerSocketEventHandler(SOCKET sock, uint8 u8Msg, void * pvMsg)
{
     if(u8Msg == SOCKET_MSG_BIND)
     {
          tstrSocketBindMsg *pstrBind = (tstrSocketBindMsg*)pvMsg;
          if(pstrBind->status == 0)
          {
               // call Recv
               recvfrom(serverSocketHdl, rxBuffer, sizeof(rxBuffer), 0);
          }
          else
          {
               printf("Bind Failed\n");
          }
     }
     else if(u8Msg == SOCKET_MSG_RECV)
     {
          tstrSocketRecvMsg *pstrRecvMsg = (tstrSocketRecvMsg*)pvMsg;
          if((pstrRecvMsg->pu8Buffer != NULL) && (pstrRecvMsg->s16BufferSize > 0))
          {
               // Perform data exchange.
               uint8     acSendBuffer[256];
               uint16    u16MsgSize;

               // Fill in the acSendBuffer with some data

               // Send some data to the same address.
               sendto(acceptedSocketHdl, acSendBuffer, u16MsgSize, 0,
                    pstrRecvMsg-> strRemoteAddr, sizeof(pstrRecvMsg-> strRemoteAddr));

               // call Recv
               recvfrom(serverSocketHdl, rxBuffer, sizeof(rxBuffer), 0);

               // Close the socket when finished.
               close(serverSocketHdl);
          }
     }
}

/* This function needs to be called from main function. For the callbacks to be invoked correctly, the API
m2m_wifi_handle_events should be called continuously from main.
*/
void udpStartServer(uint16 u16ServerPort)
{
     struct sockaddr_in  strAddr;
     // Initialize the socket layer.
     socketInit();

     // Register socket application callbacks.
     registerSocketCallback(udpServerSocketEventHandler, NULL);
```

```
    // Create the server listen socket.
    listenSocketHdl = socket(AF_INET, SOCK_DGRAM, 0);
    if(listenSocketHdl >= 0)
    {
        strAddr.sin_family              = AF_INET;
        strAddr.sin_port                = _htons(u16ServerPort);
        strAddr.sin_addr.s_addr         = 0; //INADDR_ANY
        bind(serverSocketHdl, (struct sockaddr*)&strAddr, sizeof(struct sockaddr_in));
    }
}
```

7 Transport Layer Security (TLS)

Transport Layer Security layer sits on top of TCP and provides security services including privacy, authenticity and message integrity. Various security methods are available with TLS in WINC firmware.

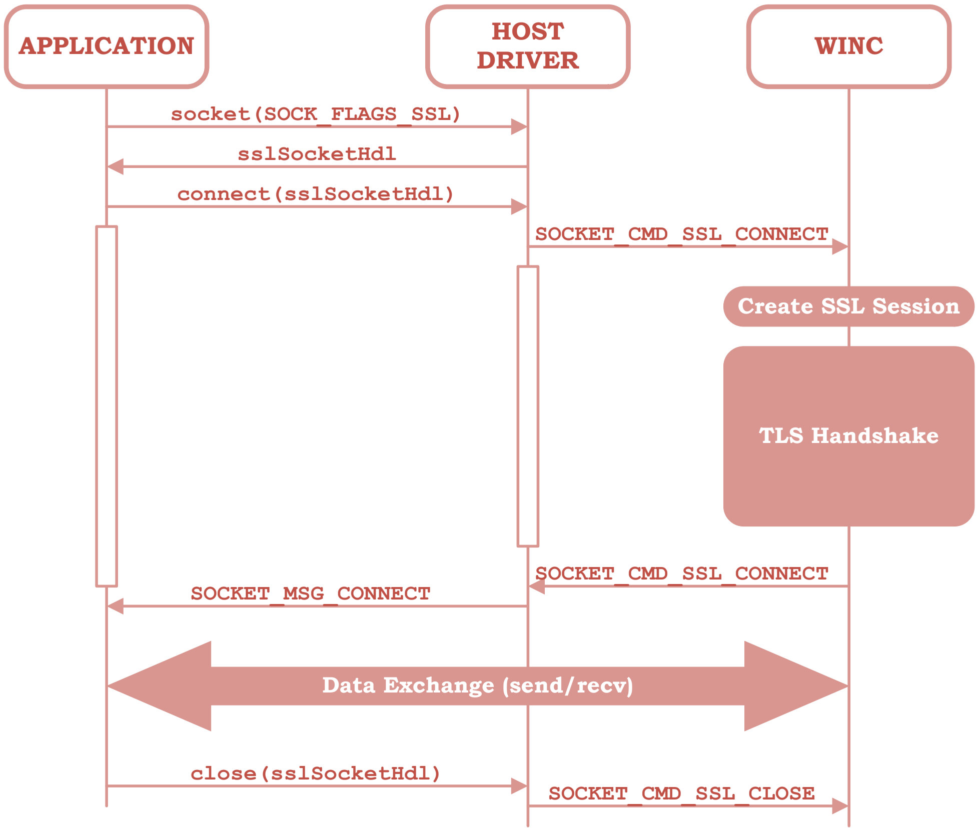
INFO WINC implements the Transport Layer Security protocol TLS v1.0, client mode.

7.1 TLS Connection Establishment

From the application's point of view, the TLS functionality is wrapped behind the socket APIs. This hides the complexity of TLS from the application which could use the TLS in the same fashion as the TCP client. The main difference between TLS sockets and regular TCP sockets is that the application sets the `SOCKET_FLAGS_SSL` while creating the TLS client socket. The detailed sequence of TLS connection establishment is described in Figure 7-1.

TIPS Do not miss both the `SOCKET_FLAGS_SSL` flag and the correct port number in your TLS application. For instance an HTTP client application shall use no flags when calling `socket` API function and `connect` to port 80. The same application source code becomes an HTTPS client application if you use the flag `SOCKET_FLAGS_SSL` and change the port number to `connect` to port 433.

Figure 7-1. TLS Connection Establishment



7.2 Server Certificate Installation

7.2.1 Technical Background

7.2.1.1 Public Key Infrastructure

The TLS security is based on the Public Key Infrastructure PKI, in which:

- A server has its public key stored in a digital certificate with X.509 standard format
- The server must have its X.509 certificate issued by Certificate Authority CA which is in turn might be certified by another CA
- This structure forms a chain of X.509 certificates known as chain of trust
- The top most CA of the Chain is known to be the *Trusted Root Certificate Authority* of the chain

7.2.1.2 TLS Server Authentication

- When a TLS client initiates a connection with a server, the server sends its X.509 certificate chain (may or may not include the root certificate) to the client
- The client must authenticate the Server (verify the Server identity) before starting data exchange
- The client must verify the entire certificate chain and also verify that the root certificate authority of the chain is in the client's trusted root certificate store

7.2.2 Adding a Certificate to the WINC Trusted Root Certificate Store

- Before connecting to a TLS Server, the root certificate of the server must be installed on the WINC1500. If this is not done, the TLS Connection to the server is aborted locally by WINC
- The root certificate must be in **DER** format. If it is not provided in **DER** format, it must be converted before installation. See Appendix A for certificate formats and conversion methods.
- To install the certificate, execute **root_certificate_downloader.exe** with the following syntax:

```
root_certificate_downloader.exe -n N File1.cer File2.cer .. FileN.cer
```

Refer to Appendix C for more information on how to download X509 certificates on WINC serial flash.

7.3 WINC TLS Limitations

7.3.1 Modes of Operation

The current TLS implementation supports TLSv1.0 Client operation only. TLS Server is not supported.

7.3.2 Concurrent Connections

Only 2 TLS concurrent connections are allowed.

7.3.3 Supported Cipher Suites

The current implementation is limited to the following cipher suites:

- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_256_CBC_SHA256

7.3.4 Supported Hash Algorithms

The current implementation supports MD5, SHA-1, and SHA256 hash algorithms.

7.4 SSL Client Code Example

```
SOCKET    sslSocketHdl;
uint8     rxBuffer[256];

/* Socket event handler.
*/
void SSL_SocketEventHandler(SOCKET sock, uint8 u8Msg, void * pvMsg)
{
     if(sock == sslSocketHdl)
     {
          if(u8Msg == SOCKET_MSG_CONNECT)
          {
               // Connect event
               tstrSocketConnectMsg *pstrConnect = (tstrSocketConnectMsg*)pvMsg;
               if(pstrConnect->s8Error == 0)
               {
                    // Perform data exchange.
                    uint8     acSendBuffer[256];
                    uint16    u16MsgSize;
                    // Fill in the acSendBuffer with some data here

                    // Send some data.
                    send(sock, acSendBuffer, u16MsgSize, 0);

                    // Recv response from server.
                    recv(sslSocketHdl, rxBuffer, sizeof(rxBuffer), 0);
               }
               else
               {
                    printf("SSL Connection Failed\n");
               }
          }
          else if(u8Msg == SOCKET_MSG_RECV)
          {
               tstrSocketRecvMsg *pstrRecvMsg = (tstrSocketRecvMsg*)pvMsg;
               if((pstrRecvMsg->pu8Buffer != NULL) && (pstrRecvMsg->s16BufferSize > 0))
               {
                    // Process the received message here

                    // Close the socket if finished.
                    close(sslSocketHdl);
               }
          }
     }
}

/* This is the DNS callback. The response of gethostbyname is here.
*/
void dnsResolveCallback(uint8* pu8HostName, uint32 u32ServerIP)
{
     struct sockaddr_in  strAddr;

     if(u32ServerIP != 0)
     {
          sslSocketHdl = socket(AF_INET,SOCK_STREAM,u8Flags);
          if(sslSocketHdl >= 0)
          {
               strAddr.sin_family         = AF_INET;
               strAddr.sin_port           = _htons(443);
               strAddr.sin_addr.s_addr    = u32ServerIP;
               connect(sslSocketHdl, (struct sockaddr*)&strAddr, sizeof(struct sockaddr_in));
          }
     }
     else
     {
          printf("DNS Resolution Failed\n");
     }
}

/* This function needs to be called from main function. For the callbacks to be invoked correctly, the API
m2m_wifi_handle_events should be called continuously from main.
*/
void SSL_Connect(char *pcServerURL)
{
     // Initialize the socket layer.
     socketInit();

     // Register socket application callbacks.
     registerSocketCallback(SSL_SocketEventHandler, dnsResolveCallback);

     // Resolve Server URL.
     gethostbyname((uint8*)pcServerURL);
}
```

8 Wi-Fi AP Mode

8.1 Overview

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

8.2 Setting WINC AP Mode

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

There are two functions to enable/disable AP mode:

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

For more information about structure and APIs, refer to the API reference in Appendix F API Reference.

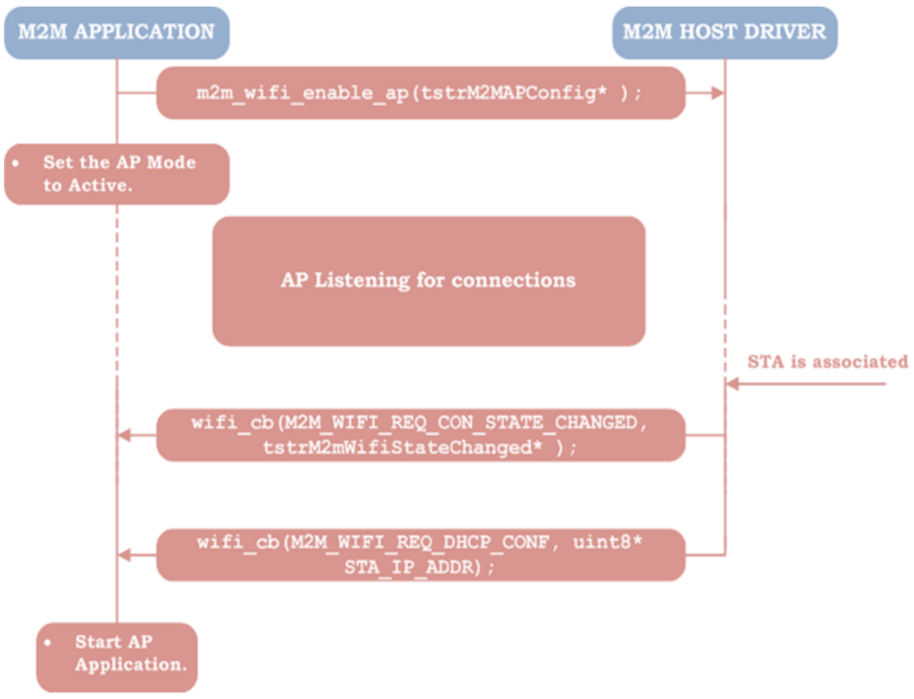
8.3 Limitations

- AP mode supports OPEN and WEP security only
- The AP can only support a single associated station. Further connect attempts will be rejected.
- Concurrency (simultaneous STA/P2P and AP mode) is not supported. Before activating the AP mode, host MCU application should disable the mode currently running.

8.4 Sequence Diagram

Once AP mode has been established, no data interface exists until after a station associates to the AP. Therefore the application needs to wait until it receives a notification via an event callback. This process is shown in Figure 8-1.

Figure 8-1. WINC AP Mode Establishment



M2M APPLICATION
M2M HOST DRIVER
m2m_wifi_enable_ap(tstrM2MAPConfig*);
• Set the AP Mode to Active.
AP Listening for connections
STA is associated
wifi_cb(M2M_WIFI_REQ_CON_STATE_CHANGED, tstrM2mWifiStateChanged*);
wifi_cb(M2M_WIFI_REQ_DHCP_CONF, uint8* STA_IP_ADDR);
• Start AP Application.

8.5 AP Mode Code Example

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

```
#include "m2m_wifi.h"
#include "m2m_types.h"
void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
     switch(u8WiFiEvent)
     {
          case M2M_WIFI_REQ_DHCP_CONF:
          {
               uint8 *pu8IPAddress = (uint8*)pvMsg;
               printf("Associated STA has IP Address \"%u.%u.%u.%u\"\n", pu8IPAddress[0],
                              pu8IPAddress[1], pu8IPAddress[2], pu8IPAddress[3]);
          }
          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, "WINC_SSID");        // Set SSID
          apConfig.u8SsidHide = SSID_MODE_VISIBLE;      // Set SSID to be broadcasted
          apConfig.u8ListenChannel = 1;                 // Set Channel

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

          // IP Address
          apConfig.au8DHCPServerIP[0] = 192;
          apConfig.au8DHCPServerIP[1] = 168;
          apConfig.au8DHCPServerIP[2] = 1;
          apConfig.au8DHCPServerIP[3] = 1;

          // Start AP mode
          m2m_wifi_enable_ap(&apConfig);
          while(1)
          {
               m2m_wifi_handle_events(NULL);
          }
     }
}
```

9 Wi-Fi Direct P2P Mode

9.1 Overview

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

9.2 WINC P2P Capabilities

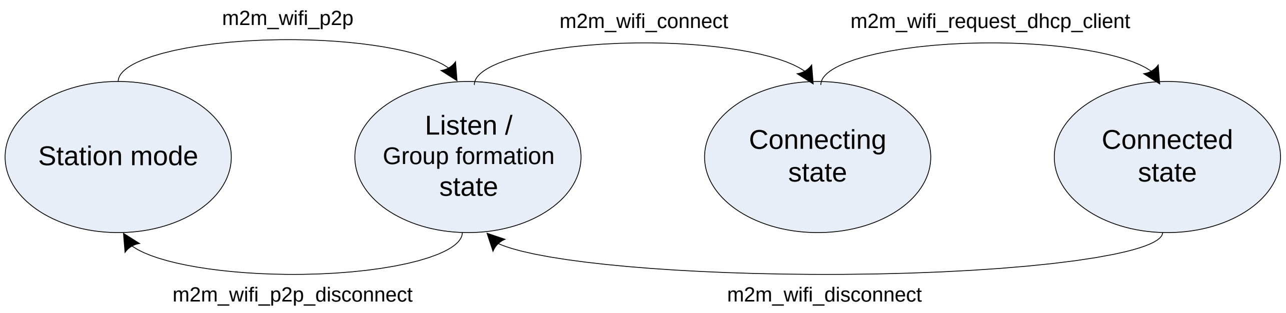
- P2P client mode is supported
- P2P device discovery
- P2P listen state

9.3 WINC P2P Limitations

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

9.4 WINC P2P States

Figure 9-1. P2P Mode State Diagram



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

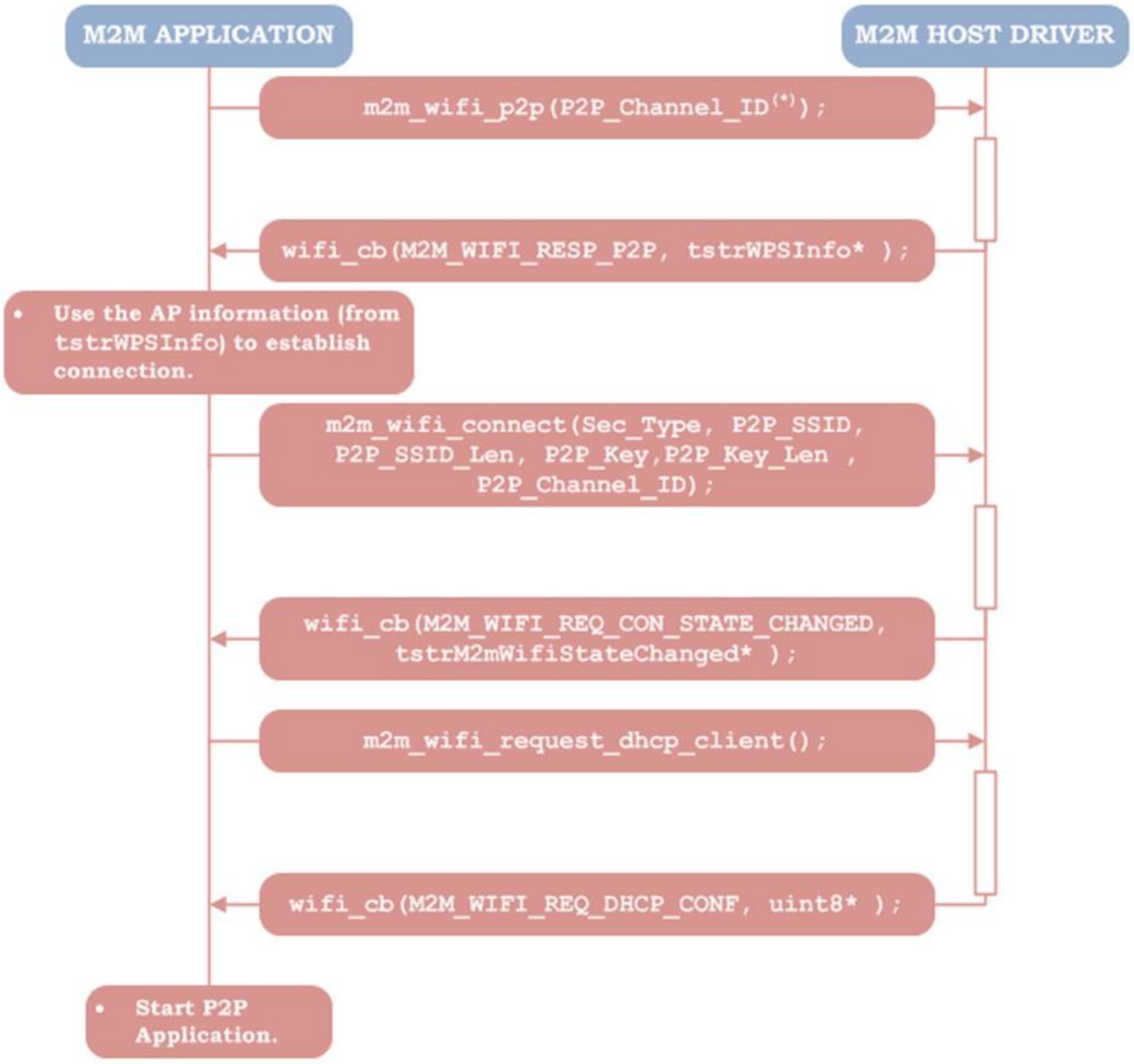
9.5 WINC P2P Listen State

The WINC 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 WINC firmware in the listening state at a certain listen channel defined through the `MAIN_WLAN_CHANNEL`.

9.6 WINC P2P Connection State

The peer P2P device will initiate group owner (GO) negotiation and the WINC device will always decline to become group owner. Assuming the peer device **will take** the **GO role**, the WINC will then perform a WPS exchange to establish a mutual shared key. The information about the remote device (which is now acting as an AP), is received by an event via the Wi-Fi callback with the P2P GO information. The Application can then use this information to connect to the GO in the same manner that the WINC connects to any conventional AP (using the **m2m_wifi_connect** function). The following sequence diagram shows the above connection flow for the WINC P2P device:

Figure 9-2. P2P Connection Flow


M2M APPLICATION
M2M HOST DRIVER
m2m_wifi_p2p(P2P_Channel_ID(*));
wifi_cb(M2M_WIFI_RESP_P2P, tstrWPSInfo*);
• Use the AP information (from tstrWPSInfo) to establish connection.
m2m_wifi_connect(Sec_Type, P2P_SSID, P2P_SSID_Len, P2P_Key,P2P_Key_Len , P2P_Channel_ID);
wifi_cb(M2M_WIFI_REQ_CON_STATE_CHANGED, tstrM2mWifiStateChanged*);
m2m_wifi_request_dhcp_client();
wifi_cb(M2M_WIFI_REQ_DHCP_CONF, uint8*);
• Start P2P Application.

9.7 WINC P2P Disconnection State

To terminate the P2P connection, the GO can send a disconnection that is received through the Wi-Fi callback with the event **M2M_WIFI_RESP_CON_STATE_CHANGED**. However, this will not change the P2P listen state, unless a p2p disable request is made.

9.8 P2P Mode Code Example

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

#define MAIN_WLAN_DEVICE_NAME    "WINC1500_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) {
                m2m_wifi_request_dhcp_client();
            } else if (pstrWifiState->u8CurrState == M2M_WIFI_DISCONNECTED) {
                printf("Wi-Fi disconnected\r\n");
            }
            break;
        }

        case M2M_WIFI_REQ_DHCP_CONF:
        {
            uint8_t *pu8IPAddress = (uint8_t *)pvMsg;
            printf("Wi-Fi connected\r\n");
            printf("Wi-Fi IP is %u.%u.%u.%u\r\n",
                    pu8IPAddress[0], pu8IPAddress[1], pu8IPAddress[2], pu8IPAddress[3]);
            break;
        }

        default:
        {
            break;
        }
    }
}

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

    // Initialize the BSP.
    nm_bsp_init();

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

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

    // Set device name to be shown in peer device.
    ret = m2m_wifi_set_device_name((uint8_t *)MAIN_WLAN_DEVICE_NAME,
                                             strlen(MAIN_WLAN_DEVICE_NAME));
    if (M2M_SUCCESS != ret) {
        printf("main: m2m_wifi_set_device_name call error!\r\n");
        while (1) {
        }
    }

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

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

10 Provisioning

For normal operation the WINC device needs certain parameters to be loaded. In particular, when operating in station mode, it needs to know the identity (SSID) and credentials of the access point to which it will connect. The entry of this information is facilitated through the following provisioning steps.

The current WINC software supports two methods of provisioning:

- HTTP-based (browser) provisioning while WINC is in AP mode
- Wi-Fi Protected Setup (WPS)

10.1 Limitations

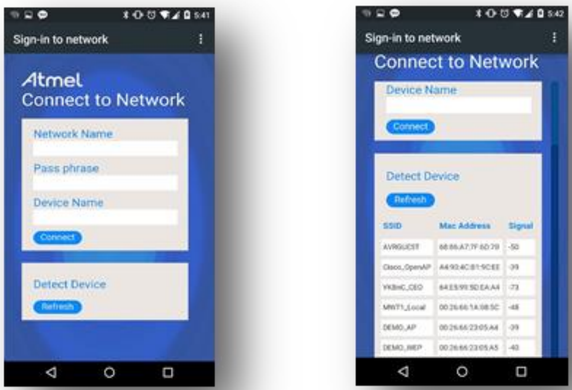
The current implementation of the HTTP Provisioning has the following limitations:

- WINC AP limitations apply in provisioning mode. See Section 8.3: Limitations for a list of AP mode limitations.
- Provisioning uses AP mode with open security. No Wi-Fi security nor application level security (e.g. TLS) is used and therefore the AP credentials entered by the user are sent on the clear and can be seen by eavesdroppers.
- The WINC Provisioning home page is a static HTML page. No server-side scripting allowed in WINC HTTP server.
- Only APs with WPA-personal security (passphrase based) and no security (Open network) can be provisioned. WEP and WPA-Enterprise APs cannot be provisioned.
- The Provisioning is responsible to deliver the connection parameters to the application, the connection procedure and the connection parameters validity its application responsibility

10.2 HTTP Provisioning

In this method, the WINC is placed in AP mode and another device with a browser capability (mobile phone, tablet, PC…etc.) is instructed to connect to the WINC HTTP server. Once connected, the desired configuration can be entered.

The HTTP provisioning home page is as shown in Figure 10-1.



SSID	Mac Address	Signal
AVRGUEST	88:86:A7:7F:6D:78	-50
Cisco_OpenAP	A4:93:4C:B1:9C:EE	-39
YKBnC_CEO	64:E5:99:5D:EA:A4	-73
MWT1_Local	00:26:66:1A:08:5C	-48
DEMO_AP	00:26:66:23:05:A4	-39
DEMO_WEP	00:26:66:23:05:A5	-40

Figure 10-1. WINC HTTP Provisioning Page

10.2.1 Provisioning Control Flow

Figure 10-2. HTTP Provisioning Sequence Diagram

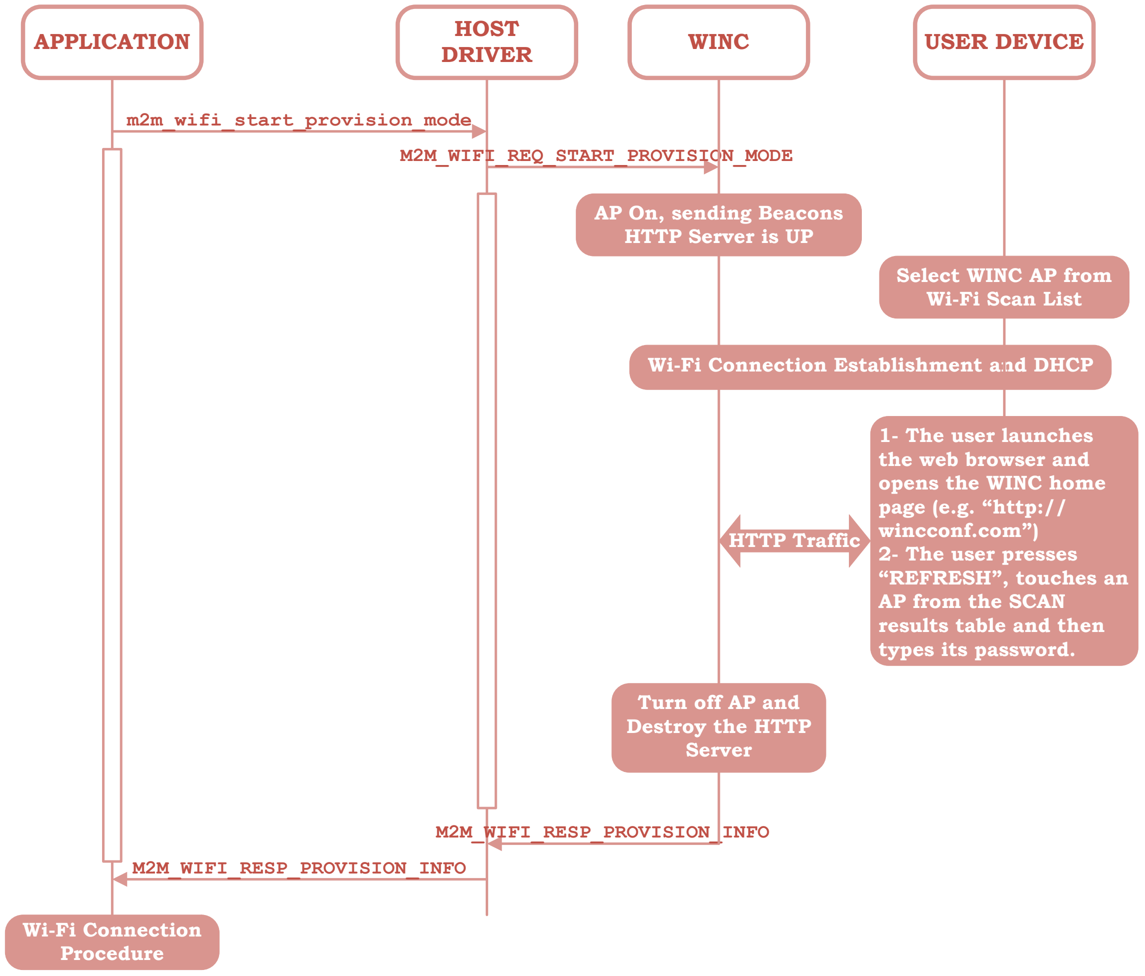

APPLICATION
HOST DRIVER
WINC
USER DEVICE
m2m_wifi_start_provision_mode
M2M_WIFI_REQ_START_PROVISION_MODE
AP On, sending Beacons HTTP Server is UP
Select WINC AP from Wi-Fi Scan List
Wi-Fi Connection Establishment and DHCP
HTTP Traffic
1- The user launches the web browser and opens the WINC home page (e.g. "http:// wincconf.com")
2- The user presses "REFRESH", touches an AP from the SCAN results table and then types its password.
Turn off AP and Destroy the HTTP Server
M2M_WIFI_RESP_PROVISION_INFO
M2M_WIFI_RESP_PROVISION_INFO
Wi-Fi Connection Procedure

Figure 10-2 shows the provisioning operation for a WINC device. The detailed steps are described as follows:

1. The WINC device starts the HTTP Provisioning mode.
2. A user with a smart phone finds the WINC AP SSID in the Wi-Fi search list.
3. The user connects to the WINC AP.
4. The user launches the web browser and writes the WINC home page in the address bar.
5. If the HTTP redirect is enabled at the WINC, any web address the WINC home page will load automatically (like connecting to a public Wi-Fi hotspot). Some phones will display a notification message "sign in to Wi-Fi networks?" which, when accepted, will load the WINC home pages load automatically. The WINC home page (shown in Figure 10-1) will appear on the browser.
6. To discover the list of Wi-Fi APs in the area, the user can press "Refresh".
7. The desired AP is then selected from the search list (by one click or one touch) and its name will appear automatically in the "Network Name" text box.
8. Then the user must then enter the correct AP passphrase (for WPA/WPA2 personal security) in the "Pass Phrase" text box. If the AP is not secured (Open network) the field should be left empty.
9. A WINC device name may be optionally configured if desired by the user in the "Device Name" text box.
10. The user presses "Connect".

The WINC will then turn off AP mode and start connecting to the provisioned AP.

10.2.2 HTTP Redirect Feature

The WINC HTTP provisioning server supports the HTTP redirect feature, which forces all HTTP traffic originating from the associated user device to be redirected to the WINC provisioning home page.

This simplifies the mechanism of loading the provisioning page instead of typing the exact web address of the http provisioning server.

To enable this feature, the redirect flag should be set when calling the API **m2m_wifi_start_provision_mode**. See the below code example for details.

10.2.3 Provisioning Code Example

```
void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    if(u8WiFiEvent == M2M_WIFI_RESP_PROVISION_INFO)
    {
        tstrM2MProvisionInfo *provInfo = (tstrM2MProvisionInfo*)pvMsg;
        if(provInfo->u8Status == M2M_SUCCESS)
        {
            // connect to the provisioned AP.
            m2m_wifi_connect((char*)provInfo->au8SSID, strlen(provInfo ->au8SSID),
                             provInfo->u8SecType, provInfo->au8Password, M2M_WIFI_CH_ALL);
            printf("PROV SSID : %s\n", provInfo->au8SSID);
            printf("PROV PSK  : %s\n", provInfo->au8Password);
        }
        else
        {
            printf("(ERR) Provisioning Failed\n");
        }
    }
}

int main()
{
    tstrWifiInitParam  param;

    // Platform specific initializations.

    // Driver initialization.
    param.pfAppWifiCb  = wifi_event_cb;
    if(!m2m_wifi_init(&param))
    {
        tstrM2MAPConfig apConfig;
        uint8 bEnableRedirect = 1;

        strcpy(apConfig.au8SSID, "WINC_AP");
        apConfig.u8ListenChannel = 1;
        apConfig.u8SecType       = M2M_WIFI_SEC_OPEN;
        apConfig.u8SsidHide      = 0;

        // IP Address
        apConfig.au8DHCPServerIP[0]    = 192;
        apConfig.au8DHCPServerIP[1]    = 168;
        apConfig.au8DHCPServerIP[2]    = 1;
        apConfig.au8DHCPServerIP[0]    = 1;

        m2m_wifi_start_provision_mode(&apConfig, "atmelconfig.com", bEnableRedirect);
        while(1)
        {
            m2m_wifi_handle_events(NULL);
        }
    }
}
```

10.3 Wi-Fi Protected Setup (WPS)

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

10.3.1 WPS Configuration Methods

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

1. **PBC (Push button) method**: a physical button is pressed on the AP which puts the AP into WPS mode for a limited period of time. WPS is initiated on the WINC1500 by calling **`m2m_wifi_wps`** with input parameter **`WPS_PBC_TRIGGER`**.
2. **PIN method**: The AP is always available for WPS initiation but requires proof that the user has knowledge of an 8 digit PIN, usually printed on the body of the AP. Because WINC 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 WINC device. Some APs allow the PIN to be changed through configuration. WPS is initiated on the WINC1500 by calling **`m2m_wifi_wps`** with input parameter **`WPS_PIN_TRIGGER`**. Given the difficulty of this approach it is not recommend for most applications.

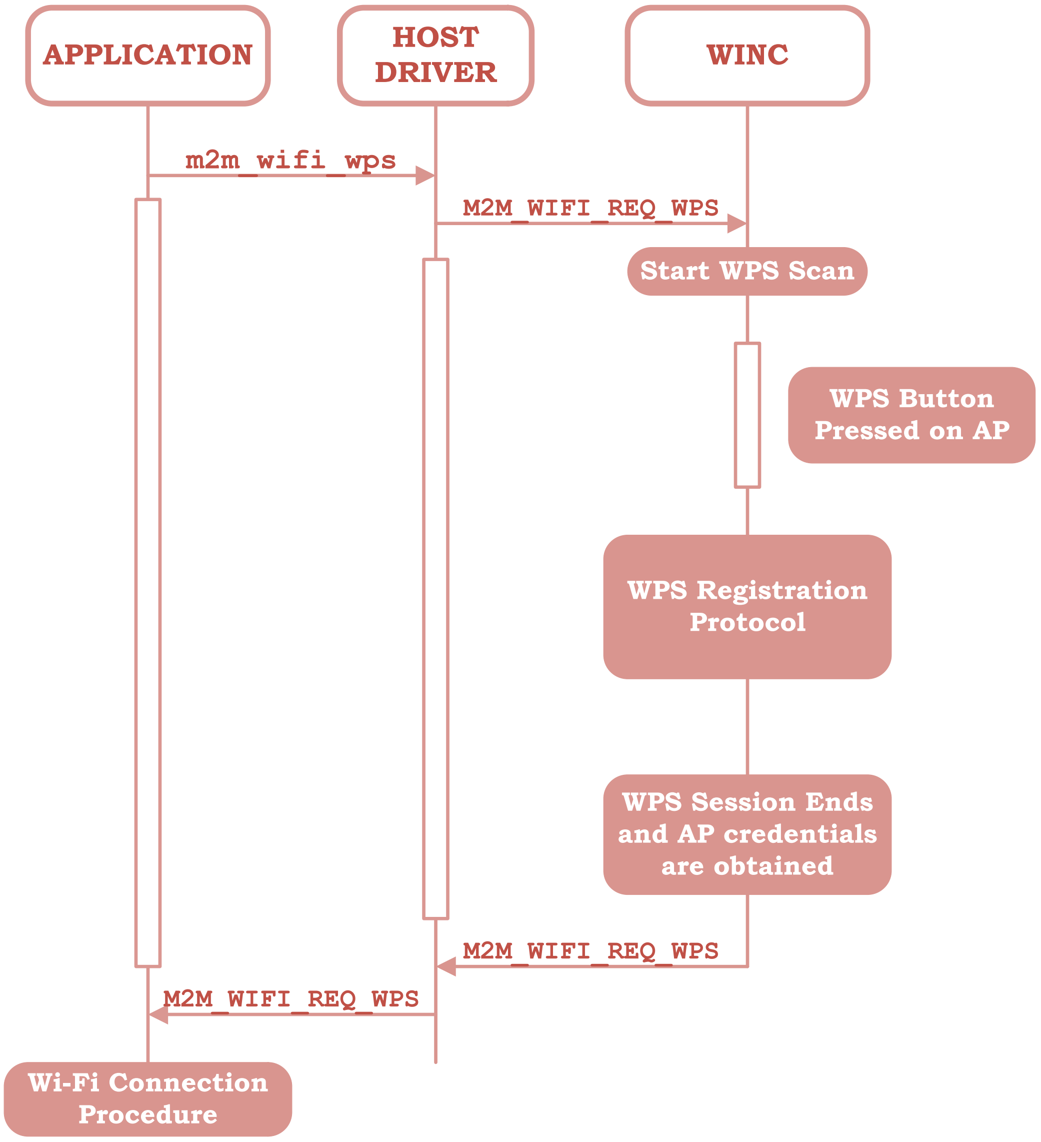
The flow of messages and actions for WPS operation is shown in Figure 10-3.

10.3.2 WPS Limitations

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

10.3.3 WPS Control Flow

Figure 10-3. WPS Operation for Push Button Trigger



10.3.4 WPS Code Example

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

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

int main()
{
    tstrWifiInitParam   param;

    // Platform specific initializations.

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

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

11 Over-The-Air Upgrade

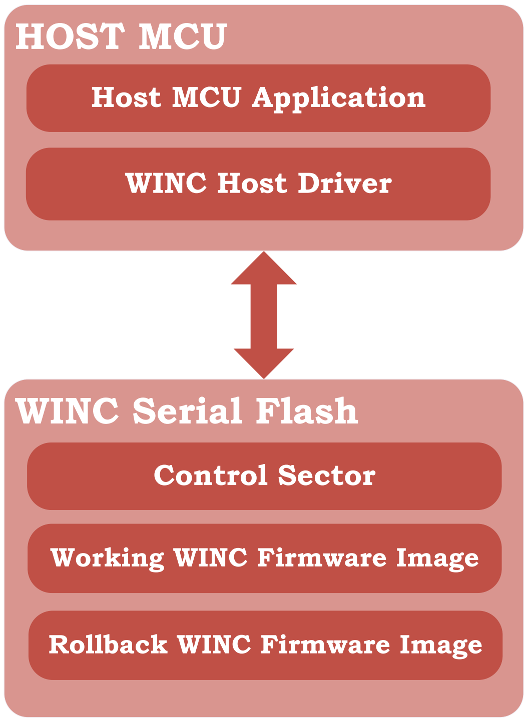
11.1 Overview

The WINC supports over-the-air upgrade of firmware on internal serial flash, No host flash memory resources are required to store the firmware. The WINC uses an internal HTTP client (no HTTPS supported) to retrieve the firmware from a remote server.

11.2 OTA Image Architecture

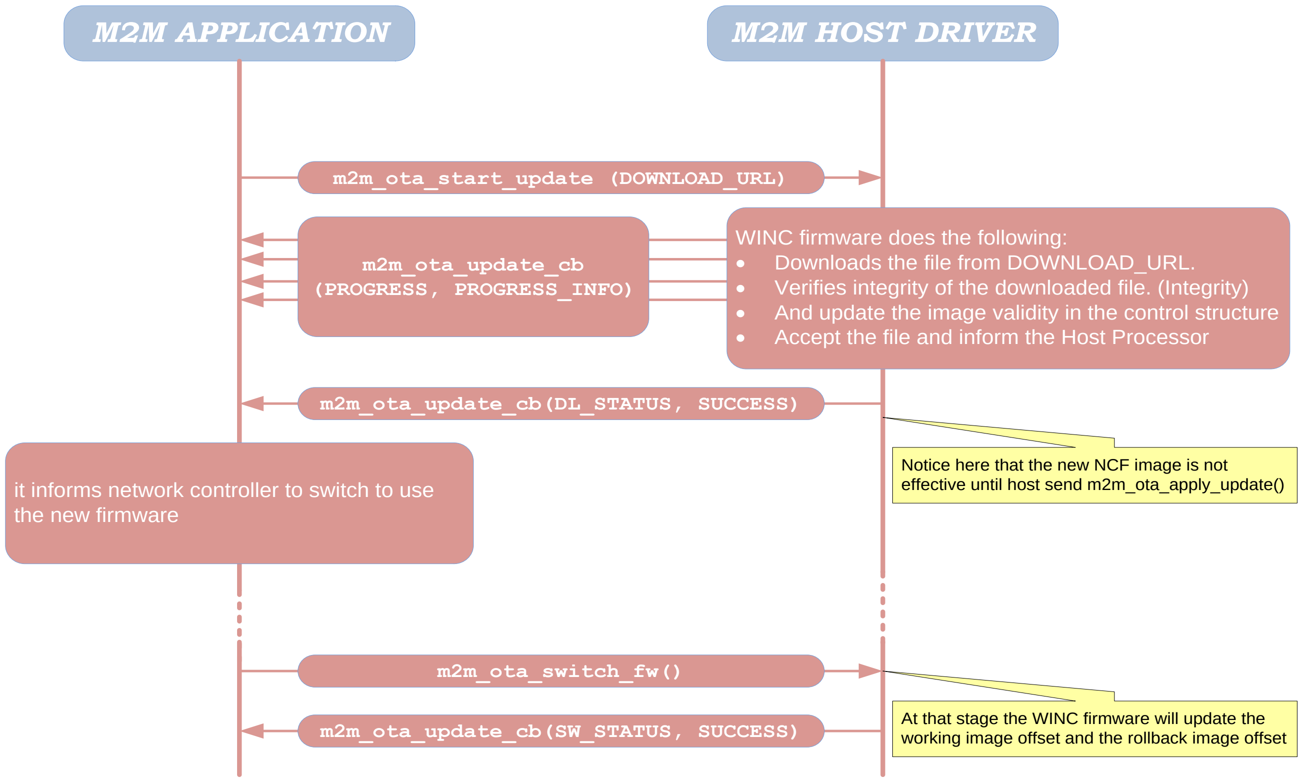
The WINC serial flash can store two copies of the firmware image: a working image and a rollback image. Upon first-time boot, the working image is the factory image and the rollback image is invalid (empty). The WINC has insufficient internal memory to save the whole image in RAM during an OTA upgrade so each block of downloaded data is written to the flash as it is received. In the event that the OTA fails, the existing (Working) image is retained and the rollback image is invalidated. If the transfer succeeds, the flash control structure is updated to reflect a new working image and the existing image is marked as a valid rollback image.

Figure 11-1. OTA Image Organization



11.3 OTA Download Sequence Diagram

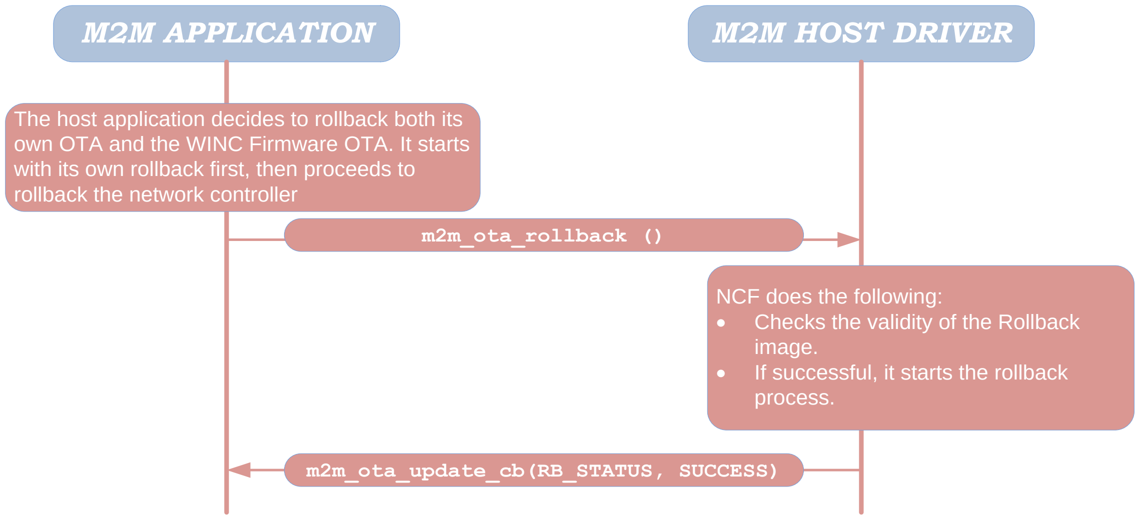
Figure 11-2. OTA Image Download and Install



M2M APPLICATION
M2M HOST DRIVER
m2m_ota_start_update (DOWNLOAD_URL)
m2m_ota_update_cb (PROGRESS, PROGRESS_INFO)
WINC firmware does the following:
• Downloads the file from DOWNLOAD_URL.
• Verifies integrity of the downloaded file. (Integrity)
• And update the image validity in the control structure
• Accept the file and inform the Host Processor
m2m_ota_update_cb(DL_STATUS, SUCCESS)
Notice here that the new NCF image is not effective until host send m2m_ota_apply_update()
it informs network controller to switch to use the new firmware
m2m_ota_switch_fw()
m2m_ota_update_cb(SW_STATUS, SUCCESS)
At that stage the WINC firmware will update the working image offset and the rollback image offset

11.4 OTA Firmware Rollback

Figure 11-3. OTA Image Rollback Sequence



M2M APPLICATION
M2M HOST DRIVER
The host application decides to rollback both its own OTA and the WINC Firmware OTA. It starts with its own rollback first, then proceeds to rollback the network controller
m2m_ota_rollback ()
NCF does the following:
• Checks the validity of the Rollback image.
• If successful, it starts the rollback process.
m2m_ota_update_cb(RB_STATUS, SUCCESS)

11.5 OTA Limitations

- No HTTPS file download supported
- Rollback is allowed only after successful OTA download
- Rollback image is overwritten by any new successful or failed OTA attempt
- A WINC device with 4Mb flash is required to perform OTA

11.6 OTA Code Example

INFO For more details on the code examples refer to [R02].

```
/*!<OTA update callback typedef> */
static void OtaUpdateCb(uint8 u8OtaUpdateStatusType ,uint8 u8OtaUpdateStatus)
{
        if(u8OtaUpdateStatusType == DL_STATUS) {
                if(u8OtaUpdateStatus == OTA_STATUS_SUCSESS) {
                        //switch to the upgraded firmware
                        m2m_ota_switch_firmware();
                }
        }
        else if(u8OtaUpdateStatusType == SW_STATUS) {
                if(u8OtaUpdateStatus == OTA_STATUS_SUCSESS) {
                        M2M_INFO("Now OTA suceesfully done");
                        //start the host SW upgrade then system reset is required (Reintilize the
driver)
                }
        }
}
void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
        case M2M_WIFI_REQ_DHCP_CONF:
        {
                //after suceesfull connection, start the over air upgrade
                m2m_ota_start_update(OTA_URL);
        }
        break;
        default:
        break;
}
int main (void)
{
        tstrWifiInitParam param;
        tstr1xAuthCredentials gstrCred1x    = AUTH_CREDENTIALS;
        nm_bsp_init();
        m2m_memset((uint8*)&param, 0, sizeof(param));
        param.pfAppWifiCb = wifi_event_cb;

        //intilize the WINC Driver
        ret = m2m_wifi_init(&param);
        if (M2M_SUCCESS != ret)
        {
                M2M_ERR("Driver Init Failed <%d>\n",ret);
                while(1);
        }
        //intilize the ota module
        m2m_ota_init(OtaUpdateCb,NULL);
        //connect to AP that provide connection to the OTA server
        m2m_wifi_default_connect();
        while(1) {
                        while(m2m_wifi_handle_events(NULL) != M2M_SUCCESS) {}
        }
}
```

12 Multicast Sockets

12.1 Overview

The purpose of the multicast filters is to provide the ability to send/receive messages to/from multicast addresses. This feature is useful for one-to-many communication over networks, whether it's intended to send Internet Protocol (IP) datagrams to a group of interested receivers in a single transmission, participate in a zero-configuration networking or listening to a multicast stream or any other application.

12.2 How to use Filters

Whenever the application wishes to use a multicast IP address, for either sending or receiving, a filter is needed. The application can establish this through setting the **`IP_ADD_MEMBERSHIP`** option for the required socket accompanied by the multicast address that the application wants to use. If subsequently the host wants to stop receiving the multicast stream it should set the **`IP_DROP_MEMBERSHIP`** option for the required socket accompanied with the multicast address.

Adding or removing a multicast address filter will cause WINC chip firmware to add/remove both MAC layer filter and IP layer filter in order to pass or /prevent messages from reaching to host.

12.3 Multicast Socket Code Example

In order to illustrate the functionality, a simple example is implemented where the host application responds to mDNS (Multicast Domain Name System) queries sent from a Computer/Mobile application. The Computer/Mobile is looking for devices which support the zero configuration service as indicated by an mDNS response. The WINC responds, announcing its presence and its capability of sending and receiving multicast messages.

The example consists of a UDP server that binds on port 5353 (mDNS port) and waits for messages, parsing them and replying with a previously saved response message.

- Server Initialization

```
void MDNS_ServerInit()
{
        tstrSockAddr    strAddr ;
        unsigned int MULTICAST_IP =  0xE00000FB; //224.0.0.251
        socketInit();
        dns_server_sock = socket( AF_INET, SOCK_DGRAM,0);
        MDNS_INFO("DNS_server_init \n");
        setsockopt(dns_server_sock,1,IP_ADD_MEMBERSHIP,&MULTICAST_IP,sizeof(MULTICAST_IP));
        strAddr.u16Port =HTONS(MDNS_SERVER_PORT);
        bind(dns_server_sock,(struct sockaddr*)&strAddr,sizeof(strAddr));
        registerSocketCallback(UDP_SocketEventHandler,AppServerCb);

   }
```

- Sockets Events Handler:

```
void MDNS_RecvfromCB(signed char sock,unsigned char *pu8RxBuffer,signed short s16DataSize,
                                           unsigned char *pu8IPAddr,unsigned short u16Port,void *pvArg)
{
     MDNS_INFO("DnsServer_RecvfromCB \n");
     if((pu8RxBuffer != 0) && (s16DataSize > 0))
     {
          tstrDnsHdr strDnsHdr;
          strdnsquery;
          MDNS_INFO("DNS Packet Recieved  \n");
```

```
        if(MDNS_ParseQuery(&pu8RxBuffer[0], &strDnsHdr,&strDnsQuery))
                MDNS_SendResp (sock,pu8IPAddr, u16Port,&strDnsHdr,&strDnsQuery );
    }
    else
    {
        MDNS_INFO("DnsServer_RecvfromCB Error !\n");
    }
}
```

- Server Socket Callback

```
void MDNS_RecvfromCB(signed char  sock,unsigned char *pu8RxBuffer,signed short
s16DataSize,unsigned char *pu8IPAddr,unsigned short     u16Port,void *pvArg)
{
        MDNS_INFO("DnsServer_RecvfromCB \n");
        if((pu8RxBuffer != 0) && (s16DataSize > 0))
        {
                tstrDnsHdr strDnsHdr  ;
                strdnsquery ;
                MDNS_INFO("DNS Packet Recieved  \n");

                if(MDNS_ParseQuery(&pu8RxBuffer[0], &strDnsHdr,&strDnsQuery))
                MDNS_SendResp (sock,pu8IPAddr, u16Port,&strDnsHdr,&strDnsQuery );
        }
        else
        {
                MDNS_INFO("DnsServer_RecvfromCB Error !\n");
        }
}
```

- Parse mDNS Query:

```
int MDNS_ParseQuery(unsigned char * pu8RxBuffer, tstrDnsHdr *pstrDnsHdr, strdnsquery *pstrDnsQuery  )
{
     unsigned char  dot_size,temp=0;
     unsigned short n=0,i=0,u16index=0;
     int  bDNSmatch = 0;
     /*  ----Identification--------------------------|QR| Opcode       |AA|TC|RD|RA|Z|AD|CD|Rcode   | */
     /*   ----Total Questions------------------------|-----------------Total Answer RRs---------------
*/
     /*   ----Total Authority RRs--------------------|----------------Total Additional RRs------------
*/
     /*   ---------------------------------      Questions         ---------------------------------
*/
     /*   ------------------------------------ Answer RRs  ------------------------------------------*/
     /*   ----------------------------------- Authority RRs        ----------------------------------
*/
     /*   -----------------------------------Additional RRs       ----------------------------------
*/
     MDNS_INFO("Parsing DNS Packet\n");
     pstrDnsHdr->id = (( pu8RxBuffer[u16index]<<8)| (pu8RxBuffer[u16index+1]));
     MDNS_INFO ("id =  %.4x \n",pstrDnsHdr->id);
     u16index+=2;
     pstrDnsHdr->flags1= pu8RxBuffer[u16index++];
     pstrDnsHdr->flags2= pu8RxBuffer[u16index++];
     MDNS_INFO ("flags =  %.2x %.2x \n",pstrDnsHdr->flags1,pstrDnsHdr->flags2);
```

```
    pstrDnsHdr->numquestions = ((pu8RxBuffer[u16index]<<8)| (pu8RxBuffer[u16index+1]));
    MDNS_INFO ("numquestions =  %.4x \n",pstrDnsHdr->numquestions);
    u16index+=2;
    pstrDnsHdr->numanswers = ((pu8RxBuffer[u16index]<<8)| (pu8RxBuffer[u16index+1]));
    MDNS_INFO ("numanswers =  %.4x \n",pstrDnsHdr->numanswers);
    u16index+=2;
    pstrDnsHdr->numauthrr = ((pu8RxBuffer[u16index]<<8)| (pu8RxBuffer[u16index+1]));
    MDNS_INFO ("numauthrr =  %.4x \n",pstrDnsHdr->numauthrr);
    u16index+=2;
    pstrDnsHdr->numextrarr = ((pu8RxBuffer[u16index]<<8)| (pu8RxBuffer[u16index+1]));
    MDNS_INFO ("numextrarr =  %.4x \n",pstrDnsHdr->numextrarr);
    u16index+=2;
    dot_size =pstrDnsQuery->query[n++]= pu8RxBuffer[u16index++];
    pstrDnsQuery->u16size=1;
    while (dot_size--!=0) //(pu8RxBuffer[++u16index] != 0)
    {
         pstrDnsQuery->query[n++]=pstrDnsQuery->queryForChecking[i++]=pu8RxBuffer[u16index++] ;
         pstrDnsQuery->u16size++;
         gu8pos=temp;
         if (dot_size == 0 )
         {
              pstrDnsQuery->queryForChecking[i++]= '.' ;
              temp=u16index;
              dot_size =pstrDnsQuery->query[n++]= pu8RxBuffer[u16index++];
              pstrDnsQuery->u16size++;
         }
    }
    pstrDnsQuery->queryForChecking[--i] = 0;

    MDNS_INFO("parsed query <%s>\n",pstrDnsQuery->queryForChecking);
    // Search for any match in the local DNS table.
    for(n = 0; n < DNS_SERVER_CACHE_SIZE; n++)
    {
         MDNS_INFO("Saved URL <%s>\n",gpacDnsServerCache[n]);
         if(strcmp(gpacDnsServerCache[n], pstrDnsQuery->queryForChecking) ==0)
         {
              bDNSmatch= 1;
              MDNS_INFO("MATCH \n");
         }
         else
         {
         MDNS_INFO("Mismatch\n");
         }
    }
    pstrDnsQuery->u16class = ((pu8RxBuffer[u16index]<<8)| (pu8RxBuffer[u16index+1]));
    u16index+=2;
    pstrDnsQuery->u16type= ((pu8RxBuffer[u16index]<<8)| (pu8RxBuffer[u16index+1]));
    return bDNSmatch;

   }
```

- Send mDNS Response:

```
void MDNS_SendResp (signed char sock,unsigned char * pu8IPAddr,
                        unsigned short u16Port,tstrDnsHdr *pstrDnsHdr,strdnsquery *pstrDnsQuery)
{
     unsigned short u16index=0;
     tstrSockAddr strclientAddr ;
     unsigned char * pu8sendBuf;
     char * serviceName2 = (char*)malloc(sizeof(serviceName)+1);
     unsigned int MULTICAST_IP =  0xFB0000E0;
     pu8sendBuf= gPu8Buf;
     memcpy(&strclientAddr.u32IPAddr,&MULTICAST_IP,IPV4_DATA_LENGTH);
     strclientAddr.u16Port=u16Port;
     MDNS_INFO("%s \n",pstrDnsQuery->query);
     MDNS_INFO("Query Size = %d \n",pstrDnsQuery->u16size);
     MDNS_INFO("class = %.4x \n",pstrDnsQuery->u16class);
     MDNS_INFO("type  = %.4x \n",pstrDnsQuery->u16type);
     MDNS_INFO("PREPARING DNS ANSWER BEFORE SENDING\n");

     /*----------------------------ID 2 Bytes -----------------------------*/
     pu8sendBuf [u16index++] =0;  //( pstrDnsHdr->id>>8);
     pu8sendBuf [u16index++] =  0;//( pstrDnsHdr->id)&(0xFF);
     MDNS_INFO ("(ResPonse) id = %.2x %.2x  \n", pu8sendBuf[u16index-2],pu8sendBuf[u16index-1]);
     /*----------------------------Flags 2 Bytes----------------------------*/
     pu8sendBuf [u16index++] =  DNS_RSP_FLAG_1;
     pu8sendBuf [u16index++] =  DNS_RSP_FLAG_2;
     MDNS_INFO ("(ResPonse) Flags = %.2x %.2x  \n", pu8sendBuf[u16index-2],pu8sendBuf[u16index-1]);
     /*----------------------------No of Questions--------------------------*/
     pu8sendBuf [u16index++] =0x00;
     pu8sendBuf [u16index++] =0x01;
     MDNS_INFO ("(ResPonse) Questions  = %.2x %.2x  \n", pu8sendBuf[u16index-2],pu8sendBuf[u16index-
1]);
     /*---------------------------No of Answers----------------------------*/
     pu8sendBuf [u16index++] =0x00;
     pu8sendBuf [u16index++] =0x01;
     MDNS_INFO ("(ResPonse) Answers = %.2x %.2x  \n", pu8sendBuf[u16index-2],pu8sendBuf[u16index-1]);
     /*---------------------------No of Authority RRs------------------------*/
     pu8sendBuf [u16index++] =0x00;
     pu8sendBuf [u16index++] =0x00;
     MDNS_INFO ("(ResPonse) Authority RRs = %.2x %.2x  \n", pu8sendBuf[u16index-2],pu8sendBuf[u16index-
1]);
     /*----------------------------No of Additional RRs----------------------*/
     pu8sendBuf [u16index++] =0x00;
     pu8sendBuf [u16index++] =0x00;
     MDNS_INFO ("(ResPonse) Additional RRs = %.2x %.2x  \n", pu8sendBuf[u16index-
2],pu8sendBuf[u16index-1]);
     /*--------------------------------Query----------------------------*/
     memcpy(&pu8sendBuf[u16index],pstrDnsQuery->query,pstrDnsQuery->u16size);
     MDNS_INFO("\nsize = %d \n",pstrDnsQuery->u16size);
     u16index+=pstrDnsQuery->u16size;
     /*-------------------------------Query Type----------------------------*/
     pu8sendBuf [u16index++] = ( pstrDnsQuery->u16type>>8);//MDNS_TYPE>>8;
     pu8sendBuf [u16index++] = ( pstrDnsQuery->u16type)&(0xFF);//(MDNS_TYPE&0xFF);
     MDNS_INFO ("Query Type =  %.2x %.2x \n", pu8sendBuf[u16index-2],pu8sendBuf[u16index-1]);
     /*-----------------------------Query Class----------------------------------*/
     pu8sendBuf [u16index++] =MDNS_CLASS>>8;//(( pstrDnsQuery->u16class>>8)|0x80);
     pu8sendBuf [u16index++] = (MDNS_CLASS & 0xFF);//( pstrDnsQuery->u16class)&(0xFF);
```

```
      MDNS_INFO ("Query Class =  %.2x %.2x \n", pu8sendBuf[u16index-2],pu8sendBuf[u16index-1]);

      /*########################Answers########################*/
      /*------------------------------Name--------------------------------*/
      pu8sendBuf [u16index++]=  0xC0 ; //pointer to query name location
      pu8sendBuf [u16index++]= 0x0C ; // instead of writing the whole query name again
      /*-----------------------------Type----------------------------------*/
      pu8sendBuf [u16index++] =MDNS_TYPE>>8;  //Type 12 PTR (domain name Pointer).
      pu8sendBuf [u16index++] =(MDNS_TYPE&0xFF);
      /*------------------------------Class----------------------------------*/
      pu8sendBuf [u16index++] =0x00;//MDNS_CLASS;  //Class IN, Internet.
      pu8sendBuf [u16index++] =0x01;// (MDNS_CLASS & 0xFF);
      /*-----------------------------TTL----------------------------------*/
      pu8sendBuf [u16index++] =(TIME_TO_LIVE >>24);
      pu8sendBuf [u16index++] =(TIME_TO_LIVE >>16);
      pu8sendBuf [u16index++] =(TIME_TO_LIVE >>8);
      pu8sendBuf [u16index++] =(TIME_TO_LIVE );
      /*---------------------------Date Length----------------------------------*/
      pu8sendBuf [u16index++] =(sizeof(serviceName)+2)>>8;//added 2 bytes for the pointer
      pu8sendBuf [u16index++] =(sizeof(serviceName)+2);
      /*-----------------------------DATA--------------------------------*/
      convertServiceName(serviceName,sizeof(serviceName),serviceName2);
      memcpy(&pu8sendBuf[u16index],serviceName2,sizeof(serviceName)+1);
      u16index+=sizeof(serviceName);
      pu8sendBuf [u16index++] =0xC0;//Pointer to .local (from name)
      pu8sendBuf [u16index++] =gu8pos;//23
      /*#############################################################*/
      strclientAddr.u16Port=HTONS(MDNS_SERVER_PORT);
      // MultiCast RESPONSE
      sendto( sock, pu8sendBuf,(uint16)u16index,0,(struct
 sockaddr*)&strclientAddr,sizeof(strclientAddr));
      strclientAddr.u16Port=u16Port;
      memcpy(&strclientAddr.u32IPAddr,pu8IPAddr,IPV4_DATA_LENGTH);
 }
```

- Service Name:

```
static char gpacDnsServerCache[DNS_SERVER_CACHE_SIZE][MDNS_HOSTNAME_SIZE] =
{
        "_services._dns-sd._udp.local","_workstation._tcp.local","_http._tcp.local"
};
unsigned char   gPu8Buf [MDNS_BUF_SIZE];
unsigned char   gu8pos ;
signed   char   dns_server_sock ;

#define serviceName "_ATMELWIFI._tcp"
```

13 WINC Serial Flash Memory

13.1 Overview and Features

The WINC has internal serial (SPI) flash memory of either 2 or 4Mb capacity. The flash memory is used to store:

- User confimbitguration
- Firmware
- Connection Profiles

During startup and mode changes firmware is loaded from the serial flash into program memory (IRAM) in which firmware is executed. The flash is accessed at other points during runtime to retrieve configuration and profile data.

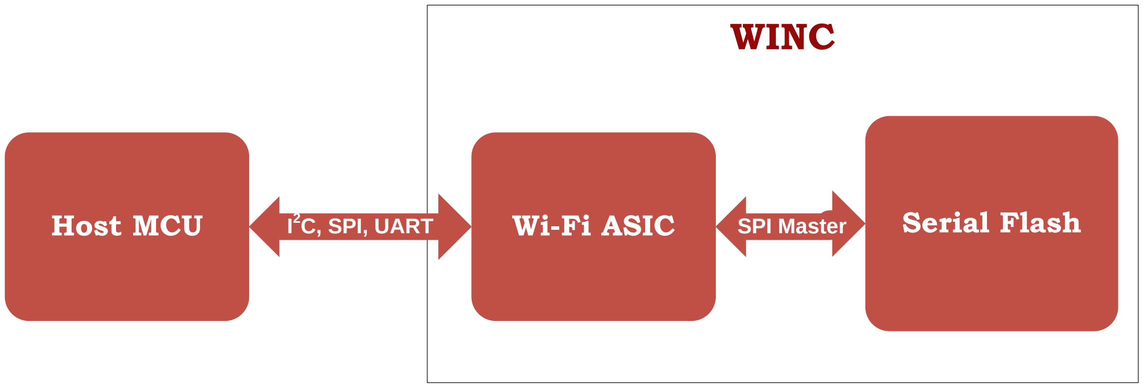
The 2Mb flash can store one firmware image. The 4Mb flash is required for OTA feature in order to store both working and rollback images.

The flash memory can be read, written to and erased directly from the host without co-operation with the WINC firmware. However, if operational firmware is already loaded, it is necessary to halt any running WINC firmware first before accessing the serial flash to avoid access conflict between host and the WINC processor.

13.2 Accessing to Serial Flash

- The host has transparent access to the serial (SPI) flash through WINC SPI master
- The host can program the serial (SPI) flash without need for operational firmware in the WINC. The function `m2m_wifi_download_mode` must be called first.

Figure 13-1. System Block Diagram showing SPI Flash Connection



13.3 Read/Write/Erase Operations

SPI Flash can be accessed to be read, written, and erased.

It is required to change WINC's mode to "*download mode*" first before any attempt to access the SPI Flash by calling:

```
sint32 m2m_wifi_download_mode();
```

All SPI flash functions are blocking. A return of `M2M_SUCCESS` indicates that the requested operation has been completed successfully.

The following is a list of flash functions that may be used:

- Query the size of the SPI Flash:

```
uint32 spi_flash_get_size();
```

This function returns with size of SPI Flash in Mb.

- Read data from the SPI Flash:

    `sint8 spi_flash_read(uint8 *pu8Buf, uint32 u32offset, uint32 u32Sz)`

  Where the size of data is limited by SPI Flash size.

- Erase sectors in the SPI Flash:

    `sint8 spi_flash_erase(uint32 u32Offset, uint32 u32Sz)`

  Note: The size is limited by the SPI Flash size.

  Before writing to any sector, this sector has to be erased first. So if some data needs to be changed within a sector, it is advised to read the sector first, modify the data then erase and write the whole sector again.

- Write data to the SPI Flash:

    `sint8 spi_flash_write(uint8* pu8Buf, uint32 u32Offset, uint32 u32Sz)`

  If the application wants to write any number of bytes within any sector, it has to erase the entire sector first. It may be necessary to read the entire sector, erase the sector and then write back with modifications. It is also recommended to verify that data had been written after it returns success by reading data again and compare it with the original.

13.4 Serial (SPI) Flash Map

The following map is valid for SPI Flash with size equals 4Mb (512KB)

Section	Offset [KB]	Size [KB]
Configuration	0	20
Firmware Image 1	20	194
Connection Profiles	214	42
Firmware Image 2	256	194
Connection Profiles*	450	62
Total		**512**

- Configuration section assumes that default firmware to run is Firmware Image 1
- In case of 2Mb SPI Flash: Firmware Image 2 and Connection Profiles* sections (used by the OTA feature) below it, will not exist

13.5 Flash Read, Erase, Write Code Example

```
#include "spi_flash.h"
#define DATA_TO_REPLACE  "THIS IS A NEW SECTOR IN FLASH"

int main()
{
     uint8au8FlashContent[FLASH_SECTOR_SZ] = {0};
     uint32u32FlashTotalSize = 0, u32FlashOffset = 0;

     // Platform specific initializations.

     ret = m2m_wifi_download_mode();
     if(M2M_SUCCESS != ret)
     {
            printf("Unable to enter download mode\r\n");
      }
      else
      {
            u32FlashTotalSize = spi_flash_get_size();
      }

      while((u32FlashTotalSize > u32FlashOffset) && (M2M_SUCCESS == ret))
      {
            ret = spi_flash_read(au8FlashContent, u32FlashOffset, FLASH_SECTOR_SZ);
            if(M2M_SUCCESS != ret)
            {
                  printf("Unable to read SPI sector\r\n");
                  break;
            }
            memcpy(au8FlashContent, DATA_TO_REPLACE, strlen(DATA_TO_REPLACE));

            ret = spi_flash_erase(u32FlashOffset, FLASH_SECTOR_SZ);
            if(M2M_SUCCESS != ret)
            {
                  printf("Unable to erase SPI sector\r\n");
                  break;
            }

            ret = spi_flash_write(au8FlashContent, u32FlashOffset, FLASH_SECTOR_SZ);
            if(M2M_SUCCESS != ret)
            {
                  printf("Unable to write SPI sector\r\n");
                  break;
            }
            u32FlashOffset += FLASH_SECTOR_SZ;
      }

      if(M2M_SUCCESS == ret)
      {
            printf("Successful operations\r\n");
      }
      else
      {
            printf("Failed operations\r\n");
      }

      while(1);
      return M2M_SUCCESS;
}
```

14 Wi-Fi Sniffer Mode

14.1 Overview

In this mode, WINC receives all traffic on the current wireless channel with the ability to apply filters and configuration. This mode operates without making a connection to an AP and will return all frames captured from the air subject to the configured filters.

The received frames are delivered to the application. Delivered frames contain two parts:

- A structure holding the Wi-Fi frame header parameters (frame type, frame sub-type, BSSID…etc.)
- A buffer holding the data payload of the Wi-Fi frame

The Application also has the ability to compose and send RAW Wi-Fi frames. The application is responsible for the format, the WINC will transmit it on the air as is.

TIPS WINC must be disconnected before activating this mode.

14.2 Sniffer (Monitoring) Mode APIs

There are two API functions to enable or/disable WINC Sniffer (monitoring) mode. An API function is dedicated to send packets in this mode.

```
sint8 m2m_wifi_enable_monitoring_mode(tstrM2MWifiMonitorModeCtrl   *pstrMtrCtrl,
                                            uint8 *pu8PayloadBuffer,
                                            uint16 u16BufferSize,
                                            uint16 u16DataOffset
                                           );

sint8 m2m_wifi_disable_monitoring_mode(void);

sint8 m2m_wifi_send_wlan_pkt(uint8 *pu8WlanPacket,
                             uint16 u16WlanHeaderLength,
                             uint16 u16WlanPktSize
                            );
```

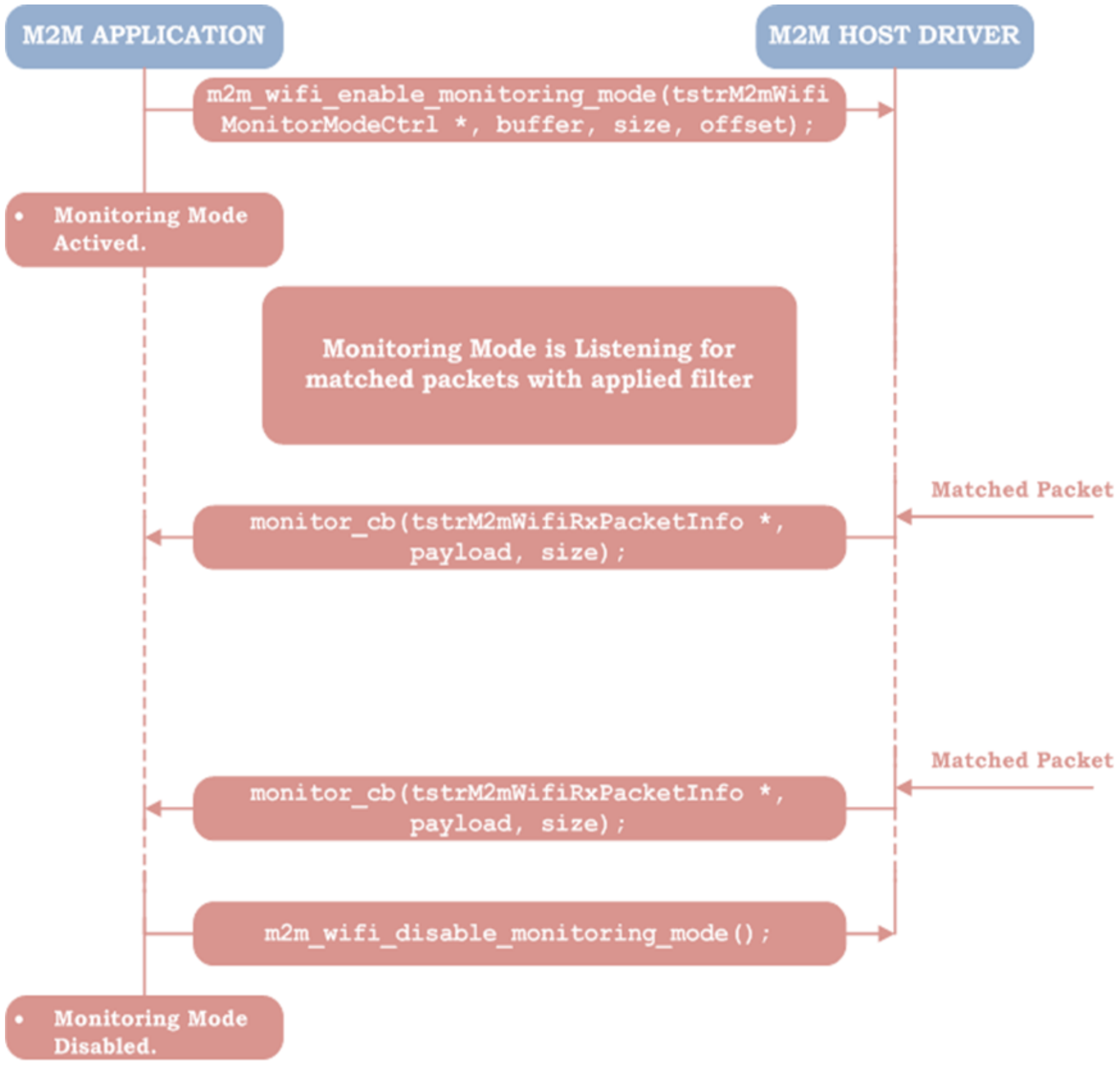
14.3 Monitoring Parameters

Prior to enabling monitoring the application needs to populate a structure of type `tstrM2MWifiMonitorModeCtrl` to specify which frames should be captured. The other parameters define a buffer into which the captured data will be placed. Once monitoring is active, the monitoring call back will be called for each frame matching the filter. A structure of type `tstrM2MWifiRxPacketInfo` is passed to the call back with details of the frame.

14.4 Sequence Diagram

Figure 14-1 shows how to setup Monitoring mode, receive matched packets and disable monitoring mode.

Figure 14-1. Monitoring Mode Sequence Diagram


M2M APPLICATION
M2M HOST DRIVER
m2m_wifi_enable_monitoring_mode(tstrM2mWifi MonitorModeCtrl *, buffer, size, offset);
• Monitoring Mode Actived.
Monitoring Mode is Listening for matched packets with applied filter
Matched Packet
monitor_cb(tstrM2mWifiRxPacketInfo *, payload, size);
Matched Packet
monitor_cb(tstrM2mWifiRxPacketInfo *, payload, size);
m2m_wifi_disable_monitoring_mode();
• Monitoring Mode Disabled.

14.5 Code Example

For more details on the code examples refer to [R02].

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

/* Declare receive buffer */
uint8 gmgmt[1600];

/* Callback functions */
void wifi_cb(uint8 u8WiFiEvent, void * pvMsg)
{
        ;
}
void wifi_monitoring_cb(tstrM2MWifiRxPacketInfo *pstrWifiRxPacket, uint8 *pu8Payload, uint16
u16PayloadSize)
{
        if((NULL != pstrWifiRxPacket) && (0 != u16PayloadSize))
        {
                if(MANAGEMENT == pstrWifiRxPacket->u8FrameType)
                {
                        M2M_INFO("***# MGMT PACKET #***\n");
                }
                else if(DATA_BASICTYPE == pstrWifiRxPacket->u8FrameType)
                {
                        M2M_INFO("***# DATA PACKET #***\n");
                }
                else if(CONTROL == pstrWifiRxPacket->u8FrameType)
                {
                        M2M_INFO("***# CONTROL PACKET #***\n");
                }
        }
}

int main()
{
        tstrWifiInitParam param;

        /* Platform specific initializations. */
        param.pfAppWifiCb = wifi_cb;
        param.pfAppMonCb  = wifi_monitoring_cb;

        if(!m2m_wifi_init(&param))
        {
                tstrM2MWifiMonitorModeCtrl  strMonitorCtrl = {0};
                /* Enable Monitor Mode with filter to receive all data frames on channel 1 */
                strMonitorCtrl.u8ChannelID      = 1;
                strMonitorCtrl.u8FrameType      = DATA_BASICTYPE;
                strMonitorCtrl.u8FrameSubtype   = M2M_WIFI_FRAME_SUB_TYPE_ANY;
                m2m_wifi_enable_monitoring_mode(&strMonitorCtrl, gmgmt, sizeof(gmgmt), 0);

                while(1)
                {
                        m2m_wifi_handle_events(NULL);
                }
        }
        return 0;
}
```

15 Writing a Simple Networking Application

This chapter provides a step-by-step tutorial on how to build a networking application from scratch. For details on getting started with the Atmel Studio and how to setup the environment and obtain the example application and AtmelWirelessConnect APK smart phone application, refer to [R01].

15.1 Prerequisites

- Hardware Prerequisites
  - Atmel SAMD21-XPRO Evaluation kit
  - Atmel IO1 Xplained ATIO1-XPRO board
  - Atmel WINC1500 extension board
  - Micro-USB Cable (Micro-A / Micro-B)
  - Android Phone
- Software Prerequisites
  - Atmel Studio 6.2 (build 1153) or higher
  - Atmel Software Frameworks 3.15.0
  - Wi-Fi Network Controller demo for SAM D21
  - Android apk AtmelWirelessConnect application

15.2 Solution Overview

The goal of this project is to develop an IOT application, capable of sending temperature information to any phone or tablet on the network while offering a way to remotely control the LED on the SAM D21 Xplained Pro board.

To develop and run this project you need to start with the downloaded empty Wi-Fi example project.

INFO

If you cannot use an Android phone to test the solution, you can still use Wireshark to see the traffic generated by the IOT sensor application.

TO DO WINC1500 Wi-Fi extension and IO1 extension should be plugged into SAM D21 Xplained Pro EXT1 and EXT2 respectively.

15.3 Project Creation

TO DO Open the empty Wi-Fi example Project.

- Open Atmel Studio 6.2
- Click on “File” then “Open Project/Solution…”
- Select the Empty Wi-Fi example project on your hard drive

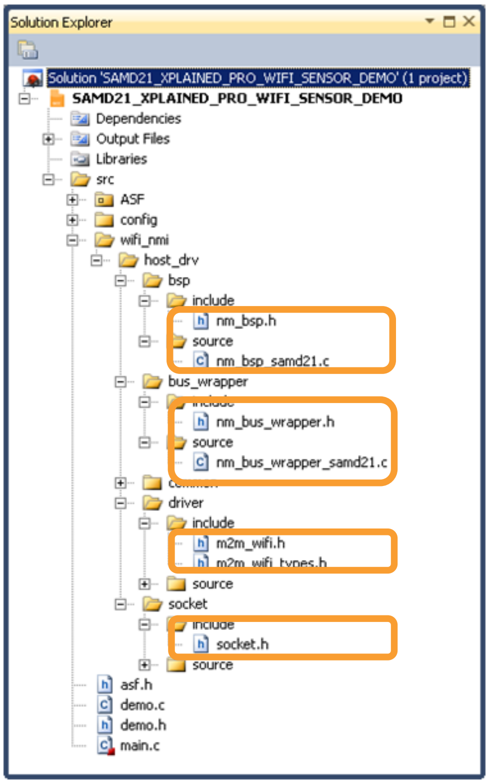
15.4 Wi-Fi Software API Files

The table below lists the main files from the Wi-Fi Software API located.

File	Description
`m2m_wifi.h` `m2m_wifi.c`	Provide entry point, Wi-Fi configuration API
`socket.h` `socket.c`	Provide socket API
`nmbsp.h` `nm_bsp_samd21.c`	Provide BSP APIs needed by Host Driver
`nm_bus_wrap-per_samd21.c`	Provide bus wrapper APIs needed by Host Driver

In order to add Wi-Fi connectivity into an existing user example project, the complete “wifi_nmi” folder should be added to the user project.

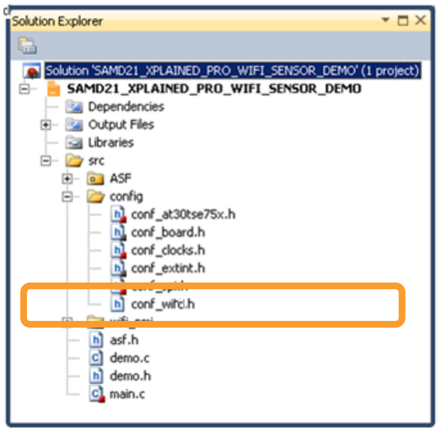
TO DO Locate these files in your project:



```
Solution Explorer
Solution 'SAMD21_XPLAINED_PRO_WIFI_SENSOR_DEMO' (1 project)
SAMD21_XPLAINED_PRO_WIFI_SENSOR_DEMO
    Dependencies
    Output Files
    Libraries
    src
        ASF
        config
        wifi_nmi
            host_drv
                bsp
                    include
                        nm_bsp.h
                    source
                        nm_bsp_samd21.c
                bus_wrapper
                    include
                        nm_bus_wrapper.h
                    source
                        nm_bus_wrapper_samd21.c
                common
                driver
                    include
                        m2m_wifi.h
                        m2m_wifi_types.h
                    source
                socket
                    include
                        socket.h
                    source
        asf.h
        demo.c
        demo.h
        main.c
```

`m2m_wifi_type.h` is an internal type definition header file.

The configuration of the Wi-Fi Software API is fairly easy and relies on three configuration files.



```
Solution Explorer
Solution 'SAMD21_XPLAINED_PRO_WIFI_SENSOR_DEMO' (1 project)
SAMD21_XPLAINED_PRO_WIFI_SENSOR_DEMO
    Dependencies
    Output Files
    Libraries
    src
        ASF
        config
            conf_at30tse75x.h
            conf_board.h
            conf_clocks.h
            conf_extint.h
            conf_spi.h
            conf_winc.h
        wifi_nmi
        asf.h
        demo.c
        demo.h
        main.c
```

The file **`conf_winc.h`** provides configuration for the following:

- `CONF_WIFI_M2M_RESET_PIN:` reset pin definition (RESET_N)
- `CONF_WIFI_M2M_CHIP_ENABLE_PIN:` reset pin definition (CHIP_EN)
- `CONF_WIFI_M2M_INT_PIN:` Interrupt line (IRQN)
- `CONF_WIFI_M2M_DEUG:` debug enable

15.5 Reading Temperature Sensor and Controlling LED Status

The empty Wi-Fi example Project comes with a specific driver for the AT30TSE temperature sensor, located on the IO1 extension.

The driver implementation can be found in the “ASF\sam0\components\sensor\at30tse75x” folder of the Solution Explorer.

Retrieving the temperature information is easy and can be performed in three steps:

1. Include the driver header file.

```
#include “asf.h”
```

2. Initialize the temperature sensor driver.

```
at30tse_init();
```

3. Retrieve the current temperature value.

```
double temp = at30tse_read_temperature();
```

INFO The empty Wi-Fi example Project already includes the peripheral dependencies for the temperature sensor.
The temperature sensor configuration file “conf_at30tse75x.h” is already configured to use the peripheral on EXT2.

15.6 Step By Step Development

The main.c file from the empty Wi-Fi example Project should already handle the system initialization and start. It is also responsible for calling a demo_start() function (declared in demo.h and implemented in the demo.c file).

```
int main(void)
{
        system_init();

        /* Initialize the UART console. */
        configure_console();

        /* Initialize the delay driver. */
        delay_init();

        /* Enable SysTick interrupt for non busy wait delay. */
        if (SysTick_Config(system_cpu_clock_get_hz() / 1000)) {
                puts("main: SysTick configuration error!");
                while (1);
        }

        /* Output example information */
        puts(STRING_HEADER);

        /* Start the demo task. */
        demo_start();

        return 0;
}
```

The `demo_start()` function is the application main loop which implements the routines for connecting to the network and sending temperature reports using the Wi-Fi Software API.

```
/**
 * \brief Demo main routine.
 */
void demo_start(void)
{
        while (1) {
                /* TODO: Implement IOT feature here. */
        }
}
```

TO DO Implement the following steps to implement the IOT temperature sensor application.

1. **Reset the WINC1500 Module.**

   To do so, we need to call nm_bsp_init(), which initializes the CHIP_EN and RESET_N GPIOs.

```
void demo_start(void)
{
        /* Reset network controller */
        nm_bsp_init();

        while (1) {
                /* TODO: Implement IOT feature here. */
        }
}
```

2. **Initialize the Wi-Fi Software API.**

   To do so, we need to declare the **`tstrWifiInitParam`** structure which contains a pointer to the Wi-Fi callback functions. A pointer to **`tstrWifiInitParam`** structure is passed to **`m2m_wifi_init()`**.To indicate successful initialization, **`m2m_wifi_init()`** returns **`M2M_SUCCESS`**. Later on, we call **`socketInit()`** and register the socket callback.

```
        tstrWifiInitParam param;
        sint8 ret;

        /* Initialize Wi-Fi parameters structure. */
        param.pfAppWifiCb = m2m_wifi_state;

        ret = m2m_wifi_init(&param);
        if (M2M_SUCCESS != ret) {
                puts("demo_start: nm_drv_init call error!");
                while (1)
                ;
        }
        /* Initialize Socket module */
        socketInit();
        registerSocketCallback(m2m_wifi_socket_handler, NULL);
```

3. **Implement an empty m2m_wifi_state() function.**

   In order to receive Wi-Fi events from “m2m_wifi” module, implement a skeleton M2M Wi-Fi callback function.

```
static void m2m_wifi_state(uint8 u8MsgType, void *pvMsg)
{
        switch (u8MsgType) {
        default:
                break;
        }
}
```

4. **Implement an empty m2m_wifi_socket_handler.**

   In order to receive socket event from "socket" module, implement a skeleton M2M Socket callback function.

```
static void m2m_wifi_socket_handler(SOCKET sock, uint8 u8Msg, void *pvMsg)
{
        /* TODO: Check for socket event here. */
}
```

INFO All WINC1500 socket operations are non-blocking asynchronous operations.

When **`m2m_wifi_socket_handler()`** is called, it indicates a specific asynchronous socket operation has been done for a specified `SOCKET sock`. The completed socket operation type is indicated in `u8Msg` parameter and any message payload (e.g. received data) is provided in the last parameter `pvMsg`.

Example of non-blocking asynchronous socket operations completions are: `SOCKET_MSG_BIND, SOCKET_MSG_LISTEN, SOCKET_MSG_ACCEPT, SOCKET_MSG_CONNECT, SOCKET_MSG_RECV, SOCKET_MSG_SEND, SOCKET_MSG_SENDTO`, and `SOCKET_MSG_RECVFROM` to indicate completion of **`bind()`**, **`listen()`**, **`accept()`**, **`connect()`**, **`recv()`**, **`send()`**, **`sendto()`**, **`recvfrom()`** respectively.

5. **Initialize the temperature sensor.**

```
/* Initialize temperature sensor. */
at30tse_init();
```

6. **Initialize LED0 to off state.**

   During connection phase, LED0 of SAM D21 will be off. LED0 will turn on when the DHCP address is acquired. After DHCP, the Android app will be able to control it remotely.

```
/* Turn LED0 off initially. */
port_pin_set_output_level(LED_0_PIN, true);
```

RESULT Upon completion of this step, the code should look like the following:

```
static void m2m_wifi_socket_handler(SOCKET sock, uint8 u8Msg, void *pvMsg)
{
	/* TODO: Check for socket event here. */
}

static void m2m_wifi_state(uint8 u8MsgType, void *pvMsg)
{
	switch (u8MsgType) {
		default:
		break;
	}
}

/**
 * \brief Demo main routine.
 */
void demo_start(void)
{
	tstrWifiInitParam param;
	sint8 ret;

	/* Initialize Wi-Fi parameters structure. */
	param.pfAppWifiCb = m2m_wifi_state;

	/* Turn LED0 off initially. */
	port_pin_set_output_level(LED_0_PIN, true);

	/* Initialize temperature sensor. */
	at30tse_init();

	/* Reset network controller */
	nm_bsp_init();

	/* Initialize Wi-Fi driver with data and Wi-Fi status callbacks. */
	ret = m2m_wifi_init(&param);
	if (M2M_SUCCESS != ret) {
		puts("demo_start: nm_drv_init call error!");
		while (1)
		;
	}
	/* Initialize Socket module */
	socketInit();
	registerSocketCallback(m2m_wifi_socket_handler, NULL);

	while (1) {
		/* TODO: Implement IOT feature here. */
	}
}
```

INFO The initialization stage is done. Now, connect to the network and open the required communication sockets.

7. **Add event handler to application main loop.**

   To do so, add a call to m2m_wifi_handle_events() inside the application loop.

   The API `m2m_wifi_handle_events()` checks for pending events and dispatches events to m2m_wifi module or socket module and returns to caller.

```
        while (1) {
               /* Handle pending events from network controller. */
               m2m_wifi_handle_events(NULL);
        }
```

TIPS Since the demo application does not use operating system, `m2m_wifi_handle_events()` is called in the application main loop. If the application used an operating system, it is required to create a dedicated task to call `m2m_wifi_handle_events()`. The task shall sleep until an interrupt on IRQN line, then it handles the deferred work by ISR and calls `m2m_wifi_handle_events()`.

TIPS Both of the application defined callbacks in "demo.c": `m2m_wifi_socket_handler()` and `m2m_wifi_state()` are called in the context of `m2m_wifi_handle_events()`.

8. **Start association.**

   The SSID and passphrase of the router are defined in the demo.h file, using the following defines; `DEMO_WLAN_SSID` and `DEMO_WLAN_PSK`. These should be updated with your local SSID and passphrase.

   The `m2m_wifi_connect()` function request the WINC1500 Wi-Fi module to start association with the local SSID.

```
        /* Connect to router. */
        m2m_wifi_connect((char *)DEMO_WLAN_SSID, sizeof(DEMO_WLAN_SSID),
               DEMO_WLAN_AUTH, (char *)DEMO_WLAN_PSK, M2M_WIFI_CH_ALL);
```

   The parameter `DEMO_WLAN_AUTH` specifies the local AP security type which can be either: OPEN, WEP, WPA/WPA2 or enterprise security. The last parameter `M2M_WIFI_CH_ALL` forces the WINC1500 to scan for the local AP on all channels.

TIPS The WINC1500 firmware contains a "**fast boot feature**" which optimizes the association time across power cycles. Firmware stores the channel on which the last successful association occurred. During next association attempt, firmware probes if the local AP still on the same channel (which is most likely to happen since AP does not change channel frequently).
If local AP is not found, then firmware will trigger a new scan.

9. **Handle Wi-Fi connection state change in m2m_wifi_state().**

   Upon association success, the host driver will call m2m_wifi_state() with `M2M_WIFI_RESP_CON_STATE_CHANGED` event to indicate the association state change. The event message payload indicates whether connection succeeds or fails.

   Add the following code to `m2m_wifi_state()` to handle `M2M_WIFI_RESP_CON_STATE_CHANGED`.

```
/** Wi-Fi status variable. */
static volatile uint8 wifi_connected = 0;
...
...
...

static void m2m_wifi_state(uint8 u8MsgType, void *pvMsg)
{
         switch (u8MsgType) {
         case M2M_WIFI_RESP_CON_STATE_CHANGED: {
                  tstrM2mWifiStateChanged *pstrWifiState = (tstrM2mWifiStateChanged*) pvMsg;
                  if (pstrWifiState->u8CurrState == M2M_WIFI_CONNECTED) {
                           puts("m2m_wifi_state: M2M_WIFI_RESP_CON_STATE_CHANGED: CONNECTED");
                           m2m_wifi_request_dhcp_client();
                  }
                  else if(pstrWifiState->u8CurrState == M2M_WIFI_DISCONNECTED) {
                           puts("m2m_wifi_state: M2M_WIFI_RESP_CON_STATE_CHANGED: DISCONNECTED");
                           wifi_connected = 0;
                           m2m_wifi_connect((char *)DEMO_WLAN_SSID, sizeof(DEMO_WLAN_SSID),
                                             DEMO_WLAN_AUTH, (char *)DEMO_WLAN_PSK, M2M_WIFI_CH_ALL);
                  }
         break;
         }
         ...
         ...
         ...
```

When association succeeds, `M2M_WIFI_CONNECTED` event is received. This implies the start of the DHCP client to obtain IP address by calling `m2m_wifi_request_dhcp_client()`. If association fails or a disconnection happens, `M2M_WIFI_DISCONNECTED` is received and the demo app will attempt to reconnect again using `m2m_wifi_connect()`.

10. Obtaining DHCP configuration.

When DHCP client in WINC1500 Firmware obtains IP address from local AP, the demo app receives information about the obtained IP address. `m2m_wifi_state()` callback is invoked with **`M2M_WIFI_REQ_DHCP_CONF`** event type.

Add the following code to **`m2m_wifi_state()`** to handle **`M2M_WIFI_REQ_DHCP_CONF`**.

```
/** Wi-Fi status variable. */
static volatile uint8 wifi_connected = 0;
...
...
...
static void m2m_wifi_state(uint8 u8MsgType, void *pvMsg)
{
         switch (u8MsgType) {
         case M2M_WIFI_RESP_CON_STATE_CHANGED: {
                  ...
                  ...
         break;
         }
         case M2M_WIFI_REQ_DHCP_CONF: {
                  uint8 *pu8IPAddress = (uint8*) pvMsg;
                  wifi_connected = 1;
                  /* Turn LED0 on to declare that IP address received. */
                  port_pin_set_output_level(LED_0_PIN, false);
                  printf("m2m_wifi_state: M2M_WIFI_REQ_DHCP_CONF: IP is %u.%u.%u.%u\n",
                                   pu8IPAddress[0], pu8IPAddress[1], pu8IPAddress[2], pu8IPAddress[3]);
                  /*TODO: add socket initialization here. */

         break;
```

TIPS

Notice that the demo application will set the global `wifi_connected` to 1 to indicate that association is complete and IP address is obtained. The global `wifi_connected` will be reset to zero if connection is lost. Notice that LED0 will turn at this time.

RESULT

Upon completion of this step, the code should look like the following:

```
/** Wi-Fi status variable. */
static volatile uint8 wifi_connected = 0;
...
...
...

static void m2m_wifi_socket_handler(SOCKET sock, uint8 u8Msg, void *pvMsg)
{
        /* TODO: Check for socket event here. */
}

static void m2m_wifi_state(uint8 u8MsgType, void *pvMsg)
{
        switch (u8MsgType) {
                case M2M_WIFI_RESP_CON_STATE_CHANGED: {
                        tstrM2mWifiStateChanged *pstrWifiState = (tstrM2mWifiStateChanged*) pvMsg;
                        if (pstrWifiState->u8CurrState == M2M_WIFI_CONNECTED) {
                                puts("m2m_wifi_state: M2M_WIFI_RESP_CON_STATE_CHANGED: CONNECTED");
                                m2m_wifi_request_dhcp_client();
                        }
                        else if(pstrWifiState->u8CurrState == M2M_WIFI_DISCONNECTED) {
                                puts("m2m_wifi_state: M2M_WIFI_RESP_CON_STATE_CHANGED: DISCONNECTED");
                                wifi_connected = 0;
                                m2m_wifi_connect((char *)DEMO_WLAN_SSID, sizeof(DEMO_WLAN_SSID),
                                DEMO_WLAN_AUTH, (char *)DEMO_WLAN_PSK, M2M_WIFI_CH_ALL);
                        }
                        break;
                }
                case M2M_WIFI_REQ_DHCP_CONF: {
                        uint8 *pu8IPAddress = (uint8*) pvMsg;
                        wifi_connected = 1;
                        /* Turn LED0 on to declare that IP address received. */
                        port_pin_set_output_level(LED_0_PIN, false);
                        printf("m2m_wifi_state: M2M_WIFI_REQ_DHCP_CONF: IP is %u.%u.%u.%u\n",
                        pu8IPAddress[0], pu8IPAddress[1], pu8IPAddress[2], pu8IPAddress[3]);
                        break;
                }
                default: {
                        break;
                }
        }
}

/**
* \brief Demo main routine.
*/
void demo_start(void)
{
        tstrWifiInitParam param;
        sint8 ret;

        /* Initialize Wi-Fi parameters structure. */
        param.pfAppWifiCb = m2m_wifi_state;

        /* Initialize temperature sensor. */
        at30tse_init();

        /* Reset network controller */
        nm_bsp_init();

        /* Initialize Wi-Fi driver with data and Wi-Fi status callbacks. */
```

INFO The association stage is done. In next section we will open sockets and start sending and receiving data.

11. **Open sockets and send transmit and receive data.**

    We should send data to and receive data from the Android application. Hence, we need to create two UDP sockets, one as server (for receiving: `rx_socket`) and one as client (for sending: `tx_socket`).

    The demo Android app shall be connected to the same local AP and broadcasts a message on the local AP WLAN. All WINC1500 devices on the local WLAN receive the broadcast message. The message contains `s_msg_temp_report` structure. The message contains the name of the device which should receive the message. Each device matches the name member to a string DEMO_PRODUCT_NAME before changing the LED state as provided in led member in `s_msg_temp_report` message.

    In return, the device broadcasts two messages every 1 sec on local AP WLAN, namely: keep alive message `t_msg_temp_keepalive` and temperature report message `t_msg_temp_report`. Each device provides its name in transmitted messages in name field as defined in DEMO_PRODUCT_NAME. The `t_msg_temp_keepalive` message allows the demo Android app to discover the existence of WINC1500 devices on local WLAN and displays to the user a list of discovered devices. When user chooses to view temperature from a specific device, the Android app matches the received `t_msg_temp_report` message name field to the name of the user selected device before plotting temperature value.

```
typedef struct s_msg_temp_keepalive {
         uint8_t id0;
         uint8_t id1;
         uint8_t name[9];
         uint8_t type;
} t_msg_temp_keepalive;
typedef struct s_msg_temp_report {
         uint8_t id0;
         uint8_t id1;
         uint8_t name[9];
         uint8_t led;
         uint32_t temp;
} t_msg_temp_report;
```

TO DO Locate the definition of `DEMO_PRODUCT_NAME` in demo.h and modify it to a unique name that you prefer. Notice that maximum name length is eight characters.

To get started with transmit and receive, add the following code to the demo application:

```
/** RX and TX socket handlers. */
static SOCKET rx_socket = -1;
static SOCKET tx_socket = -1;
...
...
...
void demo_start(void)
{
...
...
...
         while (1) {
                  /* Handle pending events from network controller. */
                  m2m_wifi_handle_events(NULL);

                  if (wifi_connected == M2M_WIFI_CONNECTED) {
                           /* Open server socket. */
                           if (rx_socket < 0) {
                                    if ((rx_socket = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
                                             puts("demo_start: failed to create RX UDP client socket error!");
                                             continue;
                                    }
                                    bind(rx_socket, (struct sockaddr *)&addr, sizeof(struct sockaddr_in));
                           }

                           /* Open client socket. */
                           if (tx_socket < 0) {
                                    if ((tx_socket = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
                                             puts("demo_start: failed to create TX UDP client socket error!");
                                             continue;
                                    }
                           }
                  }
         }
}
```

The above code open two sockets and attempts to bind on the UDP server socket. When bind is complete, `m2m_wifi_socket_handler()` is called with an `u8Msg` = `SOCKET_MSG_BIND` success indication. We need to add a handler for `SOCKET_MSG_BIND` inside `m2m_wifi_socket_handler()`.

```
/** Receive buffer definition. */
#define TEST_BUFFER_SIZE 1460
static uint8 gau8SocketTestBuffer[TEST_BUFFER_SIZE];

/** RX and TX socket handlers. */
static SOCKET rx_socket = -1;
static SOCKET tx_socket = -1;
...
...
...
static void m2m_wifi_socket_handler(SOCKET sock, uint8 u8Msg, void *pvMsg)
{
         /* Check for socket event on RX socket. */
         if (sock == rx_socket) {
                  if (u8Msg == SOCKET_MSG_BIND) {
                           tstrSocketBindMsg *pstrBind = (tstrSocketBindMsg *)pvMsg;
                           if (pstrBind && pstrBind->status == 0) {
                                    /* Prepare next buffer reception. */
                                    recvfrom(sock, gau8SocketTestBuffer, TEST_BUFFER_SIZE, 0);
                           }
                           else {
                                    puts("m2m_wifi_socket_handler: bind error!");
                           }
                  }
         }
`
```

After successful bind, the demo app will start receiving data from UDP server. The demo app allocates a static buffer `gau8SocketTestBuffer` to receive incoming data.

TIPS Notice that recvfrom() is called after successful bind() callback.

The call to **recvfrom()** is non-blocking and execution will return again to application main loop. When data is received, **m2m_wifi_socket_handler()** is called with an **u8Msg = SOCKET_MSG_RECVFROM** and **pvMsg** is a pointer to **tstrSocketRecvMsg** structure.

```
/** RX and TX socket handlers. */
static SOCKET rx_socket = -1;
static SOCKET tx_socket = -1;
...
...
static void m2m_wifi_socket_handler(SOCKET sock, uint8 u8Msg, void *pvMsg)
{
         /* Check for socket event on RX socket. */
         if (sock == rx_socket) {
                  if (u8Msg == SOCKET_MSG_BIND) {
                           ...
                           ...
                  else if (u8Msg == SOCKET_MSG_RECVFROM) {
                           tstrSocketRecvMsg *pstrRx = (tstrSocketRecvMsg *)pvMsg;
                           if (pstrRx->pu8Buffer && pstrRx->s16BufferSize) {

                                    /* Check for server report and update led status if necessary. */
                                    t_msg_temp_report report;
                                    memcpy(&report, pstrRx->pu8Buffer, sizeof(t_msg_temp_report));
                                    if (report.id0 == 0 && report.id1 == 2 &&
                                             strstr((char *)report.name, DEMO_PRODUCT_NAME)) {
                                             puts("wifi_nc_data_callback: received app message");
                                             port_pin_set_output_level(LED_0_PIN, report.led ? true : false);
                                             delay = 0;
                                    }

                                    /* Prepare next buffer reception. */
                                    recvfrom(sock, gau8SocketTestBuffer, TEST_BUFFER_SIZE, 0);
                           } else {
                                    if (pstrRx->s16BufferSize == SOCK_ERR_TIMEOUT) {
                                             /* Prepare next buffer reception. */
                                             recvfrom(sock, gau8SocketTestBuffer, TEST_BUFFER_SIZE, 0);
                                    }
                           }
                  }
         }
}
```

The structure **tstrSocketRecvMsg** provides information about the RX buffer pointer for current socket as well as the received size. As explained previously, the received message contains is **t_msg_temp_report** structure which contains LED on/off command to a specific device. The device changes the LED status only when the received **t_msg_temp_report** .name[9] field matches its **DEMO_PRODUCT_NAME**.

12. Send a discovery frame and temperature information.

In order to send UDP packets every 1 sec, the application's main loop keeps track of the current time in milliseconds and waits for **DEMO_REPORT_INTERVAL** duration to elapse before the sending the next packet. The duration **DEMO_REPORT_INTERVAL** is defined in demo.h and set to 1000ms.

Add the following code listing to your demo application:

```
/** Message format declarations. */
static t_msg_temp_keepalive msg_temp_keepalive = {
         .id0 = 0, .id1 = 1, .name = DEMO_PRODUCT_NAME, .type = 2,
};

static t_msg_temp_report msg_temp_report = {
         .id0 = 0, .id1 = 2, .name = DEMO_PRODUCT_NAME, .led = 0, .temp = 0,
};

void demo_start(void)
{
         while (1) {
                  /* Handle pending events from network controller. */
                  m2m_wifi_handle_events(NULL);

                  if (wifi_connected == M2M_WIFI_CONNECTED
                  && (ms_ticks - delay > DEMO_REPORT_INTERVAL)) {

                           /* Open server socket. */
                           if (rx_socket < 0) {
                                    ...
                                    ...
                           }
                           /* Open client socket. */
                           if (tx_socket < 0) {
                                    ...
                                    ...
                           }

                           /* Send client discovery frame. */
                           sendto(tx_socket, &msg_temp_keepalive, sizeof(t_msg_temp_keepalive), 0,
                           (struct sockaddr *)&addr, sizeof(addr));

                           /* Send client report. */
                           msg_temp_report.temp = (uint32_t)(at30tse_read_temperature() * 100);
                           msg_temp_report.led = !port_pin_get_output_level(LED_0_PIN);
                           ret = sendto(tx_socket, &msg_temp_report, sizeof(t_msg_temp_report), 0,
                           (struct sockaddr *)&addr, sizeof(addr));

                           if (ret == M2M_SUCCESS) {
                                    puts("demo_start: sensor report sent");
                                    } else {
                                    puts("demo_start: failed to send status report error!");
                           }
                  }
         }
}
```

RESULT The IOT temperature sensor application is now complete and you are ready to program the SAM D21 Xplained Pro board. Your final code should be like the following listing:

```
/**
 *
 * \file
 *
 * \brief Wi-Fi NMI temperature sensor demo.
 *
 * Copyright (c) 2014 Atmel Corporation. All rights reserved.
 *
 * \asf_license_start
 *
 * \page License
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 *    this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright notice,
 *    this list of conditions and the following disclaimer in the documentation
 *    and/or other materials provided with the distribution.
 *
 * 3. The name of Atmel may not be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * 4. This software may only be redistributed and used in connection with an
 *    Atmel microcontroller product.
 *
 * THIS SOFTWARE IS PROVIDED BY ATMEL "AS IS" AND ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE
 * EXPRESSLY AND SPECIFICALLY DISCLAIMED. IN NO EVENT SHALL ATMEL BE LIABLE FOR
 * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
 * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *
 * \asf_license_stop
 *
 */

#include <string.h>
#include <ctype.h>
#include <stdio.h>
#include "asf.h"
#include "demo.h"
#include "bsp/include/nm_bsp.h"
#include "driver/include/m2m_wifi.h"
#include "socket/include/socket.h"
#include "conf_wifi_m2m.h"

/** Message format definitions. */
typedef struct s_msg_temp_keepalive {
         uint8_t id0;
         uint8_t id1;
         uint8_t name[9];
         uint8_t type;
} t_msg_temp_keepalive;

typedef struct s_msg_temp_report {
         uint8_t id0;
         uint8_t id1;
         uint8_t name[9];
         uint8_t led;
         uint32_t temp;
```

Listing continued below…

Listing page (1/5)

```
} t_msg_temp_report;

/** Message format declarations. */
static t_msg_temp_keepalive msg_temp_keepalive =
{
         .id0 = 0,
         .id1 = 1,
         .name = DEMO_PRODUCT_NAME,
         .type = 2,
};

static t_msg_temp_report msg_temp_report =
{
         .id0 = 0,
         .id1 = 2,
         .name = DEMO_PRODUCT_NAME,
         .led = 0,
         .temp = 0,
};

/** Receive buffer definition. */
#define TEST_BUFFER_SIZE 1460
static uint8 gau8SocketTestBuffer[TEST_BUFFER_SIZE];

/** RX and TX socket handlers. */
static SOCKET rx_socket = -1;
static SOCKET tx_socket = -1;

/** Wi-Fi status variable. */
static volatile uint8 wifi_connected = 0;

/** Global counter delay for timer. */
static uint32_t delay = 0;

/** SysTick counter for non busy wait delay. */
extern uint32_t ms_ticks;

/**
 * \brief Callback to get the Data from socket.
 *
 * \param[in] sock socket handler.
 * \param[in] u8Msg socket event type. Possible values are:
 *  - SOCKET_MSG_BIND
 *  - SOCKET_MSG_LISTEN
 *  - SOCKET_MSG_ACCEPT
 *  - SOCKET_MSG_CONNECT
 *  - SOCKET_MSG_RECV
 *  - SOCKET_MSG_SEND
 *  - SOCKET_MSG_SENDTO
 *  - SOCKET_MSG_RECVFROM
 * \param[in] pvMsg is a pointer to message structure. Existing types are:
 *  - tstrSocketBindMsg
 *  - tstrSocketListenMsg
 *  - tstrSocketAcceptMsg
 *  - tstrSocketConnectMsg
 *  - tstrSocketRecvMsg
 */
static void m2m_wifi_socket_handler(SOCKET sock, uint8 u8Msg, void *pvMsg)
{
         /* Check for socket event on RX socket. */
         if (sock == rx_socket) {
                  if (u8Msg == SOCKET_MSG_BIND) {
                           tstrSocketBindMsg *pstrBind = (tstrSocketBindMsg *)pvMsg;
                           if (pstrBind && pstrBind->status == 0) {
                                    /* Prepare next buffer reception. */
                                    recvfrom(sock, gau8SocketTestBuffer, TEST_BUFFER_SIZE, 0);
                           }
                           else {
                                    puts("m2m_wifi_socket_handler: bind error!");
                           }
                  }
                  else if (u8Msg == SOCKET_MSG_RECVFROM) {
                           tstrSocketRecvMsg *pstrRx = (tstrSocketRecvMsg *)pvMsg;
```

Listing continued below…

Listing page (2/5)

```
                                        if (pstrRx->pu8Buffer && pstrRx->s16BufferSize) {

                                                /* Check for server report and update led status if necessary. */
                                                t_msg_temp_report report;
                                                memcpy(&report, pstrRx->pu8Buffer, sizeof(t_msg_temp_report));
                                                if (report.id0 == 0 && report.id1 == 2
                                                        && strstr((char *)report.name, DEMO_PRODUCT_NAME)) {
                                                        puts("wifi_nc_data_callback: received app message");
                                                        port_pin_set_output_level(LED_0_PIN, report.led ? true : false);
                                                        delay = 0;
                                                }

                                                /* Prepare next buffer reception. */
                                                recvfrom(sock, gau8SocketTestBuffer, TEST_BUFFER_SIZE, 0);
                                        }
                                        else {
                                                if (pstrRx->s16BufferSize == SOCK_ERR_TIMEOUT) {
                                                        /* Prepare next buffer reception. */
                                                        recvfrom(sock, gau8SocketTestBuffer, TEST_BUFFER_SIZE, 0);
                                                }
                                        }
                                }
                        }
}

/**
 * \brief Callback to get the Wi-Fi status update.
 *
 * \param[in] u8MsgType type of Wi-Fi notification. Possible types are:
 *  - [M2M_WIFI_RESP_CURRENT_RSSI](@ref M2M_WIFI_RESP_CURRENT_RSSI)
 *  - [M2M_WIFI_RESP_CON_STATE_CHANGED](@ref M2M_WIFI_RESP_CON_STATE_CHANGED)
 *  - [M2M_WIFI_RESP_CONNTION_STATE](@ref M2M_WIFI_RESP_CONNTION_STATE)
 *  - [M2M_WIFI_RESP_SCAN_DONE](@ref M2M_WIFI_RESP_SCAN_DONE)
 *  - [M2M_WIFI_RESP_SCAN_RESULT](@ref M2M_WIFI_RESP_SCAN_RESULT)
 *  - [M2M_WIFI_REQ_WPS](@ref M2M_WIFI_REQ_WPS)
 *  - [M2M_WIFI_RESP_IP_CONFIGURED](@ref M2M_WIFI_RESP_IP_CONFIGURED)
 *  - [M2M_WIFI_RESP_IP_CONFLICT](@ref M2M_WIFI_RESP_IP_CONFLICT)
 *  - [M2M_WIFI_RESP_P2P](@ref M2M_WIFI_RESP_P2P)
 *  - [M2M_WIFI_RESP_AP](@ref M2M_WIFI_RESP_AP)
 *  - [M2M_WIFI_RESP_CLIENT_INFO](@ref M2M_WIFI_RESP_CLIENT_INFO)
 * \param[in] pvMsg A pointer to a buffer containing the notification parameters
 * (if any). It should be casted to the correct data type corresponding to the
 * notification type. Existing types are:
 *  - tstrM2mWifiStateChanged
 *  - tstrM2MWPSInfo
 *  - tstrM2MP2pResp
 *  - tstrM2MAPResp
 *  - tstrM2mScanDone
 *  - tstrM2mWifiscanResult
 */
static void m2m_wifi_state(uint8 u8MsgType, void *pvMsg)
{
        switch (u8MsgType) {
                case M2M_WIFI_RESP_CON_STATE_CHANGED: {
                        tstrM2mWifiStateChanged *pstrWifiState = (tstrM2mWifiStateChanged*) pvMsg;
                        if (pstrWifiState->u8CurrState == M2M_WIFI_CONNECTED) {
                                puts("m2m_wifi_state: M2M_WIFI_RESP_CON_STATE_CHANGED: CONNECTED");
                                m2m_wifi_request_dhcp_client();
                        }
                        else if(pstrWifiState->u8CurrState == M2M_WIFI_DISCONNECTED) {
                                puts("m2m_wifi_state: M2M_WIFI_RESP_CON_STATE_CHANGED: DISCONNECTED");
                                wifi_connected = 0;
                                m2m_wifi_connect((char *)DEMO_WLAN_SSID, sizeof(DEMO_WLAN_SSID),
                                                DEMO_WLAN_AUTH, (char *)DEMO_WLAN_PSK, M2M_WIFI_CH_ALL);
                        }
                        break;
                }
                case M2M_WIFI_REQ_DHCP_CONF: {
                        uint8 *pu8IPAddress = (uint8*) pvMsg;
                        wifi_connected = 1;
                        /* Turn LED0 on to declare that IP address received. */
                        port_pin_set_output_level(LED_0_PIN, false);
```

Listing continued below…

Listing page (3/5)

```
                              printf("m2m_wifi_state: M2M_WIFI_REQ_DHCP_CONF: IP is %u.%u.%u.%u\n",
                                                   pu8IPAddress[0], pu8IPAddress[1], pu8IPAddress[2], pu8IPAddress[3]);
                              break;
                    }
          default: {
                              break;
                    }
          }
}

/**
 * \brief Sensor thread entry.
 *
 * \param[in] params unused parameter.
 */
void demo_start(void)
{
          tstrWifiInitParam param;
          struct sockaddr_in addr;
          sint8 ret;

          /* Initialize Wi-Fi parameters structure. */
          param.pfAppWifiCb = m2m_wifi_state;

          /* Initialize socket address structure. */
          addr.sin_family    = AF_INET;
          addr.sin_port = _htons(DEMO_SERVER_PORT);
          addr.sin_addr.s_addr = 0xFFFFFFFF;

          /* Turn LED0 off initially. */
          port_pin_set_output_level(LED_0_PIN, true);

          /* Initialize temperature sensor. */
          at30tse_init();

          /* Reset network controller */
          nm_bsp_init();

          /* Initialize Wi-Fi driver with data and Wi-Fi status callbacks. */
          ret = m2m_wifi_init(&param);
          if (M2M_SUCCESS != ret) {
                    puts("demo_start: nm_drv_init call error!");
                    while (1)
                              ;
          }

          /* Initialize Socket module */
          socketInit();
          registerSocketCallback(m2m_wifi_socket_handler, NULL);

          /* Connect to router. */
          m2m_wifi_connect((char *)DEMO_WLAN_SSID, sizeof(DEMO_WLAN_SSID),
                              DEMO_WLAN_AUTH, (char *)DEMO_WLAN_PSK, M2M_WIFI_CH_ALL);

          while (1) {
                    /* Handle pending events from network controller. */
                    m2m_wifi_handle_events(NULL);

                    if ((wifi_connected == 1) && (ms_ticks - delay > DEMO_REPORT_INTERVAL)) {
                              delay = ms_ticks;

                              /* Open server socket. */
                              if (rx_socket < 0) {
                                        if ((rx_socket = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
                                                  puts("demo_start: failed to create RX UDP client socket error!");
                                                  continue;
                                        }
                                        bind(rx_socket, (struct sockaddr *)&addr, sizeof(struct sockaddr_in));
                              }

                              /* Open client socket. */
                              if (tx_socket < 0) {
                                        if ((tx_socket = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
```

Listing continued below…

Listing page (4/5)

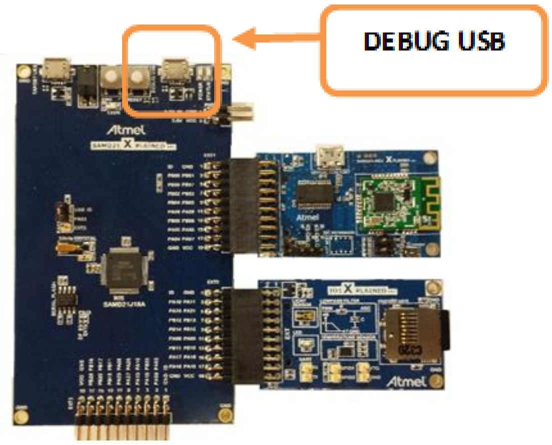
```
                                        puts("demo_start: failed to create TX UDP client socket error!");
                                        continue;
                                }
                        }

                        /* Send client discovery frame. */
                        sendto(tx_socket, &msg_temp_keepalive, sizeof(t_msg_temp_keepalive), 0,
                                        (struct sockaddr *)&addr, sizeof(addr));

                        /* Send client report. */
                        msg_temp_report.temp = (uint32_t)(at30tse_read_temperature() * 100);
                        msg_temp_report.led = !port_pin_get_output_level(LED_0_PIN);
                        ret = sendto(tx_socket, &msg_temp_report, sizeof(t_msg_temp_report), 0,
                                        (struct sockaddr *)&addr, sizeof(addr));

                        if (ret == M2M_SUCCESS) {
                                puts("demo_start: sensor report sent");
                        } else {
                                puts("demo_start: failed to send status report error!");
                        }

                }
        }
}
```

Listing page (5/5)

TO DO Build the solution (F7) and ensure you get no errors:

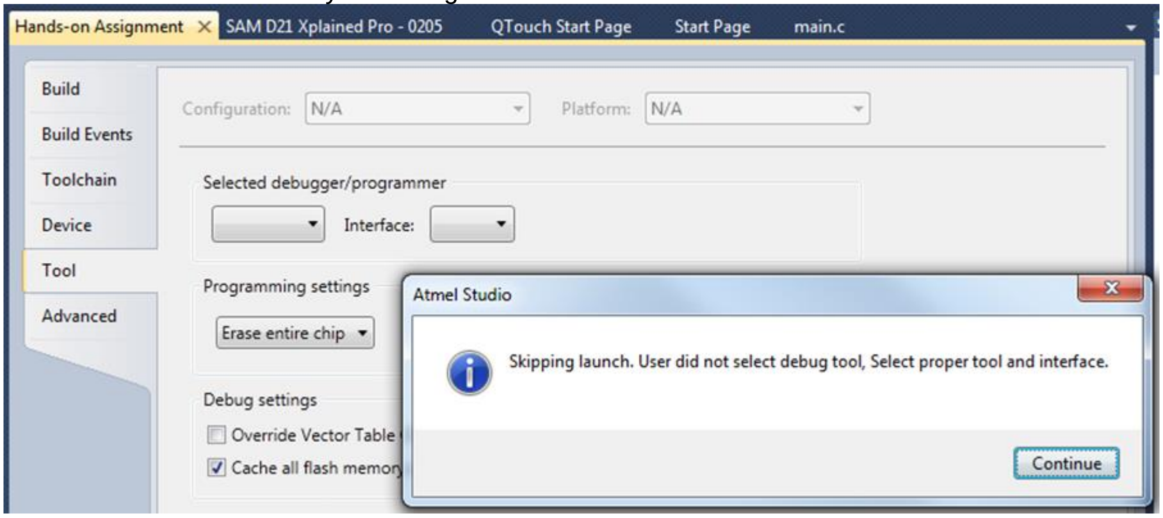
TO DO Program the SAM D21 Xplained Pro.

- Connect the WINC1500 Wi-Fi extension and the IO1 extension to the SAM D21 Xplained Pro as displayed:



- Connect the SAM D21 Xplained Pro board to your PC using DEBUG USB connector
- Program the application by clicking on the Start Debugging and Break icon:

- You will be asked to select your debug tool:



Hands-on Assignment | SAM D21 Xplained Pro - 0205 | QTouch Start Page | Start Page | main.c

Build
Build Events
Toolchain
Device
Tool
Advanced

Configuration: N/A Platform: N/A

Selected debugger/programmer
Interface:

Programming settings
Erase entire chip

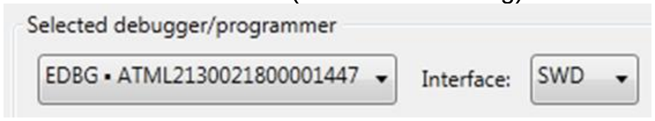
Debug settings
☐ Override Vector Table
☑ Cache all flash memor

Atmel Studio

Skipping launch. User did not select debug tool, Select proper tool and interface.

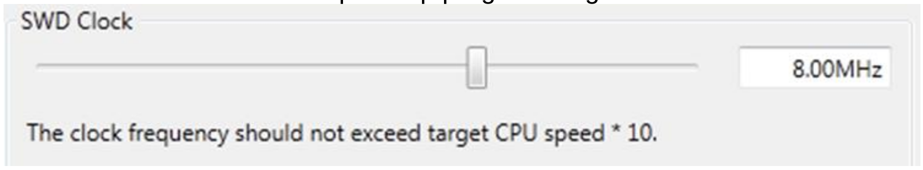
Continue

- Select EDBG and SWD (Serial Wire Debug) as Interface:

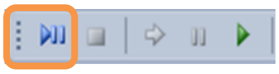


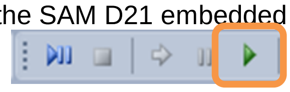
Selected debugger/programmer
EDBG • ATML2130021800001447 Interface: SWD

- Set SWD clock to 8MHz to speed up programming:

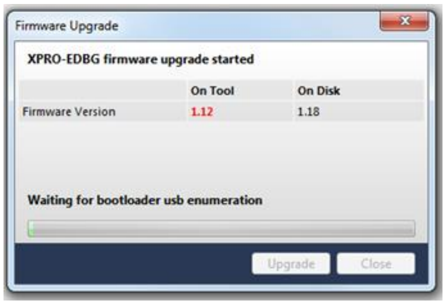


SWD Clock
8.00MHz
The clock frequency should not exceed target CPU speed * 10.

- Click again on the Start Debugging and Break icon: 

- The application will be programmed in the SAM D21 embedded flash and breaks at main function. Click on Continue to execute the application: 

INFO You may be asked to upgrade your EDBG firmware. If so, click on Upgrade:



Firmware Upgrade

XPRO-EDBG firmware upgrade started

	On Tool	On Disk
Firmware Version	1.12	1.18

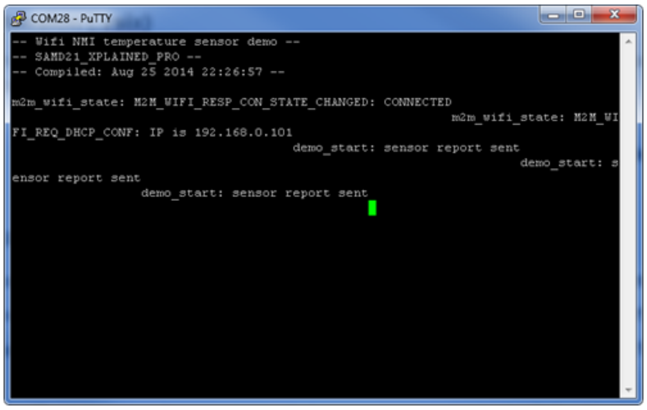
Waiting for bootloader usb enumeration

Upgrade Close

WARNING Upgrade operation may take a few minutes, **wait** for the operation to complete.

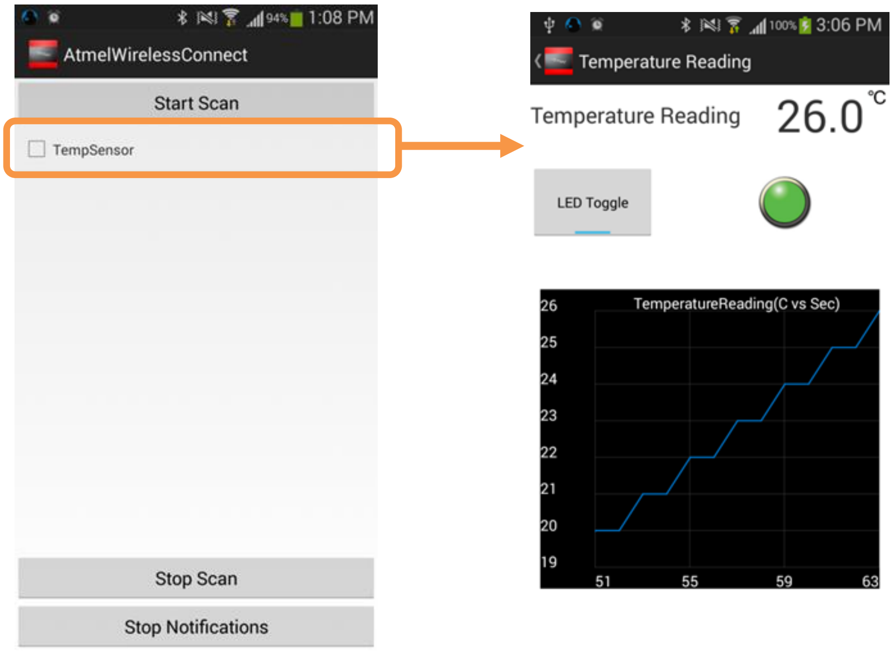
RESULT The IOT sensor application is now programmed and running.

Open the EDBG DEBUG USB serial COM port, with the following settings: 115200 bauds, 8 bit data, no parity, one stop bit and no flow control.



```
COM28 - PuTTY
-- Wifi NMI temperature sensor demo --
-- SAMD21_XPLAINED_PRO --
-- Compiled: Aug 25 2014 22:26:57 --

m2m_wifi_state: M2M_WIFI_RESP_CON_STATE_CHANGED: CONNECTED
                                                                m2m_wifi_state: M2M_WI
FI_REQ_DHCP_CONF: IP is 192.168.0.101
                                     demo_start: sensor report sent
                                                              demo_start: s
ensor report sent
                 demo_start: sensor report sent
```

- Connect the Android device to the same SSID network than the WINC1500 Wi-Fi module
- Open the Temperature sensor application on the Android device
- Hit the “Start Scan” button
- A “TempSensor” device (as defined in the `DEMO_PRODUCT_NAME` macro in demo.h file) appears



- Hit the “TempSensor” entry from the list
- The Android application now displays real-time temperature information from the SAM D21 Xplained Pro board
- Hit the “LED Toggle” button to toggle LED0 on the SAM D21 Xplained Pro board

RESULT Congratulations, you just built your first IOT sensor application.

16 Host Interface Protocol

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

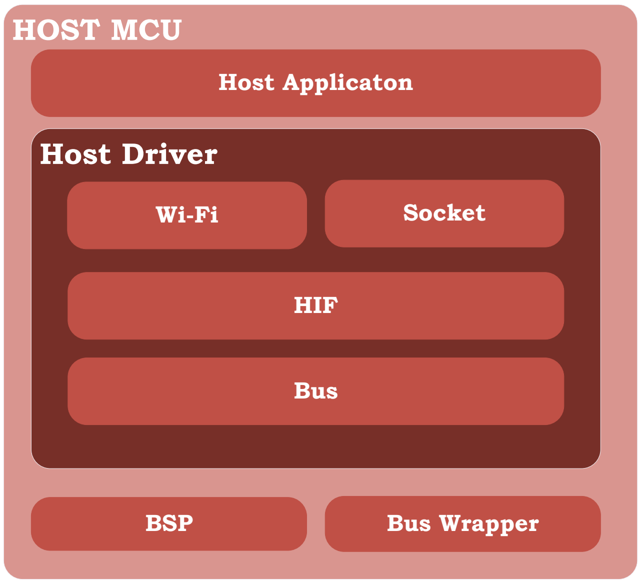
The host driver implements services asynchronously. This means that when the application calls an API to request a service action, the call is non-blocking and returns immediately, often before the action is completed. Where appropriate, notification that an action has completed is provided in a subsequent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

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

WARNING Dealing with HIF messages in host MCU application is an advanced topic. For most applications, it is recommended to use Wi-Fi and socket layers. Both layers hide the complexity of the HIF APIs.

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

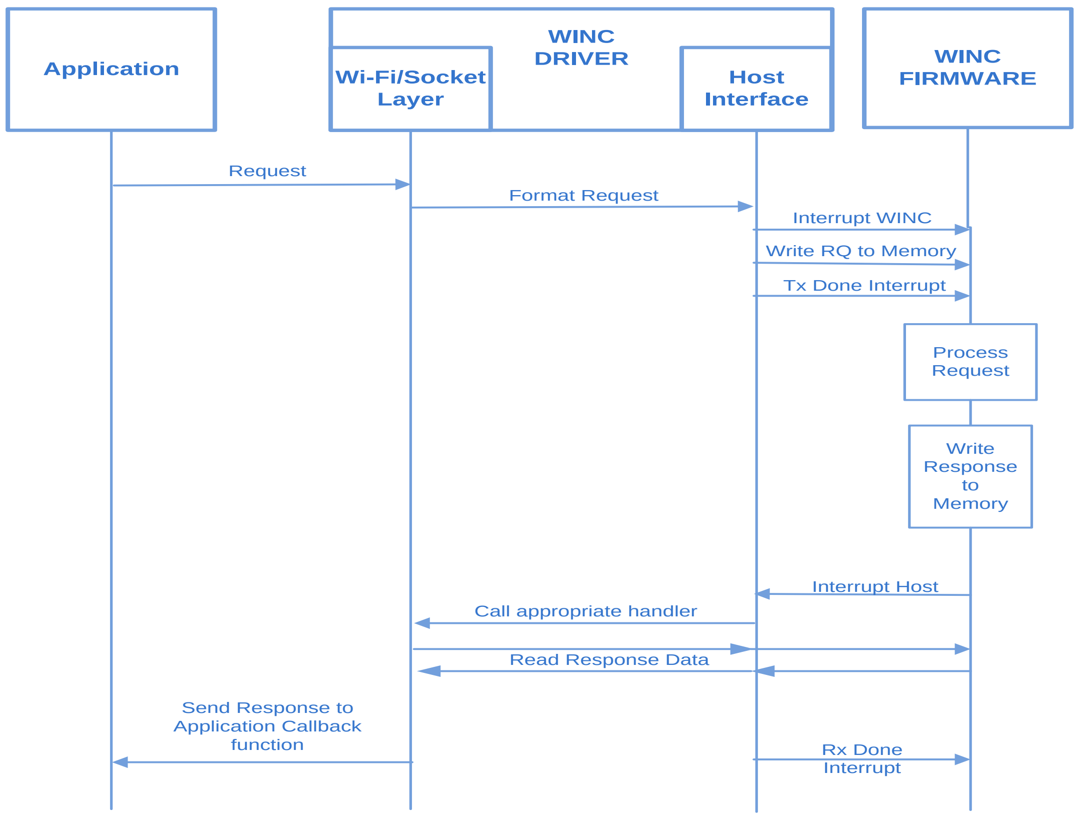
Figure 16-1. WINC Driver Layers



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

The Request/Response sequence between the Host and the WINC chip is shown in Figure 16-2.

Figure 16-2. The Request/Response Sequence Diagram



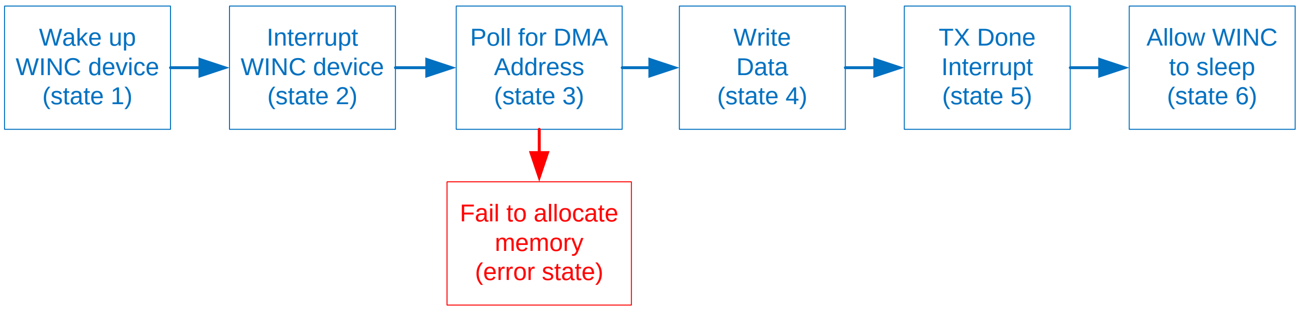
16.1 Transfer Sequence Between HIF Layer and WINC Firmware

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

16.1.1 Frame Transmit

Figure 16-3 shows the steps and states involved in sending a message from the host to the WINC device.

Figure 16-3. HIF Frame Transmit to WINC

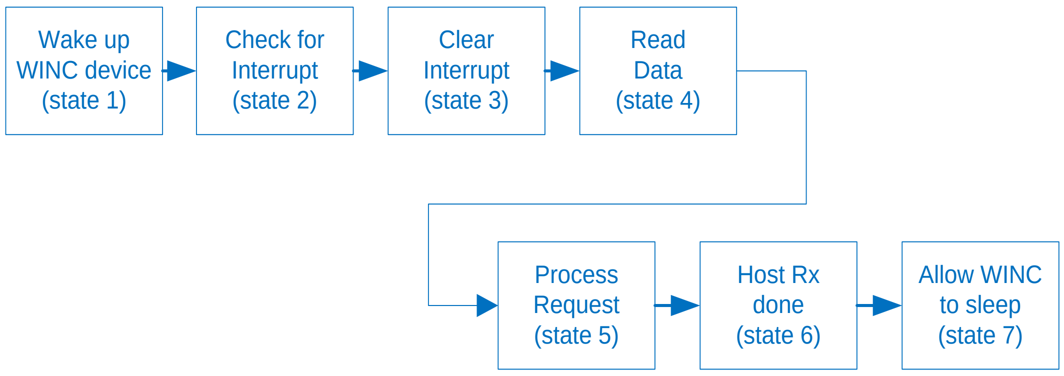

Wake up WINC device (state 1)
Interrupt WINC device (state 2)
Poll for DMA Address (state 3)
Write Data (state 4)
TX Done Interrupt (state 5)
Allow WINC to sleep (state 6)
Fail to allocate memory (error state)

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

16.1.2 Frame Receive

Figure 16-4 shows the steps and states involved in sending a message from the WINC device to the host:

Figure 16-4. HIF Frame Receive from WINC to Host

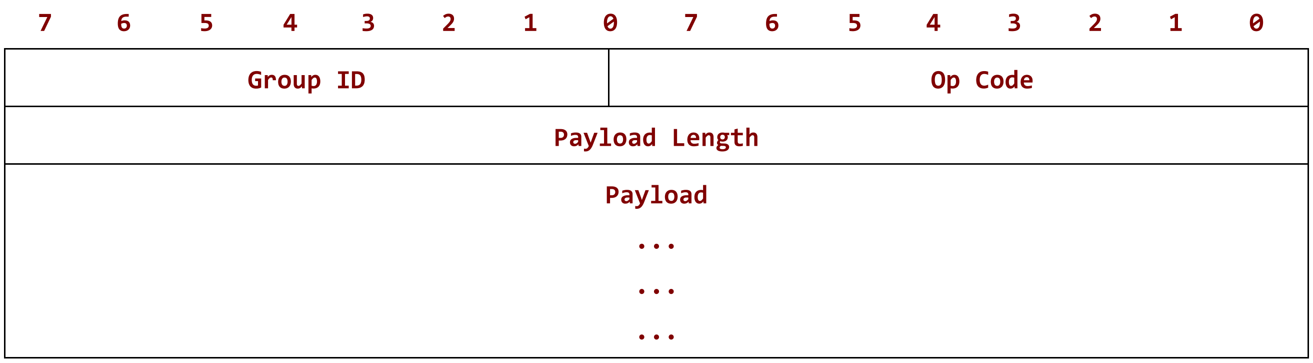

Wake up WINC device (state 1)
Check for Interrupt (state 2)
Clear Interrupt (state 3)
Read Data (state 4)
Process Request (state 5)
Host Rx done (state 6)
Allow WINC to sleep (state 7)

Step	Description
Step (1) Wake up the WINC device	Wakeup the device to be able to receive Host requests.
Step (2) Check for Interrupt	Monitor BIT[0] of WIFI_HOST_RCV_CTRL_0 register. Disable the host from receiving interrupts (until this one has been processed).
Step (3) Clear interrupt	Write zero to BIT[0] of WIFI_HOST_RCV_CTRL_0 register.
Step (4) Read Data	Get the address of the data block from WIFI_HOST_RCV_CTRL_1 register. Read Data block with size obtained from WIFI_HOST_RCV_CTRL_0 register BIT[13] <->BIT[2].
Step (5) Process Request	Parse the HIF header at the start of the Data and forward the Data to the appropriate registered Callback function.

Step	Description
Step (6) HOST RX Done	Raise an interrupt for the chip to free the memory holding the data by setting BIT[1] of WIFI_HOST_RCV_CTRL_0 register. Enable Host interrupt reception again.
Step (7) Allow WINC device to sleep	Allow the WINC device to enter sleep mode again (if it wishes).

16.2 HIF Message Header Structure

The HIF message is the data structure exchanged back and forth between the Host Interface and WINC 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_IP`, `M2M_REQ_GRP_HIF`, `M2M_REQ_GRP_OTA` corresponding to Wi-Fi, Socket, HIF and OTA respectively. A group ID can be assigned one of the values enumerated in `tenuM2mReqGrp`.
- **Op Code: (8-bit)**: is a command number. Valid command number is a value enumerated in: `tenuM2mConfigCmd and tenuM2mStaCmd`, `tenuM2mApCmd`, and `tenuM2mP2pCmd` corresponding to configuration, STA mode AP mode and P2P mode commands. See the full list of commands in the header file `m2m_types.h`.
- **Payload Length (16-bits)**: The payload length in bytes (does not include header).

16.3 HIF Layer APIs

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

The following API functions are described:

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

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

`sint8 hif_chip_wake(void):`

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

`sint8 hif_chip_sleep(void):`

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

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

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

`sint8 hif_isr(void):`

This is the Host interface interrupt service routine. It handles interrupts generated by the WINC 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 WINC chip. The location and length of the data must be known in advance and specified. This will typically have been extracted from an earlier part of a transaction.

`sint8 hif_send(uint8 u8Gid,uint8 u8Opcode,uint8 *pu8CtrlBuf,uint16 u16CtrlBufSize,uint8 *pu8DataBuf,uint16 u16DataSize, uint16 16DataOffset):`

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

16.4 Scan Code Example

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

- The application requests a Wi-Fi scan

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

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

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

- The HIF layer sends the request to the WINC chip

```
sint8 hif_send(uint8 u8Gid,uint8 u8Opcode,uint8 *pu8CtrlBuf,uint16 u16CtrlBufSize,
                              uint8 *pu8DataBuf,uint16 u16DataSize, uint16 u16DataOffset)
{
         sint8 ret = M2M_ERR_SEND;
         volatile tstrHifHdr strHif;
```

```
strHif.u8Opcode = u8Opcode&(~NBIT7);
strHif.u8Gid = u8Gid;
strHif.u16Length = M2M_HIF_HDR_OFFSET;
if(pu8DataBuf != NULL)
{
        strHif.u16Length += u16DataOffset + u16DataSize;
}
else
{
        strHif.u16Length += u16CtrlBufSize;
}
/* TX STEP (1) */
ret = hif_chip_wake();
if(ret == M2M_SUCCESS)
{
        volatile uint32 reg, dma_addr = 0;
        volatile uint16 cnt = 0;

        reg = 0UL;
        reg |= (uint32)u8Gid;
        reg |= ((uint32)u8Opcode<<8);
        reg |= ((uint32)strHif.u16Length<<16);
        ret = nm_write_reg(NMI_STATE_REG,reg);
        if(M2M_SUCCESS != ret) goto ERR1;
        reg = 0;
        /* TX STEP (2) */
        reg |= (1<<1);
        ret = nm_write_reg(WIFI_HOST_RCV_CTRL_2, reg);
        if(M2M_SUCCESS != ret) goto ERR1;
        dma_addr = 0;
        for(cnt = 0; cnt < 1000; cnt ++)
        {
                ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_2,(uint32 *)&reg);
                if(ret != M2M_SUCCESS) break;
                if (!(reg & 0x2))
                {
                        /* TX STEP (3) */
                        ret = nm_read_reg_with_ret(0x150400,(uint32 *)&dma_addr);
                        if(ret != M2M_SUCCESS) {
                /*in case of read error clear the dma address and return error*/
                                dma_addr = 0;
                        }
                        /*in case of success break */
                                                break;
                }
        }
        if (dma_addr != 0)
        {
                volatile uint32      u32CurrAddr;
                u32CurrAddr = dma_addr;
                strHif.u16Length=NM_BSP_B_L_16(strHif.u16Length);
                /* TX STEP (4) */
                ret = nm_write_block(u32CurrAddr, (uint8*)&strHif, M2M_HIF_HDR_OFFSET);
                if(M2M_SUCCESS != ret) goto ERR1;
                u32CurrAddr += M2M_HIF_HDR_OFFSET;
                if(pu8CtrlBuf != NULL)
```

```
                        {
                                ret = nm_write_block(u32CurrAddr, pu8CtrlBuf, u16CtrlBufSize);
                                if(M2M_SUCCESS != ret) goto ERR1;
                                u32CurrAddr += u16CtrlBufSize;
                        }
                        if(pu8DataBuf != NULL)
                        {
                                u32CurrAddr += (u16DataOffset - u16CtrlBufSize);
                                ret = nm_write_block(u32CurrAddr, pu8DataBuf, u16DataSize);
                                if(M2M_SUCCESS != ret) goto ERR1;
                                u32CurrAddr += u16DataSize;
                        }
                        reg = dma_addr << 2;
                        reg |= (1 << 1);
                        /* TX STEP (5) */
                        ret = nm_write_reg(WIFI_HOST_RCV_CTRL_3, reg);
                        if(M2M_SUCCESS != ret) goto ERR1;
                }
                else
                {
                        /* ERROR STATE */
                        M2M_DBG("Failed to alloc rx size\r");
                        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;}
```

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

```
static sint8 hif_isr(void)
{
        sint8 ret = M2M_ERR_BUS_FAIL;
        uint32 reg;
        volatile tstrHifHdr strHif;
        /* RX STEP (1) */
        ret = hif_chip_wake();
        if(ret == M2M_SUCCESS)
        {
                /* RX STEP (2) */
                ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_0, &reg);
                if(M2M_SUCCESS == ret)
                {
                        /* New interrupt has been received */
                        if(reg & 0x1)
                        {
                                uint16 size;
                                nm_bsp_interrupt_ctrl(0);
                                /*Clearing RX interrupt*/
                                ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_0,&reg);
```

```
            if(ret != M2M_SUCCESS)goto ERR1;
            reg &= ~(1<<0);
            /* RX STEP (3) */
            ret=nm_write_reg(WIFI_HOST_RCV_CTRL_0,reg);
            if(ret != M2M_SUCCESS)goto ERR1;
            /* read the rx size */
            ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_0, &reg);
            if(M2M_SUCCESS != ret)
            {
                    M2M_ERR("(hif) WIFI_HOST_RCV_CTRL_0 bus fail\n");
                    nm_bsp_interrupt_ctrl(1);
                    goto ERR1;
            }
            gu8HifSizeDone = 0;
            size = (uint16)((reg >> 2) & 0xfff);
            if (size > 0) {
                    uint32 address = 0;
                    /**
                    start bus transfer
                    **/
                    /* RX STEP (4) */
                    ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_1, &address);
                    if(M2M_SUCCESS != ret)
                    {
                        M2M_ERR("(hif) WIFI_HOST_RCV_CTRL_1 bus fail\n");
                        nm_bsp_interrupt_ctrl(1);
                        goto ERR1;
                    }
                    ret = nm_read_block(address, (uint8*)&strHif, sizeof(tstrHifHdr));
                    strHif.u16Length = NM_BSP_B_L_16(strHif.u16Length);
                    if(M2M_SUCCESS != ret)
                    {
                        M2M_ERR("(hif) address bus fail\n");
                        nm_bsp_interrupt_ctrl(1);
                        goto ERR1;
                    }
                    if(strHif.u16Length != size)
                    {
                        if((size - strHif.u16Length) > 4)
                        {
                            M2M_ERR("(hif) Corrupted packet Size = %u <L = %u, G = %u, OP
= %02X>\n",
                                size, strHif.u16Length, strHif.u8Gid, strHif.u8Opcode);
                            nm_bsp_interrupt_ctrl(1);
                            ret = M2M_ERR_BUS_FAIL;
                            goto ERR1;
                         }
                    }

                    /* RX STEP (5) */
                    if(M2M_REQ_GRP_WIFI == strHif.u8Gid)
                    {
                        if(pfWifiCb)
                            pfWifiCb(strHif.u8Opcode,strHif.u16Length - M2M_HIF_HDR_OFFSET,
                                        address + M2M_HIF_HDR_OFFSET);

                    }
                    else if(M2M_REQ_GRP_IP == strHif.u8Gid)
                    {
                        if(pfIpCb)
                            pfIpCb(strHif.u8Opcode,strHif.u16Length - M2M_HIF_HDR_OFFSET,
                                        address + M2M_HIF_HDR_OFFSET);
                    }
                    else if(M2M_REQ_GRP_OTA == strHif.u8Gid)
```

```
                                    {
                                          if(pfOtaCb)
                                                pfOtaCb(strHif.u8Opcode,strHif.u16Length - M2M_HIF_HDR_OFFSET,
                                                                    address + M2M_HIF_HDR_OFFSET);
                                    }
                                    else
                                    {
                                          M2M_ERR("(hif) invalid group ID\n");
                                          ret = M2M_ERR_BUS_FAIL;
                                          goto ERR1;
                                    }
                                    /* RX STEP (6) */
                                    if(!gu8HifSizeDone)
                                    {
                                          M2M_ERR("(hif) host app didn't set RX Done\n");
                                          ret = hif_set_rx_done();
                                    }
                              }
                              else
                              {
                                    ret = M2M_ERR_RCV;
                                    M2M_ERR("(hif) Wrong Size\n");
                                    goto ERR1;
                              }
                        }
                        else
                        {
#ifndef WIN32
                              M2M_ERR("(hif) False interrupt %lx",reg);
#endif
                        }
                  }
                  else
                  {
                        M2M_ERR("(hif) Fail to Read interrupt reg\n");
                        goto ERR1;
                  }
            }
            else
            {
                  M2M_ERR("(hif) FAIL to wakeup the chip\n");
                  goto ERR1;
            }
            /* RX STEP (7) */
            ret = hif_chip_sleep();
ERR1:
            return ret;
}
```

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

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

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

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

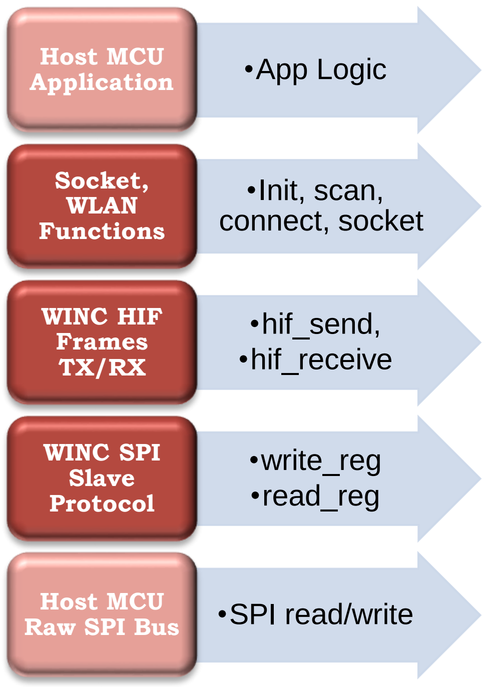
17 WINC SPI Protocol

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

The WINC SPI protocol consists of three layers:

- Layer 1: WINC SPI slave protocol, which allows the host MCU application to perform register/memory read and write operation in the WINC1500 device using raw SPI data exchange
- Layer 2: Host MCU application uses the register and memory read and write capabilities to exchange host interface frames with the WINC firmware. It also provides asynchronous callback from the WINC firmware to the host MCU through interrupts and host interface RX frames. This layer was discussed earlier in Chapter 16: Host Interface Protocol.
- Layer 3: Allows the host MCU application to exchange high level messages (e.g. Wi-Fi scan, socket connection, or TCP data received) with the WINC firmware to employ in the host MCU application logic

Figure 17-1. WINC SPI Protocol Layers



17.1 Introduction

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

- Command
- Response
- Data

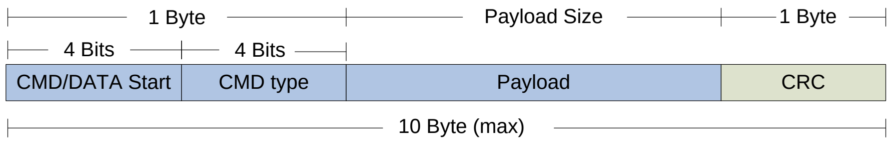
In the case of Command and Data messages, the last byte is used as data integrity check.

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

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

17.1.1 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 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: Three bytes
- B: Five bytes
- C: Six bytes
- D: Seven bytes

Type A commands include:

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

Type B commands include:

- DMA RD Transaction
- DMA WR Transaction

Type C commands include:

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

Type D commands include:

- DMA single-word WR

Full details of the frame format fields are provided in the following table:

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

Payload	A: 3 B: 5 C: 6 D: 7	The Payload field may be of Type A, B, C, or D **<u>Type A (length 3)</u>** ***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 (two 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*** *Param: none* *Payload bytes:* B0: 0 B1: 0 B2: 0 ***5- Soft reset command*** *Param: none* *Payload bytes:* B0: 0xFF B1: 0xFF B2: 0xFF **<u>Type B (length 5)</u>** ***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]

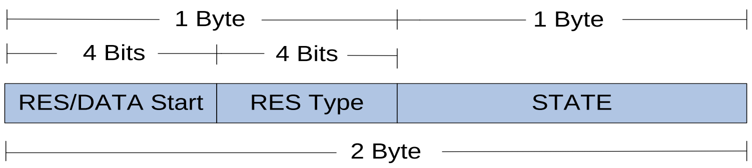
Field	Size	Description
		<u>**Type C (length 6)**</u> ***1- DMA Extended RD transaction*** *Params:* DMA Start Address: 3 bytes DMA extended count: 3 bytes *Payload bytes:* B0: ADDRESS[23:16] B1: ADDRESS[15:8] B2: ADDRESS[7:0] B3: COUNT[23:16] B4: COUNT[15:8] B5: COUNT[7:0] ***2- DMA Extended WR transaction*** *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] <u>**Type D (length 7)**</u> ***1- DMA single-word WR*** *Params:* 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 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):

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

17.1.2 Response Format

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



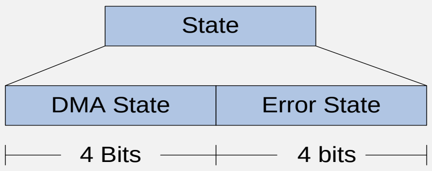
The first byte contains two four bit fields which identify the response message and the response type.

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

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

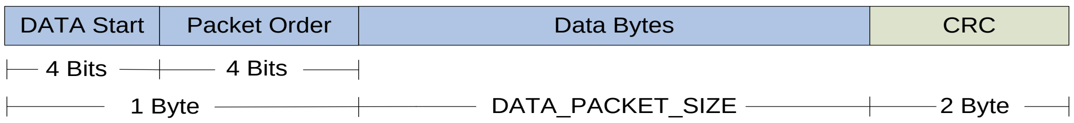
States that may be indicated are:

- 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
State	1 byte	This field is divided into two subfields:  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

17.1.3 Data Packet Format

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

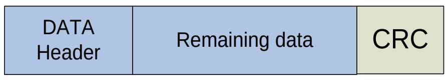


To support DMA hardware a large data transfer may be fragmented into multiple smaller Data Packets. This is controlled by the value of DATA_PACKET_SIZE which is agreed between the master and slave in software and is a fixed value such as 256B, 512B, 1KB (default), 2KB, 4KB, or 8KB. If a transfer has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

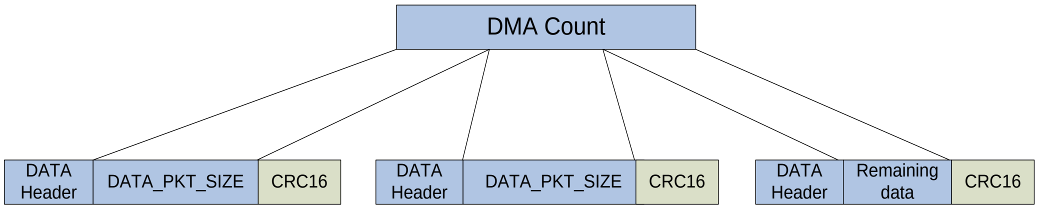
(m – (n-1)* DATA_PACKET_SIZE).This is shown diagrammatically below:

- **If DMA count <= DATA_PACKET_SIZE**

  The data packet is “*DATA_Header + DMA count +optional CRC16*“, i.e. no padding.



- **If DMA count > DATA_PACKET_SIZE**



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

"DATA_Header + remaining data + optional CRC16 ", i.e. no padding.

The frame fields are describe in detail in the following table:

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 transac-tion 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

17.1.4 Error Recovery Mechanism

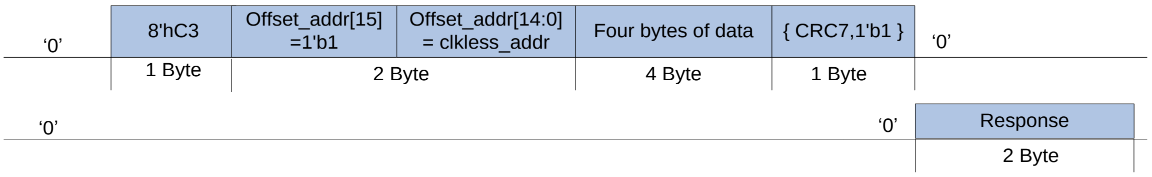
Error Type	Recovery Mechanism
Master:	
CRC error in command	1. Error response received from slave. 2. Retransmit the command
CRC error in received data	1. Issue a repeat command for the data packet that has a CRC error. 2. Slave sends a response to the previous command. 3. Slave keeps the start DMA address of the previous data packet, so it can retransmit it. 4. Receive the data packet again.

Error Type	Recovery Mechanism
No response is received from slave	• Synchronization is lost between master and slave • The worst case is when slave is in receiving data state • Solution: master should wait for max DATA_PACKET_SIZE period then generate a soft reset command
Unexpected response	Retransmit the command
TX/RX Data count error	Retransmit the command
No response to soft reset command	• Transmit all ones till 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 • waits for command retransmission
Received data CRC error	• Send response with error • wait for retransmission of the data packet
Internal general error	• The master should soft reset the slave
TX/RX Data count error	• Only the master can detect this error • Slave operates with the data count received till the count finishes or the master terminates the transaction • In both cases the master should retry the command from the beginning
No response to soft reset command	1. First received 4'b1001, it decides data start. 2. Then received packet order 4'b1111 that is reserved value. 3. Then monitors for 7 bytes all ones to decide Soft Reset action. 4. The slave should activate the output data line. 5. Waits for deactivation for the received line. 6. The slave then deactivates the output data line and returns to the CMD/DATA start monitor state.
General NOTE	• The slave should monitor the received line for command reception in any time • When a CMD start is detected, the slave will receive 8 bytes then return again to the command reception state • When the slave is transmitting data, it should also monitor for command reception • When the slave is receiving data, it will monitor for command reception between the data packets • Therefore issuing a soft reset command, should be detected in all cases

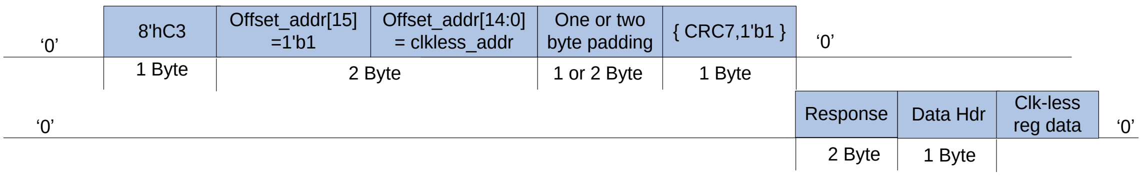
17.1.5 Clockless Registers Access

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

For clock-less register **write**: - the protocol master should wait for the response as shown here:



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

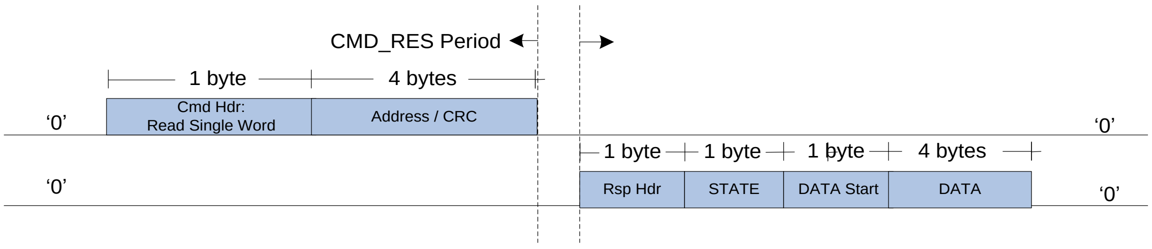


17.2 Message Flow for Basic Transactions

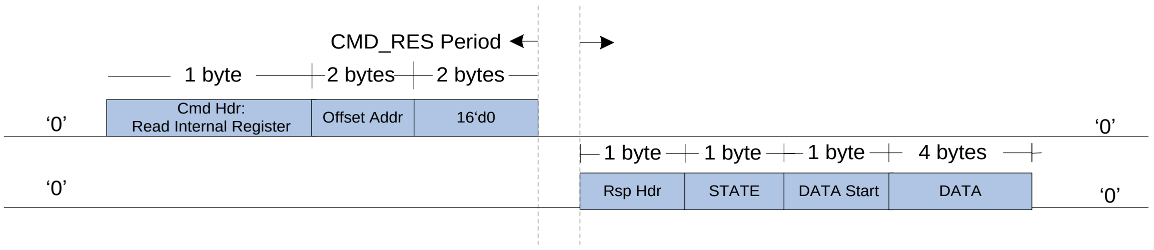
This section shows the essential message exchanges and timings associated with the following commands:

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

17.2.1 Read Single Word



17.2.2 Read Internal Register (for clockless registers)



17.2.3 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.

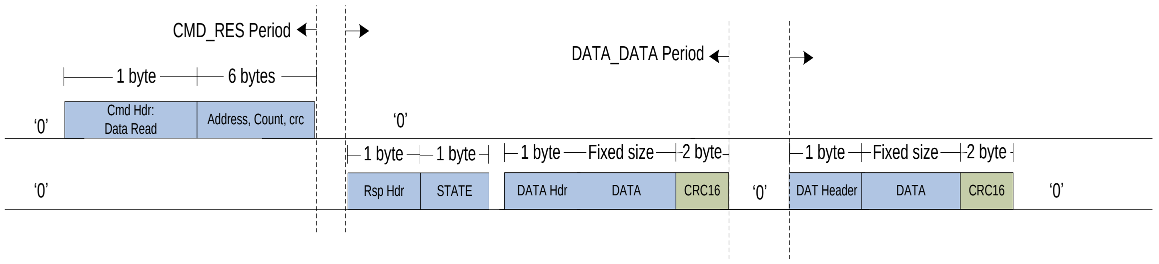
Slave: sends the data packets, separated by DATA_DATA_PERIOD[5] where DATA_DATA_PERIOD is controlled by software and has one of these values:

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

Slave: continues sending till the count ends.

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

The message sequence for this case is shown below:



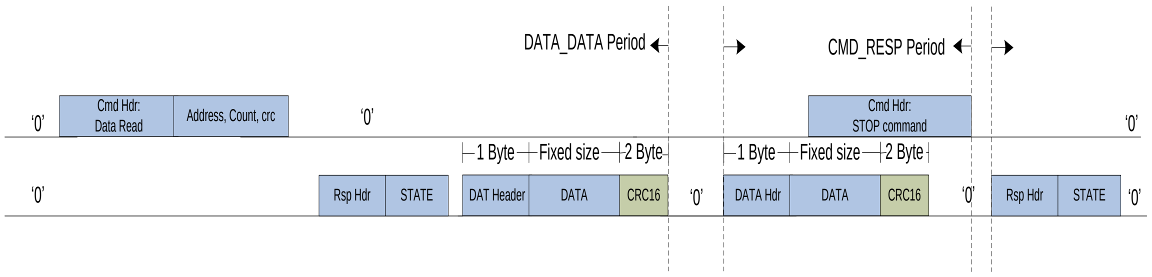
Termination command is issued:

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

Master: should monitor for RES_START after CMD_RESP_PERIOD.

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

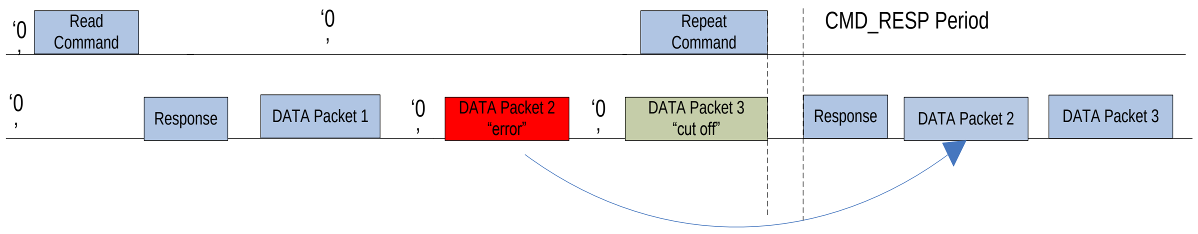
Slave: should respond to the termination command after CMD_RESP_PERIOD from the end of the termination command packet.



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

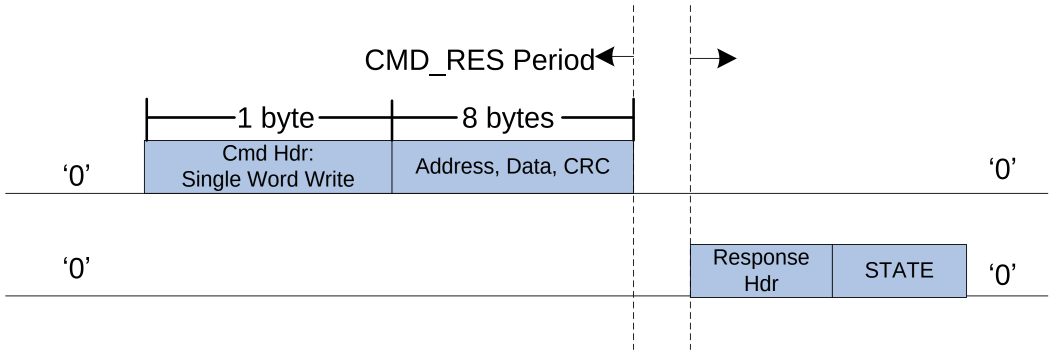
Repeat command is issued:

1. Master: can issue a repeat command at any time during the transaction.
2. Master: should monitor for RES_START after CMD_RESP_PERIOD.
3. Slave: should cut off the current running data packet, if any.
4. Slave: should respond to the repeat command after CMD_RESP_PERIOD from the end of the repeat command packet.
5. Slave: Resends the data packet that has an error then continues the transaction as normal.



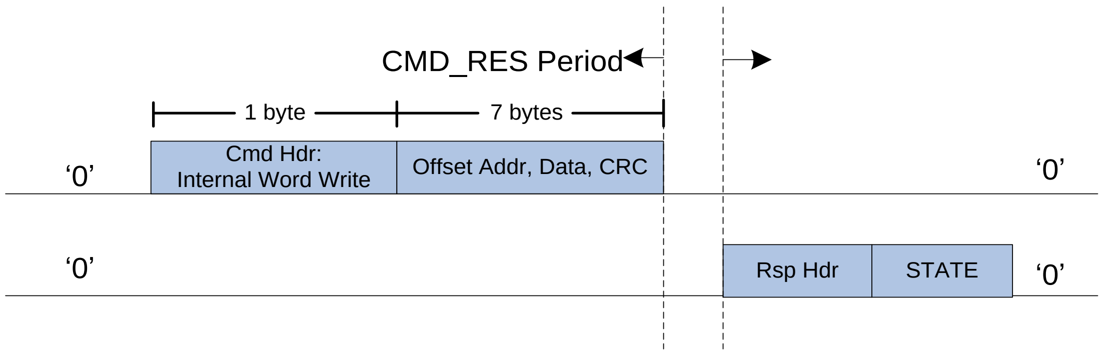
17.2.4 Write Single Word

1. Master: issues DMA single-word write command, including the data.
2. Slave: takes the data and sends a command response.



17.2.5 Write Internal Register (for clockless registers)

1. Master: issues an internal register write command, including the data.
2. Slave: takes the data and sends a command response.



17.2.6 Write Block

Case 1: Master waits for a command response:

1. Master: issues a DMA write command and waits for a response.
2. Slave: sends response after CMD_RES_PERIOD.
3. Master: sends the data packets after receiving response.
4. Slave: sends a response packet for each data packet received after DATA_RES_PERIOD.
5. Master: does not wait for the data response before sending the following data packet notes:

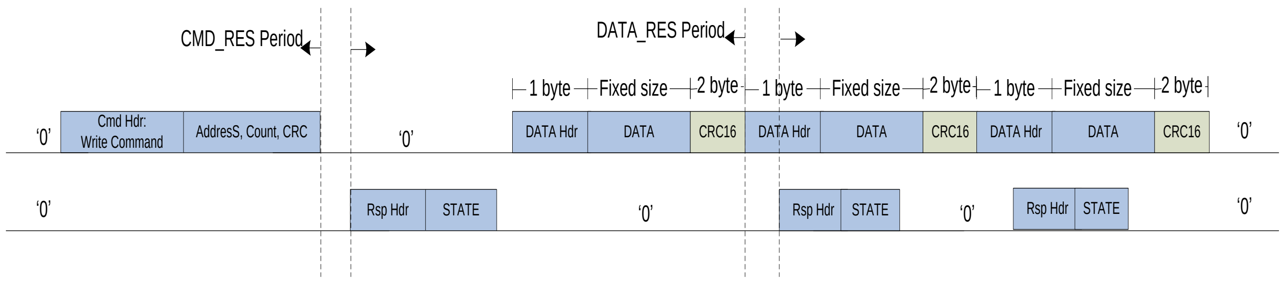
   CMD_RES_PERIOD is controlled by SW taking one of the values:

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

   The master should monitor for RES_START after CMD_RES_PERIOD

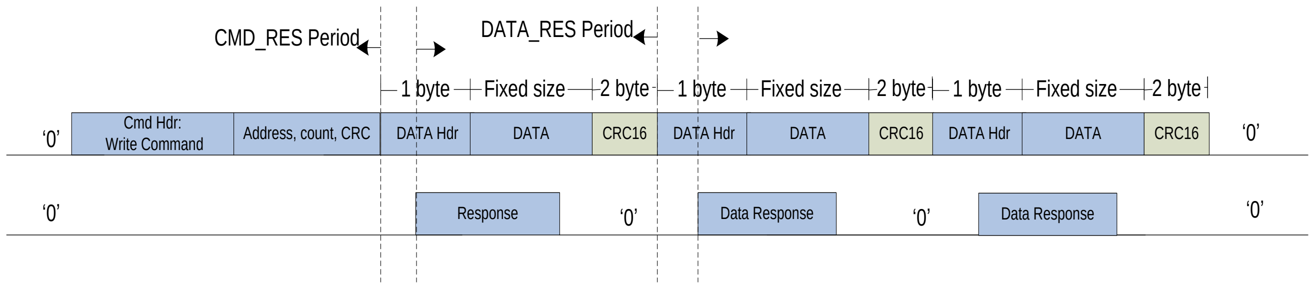
   DATA_RES_PERIOD is controlled by SW taking one of the values:

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



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

1. Master: sends the data packets directly after the command but it still monitors for a command response after CMD_RESP_PERIOD.
2. Master: retransmits the data packets if there is an error in the command.



17.3 SPI Level Protocol Example

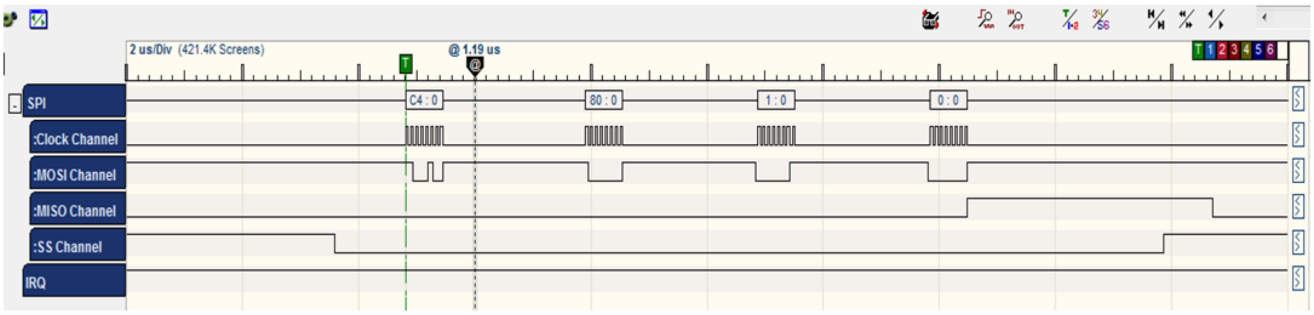
In order to illustrate how WINC SPI protocol works, SPI Bytes from the scan request example were dumped and the sequence is described below.

17.3.1 TX (Send Request)

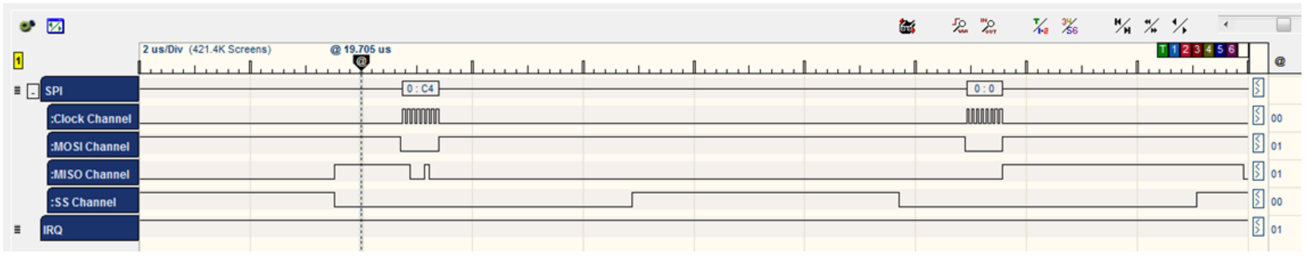
First step in hif_send() API is to wake up the chip:

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

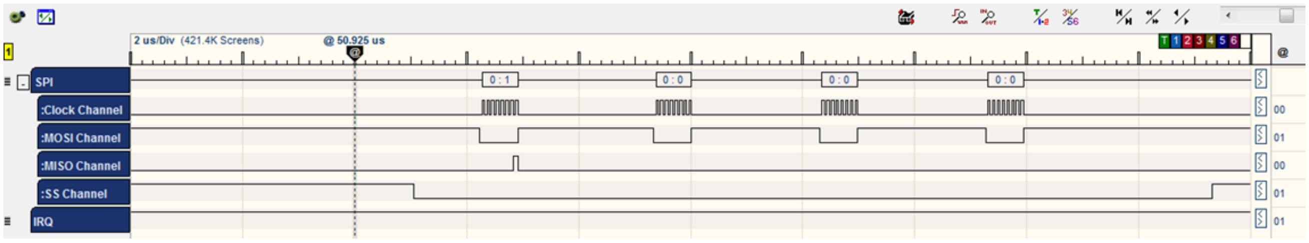
```
Command      CMD_INTERNAL_READ:  0xC4               /* internal register read */
        BYTE [0] = CMD_INTERNAL_READ
        BYTE [1] = address >> 8;                    /* address = 0x01 */
        BYTE [1] |= (1 << 7);                           /* clockless register */
        BYTE [2] = address;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



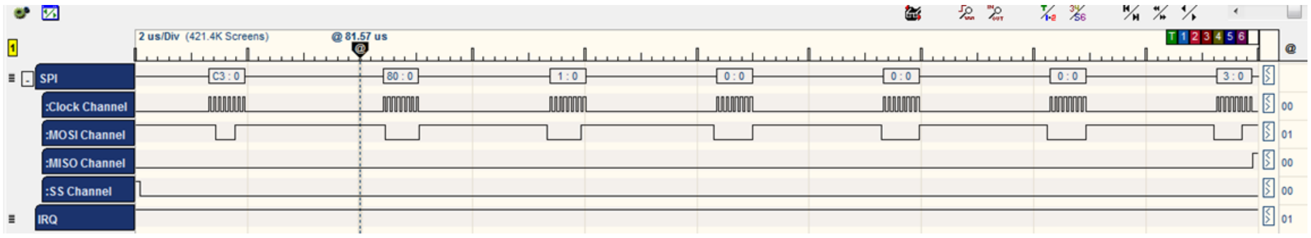
WINC acknowledges the command by sending three bytes [C4] [0] [F3].



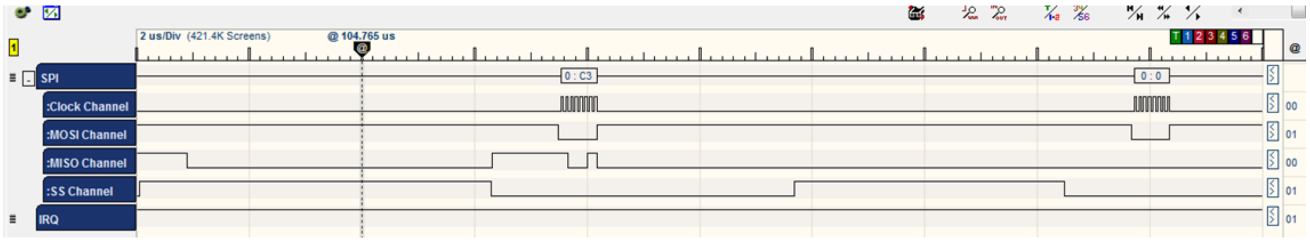
Then WINC chip sends the value of the register 0x01 which equals 0x01.



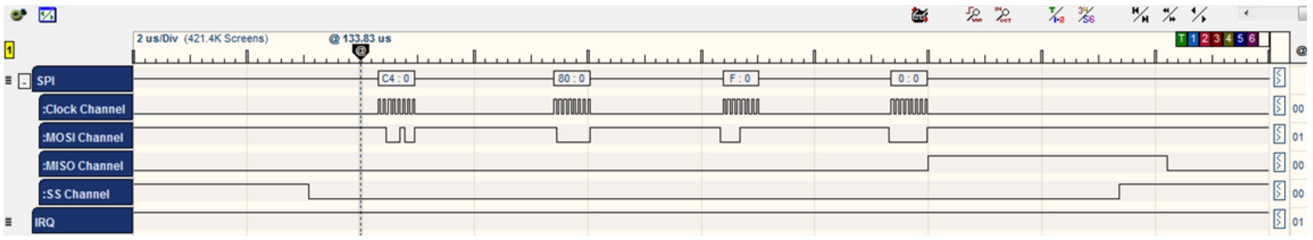
```
Command         CMD_INTERNAL_WRITE:   C3              /*    internal register write */
                BYTE [0] = CMD_INTERNAL_WRITE
                BYTE [1] = address >> 8;                     /*     address = 0x01             */
                BYTE [1] |= (1 << 7);               /*     clockless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



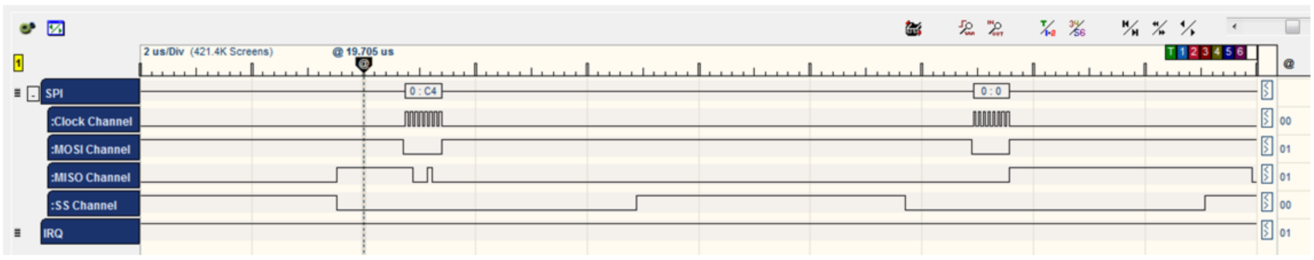
WINC acknowledges the command by sending two bytes [C3] [0].



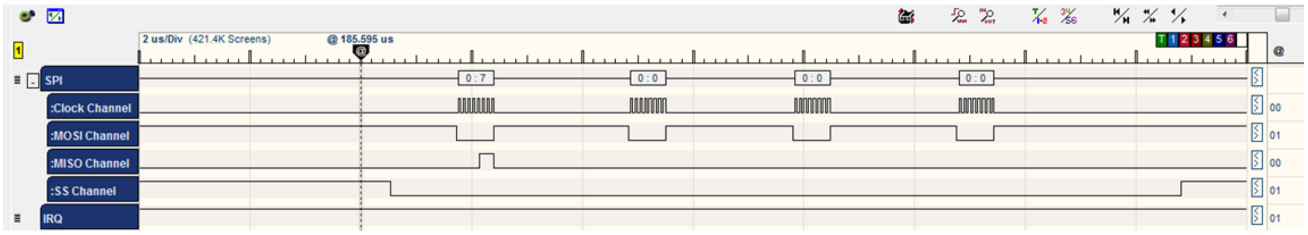
```
Command         CMD_INTERNAL_READ:    0xC4          /*     internal register read       */
                BYTE [0] = CMD_INTERNAL_READ
                BYTE [1] = address >> 8;                     /*     address = 0x0F             */
                BYTE [1] |= (1 << 7);                        /*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



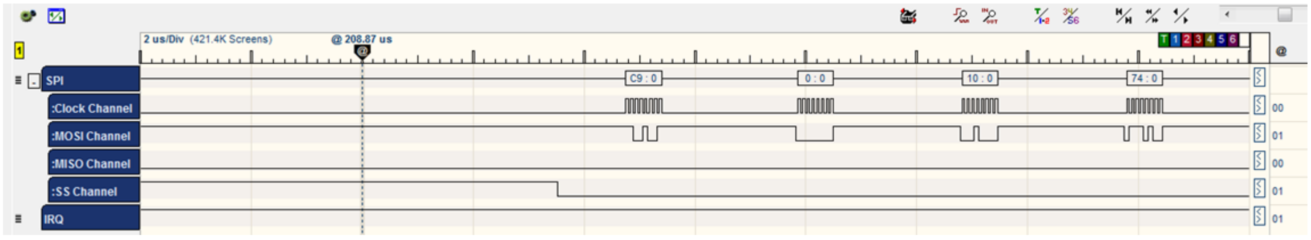
WINC acknowledges the command by sending three bytes [C4] [0] [F3].

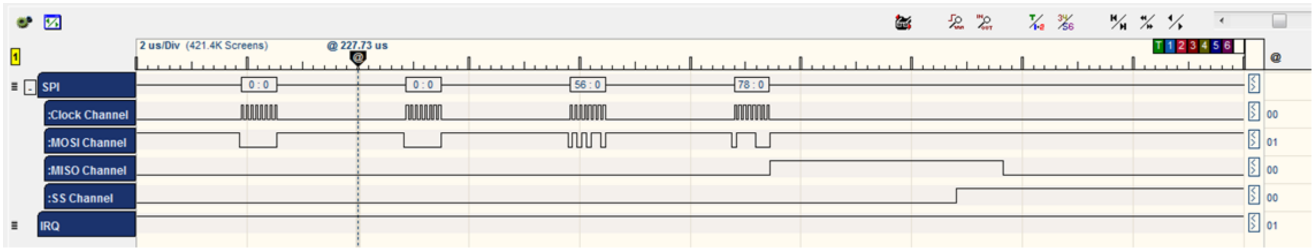


Then WINC chip sends the value of the register 0x01 which equals 0x07.

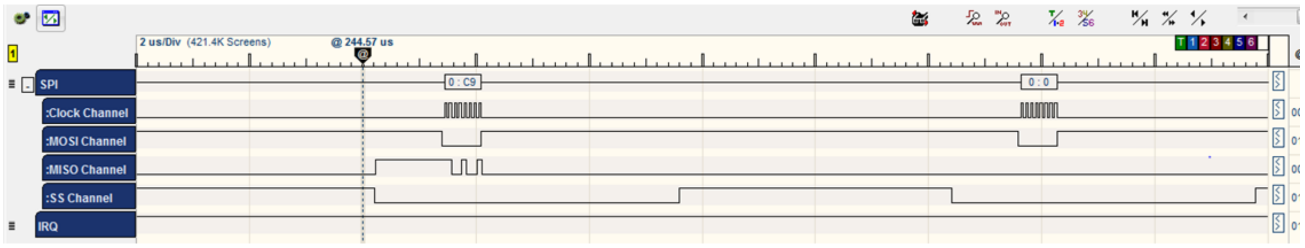


```
Command      CMD_SINGLE_WRITE:0XC9          /* single word write         */
               BYTE [0] = CMD_SINGLE_WRITE
               BYTE [1] = address >> 16;                  /* WAKE_REG address = 0x1074 */
               BYTE [2] = address >> 8;
               BYTE [3] = address;
            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





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



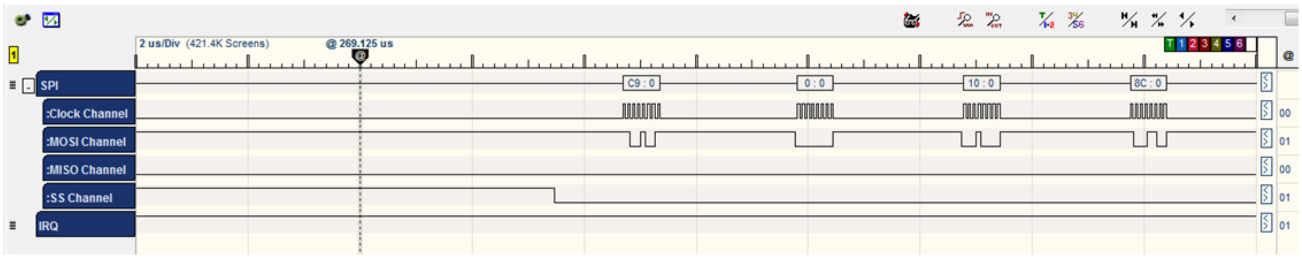
At this point, HIF finishes executing the clockless wakeup of the WINC chip.

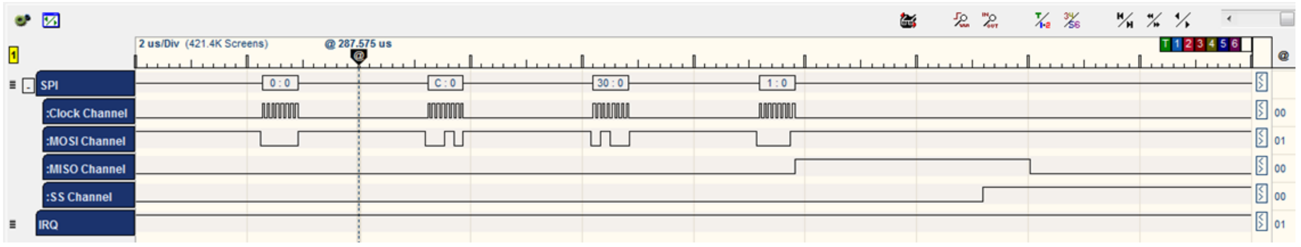
The HIF layer Prepares and Sets the HIF layer header to NMI_STATE_REG register (4 | 8 Byte header describing the packet to be sent).

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

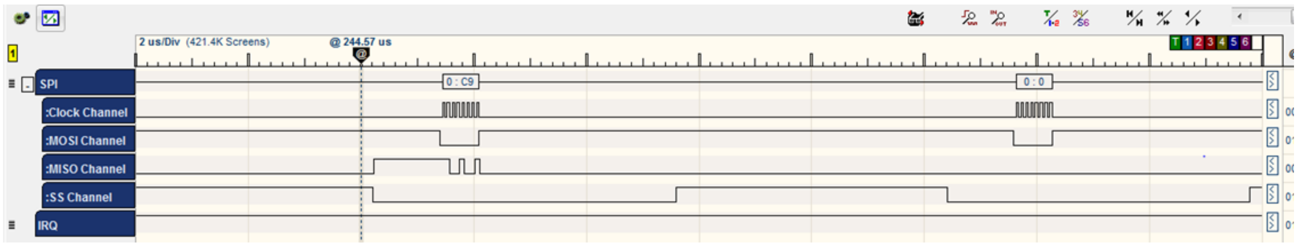
```
sint8 hif_send(uint8 u8Gid,uint8 u8Opcode,uint8 *pu8CtrlBuf,uint16 u16CtrlBufSize,
                              uint8 *pu8DataBuf,uint16 u16DataSize, uint16 u16DataOffset)
{
        volatile tstrHifHdr   strHif;
        volatile uint32 reg;
        strHif.u8Opcode      = u8Opcode&(~NBIT7);
    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

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

```
Command       CMD_SINGLE_WRITE:0XC9         /* single word write */
                BYTE [0] = CMD_SINGLE_WRITE
                BYTE [1] = address >> 16;         /* NMI_STATE_REG address = 0x180c */
                BYTE [2] = address >> 8;
                BYTE [3] = address;
                BYTE [4] = u32data >> 24;         /* Data = 0x000C3001 */
                BYTE [5] = u32data >> 16;         /* 0x0C is the length and equals 12  */
                BYTE [6] = u32data >> 8;          /* 0x30 is the Opcode =
M2M_WIFI_REQ_SET_SCAN_REGION */
                BYTE [7] = u32data;         /* 0x01 is the Group ID = M2M_REQ_GRP_WIFI  */
```

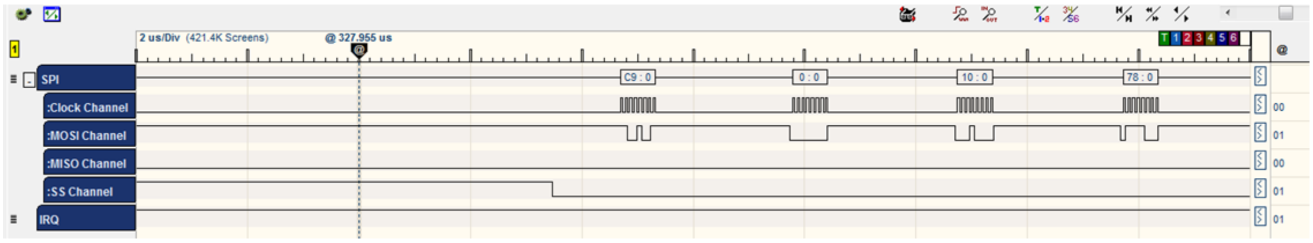


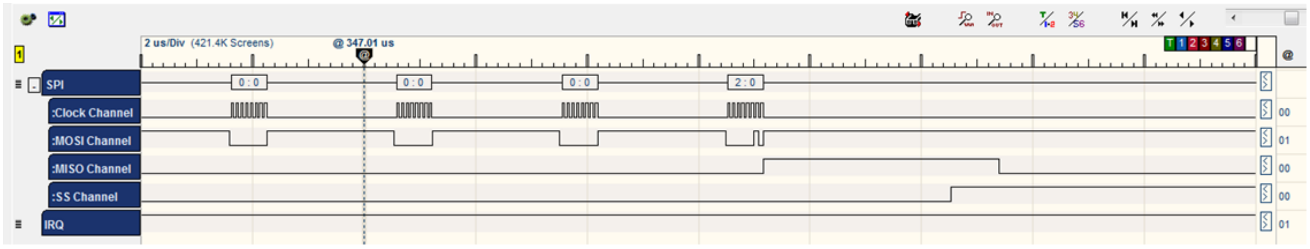


WINC acknowledges the command by sending two bytes [C9] [0].

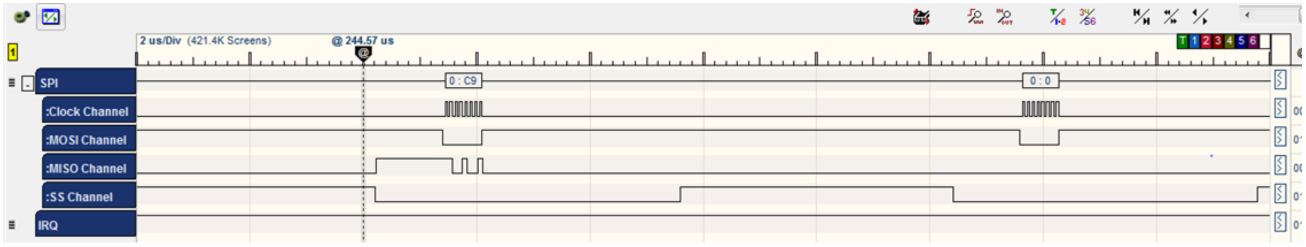


```
Command      CMD_SINGLE_WRITE:0XC9 /*     single word write */
              BYTE [0] = CMD_SINGLE_WRITE
              BYTE [1] = address >> 16;   /*     WIFI_HOST_RCV_CTRL_2address = 0x1087*/
              BYTE [2] = address >> 8;
              BYTE [3] = address;
              BYTE [4] = u32data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





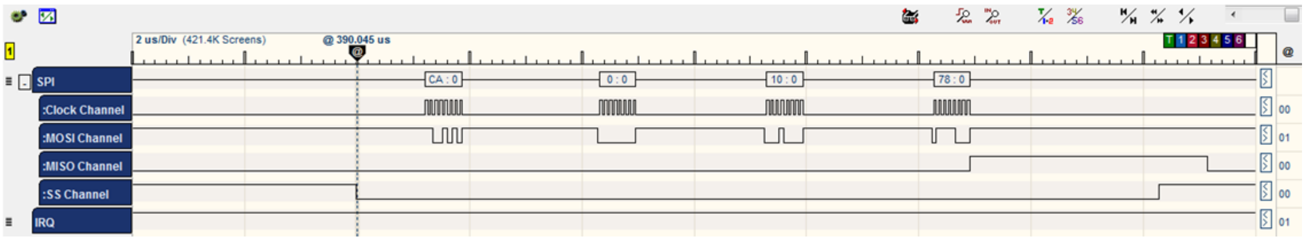
WINC acknowledges the command by sending two bytes [C9] [0].



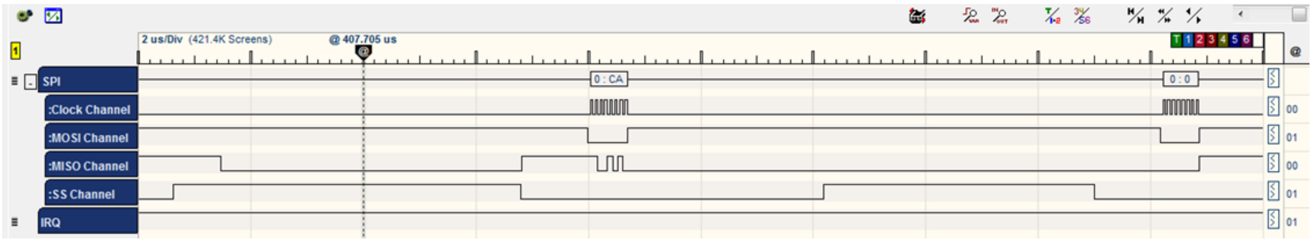
Then HIF polls for DMA address.

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

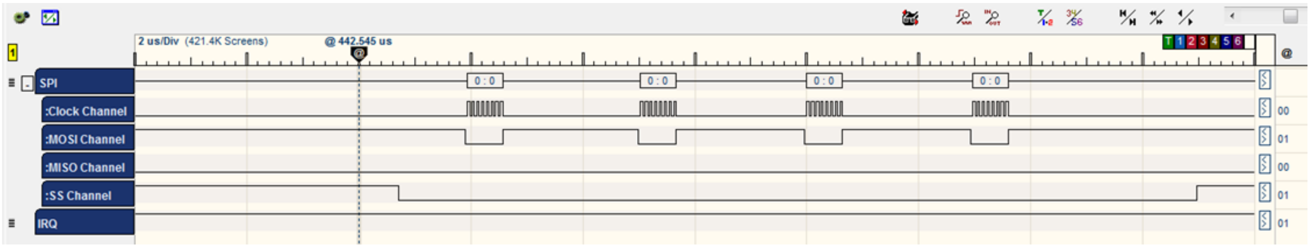
```
Command      CMD_SINGLE_READ:     0xCA          /* single word (4 bytes) read */
              BYTE [0] = CMD_SINGLE_READ
              BYTE [1] = address >> 16;         /* WIFI_HOST_RCV_CTRL_2 address = 0x1078 */
              BYTE [2] = address >> 8;
              BYTE [3] = address;
```



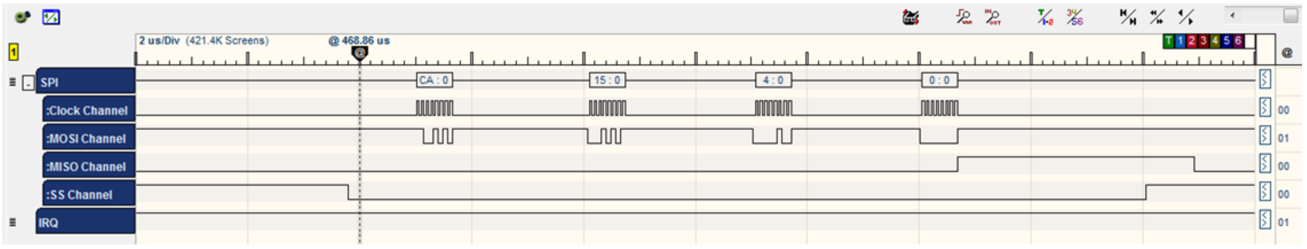
WINC acknowledges the command by sending three bytes [CA] [0] [F3].



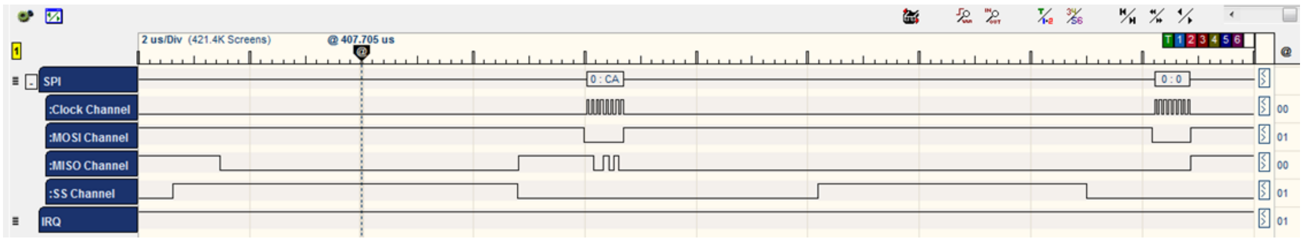
Then WINC chip send the value of the register 0x1078 which equals 0x00.



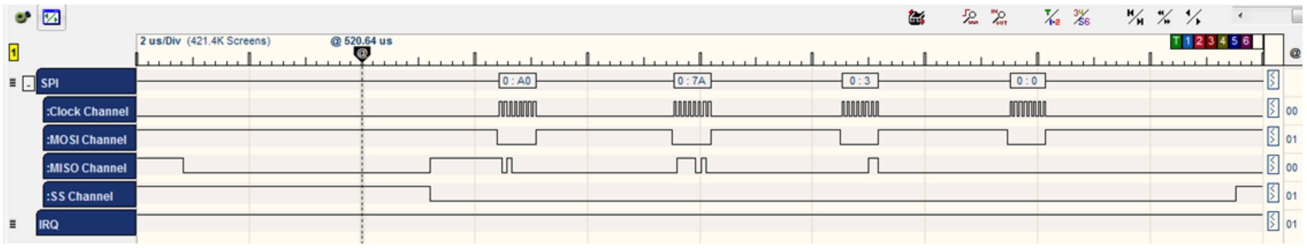
```
Command      CMD_SINGLE_READ:     0xCA          /* single word (4 bytes) read */
              BYTE [0] = CMD_SINGLE_READ
              BYTE [1] = address >> 16;         /* address = 0x1504 */
              BYTE [2] = address >> 8;
              BYTE [3] = address;
```



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



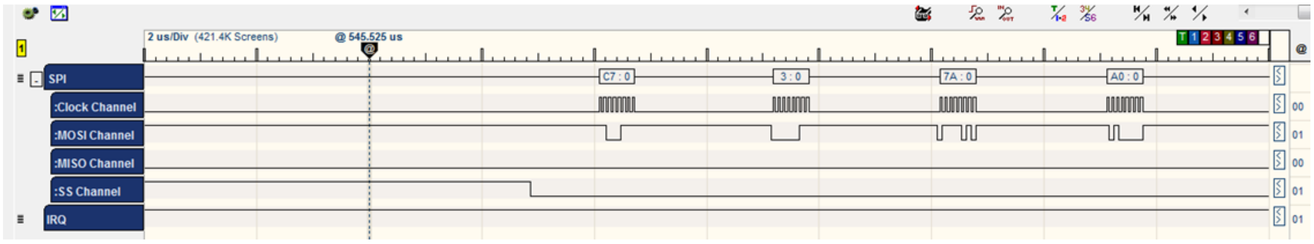
Then WINC chip send the value of the register 0x1504 which equals 0x037AA0.

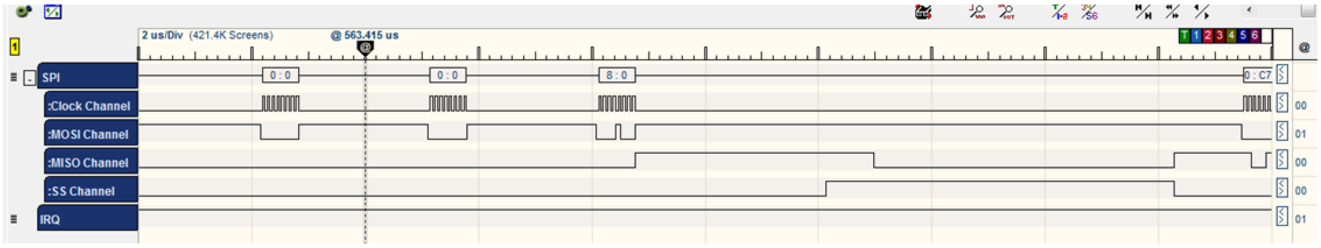


WINC writes the HIF header to the DMA memory address.

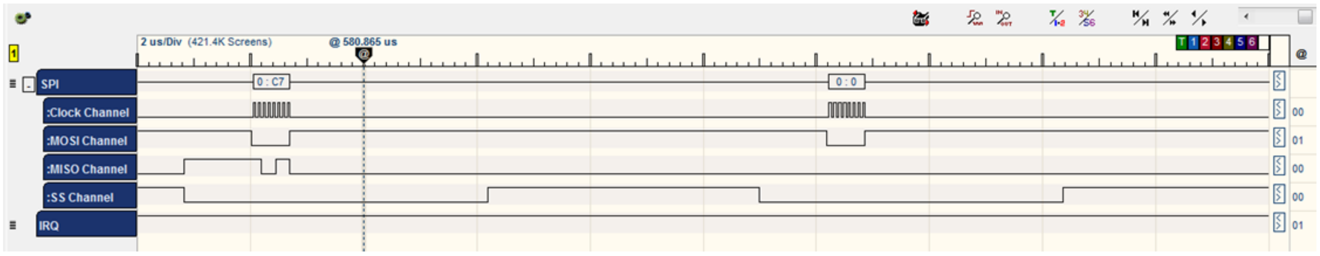
```
u32CurrAddr = dma_addr;
strHif.u16Length=NM_BSP_B_L_16(strHif.u16Length);
ret = nm_write_block(u32CurrAddr, (uint8*)&strHif, M2M_HIF_HDR_OFFSET);
```

```
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;
```

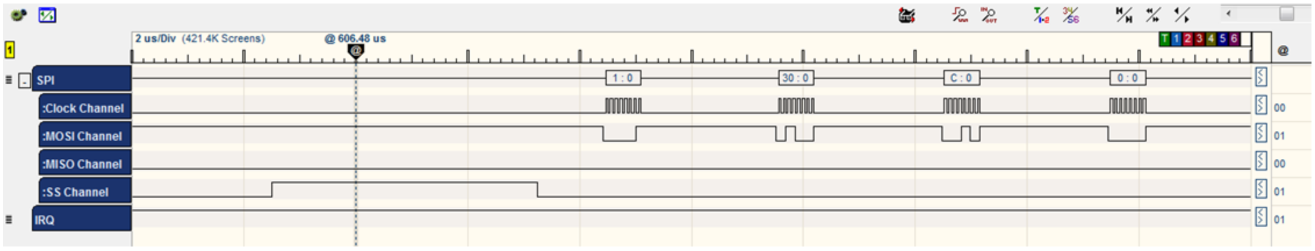


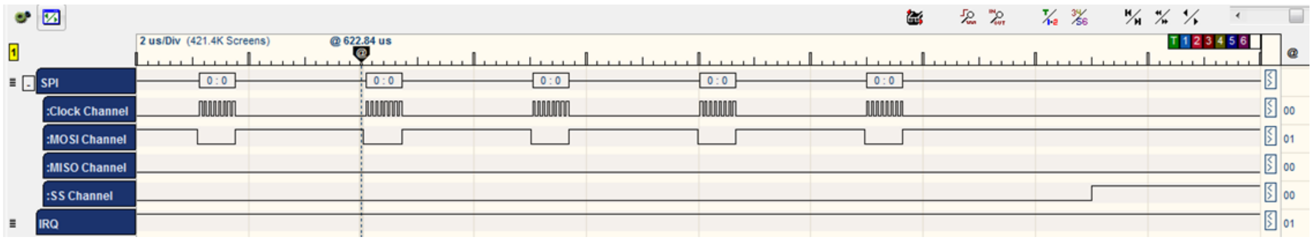


WINC acknowledges the command by sending three bytes [C7] [0] [F3].



The HIF layer writes the Data.

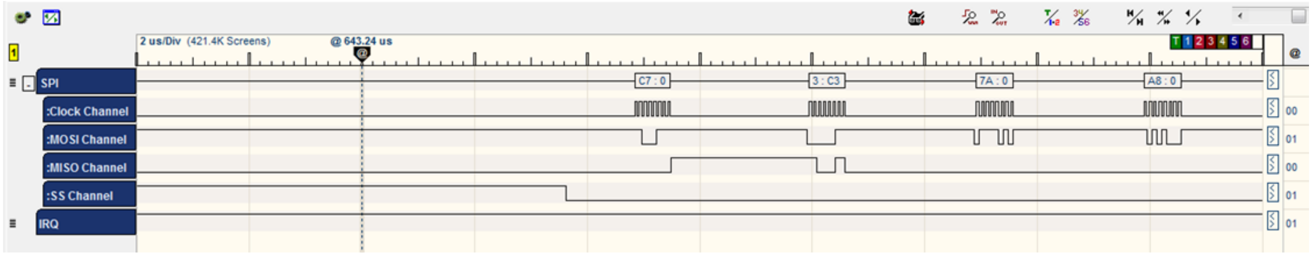


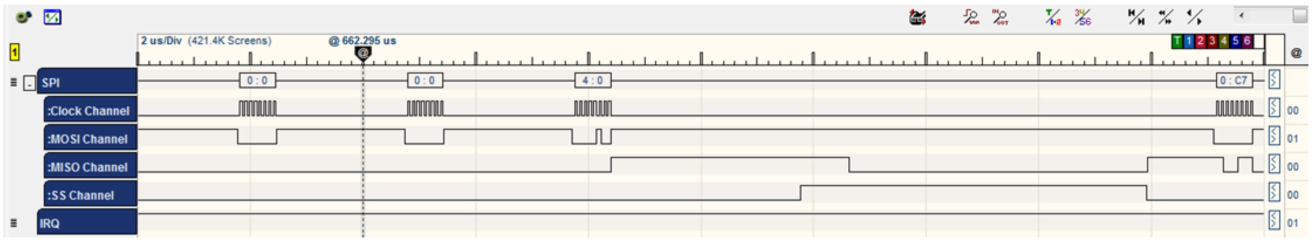


HIF writes the Control Buffer data (part of the framing of the request).

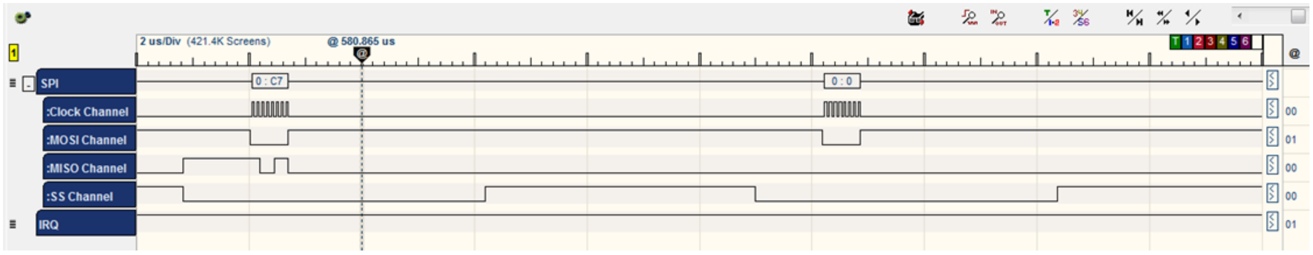
```
if (pu8CtrlBuf != NULL)
{
            ret = nm_write_block(u32CurrAddr, pu8CtrlBuf, u16CtrlBufSize);
            if(M2M_SUCCESS != ret) goto ERR1;
            u32CurrAddr += u16CtrlBufSize;
}
```

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

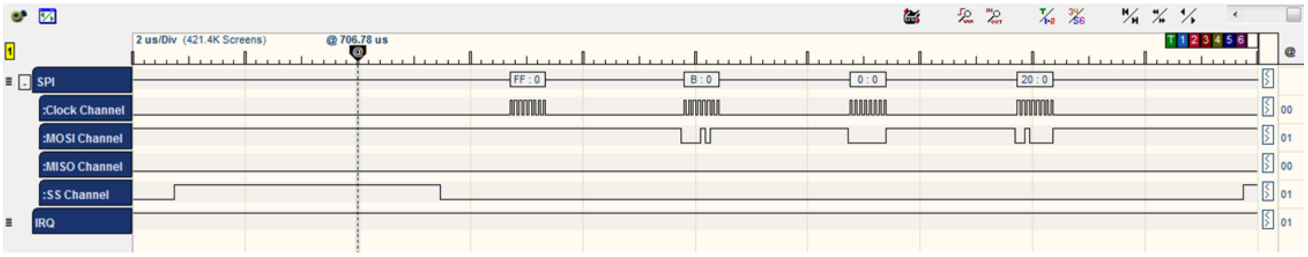




WINC acknowledges the command by sending three bytes [C7] [0] [F3].



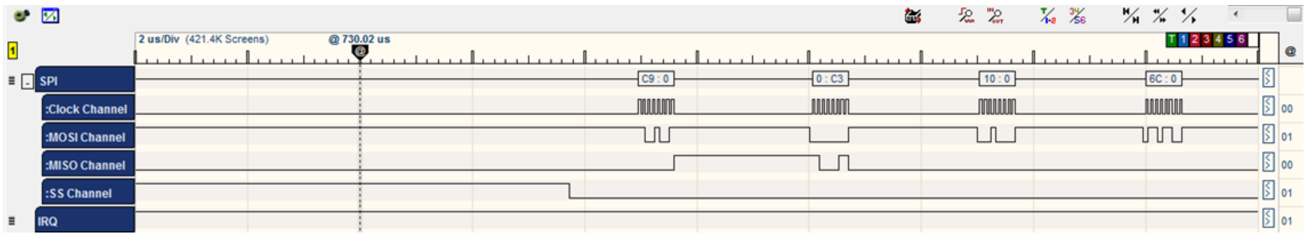
HIF layer writes the Data.

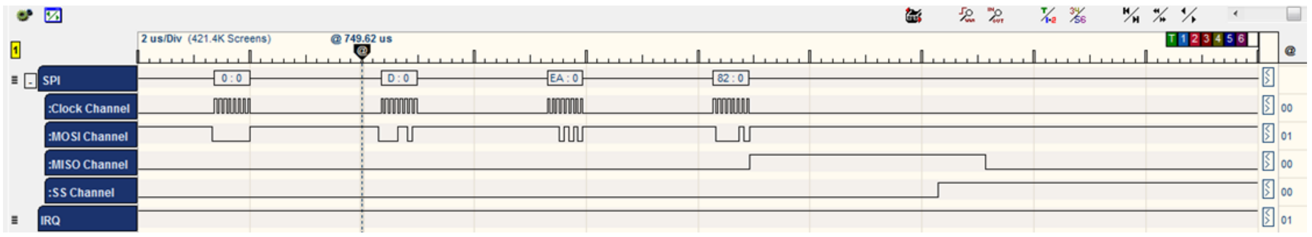


Finally, HIF finished writing the request data to memory and is going to interrupt the chip announcing that host Tx is done.

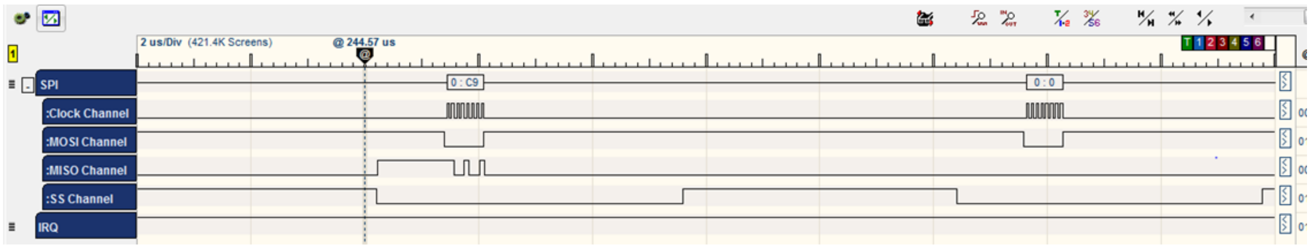
```
reg = dma_addr << 2;
reg |= (1 << 1);
ret = nm_write_reg(WIFI_HOST_RCV_CTRL_3, reg);
```

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





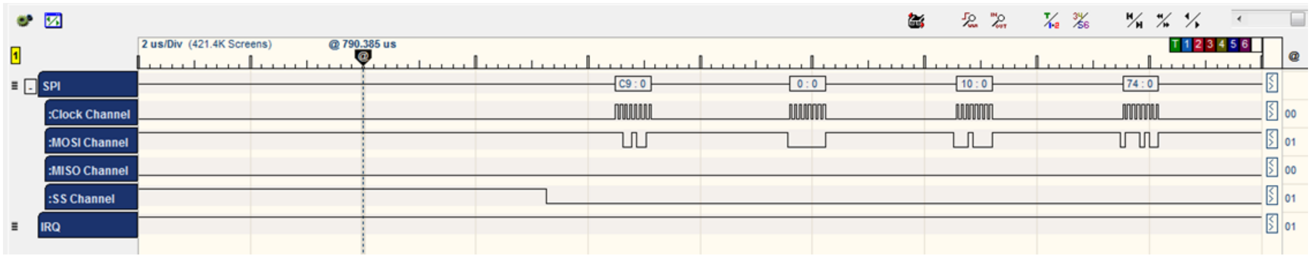
WINC acknowledges the command by sending two bytes [C9] [0].

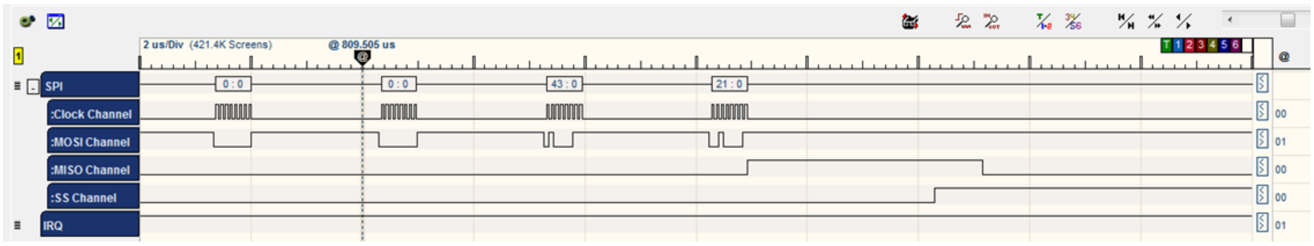


HIF layer allows the chip to enter sleep mode again.

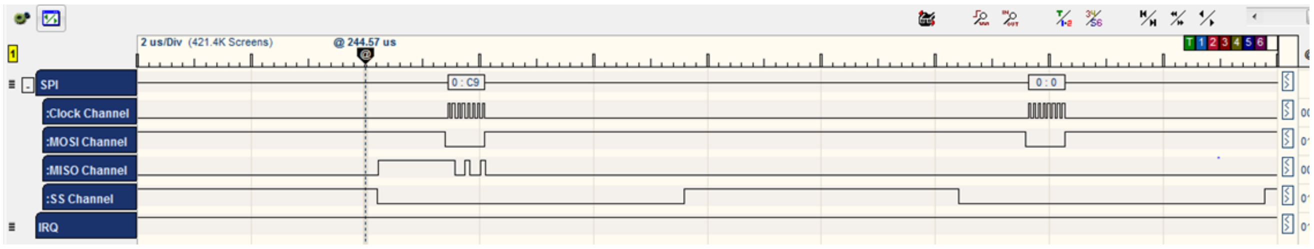
```
sint8 hif_chip_sleep(void)
{
            sint8 ret = M2M_SUCCESS;
            uint32 reg = 0;
            ret = nm_write_reg(WAKE_REG, SLEEP_VALUE);
            /* Clear bit 1 */
            ret = nm_read_reg_with_ret(0x1, &reg);
            if(reg&0x2)
            {
                  reg &=~(1 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

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

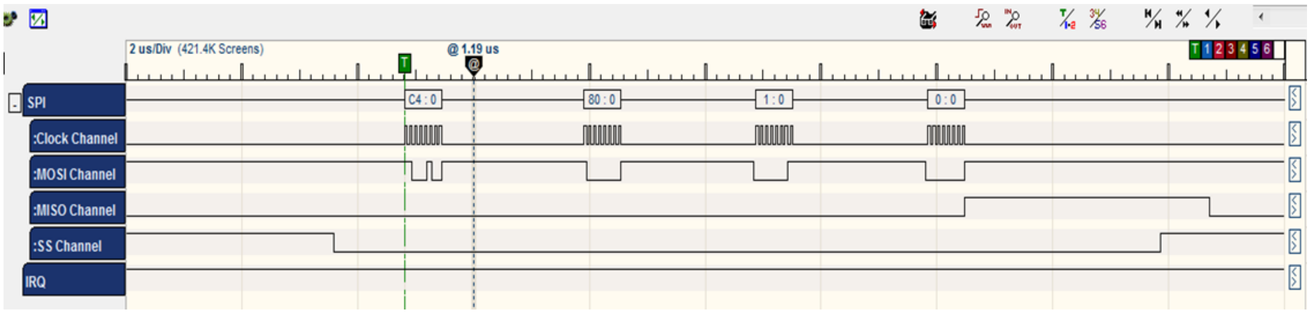




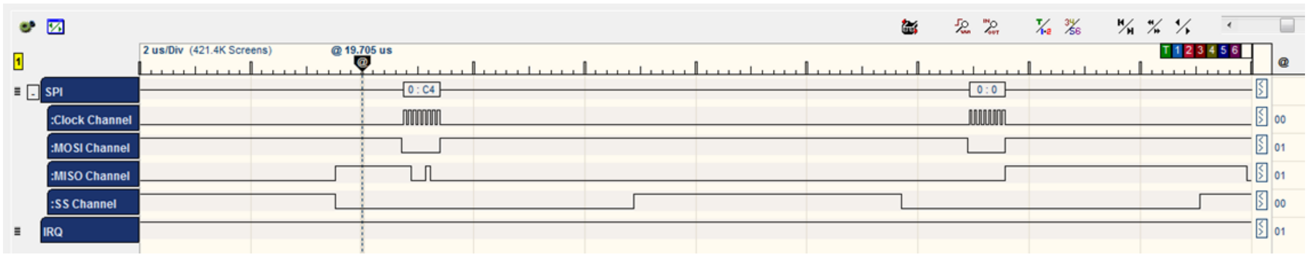
WINC acknowledges the command by sending two bytes [C9] [0].



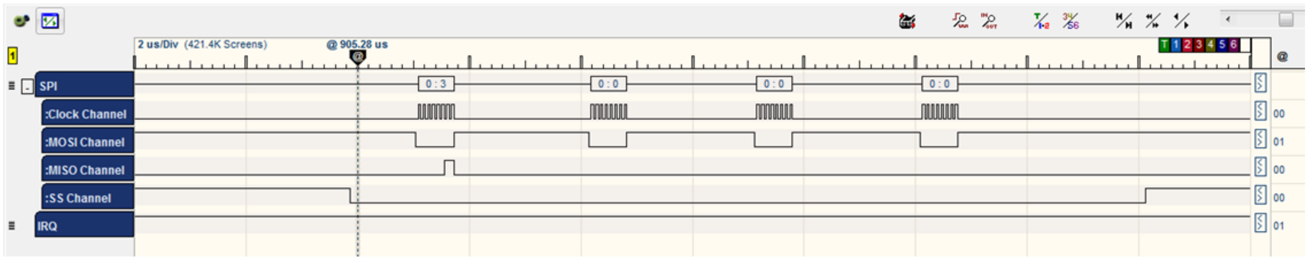
```
Command       CMD_INTERNAL_READ:   0xC4          /* internal register read   */
              BYTE [0] = CMD_INTERNAL_READ
              BYTE [1] = address >> 8;                     /* address = 0x01    */
              BYTE [1] |= (1 << 7);                        /* clockless register      */
 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



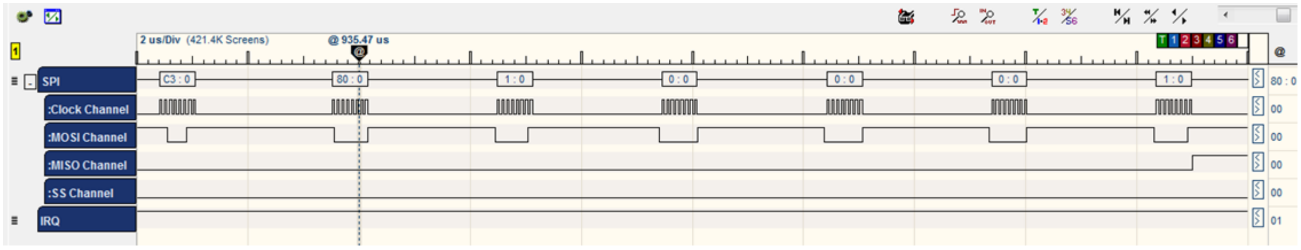
WINC acknowledges the command by sending three bytes [C4] [0] [F3].



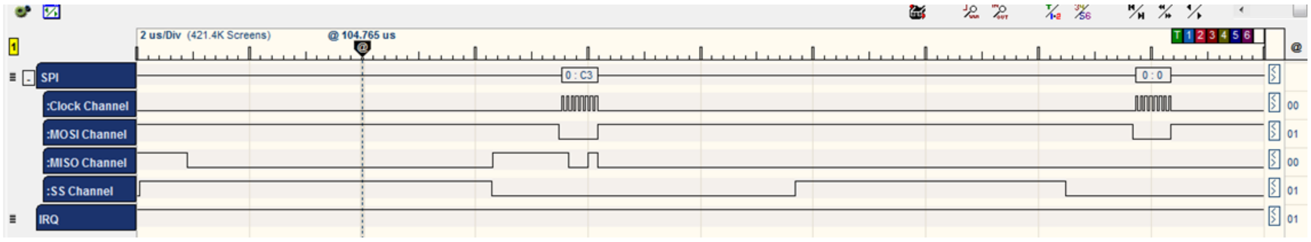
Then WINC chip sends the value of the register 0x01 which equals 0x03.



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



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



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

17.3.2 RX (Receive Response)

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

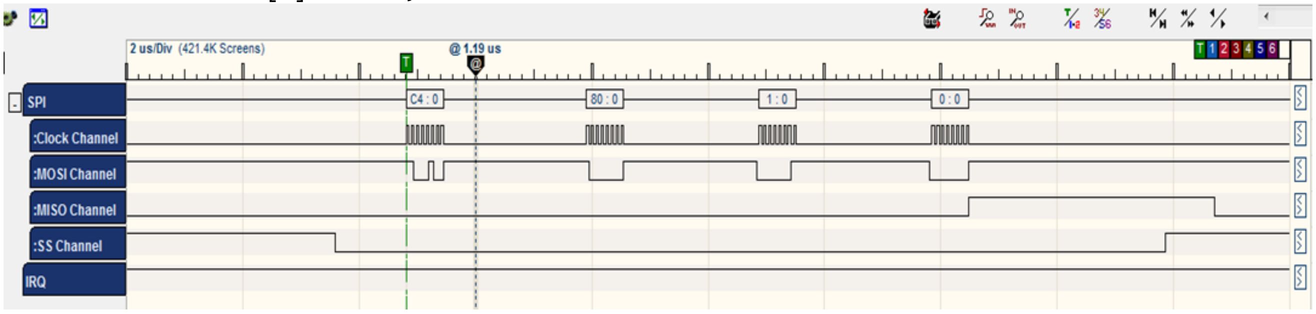
Host will handle this interrupt to receive the response.

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

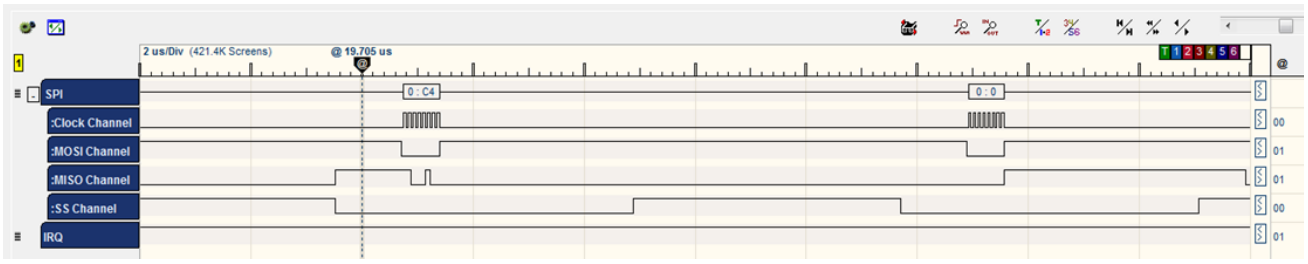
```
sint8 nm_clkless_wake(void)
{
        ret = nm_read_reg_with_ret(0x1, &reg);
        /* Set bit 1 */
        ret = nm_write_reg(0x1, reg | (1 << 1));
        // Check the clock status
```

```
                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;
        }
```

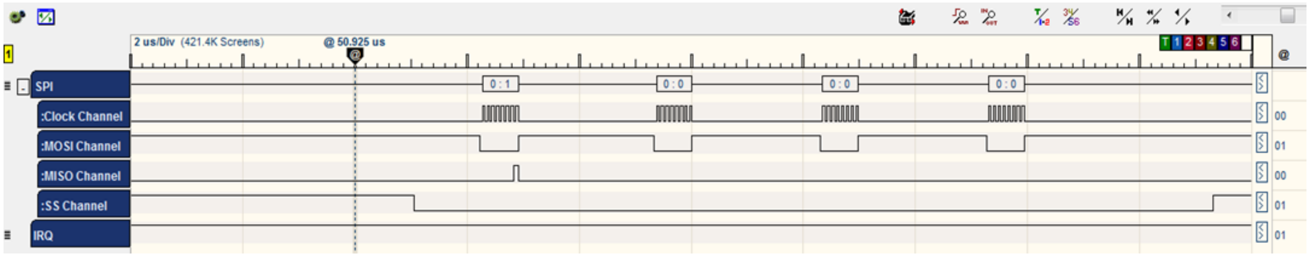
```
Command       CMD_INTERNAL_READ:   0xC4        /* internal register read */
              BYTE [0] = CMD_INTERNAL_READ
              BYTE [1] = address >> 8;                     /* address = 0x01 */
              BYTE [1] |= (1 << 7);                        /* clockless register */
             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



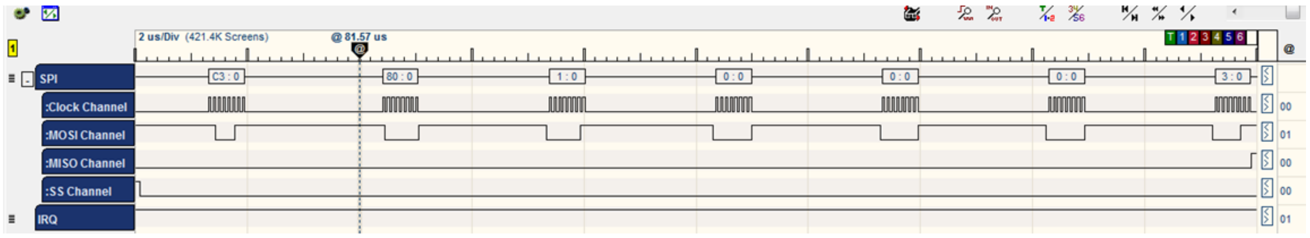
WINC acknowledges the command by sending three bytes [C4] [0] [F3].



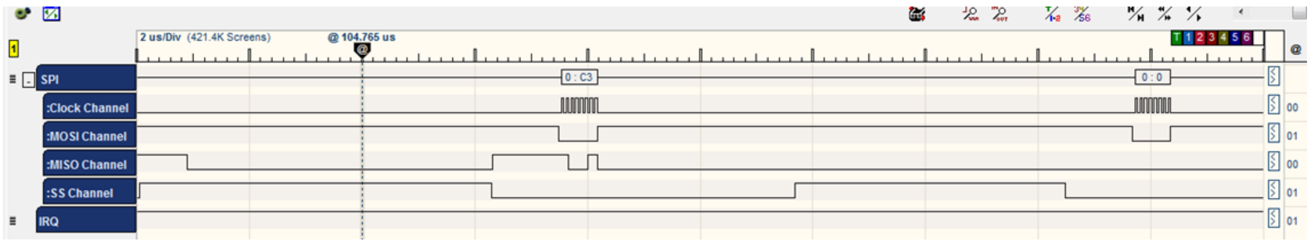
Then WINC chip sends the value of the register 0x01 which equals 0x01.



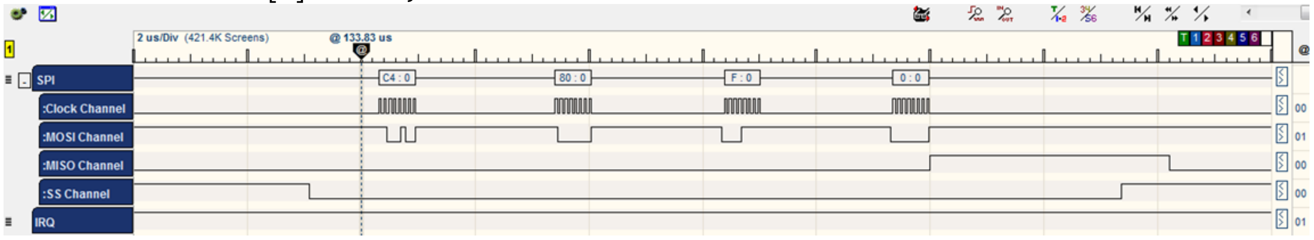
```
Command       CMD_INTERNAL_WRITE:  C3           /*    internal register write */
              BYTE [0] = CMD_INTERNAL_WRITE
              BYTE [1] = address >> 8;                     /*     address = 0x01
       */
              BYTE [1] |= (1 << 7);                        /*     clockless register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



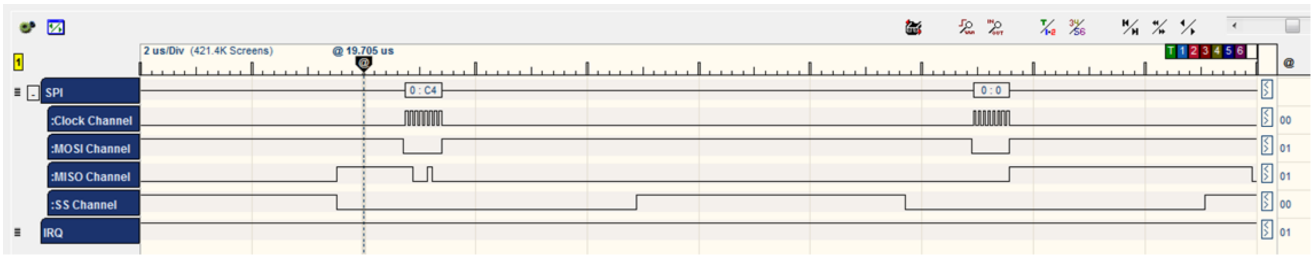
WINC acknowledges the command by sending two bytes [C3] [0].



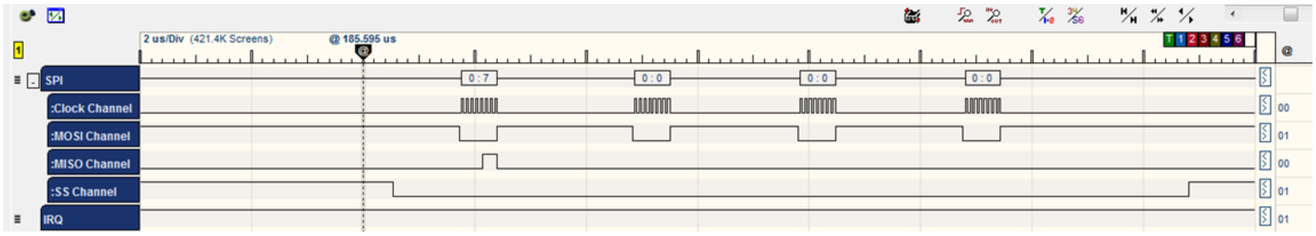
```
Command       CMD_INTERNAL_READ:   0xC4          /*     internal register read     */
              BYTE [0] = CMD_INTERNAL_READ
              BYTE [1] = address >> 8;                      /*     address = 0x0F
       */
              BYTE [1] |= (1 << 7);                         /*     clockless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



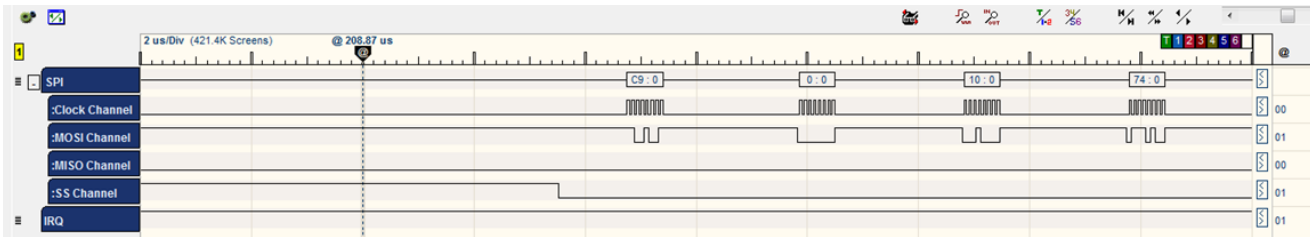
WINC acknowledges the command by sending three bytes [C4] [0] [F3].

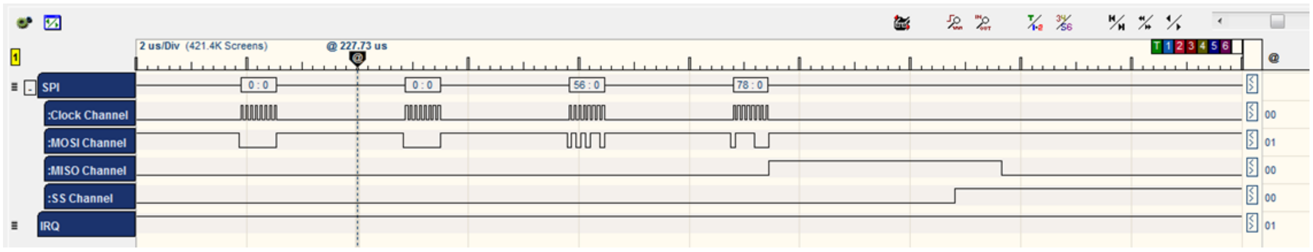


Then WINC chip sends the value of the register 0x01 which equals 0x07.

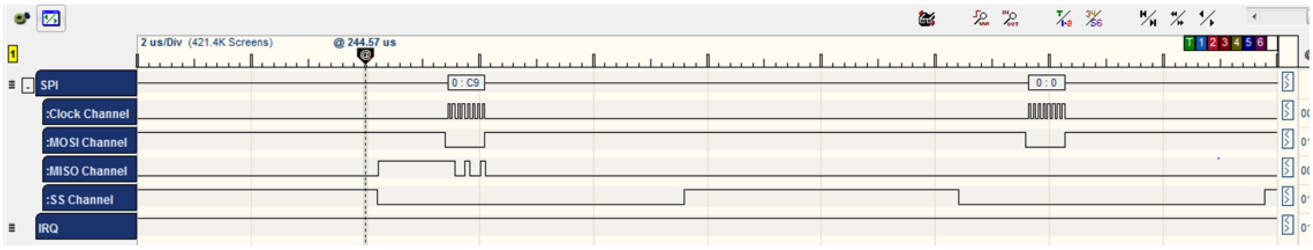


```
Command       CMD_SINGLE_WRITE:0XC9       /* single word write */
               BYTE [0] = CMD_SINGLE_WRITE
               BYTE [1] = address >> 16;         /* WAKE_REG address = 0x1074 */
               BYTE [2] = address >> 8;
               BYTE [3] = address;
               BYTE [4] = u32data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





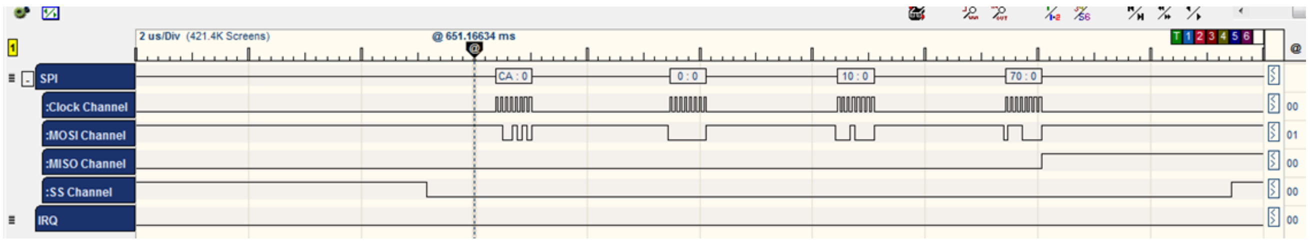
The chip acknowledges the command by sending two bytes [C9] [0].



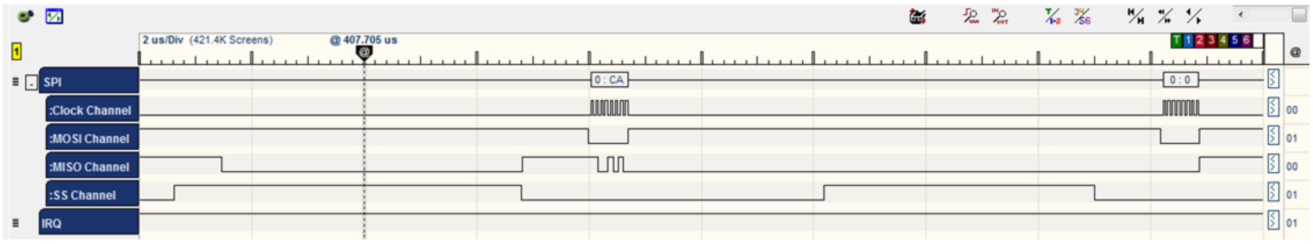
Read register WIFI_HOST_RCV_CTRL_0 to check if there is new interrupt, and if so, clear it (as it will be handled now).

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

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

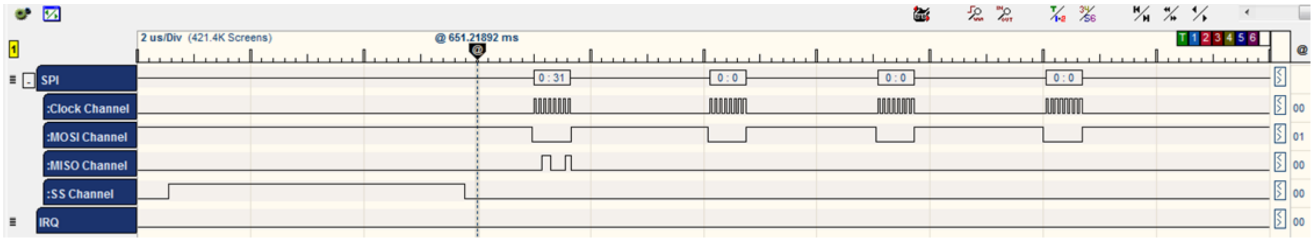
```
Command      CMD_SINGLE_READ:     0xCA          /* single word (4 bytes) read          */
              BYTE [0] = CMD_SINGLE_READ
              BYTE [1] = address >> 16;         /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
              BYTE [2] = address >> 8;
              BYTE [3] = address;
```



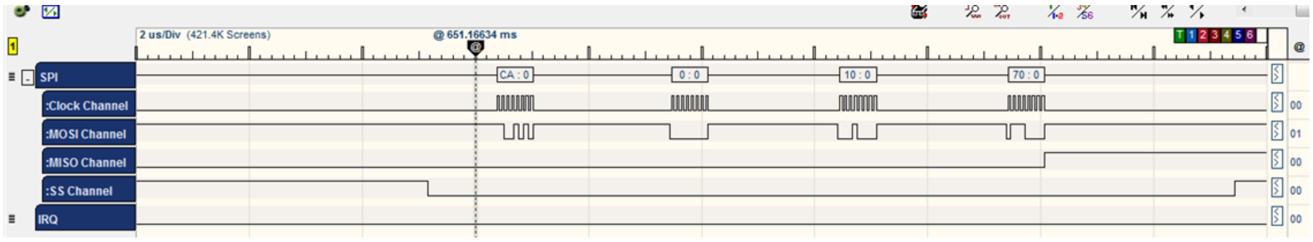
WINC acknowledges the command by sending three bytes [CA] [0] [F3].



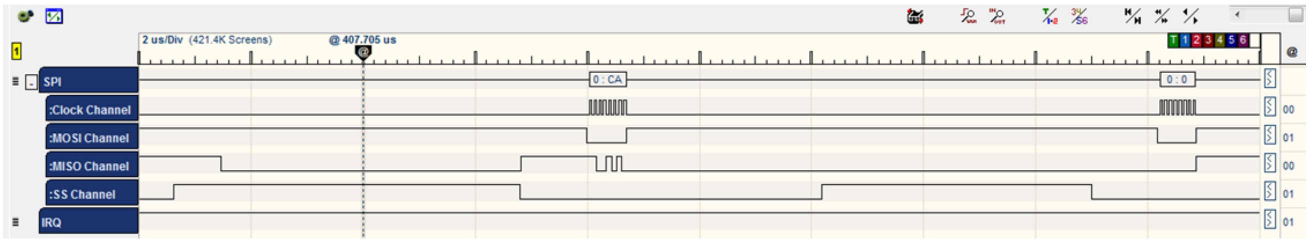
Then WINC chip sends the value of the register 0x1070 which equals 0x31.



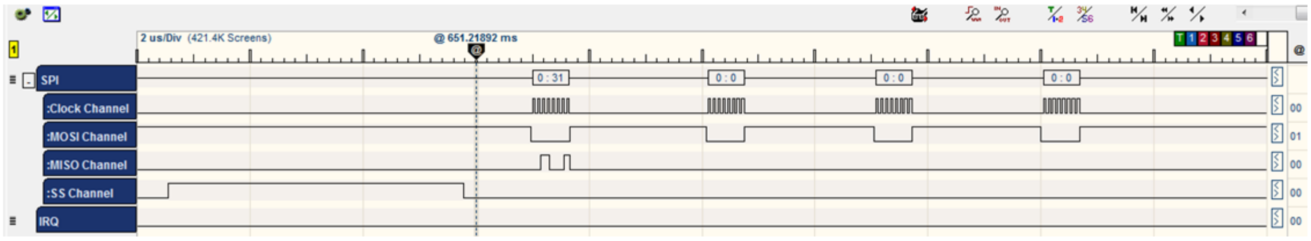
```
Command      CMD_SINGLE_READ:     0xCA          /* single word (4 bytes) read          */
              BYTE [0] = CMD_SINGLE_READ
              BYTE [1] = address >> 16;         /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
              BYTE [2] = address >> 8;
              BYTE [3] = address;
```



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

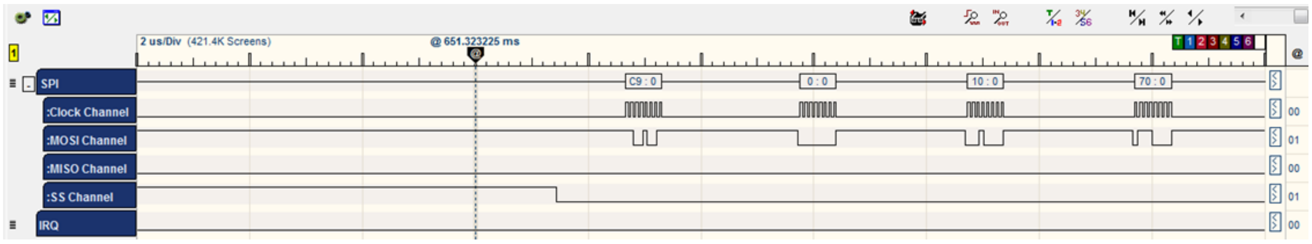


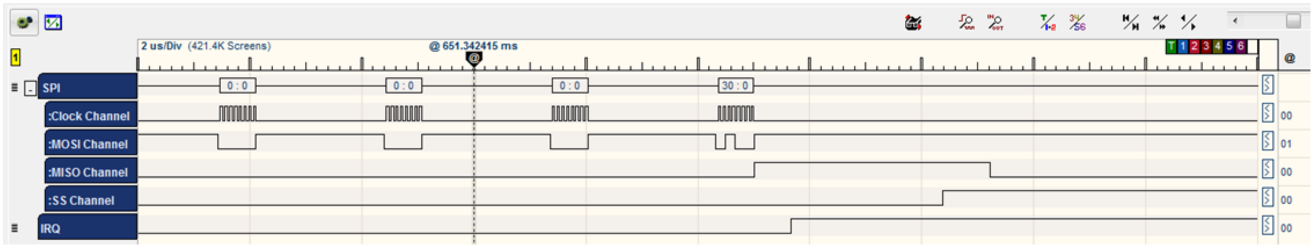
Then WINC chip sends the value of the register 0x1070 which equals 0x31.



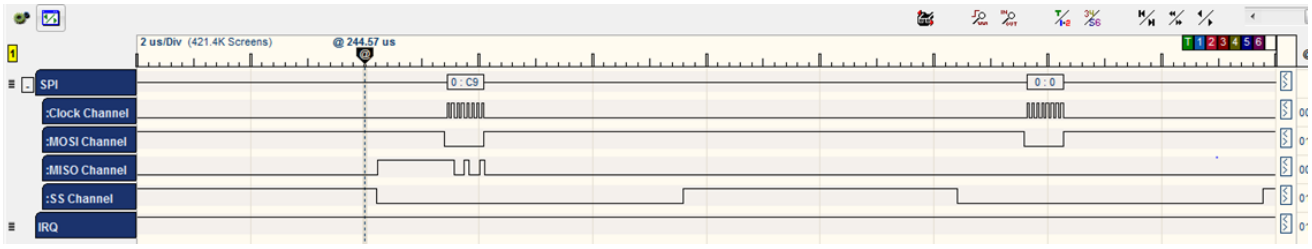
Clear the WINC Interrupt.

```
Command      CMD_SINGLE_WRITE:0XC9              /* single word write */
              BYTE [0] = CMD_SINGLE_WRITE
              BYTE [1] = address >> 16;         /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
              BYTE [2] = address >> 8;
              BYTE [3] = address;
              BYTE [4] = u32data >> 24;         /* Data = 0x30 */
              BYTE [5] = u32data >> 16;
              BYTE [6] = u32data >> 8;
              BYTE [7] = u32data;
```





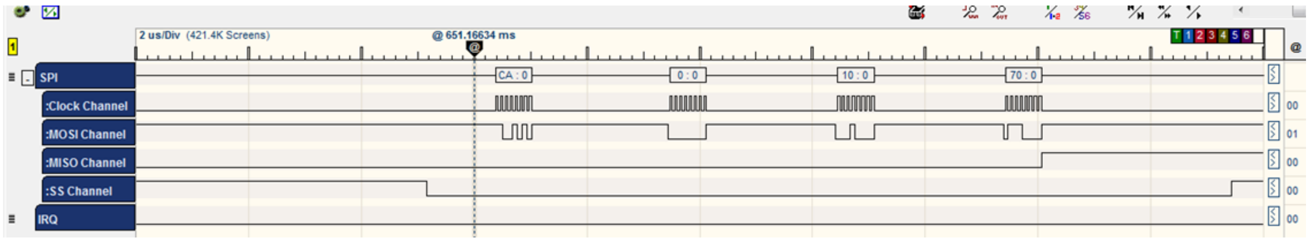
The chip acknowledges the command by sending two bytes [C9] [0].



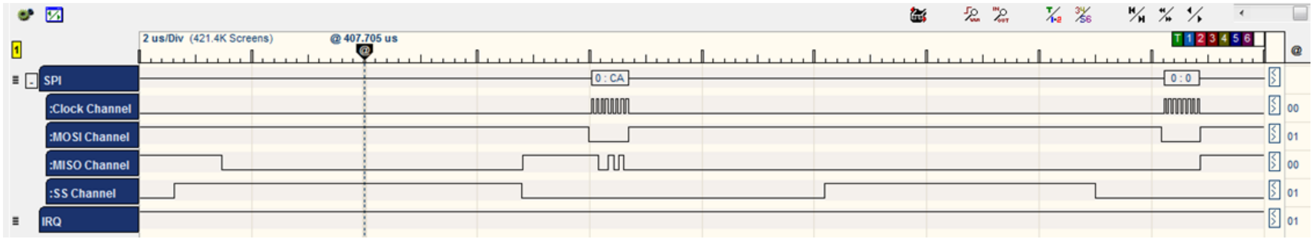
Then HIF reads the data size.

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

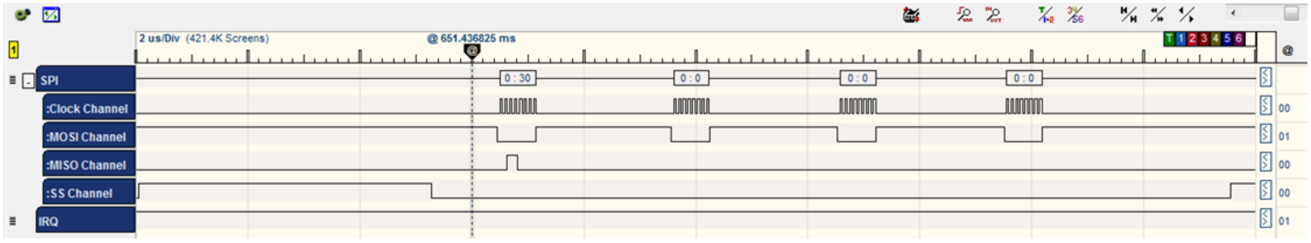
```
Command      CMD_SINGLE_READ:      0xCA         /* single word (4 bytes) read          */
              BYTE [0] = CMD_SINGLE_READ
              BYTE [1] = address >> 16;         /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
              BYTE [2] = address >> 8;
              BYTE [3] = address;
```



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



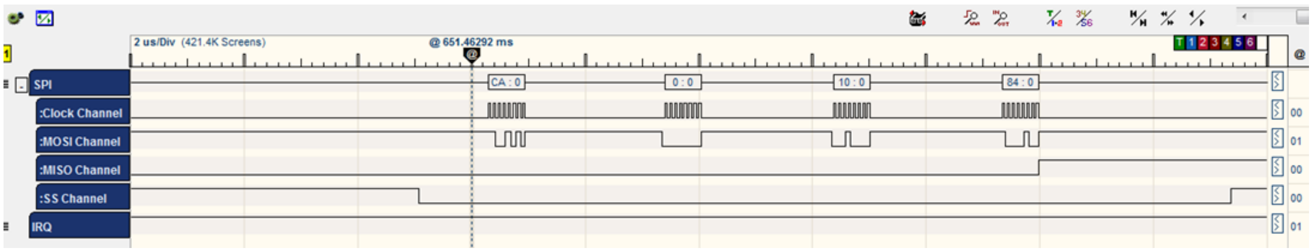
Then WINC chip sends the value of the register 0x1070 which equals 0x30.



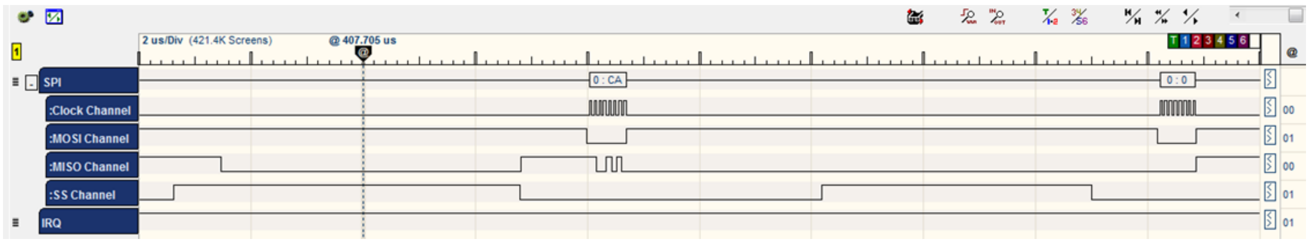
HIF reads hif header address.

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

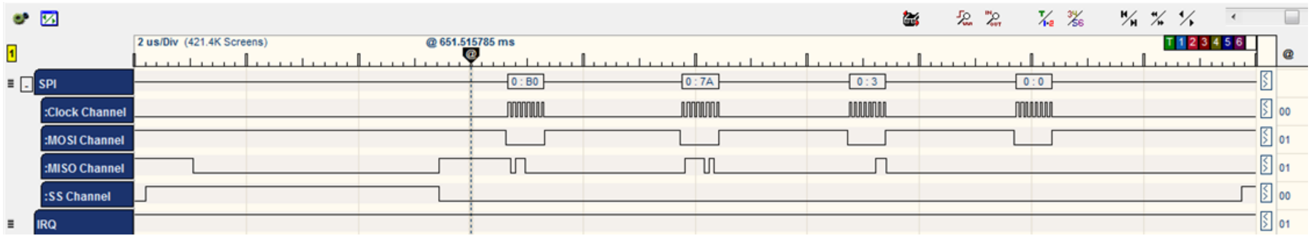
```
Command       CMD_SINGLE_READ:      0xCA          /* single word (4 bytes) read */
               BYTE [0] = CMD_SINGLE_READ
               BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_1 address = 0x1084 */
               BYTE [2] = address >> 8;
               BYTE [3] = address;
```



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



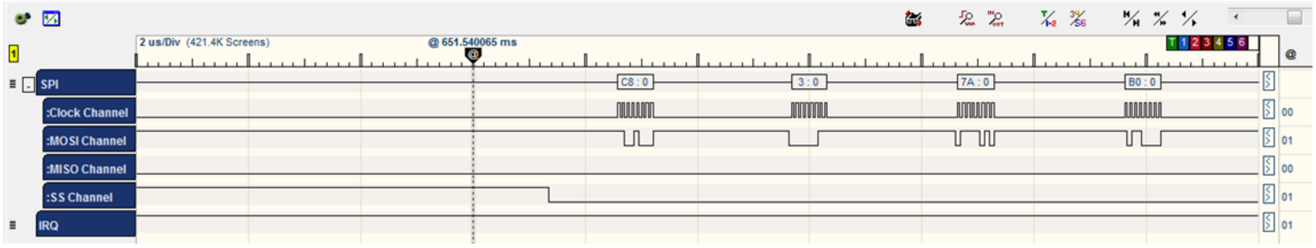
Then WINC chip sends the value of the register 0x1078 which equals 0x037AB0.

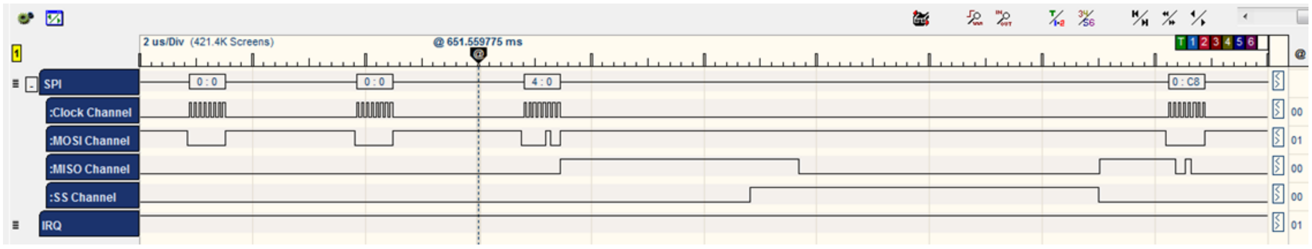


HIF reads the hif header data (as a block).

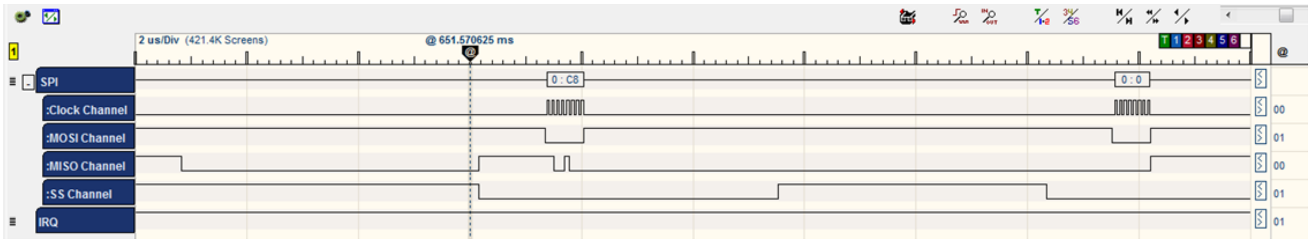
```
ret = nm_read_block(address, (uint8*)&strHif, sizeof(tstrHifHdr));
```

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

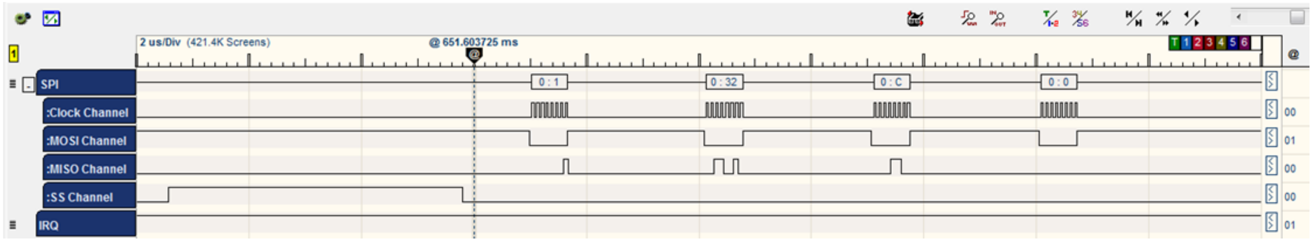




WINC acknowledges the command by sending three bytes [C8] [0] [F3].



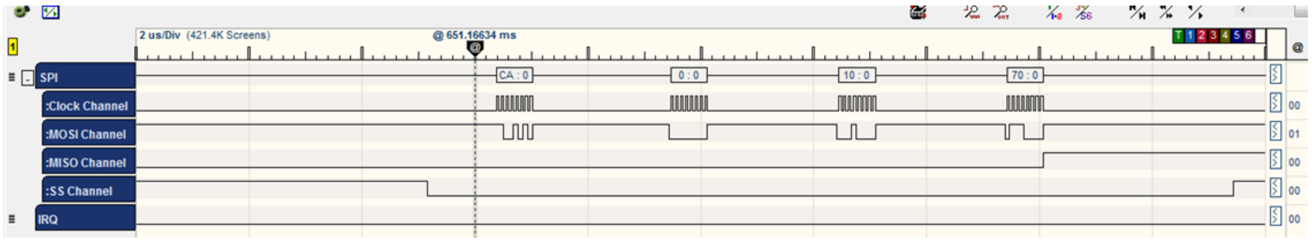
WINC sends the data block (four bytes).



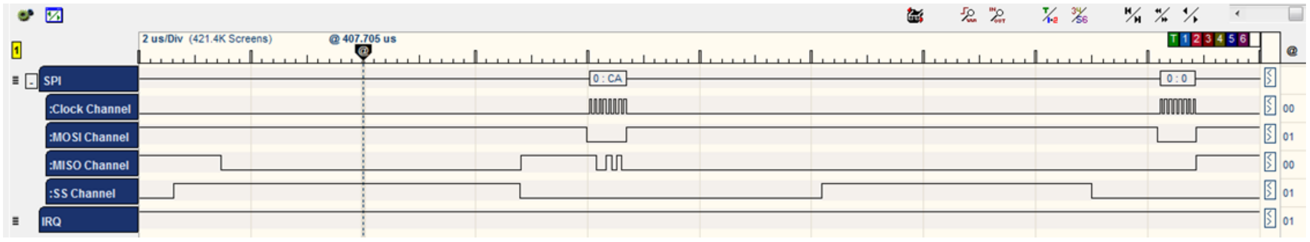
HIF then calls the appropriate handler according to the hif header received which tries to receive the Response data payload. (Note that hif_receive ( ) obtains some data again for checks.)

```
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);

}
```

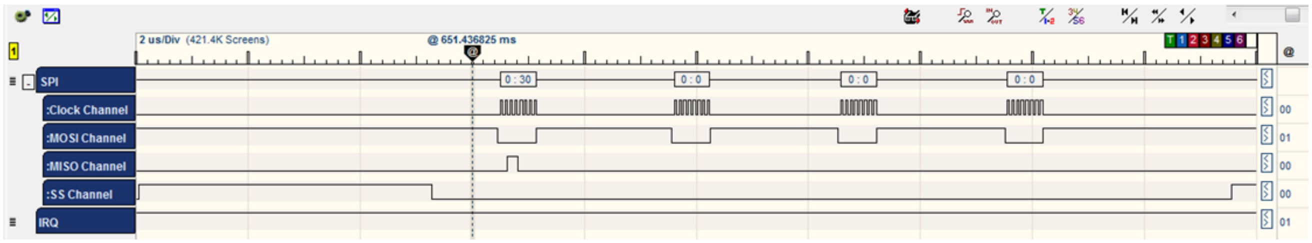
```
Command      CMD_SINGLE_READ:     0xCA         /* single word (4 bytes) read          */
              BYTE [0] = CMD_SINGLE_READ
              BYTE [1] = address >> 16;        /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
              BYTE [2] = address >> 8;
              BYTE [3] = address;
```



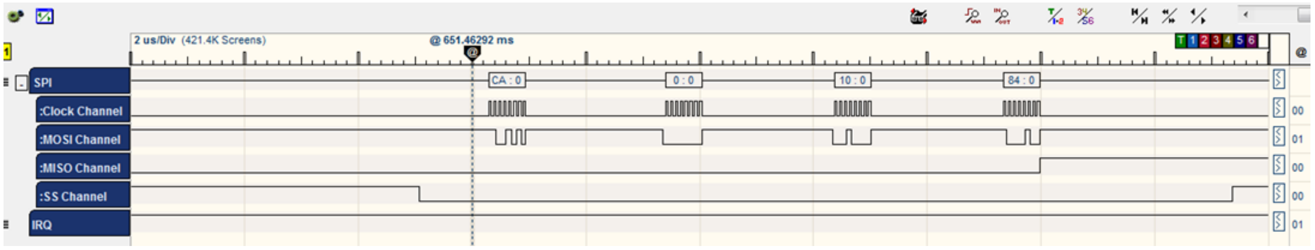
WINC acknowledges the command by sending three bytes [CA] [0] [F3].



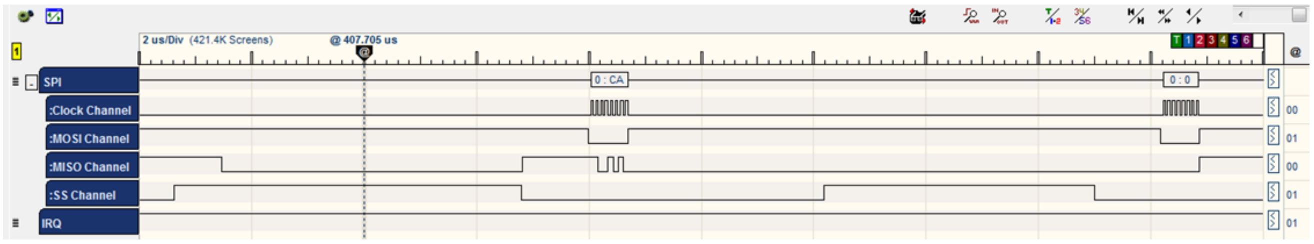
Then WINC chip sends the value of the register 0x1070 which equals 0x30.



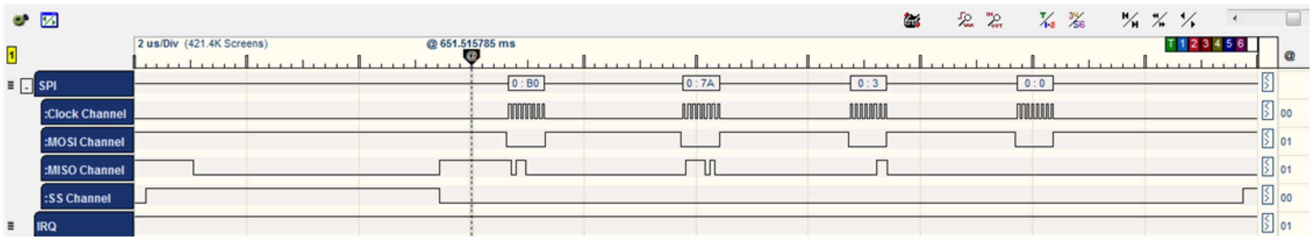
```
Command       CMD_SINGLE_READ:      0xCA          /* single word (4 bytes) read             */
               BYTE [0] = CMD_SINGLE_READ
               BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_1 address = 0x1084 */
               BYTE [2] = address >> 8;
               BYTE [3] =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



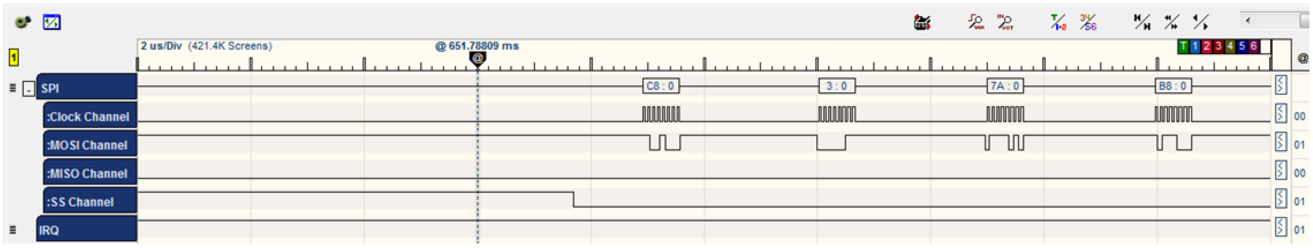
WINC acknowledges the command by sending three bytes [CA] [0] [F3].

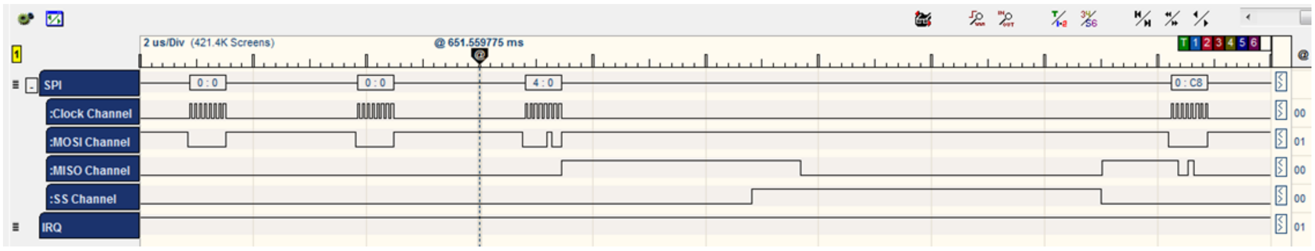


Then WINC chip sends the value of the register 0x1078 which equals 0x037AB0.

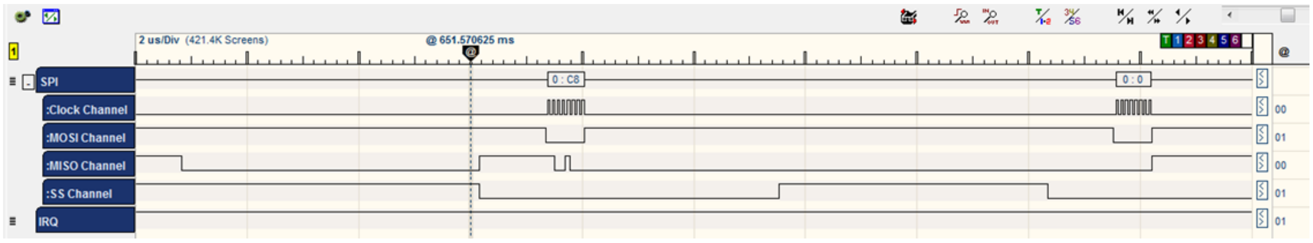


```
Command       CMD_DMA_EXT_READ:     C8           /* dma extended read */
               BYTE [0] = CMD_DMA_EXT_READ
               BYTE [1] = address >> 16;         /* address = 0x037AB8*/
               BYTE [2] = address >> 8;
               BYTE [3] = address;
               BYTE [4] = size >>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

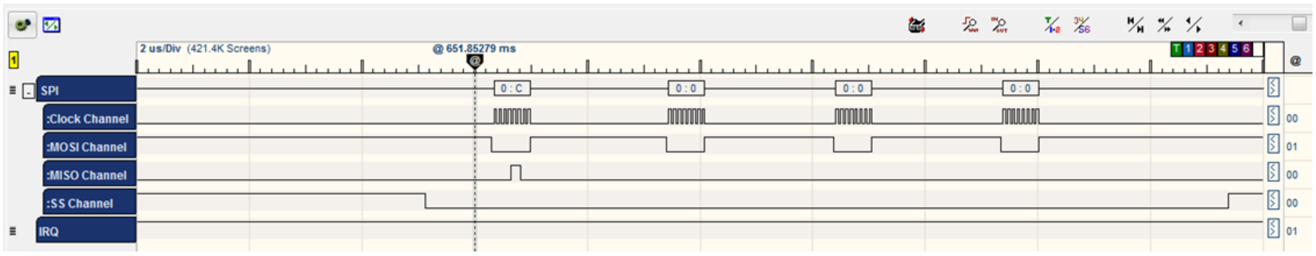




WINC acknowledges the command by sending three bytes [C8] [0] [F3].



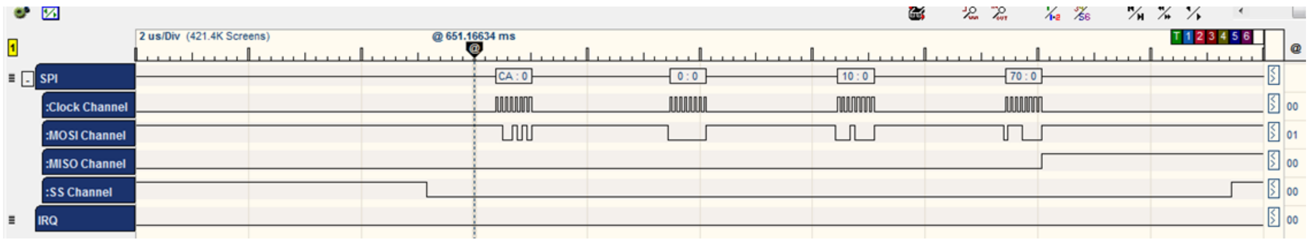
WINC sends the data block (four bytes).



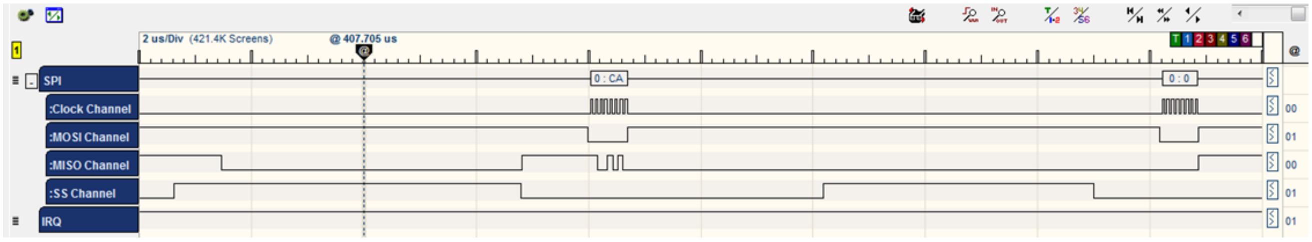
Now, after HIF layer received the response, it interrupts the chip to announce host Rx is done.

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

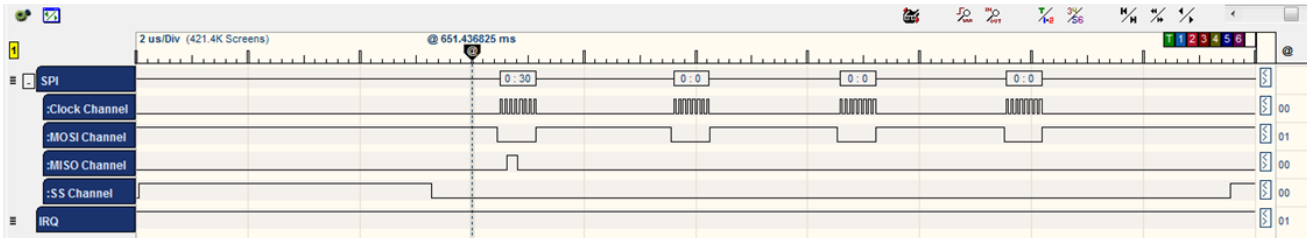
```
Command      CMD_SINGLE_READ:     0xCA         /* single word (4 bytes) read            */
              BYTE [0] = CMD_SINGLE_READ
              BYTE [1] = address >> 16;        /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
              BYTE [2] = address >> 8;
              BYTE [3] = address;
```



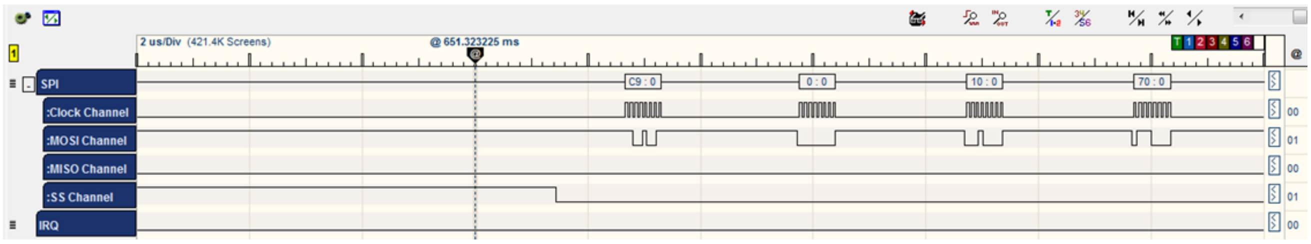
WINC acknowledges the command by sending three bytes [CA] [0] [F3].

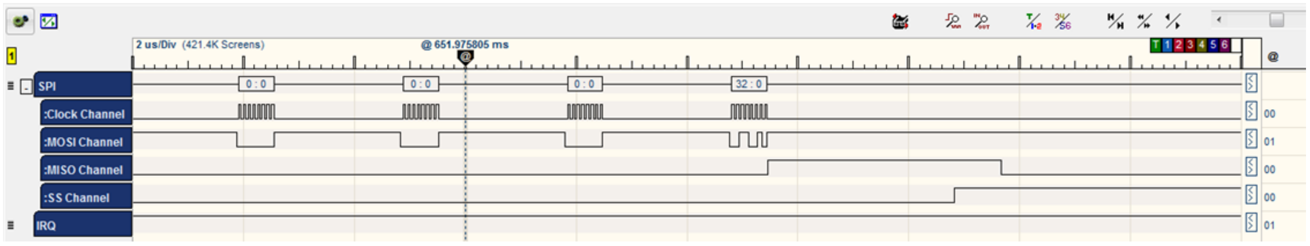


Then WINC chip sends the value of the register 0x1070 which equals 0x30.

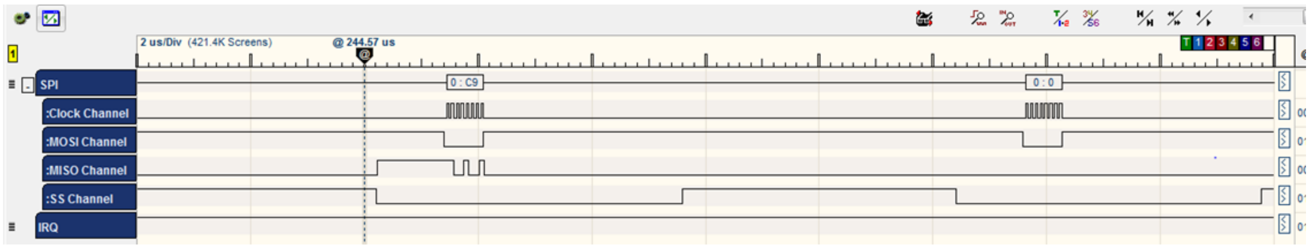


```
Command      CMD_SINGLE_WRITE:0XC9       /* single word write */
              BYTE [0] = CMD_SINGLE_WRITE
              BYTE [1] = address >> 16;         /* WIFI_HOST_RCV_CTRL_0 address = 0x1070  */
              BYTE [2] = address >> 8;
              BYTE [3] = address;
              BYTE [4] =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





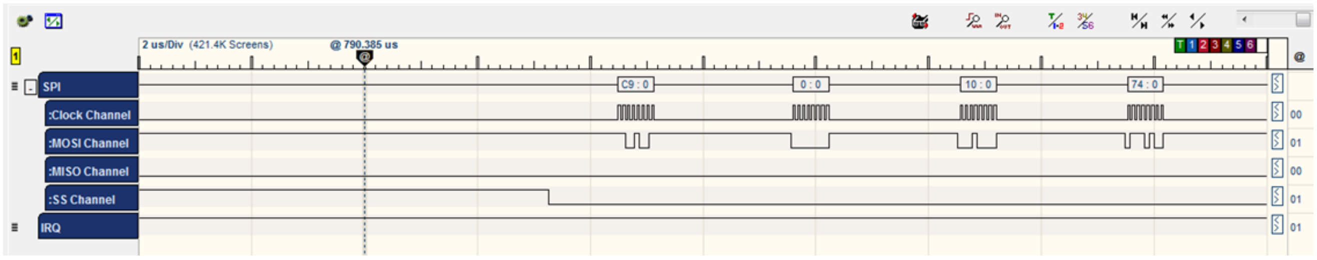
The chip acknowledges the command by sending two bytes [C9] [0].

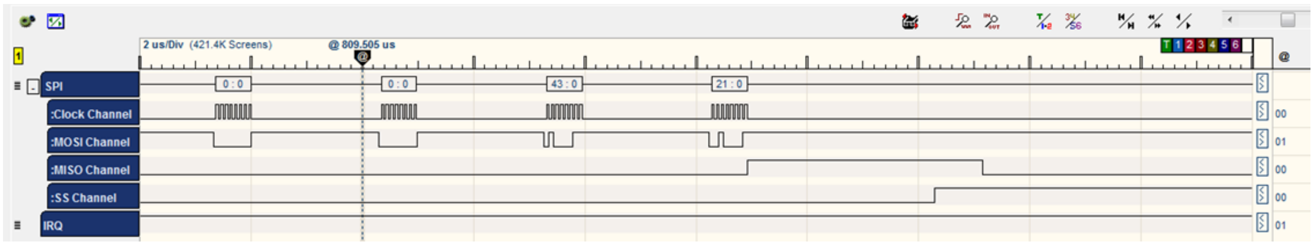


The HIF layer allows the chip to enter sleep mode again.

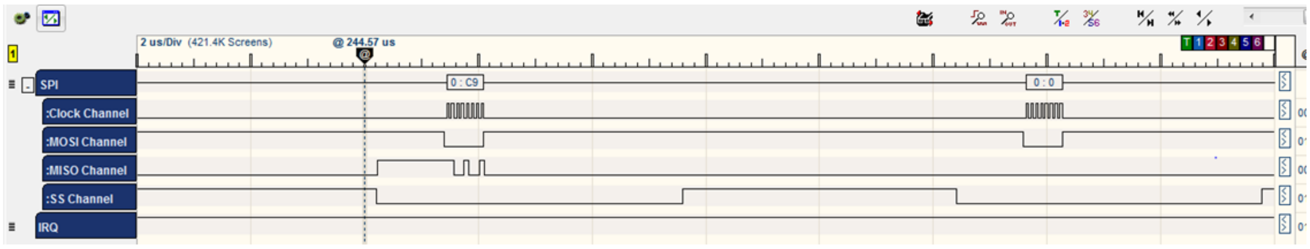
```
sint8 hif_chip_sleep(void)
{
        sint8 ret = M2M_SUCCESS;
        uint32 reg = 0;
        ret = nm_write_reg(WAKE_REG, SLEEP_VALUE);
        /* Clear bit 1 */
        ret = nm_read_reg_with_ret(0x1, &reg);
        if(reg&0x2)
        {
                reg &=~(1 << 1);
                ret = nm_write_reg(0x1,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

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



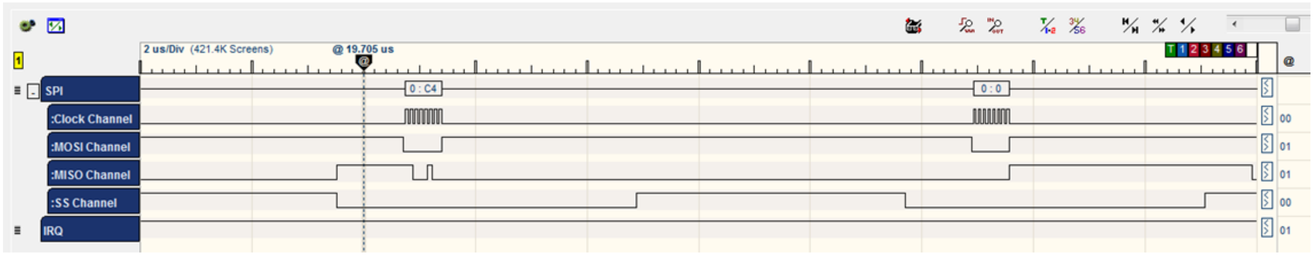


WINC acknowledges the command by sending two bytes [C9] [0].

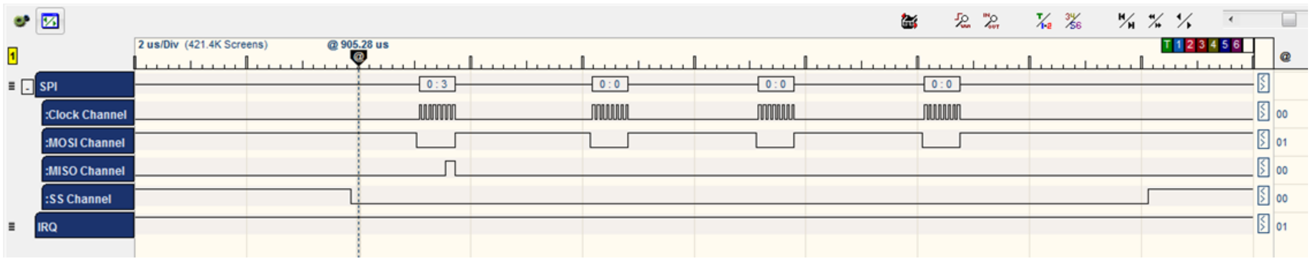


```
Command       CMD_INTERNAL_READ:   0xC4        /* internal register read   */
              BYTE [0] = CMD_INTERNAL_READ
              BYTE [1] = address >> 8;         /* address = 0x01           */
              BYTE [1] |= (1 << 7);            /* clockless register       */
              BYTE [2]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

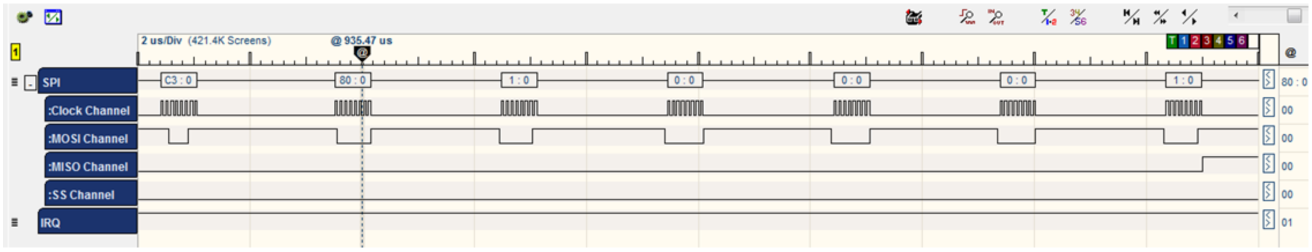
WINC acknowledges the command by sending three bytes [C4] [0] [F3].



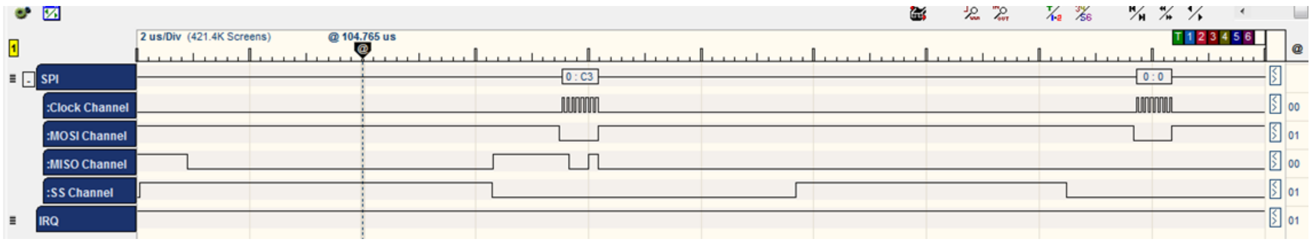
Then WINC chip sends the value of the register 0x01 which equals 0x03.



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



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



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

Appendix A. How to Generate Certificates

A.1 Introduction

This chapter explains the required procedures to create and sign custom certificates using OpenSSL, to use this guide you must install OpenSSL to your Machine.

OpenSSL is an open-source implementation of the SSL and TLS protocols. The core library, written in the C programming language, implements basic cryptographic functions and provides various utility functions.

OpenSSL can be downloaded from the following URL: https://www.openssl.org/related/binaries.html

A.2 Steps

After installing OpenSSL, open a CMD prompt and navigate to the directory where OpenSSL was installed (ex: C:\OpenSSL-Win64\bin).

- First you need to generate a key for our The CA (certification authority).To generate a 4096-bit long RSA (will create a new file CA_KEY.key to store the random key):

  CMD: **openssl genrsa -out CA_KEY.key 4096**

- Next, create your self-signed root CA certificate CA_CERT.crt; you'll need to provide some data for your Root certificate.

  CMD: **openssl req -new -x509 -days 1826 -key CA_KEY.key -out CA_CERT.crt**

- Next step is to create the custom certificate which will be signed by the CA root certificate created earlier. First, generate the key:

  CMD: **openssl genrsa -out Custom.key 4096**

- Using the key generated above, you should generate a certificate request file (csr):

  CMD: **openssl req -new -key Custom.key -out CertReq.csr**

- Finally: process the request for the certificate and get it signed by the root CA.

  CMD: **openssl x509 -req -days 730 -in CertReq.csr -CA CA_CERT.crt -CAkey CA_KEY.key -set_serial 01 -out CustomCert.crt**

Appendix B. X.509 Certificate Format and Conversion

B.1 Introduction

The most known encodings for the X.509 digital certificates are PEM and DER formats.

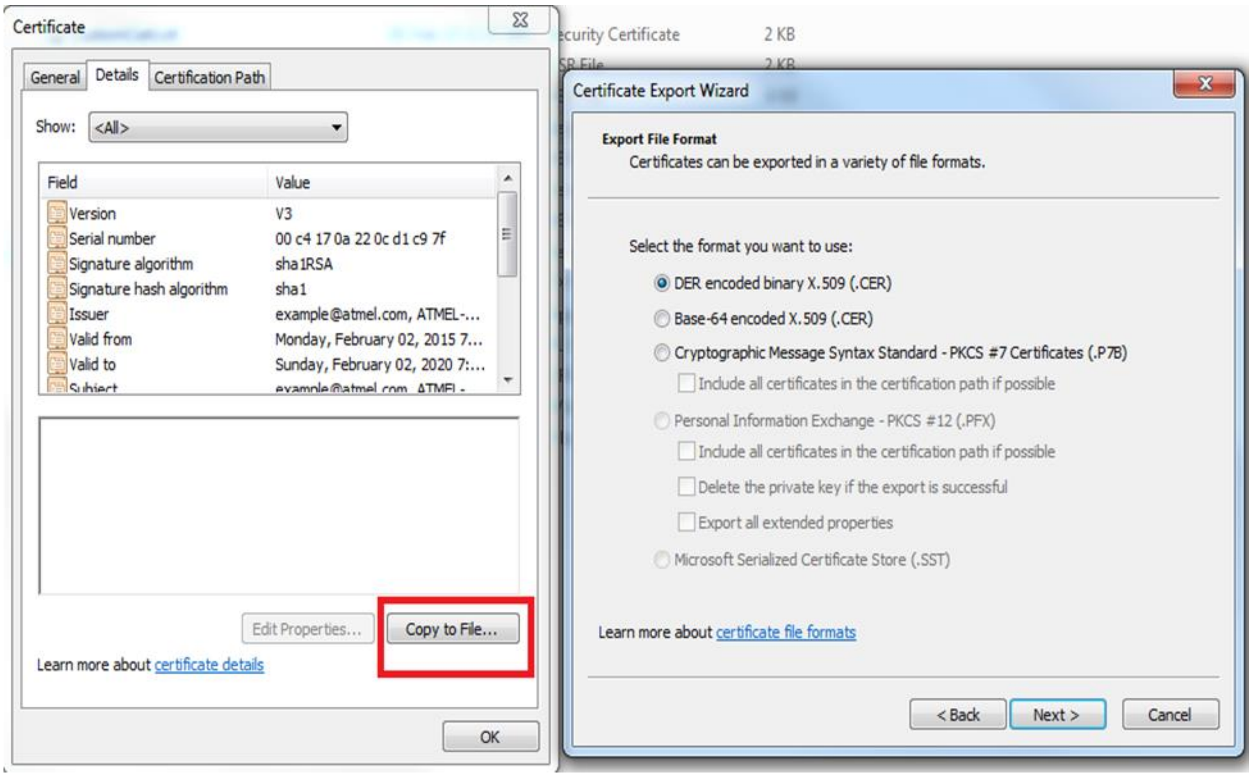
The PEM format is base64 encoding of the DER enclosed with between "-----BEGIN CERTIFICATE-----" and "-----END CERTIFICATE-----".

B.2 Conversion Between Different Formats

The current implementation of WINC root_certificate_downloader supports only DER format. So, if the certificate is not in DER it must be converted to DER. This conversion can be done by several methods:

B.2.1 Using Windows

From Windows®, double click on the .pem certificate file and then go to Details Tab and press “Copy to File”. Follow the wizard until finish.



Show: <All>

Field	Value
Version	V3
Serial number	00 c4 17 0a 22 0c d1 c9 7f
Signature algorithm	sha1RSA
Signature hash algorithm	sha1
Issuer	example@atmel.com, ATMEL-...
Valid from	Monday, February 02, 2015 7...
Valid to	Sunday, February 02, 2020 7:...
Subject	example@atmel.com, ATMEL-...

Learn more about certificate details

ecurity Certificate 2 KB

SP File 2 KB

Export File Format

Certificates can be exported in a variety of file formats.

Select the format you want to use:

- [✓] DER encoded binary X.509 (.CER)
- Base-64 encoded X.509 (.CER)
- Cryptographic Message Syntax Standard - PKCS #7 Certificates (.P7B)
  - Include all certificates in the certification path if possible
- Personal Information Exchange - PKCS #12 (.PFX)
  - Include all certificates in the certification path if possible
  - Delete the private key if the export is successful
  - Export all extended properties
- Microsoft Serialized Certificate Store (.SST)

Learn more about certificate file formats

B.2.2 Using OpenSSL

The famous OpenSSL could be used for certificate conversion by the following command.

```
openssl x509 -outform der -in certificate.pem -out certificate.der
```

B.2.3 Online Conversion

There are useful online tools which provide conversion between certificate formats, which can be found through searching online using keywords such as "OpenSSL".

Appendix C. How to Download the Certificate into WINC

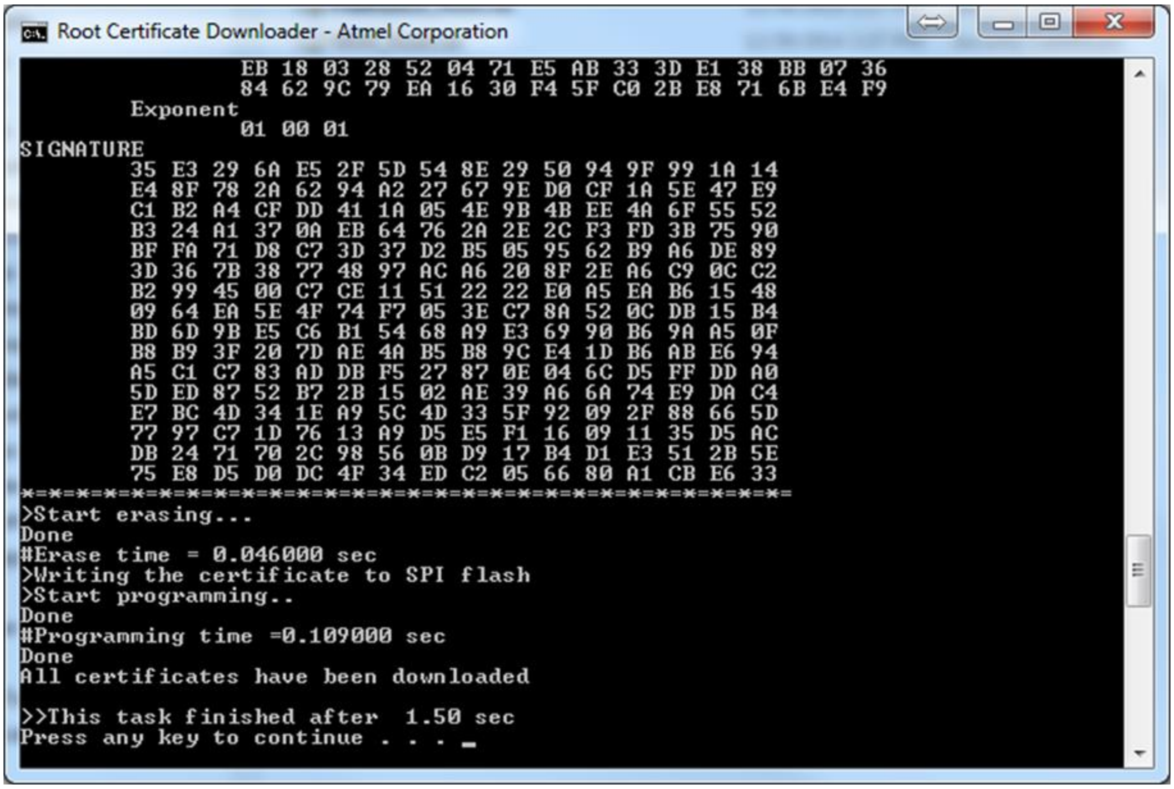
C.1 Overview

The WINC save the certificate inside the SPI flash in 4K sector (so the maximum size of all certificates in flash should be less than 4K).

C.2 Certificate Downloading

To download the root certificate Execute the batch file ***RootCertDownload.bat*** inside the release package:

- I²C Downloader
  /src/Tools/root_certificate_downloader/debug_I2C/RootCertDownload.bat
- UART Downloader
  /src/Tools/root_certificate_downloader/debug UART/RootCertDownload.bat



```
                EB 18 03 28 52 04 71 E5 AB 33 3D E1 38 BB 07 36
                84 62 9C 79 EA 16 30 F4 5F C0 2B E8 71 6B E4 F9
        Exponent
                01 00 01
SIGNATURE
        35 E3 29 6A E5 2F 5D 54 8E 29 50 94 9F 99 1A 14
        E4 8F 78 2A 62 94 A2 27 67 9E D0 CF 1A 5E 47 E9
        C1 B2 A4 CF DD 41 1A 05 4E 9B 4B EE 4A 6F 55 52
        B3 24 A1 37 0A EB 64 76 2A 2E 2C F3 FD 3B 75 90
        BF FA 71 D8 C7 3D 37 D2 B5 05 95 62 B9 A6 DE 89
        3D 36 7B 38 77 48 97 AC A6 20 8F 2E A6 C9 0C C2
        B2 99 45 00 C7 CE 11 51 22 22 E0 A5 EA B6 15 48
        09 64 EA 5E 4F 74 F7 05 3E C7 8A 52 0C DB 15 B4
        BD 6D 9B E5 C6 B1 54 68 A9 E3 69 90 B6 9A A5 0F
        B8 B9 3F 20 7D AE 4A B5 B8 9C E4 1D B6 AB E6 94
        A5 C1 C7 83 AD DB F5 27 87 0E 04 6C D5 FF DD A0
        5D ED 87 52 B7 2B 15 02 AE 39 A6 6A 74 E9 DA C4
        E7 BC 4D 34 1E A9 5C 4D 33 5F 92 09 2F 88 66 5D
        77 97 C7 1D 76 13 A9 D5 E5 F1 16 09 11 35 D5 AC
        DB 24 71 70 2C 98 56 0B D9 17 B4 D1 E3 51 2B 5E
        75 E8 D5 D0 DC 4F 34 ED C2 05 66 80 A1 CB E6 33
*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=
>Start erasing...
Done
#Erase time = 0.046000 sec
>Writing the certificate to SPI flash
>Start programming..
Done
#Programming time =0.109000 sec
Done
All certificates have been downloaded

>>This task finished after  1.50 sec
Press any key to continue . . . _
```

C.3 Adding New Certificate

- Open the file RootCertDownload.bat. There you find the following command:

root_certificate_downloader -n 2 NMA_Root.cer PROWL_Root.cer

- Update the batch for example to add NMI_root.cer (by update the n number of certificated and add the new certificate to the argument)

root_certificate_downloader -n 3 NMA_Root.cer PROWL_Root.cer NMI_root.cer

Appendix D. Firmware Image Downloader

The Firmware Downloader script use the EDBG SAMD21/W25 UART as the main interface.

1. Downloads the serial bridge application on the SAMD21/W25 using the Atmel atprogram.exe.
2. After downloading, the application will initialize the WINC in download mode and start listen on EDBG UART for UART commands.
3. The application will convert the UART commands to SPI commands.
4. The script will wait a couple of seconds until the application initialization finishes, then executes the firmware downloader then the gain builder given the UART argument and the firmware/gain sheets path argument.

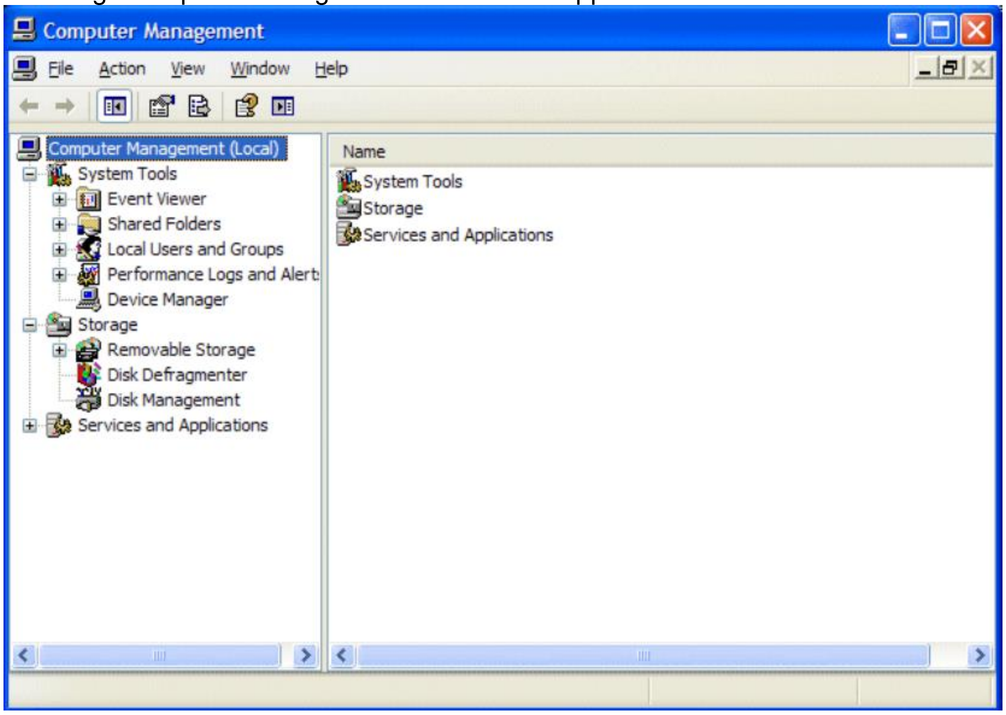
D.1 Preparing Environment

After connecting SAMD21 Debug USB and WINC Virtual COM Port USB port to computer, make sure that their drivers are already installed and correctly detected by Windows.

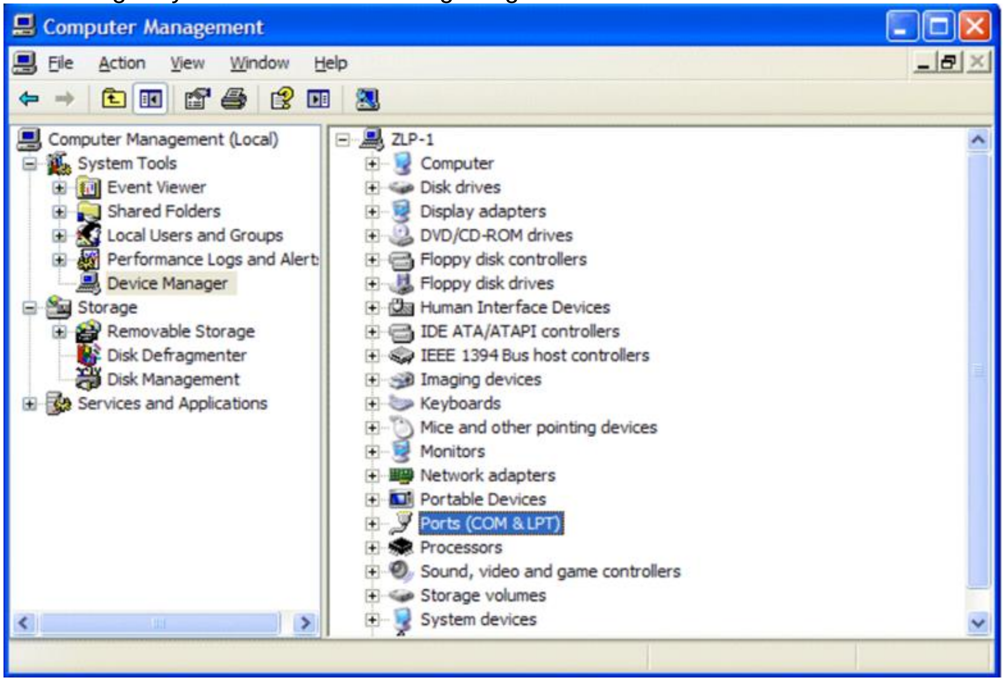
The WINC COM Port is configured (115200N-8-1) for Debug traces.

To check this, here are the steps fort Windows XP and 7:

1. Right click on the icon “My Computer”, a menu will appear. Scroll down and select “Manage”. The following “Computer Management” window will appear:

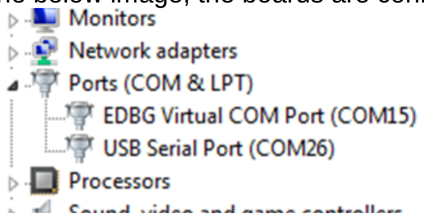


2. From the list on the left hand side, select (click on) "Device Manager". The "View" will change to something very similar to the following image:



3. In the Right Hand Panel, select (double click) Ports (COM and LPT). The "View" will change to something like the following (actual info shown depends on your PC).

   In the below image, the boards are connected and using COM15, and COM26:

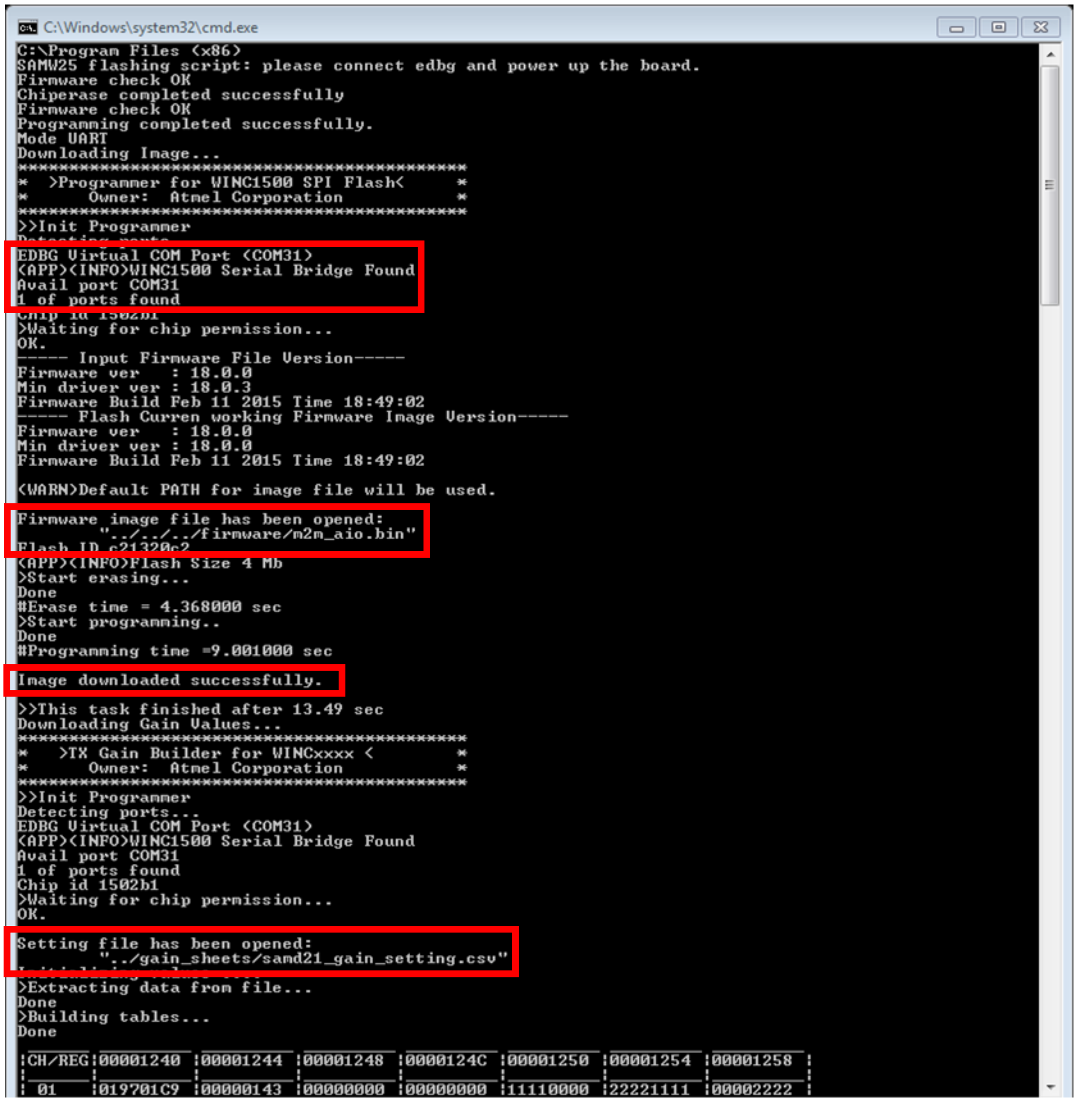


WARNING Make sure the EDBG port is not in use at Atmel Studio or with any other serial monitor before downloading Also make sure that firmware bin file is located at "./firm-ware/m2m_aio.bin" and gain files is located at "./Tools/gain_builder/gain_sheets/".

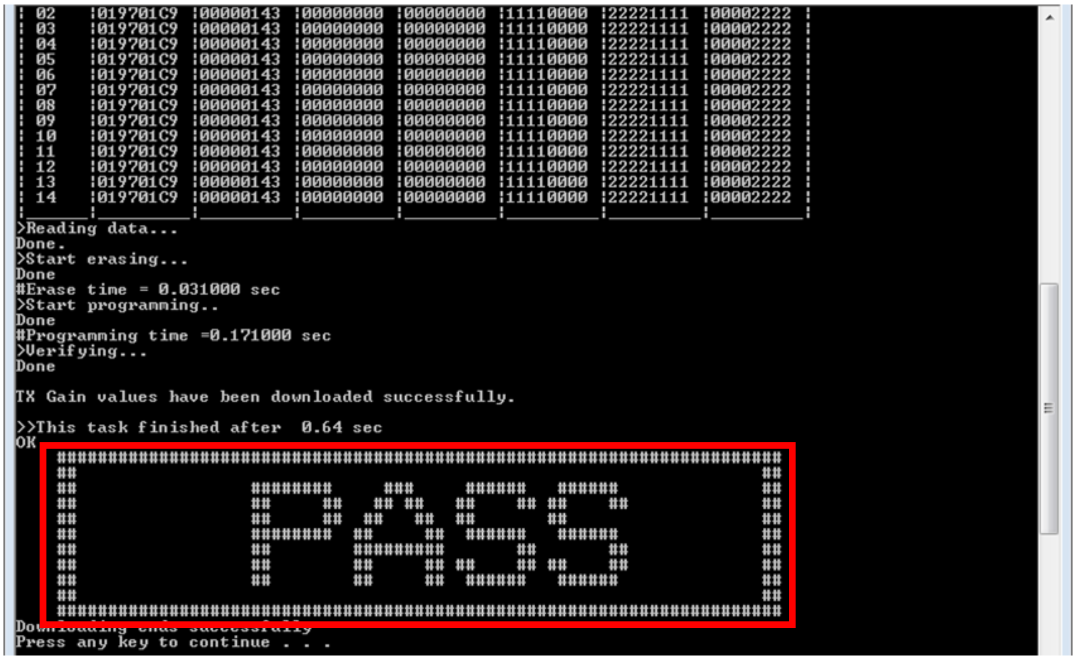
TIPS The same thing is valid for SAMW25, by running "down-load_all_sb_samw25_xplained_pro.bat".

D.2 Download Firmware

Run the "download_all_sb_samd21_xplained_pro.bat" script that is associated with the release to download firmware and gain settings for SAMD21.



```
C:\Program Files (x86)
SAMW25 flashing script: please connect edbg and power up the board.
Firmware check OK
Chiperase completed successfully
Firmware check OK
Programming completed successfully.
Mode UART
Downloading Image...
********************************************
*  >Programmer for WINC1500 SPI Flash<      *
*       Owner:  Atmel Corporation           *
********************************************
>>Init Programmer
Detecting ports...
EDBG Virtual COM Port (COM31)
(APP)(INFO)WINC1500 Serial Bridge Found
Avail port COM31
1 of ports found
Chip id 1502b1
>Waiting for chip permission...
OK.
----- Input Firmware File Version-----
Firmware ver    : 18.0.0
Min driver ver : 18.0.3
Firmware Build Feb 11 2015 Time 18:49:02
----- Flash Curren working Firmware Image Version-----
Firmware ver    : 18.0.0
Min driver ver : 18.0.0
Firmware Build Feb 11 2015 Time 18:49:02

(WARN)Default PATH for image file will be used.

Firmware image file has been opened:
        "../../../firmware/m2m_aio.bin"
Flash ID c21320c2
(APP)(INFO)Flash Size 4 Mb
>Start erasing...
Done
#Erase time = 4.368000 sec
>Start programming..
Done
#Programming time =9.001000 sec

Image downloaded successfully.

>>This task finished after 13.49 sec
Downloading Gain Values...
********************************************
*   >TX Gain Builder for WINCxxxx <         *
*       Owner:  Atmel Corporation           *
********************************************
>>Init Programmer
Detecting ports...
EDBG Virtual COM Port (COM31)
(APP)(INFO)WINC1500 Serial Bridge Found
Avail port COM31
1 of ports found
Chip id 1502b1
>Waiting for chip permission...
OK.

Setting file has been opened:
        "../gain_sheets/samd21_gain_setting.csv"
Initializing values ...
>Extracting data from file...
Done
>Building tables...
Done
 _______ _________ _________ _________ _________ _________ _________ _________
¦CH/REG¦00001240 ¦00001244 ¦00001248 ¦0000124C ¦00001250 ¦00001254 ¦00001258 ¦
¦_______¦_________¦_________¦_________¦_________¦_________¦_________¦_________¦
¦ 01    ¦019701C9 ¦00000143 ¦00000000 ¦00000000 ¦11110000 ¦22221111 ¦00002222 ¦
```



```
¦ 02    ¦019701C9 ¦00000143 ¦00000000 ¦00000000 ¦11110000 ¦22221111 ¦00002222 ¦
¦ 03    ¦019701C9 ¦00000143 ¦00000000 ¦00000000 ¦11110000 ¦22221111 ¦00002222 ¦
¦ 04    ¦019701C9 ¦00000143 ¦00000000 ¦00000000 ¦11110000 ¦22221111 ¦00002222 ¦
¦ 05    ¦019701C9 ¦00000143 ¦00000000 ¦00000000 ¦11110000 ¦22221111 ¦00002222 ¦
¦ 06    ¦019701C9 ¦00000143 ¦00000000 ¦00000000 ¦11110000 ¦22221111 ¦00002222 ¦
¦ 07    ¦019701C9 ¦00000143 ¦00000000 ¦00000000 ¦11110000 ¦22221111 ¦00002222 ¦
¦ 08    ¦019701C9 ¦00000143 ¦00000000 ¦00000000 ¦11110000 ¦22221111 ¦00002222 ¦
¦ 09    ¦019701C9 ¦00000143 ¦00000000 ¦00000000 ¦11110000 ¦22221111 ¦00002222 ¦
¦ 10    ¦019701C9 ¦00000143 ¦00000000 ¦00000000 ¦11110000 ¦22221111 ¦00002222 ¦
¦ 11    ¦019701C9 ¦00000143 ¦00000000 ¦00000000 ¦11110000 ¦22221111 ¦00002222 ¦
¦ 12    ¦019701C9 ¦00000143 ¦00000000 ¦00000000 ¦11110000 ¦22221111 ¦00002222 ¦
¦ 13    ¦019701C9 ¦00000143 ¦00000000 ¦00000000 ¦11110000 ¦22221111 ¦00002222 ¦
¦ 14    ¦019701C9 ¦00000143 ¦00000000 ¦00000000 ¦11110000 ¦22221111 ¦00002222 ¦
¦_______¦_________¦_________¦_________¦_________¦_________¦_________¦_________¦
>Reading data...
Done.
>Start erasing...
Done
#Erase time = 0.031000 sec
>Start programming..
Done
#Programming time =0.171000 sec
>Verifying...
Done

TX Gain values have been downloaded successfully.

>>This task finished after  0.64 sec
OK
    ###################################################################
    ##                                                               ##
    ##                ########    ###     ######  ######             ##
    ##                ##     ##  ## ##   ##    ## ##    ##           ##
    ##                ##     ## ##   ##  ##       ##                 ##
    ##                ######## ##     ##  ######  ######             ##
    ##                ##       #########       ##       ##           ##
    ##                ##       ##     ## ##    ## ##    ##           ##
    ##                ##       ##     ##  ######  ######             ##
    ##                                                               ##
    ###################################################################
Downloading ends successfully
Press any key to continue . . .
```

Appendix E. Gain Settings Builder

E.1 Introduction

Gain setting values, are those values used by RF with different rates to configure transmission power.

This appendix helps to calculate these values and store them in Flash to use them otherwise default values will be used.

E.2 Preparing Environment

Make sure the environment is ready for building and downloading gain settings as in Appendix D: Firmware Image Downloader.

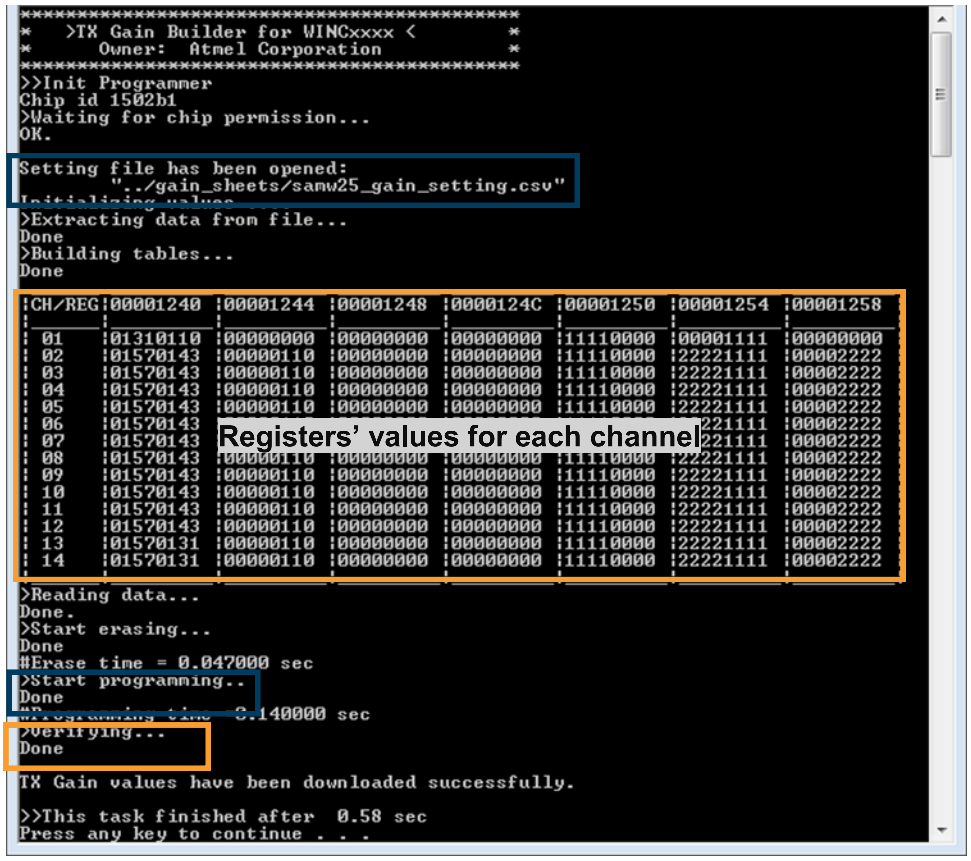
E.3 How to use

E.3.1 Method 1

- Replace the data of samd21_gain_setting.csv file in the project's folder with the new data where the file's location is the default (./Tools/gain_builder/gain_sheets/).
- Then run your application. It will calculate and store you data in Flash.

E.3.2 Method 2

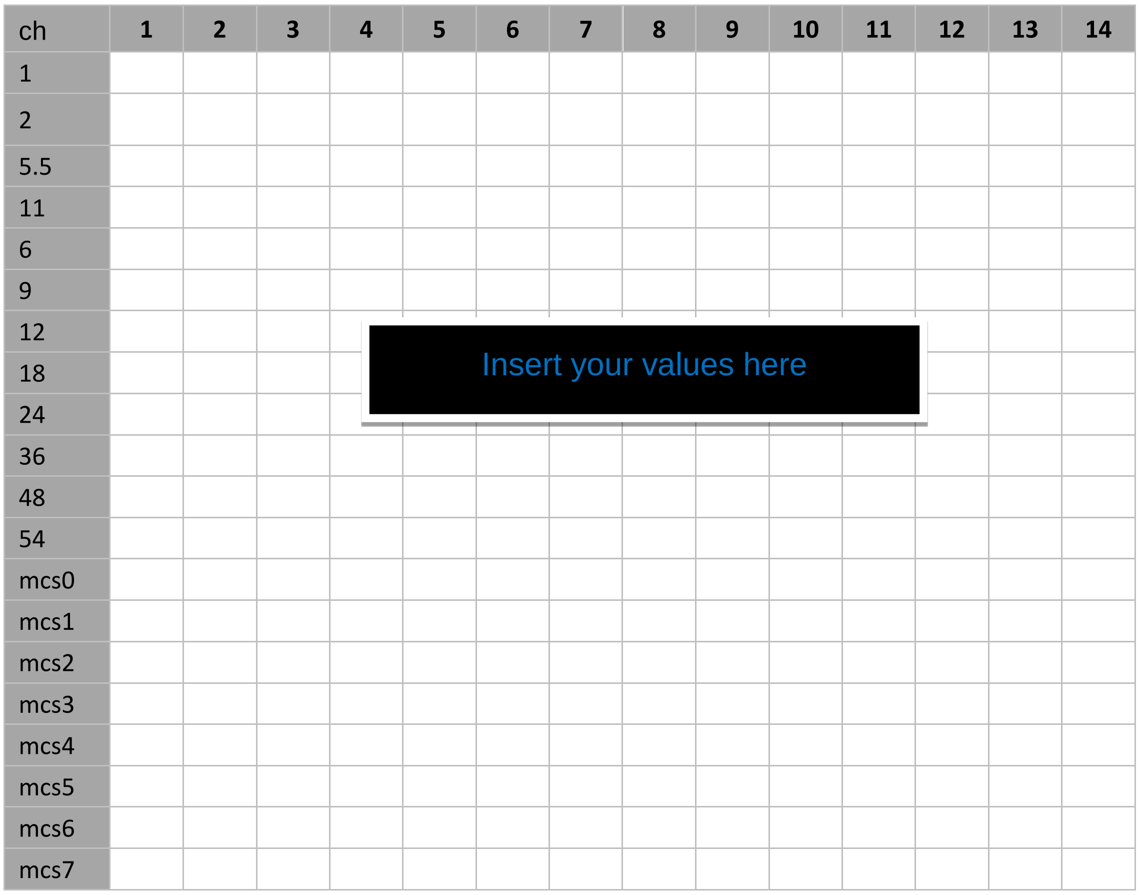
- If you have a different file with data in a different path, then open "gain_build_and_download.bat" patch file and update it with the new path and file like:
  -fw_path ../gain_sheets/samd21_gain_setting.csv → -fw_path c:/gain_values.csv
- Then run your application by double click on modified patch file



```
*******************************************
*   >TX Gain Builder for WINCxxxx <       *
*       Owner:  Atmel Corporation         *
*******************************************
>>Init Programmer
Chip id 1502b1
>Waiting for chip permission...
OK.

Setting file has been opened:
        "../gain_sheets/samw25_gain_setting.csv"
Initializing values ...
>Extracting data from file...
Done
>Building tables...
Done

|CH/REG|00001240 |00001244 |00001248 |0000124C |00001250 |00001254 |00001258
|______|_________|_________|_________|_________|_________|_________|_________
| 01   |01310110 |00000000 |00000000 |00000000 |11110000 |00001111 |00000000
| 02   |01570143 |00000110 |00000000 |00000000 |11110000 |22221111 |00002222
| 03   |01570143 |00000110 |00000000 |00000000 |11110000 |22221111 |00002222
| 04   |01570143 |00000110 |00000000 |00000000 |11110000 |22221111 |00002222
| 05   |01570143 |00000110 |00000000 |00000000 |11110000 |22221111 |00002222
| 06   |01570143 [illegible]                               221111 |00002222
| 07   |01570143 [illegible]                               221111 |00002222
| 08   |01570143 |00000110 |00000000 |00000000 |11110000 |22221111 |00002222
| 09   |01570143 |00000110 |00000000 |00000000 |11110000 |22221111 |00002222
| 10   |01570143 |00000110 |00000000 |00000000 |11110000 |22221111 |00002222
| 11   |01570143 |00000110 |00000000 |00000000 |11110000 |22221111 |00002222
| 12   |01570143 |00000110 |00000000 |00000000 |11110000 |22221111 |00002222
| 13   |01570131 |00000110 |00000000 |00000000 |11110000 |22221111 |00002222
| 14   |01570131 |00000110 |00000000 |00000000 |11110000 |22221111 |00002222
|______|_________|_________|_________|_________|_________|_________|_________
>Reading data...
Done.
>Start erasing...
Done
#Erase time = 0.047000 sec
>Start programming..
Done
#Programming time = 0.140000 sec
>Verifying...
Done

TX Gain values have been downloaded successfully.

>>This task finished after  0.58 sec
Press any key to continue . . .
```

WARNING

– The .csv file must be sorted based on gain rates like the templates
– There are two different busses to run your app I²C or UART
– Your file must have 15 columns and 21 Rows for all channels such as the following template:



ch	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1														
2														
5.5														
11														
6														
9														
12														
18														
24														
36														
48														
54														
mcs0														
mcs1														
mcs2														
mcs3														
mcs4														
mcs5														
mcs6														
mcs7														

Insert your values here

Appendix F. API Reference

F.1 WLAN Module

F.1.1 Defines

Define	Definition	Value
#defineM2M_FIRMWARE_VERSION_MAJOR_NO	Firmware Major release version number.	**18**
#define M2M_FIRMWARE_VERSION_MINOR_NO	Firmware Minor release version number.	**0**
#define M2M_FIRMWARE_VERSION_PATCH_NO	Firmware patch release version number.	**2**
#define M2M_DRIVER_VERSION_PATCH_NO	Driver patch release version number.	**0**
#define M2M_BUFFER_MAX_SIZE (1600UL)	Maximum size for the shared packet buffer.	**- 4**
#define M2M_MAC_ADDRES_LEN	The size for 802.11 MAC address.	**6**
#define M2M_ETHERNET_HDR_OFFSET	The offset of the Ethernet header within the WLAN TX Buffer.	**34**
#define M2M_ETHERNET_HDR_LEN	Length of the Ethernet header in bytes.	**14**
#define M2M_MAX_SSID_LEN	Maximum size for the Wi-Fi SSID including the NULL termination.	**33**
#define M2M_MAX_PSK_LEN	Maximum size for the WPA PSK including the NULL termination.	**65**
#define M2M_DEVICE_NAME_MAX	Maximum Size for the device name including the NULL termination.	**48**
#define M2M_LISTEN_INTERVAL	The STA uses the Listen Interval parameter to indicate to the AP how many beacon intervals it shall sleep before it retrieves the queued frames.	**1**
#define M2M_1X_PWD_MAX	The maximum size of the password including the NULL termination. It is used for RADIUS authentication in case of connecting the device to an AP secured with WPA-Enterprise.	**41**
#define M2M_CUST_IE_LEN_MAX	The maximum size of IE (Information Element).	**252**
#define M2M_CONFIG_CMD_BASE	The base value of all the host configuration commands opcodes.	**1**
#define M2M_SERVER_CMD_BASE	The base value of all the power save mode host commands codes.	**20**
#define M2M_STA_CMD_BASE	The base value of all the station mode host commands opcodes.	**40**
#define M2M_AP_CMD_BASE	The base value of all the Access Point mode host commands opcodes.	**70**
#define M2M_P2P_CMD_BASE	The base value of all the P2P mode host commands opcodes.	**90**
#define M2M_OTA_CMD_BASE	The base value of all the OTA mode host commands opcodes.	**100**

Define	Definition	Value
#define WEP_40_KEY_SIZE	Indicate the wep key size in bytes for 40 bit hex passphrase.	**((uint8)5)**
#define WEP_104_KEY_SIZE	Indicate the wep key size in bytes for 104 bit hex passphrase.	**((uint8)13)**
#define WEP_40_KEY_STRING_SIZE	Indicate the wep key size in bytes for 40 bit string passphrase.	**(uint8)10)**
#define WEP_104_KEY_STRING_SIZE	Indicate the wep key size in bytes for 104 bit string passphrase.	**((uint8)26**
#define WEP_KEY_MAX_INDEX	Indicate the max key index value for WEP authentication.	**(uint8)4**
#define M2M_SCAN_MIN_NUM_SLOTS	The min. number of scan slots performed by the WINC firmware.	**2**
#define M2M_SCAN_MIN_SLOT_TIME	The min. duration in milliseconds of a scan slots performed by the WINC firmware.	**(20)**
#define M2M_SCAN_FAIL	Indicate that the WINC firmware has failed to perform the scan operation.	**((uint8)1)**
#define M2M_JOIN_FAIL	Indicate that the WINC firmware has failed to join the BSS.	**((uint8)2)**
#define M2M_AUTH_FAIL	Indicate that the WINC firmware has failed to authenticate with the AP.	**((uint8)3)**
#define M2M_ASSOC_FAIL	Indicate that the WINC firmware has failed to associate with the AP.	**((uint8)4)**
#define M2M_SCAN_ERR_WIFI	Currently not used.	**((sint8)-2)**
#define M2M_SCAN_ERR_IP	Currently not used.	**((sint8)-3)**
#define M2M_SCAN_ERR_AP	Currently not used.	**((sint8)-4)**
#define M2M_SCAN_ERR_P2P	Currently not used.	**((sint8)-5)**
#define M2M_SCAN_ERR_WPS	Currently not used.	**((sint8)-6)**
#define M2M_DEFAULT_CONN_EMPTY_LIST	A failure response that indicates an empty network list as a result to the function call m2m_default_connect.	**((sint8)-20)**
#define M2M_DEFAULT_CONN_SCAN_MISMATCH	A failure response that indicates that no one of the cached networks was found in the scan results, as a result to the function call m2m_default_connect.	**((sint8)-21)**
#define M2M_WIFI_FRAME_TYPE_ANY	Set monitor mode to receive any of the frames types.	**0xFF**
#define M2M_WIFI_FRAME_SUB_TYPE_ANY	Set monitor mode to receive frames with any sub type.	**0xFF**
#define OTA_ROLLB_STATUS_VALID	Magic value updated in the Control structure in case of ROLLACK image Valid.	**0x12526285**
#define OTA_ROLLB_STATUS_INVALID	Magic value updated in the Control structure in case of ROLLACK image Invalid.	**0x23987718**
#define OTA_MAGIC_VALUE	Magic value set at the beginning of the OTA image header.	**0x1ABCDEF9**

Define	Definition	Value
#define OTA_SHA256_DIGEST_SIZE	Sha256 digest size in the ota image, the sha256 digest is set at the beginning of image before the OTA header.	**32**
#define OTA_SUCCESS	OTA Success status.	**0**
#define OTA_ERR_WORKING_IMAGE_LOAD_FAIL	Failure to load the firmware image.	**((sint8)-1)**
#define OTA_ERR_INVAILD_CONTROL_SEC	Control structure is being created.	**((sint8)-2)**
#define M2M_ERR_OTA_SWITCH_FAIL	Switching to the updated image failed as may be the image is invalid.	**((sint8)-3)**
#define M2M_ERR_OTA_START_UPDATE_FAIL	Ota update fail due to multiple reasons.	**((sint8)-4)**
#define M2M_ERR_OTA_ROLLBACK_FAIL	Roll-back failed due to Roll-back image is not valid.	**((sint8)-5)**
#define M2M_ERR_OTA_INVAILD_FLASH_SIZE	The OTA support at least 4MB flash size, failure to provide at least 4MB from the flash will result in this error.	**((sint8)-6)**
#define M2M_ERR_OTA_INVAILD_ARG	Invalid argument in any OTA Function.	**((sint8)-7)**

F.1.2 Enumeration/Typedef

void (* tpfOtaUpdateCb)(uint8 u8OtaUpdateStatusType,uint8 u8OtaUpdateStatus)

A callback to get OTA status update, the callback provide the status type and its status the OTA callback provides the download status, the switch to the downloaded firmware status and roll-back status.

Parameters:

in	u8OtaUpdateStatusType	Possible values are listed in tenuOtaUpdateStatusType. Possible types are: DL_STATUS SW_STATUS RB_STATUS
in	u8OtaUpdateStatus	Possible values are listed in tenuOtaUpdateStatus.

See also:

tenuOtaUpdateStatusType

tenuOtaUpdateStatus

void(* tpfOtaNotifCb)(tstrOtaUpdateInfo *)

A callback to get notification about a potential OTA update.

Parameters:

in	*pstrOtaUpdateInfo*	A structure to provide notification payload.

See also:

tstrOtaUpdateInfo

WARNING The notification is not supported (Not implemented).

enum tenuM2mConfigCmd

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

Enumerator Values	
M2M_WIFI_REQ_RESTART	Reserved for Firmware use not allowed from host driver.
M2M_WIFI_REQ_SET_MAC_ADDRESS	Set the WINC mac address (will overwrite production eFused boards).
M2M_WIFI_REQ_CURRENT_RSSI	Request the current connected AP RSSI.
M2M_WIFI_RESP_CURRENT_RSSI	Response to M2M_WIFI_REQ_CURRENT_RSSI with the RSSI value.
M2M_WIFI_REQ_SET_DEVICE_NAME	Set the WINC device name property.
M2M_WIFI_REQ_START_PROVISION_MODE	Start the provisioning mode for the M2M Device.
M2M_WIFI_RESP_PROVISION_INFO	Send the provisioning information to the host.
M2M_WIFI_REQ_STOP_PROVISION_MODE	Stop the current running provision mode.
M2M_WIFI_REQ_SET_SYS_TIME	Set time of day from host.
M2M_WIFI_REQ_ENABLE_SNTP_CLIENT	Enable the simple network time protocol to get the time from the in-ternet. This is required for security purposes.
M2M_WIFI_REQ_DISABLE_SNTP_CLIENT	Disable the simple network time protocol for applications that do not need it.
M2M_WIFI_RESP_MEMORY_RECOVER	Reserved for firmware memory debugging.
M2M_WIFI_REQ_CUST_INFO_ELEMENT	Add Custom Element to Beacon Management Frame.

enum tenuM2mServerCmd

This enum contains all the WINC commands while in PS mode. These commands are currently not supported.

Enumerator Values
M2M_WIFI_REQ_CLIENT_CTRL
M2M_WIFI_RESP_CLIENT_INFO
M2M_WIFI_REQ_SERVER_INIT

enum tenuM2mStaCmd

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

Enumerator Values	
M2M_WIFI_REQ_CONNECT	Connect with AP command.
M2M_WIFI_REQ_DEFAULT_CONNECT	Connect with default AP command.
M2M_WIFI_RESP_DEFAULT_CONNECT	Connect with default AP response.
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_REQ_WPS_SCAN	Request WPS scan command.
M2M_WIFI_RESP_SCAN_DONE	Scan complete notification response.
M2M_WIFI_REQ_SCAN_RESULT	Request Scan results command.
M2M_WIFI_RESP_SCAN_RESULT	Request Scan results response.
M2M_WIFI_REQ_WPS	Request WPS start command.
M2M_WIFI_REQ_START_WPS	This command is for internal use by the WINC and should not be used by the host driver.
M2M_WIFI_REQ_DISABLE_WPS	Request to disable WPS command.
M2M_WIFI_REQ_DHCP_CONF	Response indicating that IP address was obtained.
M2M_WIFI_RESP_IP_CONFIGURED	This command is for internal use by the WINC and should not be used by the host driver.
M2M_WIFI_RESP_IP_CONFLICT	Response indicating a conflict in obtained IP address. The user should re attempt the DHCP request.
M2M_WIFI_REQ_ENABLE_MONITORING	Request to enable monitor mode command.
M2M_WIFI_REQ_DISABLE_MONITORING	Request to disable monitor mode command.
M2M_WIFI_RESP_WIFI_RX_PACKET	Indicate that a packet was received in monitor mode.
M2M_WIFI_REQ_SEND_WIFI_PACKET	Send packet in monitor mode.
M2M_WIFI_REQ_LSN_INT	Set Wi-Fi listen interval.
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 ... etc”.
M2M_WIFI_REQ_SET_SCAN_REGION	Set scan region.
M2M_WIFI_REQ_DOZE	Used to force the WINC to sleep in manual PS mode.
M2M_WIFI_REQ_SET_MAC_MCAST	Set the WINC multicast filters.

enum tenuM2mP2pCmd

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

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

enum tenuM2mApCmd

Enumerator Values	
M2M_WIFI_REQ_ENABLE_AP	Enable AP mode command.
M2M_WIFI_REQ_DISABLE_AP	Disable AP mode command

enum tenuM2mOtaCmd

Enumerator Values	
M2M_OTA_REQ_NOTIF_SET_URL	Not supported at the current release.
M2M_OTA_REQ_NOTIF_CHECK_FOR_UP-DATE	Not supported at the current release.
M2M_OTA_REQ_NOTIF_SCHED	Not supported at the current release.
M2M_OTA_REQ_START_UPDATE	The command used to start ota update.
M2M_OTA_REQ_SWITCH_FIRMWARE	The command used to switch to the updated firmware.
M2M_OTA_REQ_ROLLBACK	The command is used rollback to the rollback image.
M2M_OTA_RESP_NOTIF_UPDATE_INFO	Not supported at the current release.
M2M_OTA_RESP_UPDATE_STATUS	The Update callback response for switching and update status.
M2M_OTA_REQ_TEST	Not supported at the current release.

enum tenuM2mIpCmd

Enumerator Values	
M2M_IP_REQ_STATIC_IP_CONF	Used to set static ip address.

enum tenuM2mConnState

Wi-Fi Connection State.

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

enum tenuM2mSecType

Wi-Fi Supported Security types.

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

enum tenuM2mSsidMode

Wi-Fi Supported SSID types.

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

enum tenuM2mScanCh

Wi-Fi RF Channels.

Enumerator Values
M2M_WIFI_CH_1
M2M_WIFI_CH_2
M2M_WIFI_CH_3
M2M_WIFI_CH_4

Enumerator Values
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

Enumerator Values
NORTH_AMERICA
ASIA

enum tenuPowerSaveModes

Power save Modes.

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

enum tenuWPSTrigger

WPS triggering methods.

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

enum tenuOtaUpdateStatus

Enumerator Values	
OTA_STATUS_SUCSESS	OTA Success with not errors.
OTA_STATUS_FAIL	OTA generic fail.
OTA_STATUS_INVAILD_ARG	Invalid or malformed download URL.
OTA_STATUS_INVAILD_RB_IMAGE	Invalid rollback image.
OTA_STATUS_INVAILD_FLASH_SIZE	Flash size on device is not enough for OTA.
OTA_STATUS_AIREADY_ENABLED	An OTA operation is already enabled.
OTA_STATUS_UPDATE_INPROGRESS	An OTA operation update is in progress

enum tenuOtaUpdateStatusType

Enumerator Values	
DL_STATUS	Download OTA file status.
SW_STATUS	Switching to the upgrade firmware status.
RB_STATUS	Roll-back status.

enum tenuWifiFrameType

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

Remarks:

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

See also:

tenuSubTypes

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

enum tenuSubTypes

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

Remarks:

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

See also:

tenuWifiFrameType

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

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

enum tenuInfoElementId

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

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

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

typedef struct tstr1xAuthCredentials

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

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

typedef struct tstrEthInitParam

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

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

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

See also:

tpfAppEthCb tpfAppWifiCb

m2m_wifi_init

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

typedef struct tstrM2MAPConfig

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

Data Field	Definition
uint8 au8SSID[M2M_MAX_SSID_LEN]	Configuration parameters for the Wi-Fi AP.AP SSID
uint8 u8ListenChanel	Wi-Fi RF Channel which the AP will operate on
uint8 u8KeyIndx	Wep key Index start from 0 to 3
uint8 u8KeySz	Wep key Size `WEP_40_KEY_STRING_SIZE` or `WEP_104_KEY_STRING_SIZE`
uint8 au8WepKey[WEP_104_KEY_STRING_SIZE+1]	Wep key null terminated

Data Field	Definition
uint8 u8SecType	Security type: Open or WEP only in the current implementation
uint8 u8SsidHide	SSID Status "Hidden(1)/Visible(0)"
uint8 au8DHCPServerIP[4]	Ap IP server address

typedef struct tstrM2mClientState

PS Client State.

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

typedef struct tstrM2MConnInfo

M2M Provisioning Information obtained from the HTTP Provisioning server.

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

typedef struct tstrM2MDefaultConnResp

Response error of the m2m_default_connect.

See also:

M2M_DEFAULT_CONN_SCAN_MISMATCH

M2M_DEFAULT_CONN_EMPTY_LIST

Data Field	Definition
sint8 s8ErrorCode	Default connect error code. possible values are: M2M_DEFAULT_CONN_EMPTY_LIST M2M_DEFAULT_CONN_SCAN_MISMATCH

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

typedef struct tstrM2MDeviceNameConfig

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

typedef struct tstrM2MIPConfig

Static IP configuration.

Note: All member IP addresses are expressed in Network Byte Order (e.g. "192.168.10.1" will be expressed as 0x010AA8C0).

Data Field	Definition
uint32 u32StaticIP	The static IP assigned to the device.
uint32 u32Gateway	IP of the Default internet gateway.
uint32 u32DNS	IP for the DNS server.
uint32 u32SubnetMask	Subnet mask for the local area network.

typedef struct tstrM2mIpCtrlBuf

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

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

See also:

tpfAppEthCb

tstrEthInitParam

WARNING Make sure that bypass mode is defined before using **tstrM2mIpCtrlBuf**.

Data Field	Definition
uint16 u16DataSize	Size of the received data in bytes.

Data Field	Definition
uint16 u16RemainigDataSize	Size of the remaining data bytes to be delivered to host.

typedef struct tstrM2mIpRsvdPkt

Received Packet Size and Data Offset.

Data Field	Definition
uint16 u16PktSz	Size of the received packet in bytes.
uint16 u16PktOffset	Size of the remaining data bytes to be delivered to host.

typedef struct tstrM2mLsnInt

Listen interval.

It is the value of the Wi-Fi STA listen interval for power saving. It is given in units of Beacon period. Periodically after the listen interval fires, the WINC is wakeup and listen to the beacon and check for any buffered frames for it from the AP.

Data Field	Definition
uint16 u16LsnInt	Listen interval in Beacon period count.
uint8 __PAD16__[2]	Padding bytes for forcing 4-byte alignment.

typedef struct tstrM2MMulticastMac

M2M add/remove multicast mac address.

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

typedef struct tstrM2MP2PConnect

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

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

typedef struct tstrM2MProvisionInfo

M2M Provisioning Information obtained from the HTTP Provisioning server.

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

typedef struct tstrM2MProvisionModeConfig

M2M Provisioning Mode Configuration.

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

typedef struct tstrM2mPs

Power save Configuration.

See also:

tenuPowerSaveModes

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

typedef struct tstrM2mReqScanResult

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

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

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

typedef struct tstrM2MScan

Wi-Fi Scan Request.

See also:

tenuM2mScanCh

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

typedef struct tstrM2mScanDone

Wi-Fi Scan Result.

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

typedef struct tstrM2MScanOption

Wi-Fi Scan Request.

Data Field	Definition
uint8 u8NumOfSlot	The min number of slots is 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 increased in the scan results min time is 10ms and the max is 250ms
uint8 __PAD16__[2]	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2Mservercmd

PS Server Cmd.

Data Field	Definition
uint8 u8cmd	PS Server CMD
uint8 __PAD24__[3]	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2mServerInit

PS Server initialization.

Data Field	Definition
uint8 u8Channel	Server Listen channel
uint8 __PAD24__[3]	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2mSetMacAddress

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

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

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

typedef struct tstrM2mSlpReqTime

Manual power save request sleep time.

Data Field	Definition
uint32 u32SleepTime	Sleep time in ms

typedef struct tstrM2mWifiConnect

Wi-Fi Connect Request.

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

typedef struct tstrM2MWifiMonitorModeCtrl

Wi-Fi Monitor Mode Filter.

This structure sets the filtering criteria for WLAN packets when monitoring mode is enable. The received packets matching the filtering parameters, are passed directly to the application.

Data Field	Definition
uint8 u8Index	Index of the desired scan result
uint8 __PAD24__[3]	Padding bytes for forcing 4-byte alignment
uint8 u8ChannelID	The monitoring channel
uint8 u8FrameType	It must use values from tenuWifiFrameType
uint8 u8FrameSubtype	It must use values from tenuSubTypes
uint8 au8SrcMacAddress[6]	ZERO means DO NOT FILTER Source address
uint8 au8DstMacAddress[6]	ZERO means DO NOT FILTER Destination address

Data Field	Definition
uint8 au8BSSID[6]	ZERO means DO NOT FILTER BSSID
uint8 __PAD24__[3]	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2MWifiRxPacketInfo

Wi-Fi RX Frame Header.

The M2M application has the ability to allow Wi-Fi monitoring mode for receiving all Wi-Fi Raw frames matching a well-defined filtering criteria. When a target Wi-Fi packet is received, the header information are extracted and assigned in this structure.

Data Field	Definition
uint8 u8FrameType	it must use values from tenuWifiFrameType
uint8 u8FrameSubtype	It must use values from tenuSubTypes
uint8 u8ServiceClass	Service class from Wi-Fi header
uint8 u8Priority	Priority from Wi-Fi header
uint8 u8HeaderLength	Frame Header length
uint8 u8CipherType	Encryption type for the rx packet
uint8 au8SrcMacAddress[6]	ZERO means DO NOT FILTER Source address
uint8 au8DstMacAddress[6]	ZERO means DO NOT FILTER Destination address
uint8 au8BSSID[6]	ZERO means DO NOT FILTER BSSID
uint16 u16DataLength	Data payload length (Header excluded)
uint16 u16FrameLength	Total frame length (Header + Data)
uint32 u32DataRateKbps	Data Rate in Kbps
sint8 s8RSSI	RSSI
uint8 __PAD24__[3]	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

Data Field	Definition
uint8 u8ch	AP RF channel
uint8 au8BSSID[6]	BSSID of the AP
uint8 au8SSID[M2M_MAX_SSID_LEN]	AP SSID
uint8 _PAD8_	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2MWifiSecInfo

Authentication credentials to connect to a Wi-Fi network.

Data Field	Definition
tuniM2MWifiAuth uniAuth	Union holding all possible authentication parameters corresponding the current security types
uint8 u8SecType	Wi-Fi network security type. See tenuM2mSecType for supported security types
uint8 __PAD__[__PADDING__]	Padding bytes for forcing 4-byte alignment

typedef struct tstrM2mWifiStateChanged

Wi-Fi Connection State.

See also:

M2M_WIFI_DISCONNECTED

M2M_WIFI_CONNECTED

M2M_WIFI_REQ_CON_STATE_CHANGED

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

typedef struct tstrM2MWifiTxPacketInfo

Wi-Fi Tx Packet Info.

The M2M Application has the ability to compose a RAW Wi-Fi frames (under the application responsibility). When transmitting a Wi-Fi packet, the application must supply the firmware with this structure for sending the target frame.

Data Field	Definition
uint16 u16PacketSize	WLAN frame length
uint16 u16HeaderLength	WLAN frame header length

typedef struct tstrM2mWifiWepParams

WEP security key parameters.

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

typedef struct tstrM2MWPSConnect

WPS configuration parameters.

See also:

tenuWPSTrigger

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

typedef struct tstrM2MWPSInfo

WPS Result.

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

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

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

See also:

tenuM2mSecType

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

typedef struct tstrOtaControlSec

Control section structure is used to define the working image and the validity of the roll-back image and its offset, also both firmware versions is kept in that structure.

Data Field	Definition
uint32 u32OtaMagicValue	magic value used to ensure the structure is valid or not
uint32 u32OtaFormatVersion	control structure format version, the value will be incremented in case of structure changed or updated
uint32 u32OtaSequenceNumber	sequence number is used while update the control structure to keep track of how many times that section updated
uint32 u32OtaLastCheckTime	Last time OTA check for update
uint32 u32OtaCurrentworkingImagOffset	Current working offset in flash
uint32 u32OtaCurrentworkingImagFirmwareVer	current working image version ex 18.0.1
uint32 u32OtaRollbackImageOffset	Roll-back image offset in flash
uint32 u32OtaRollbackImageValidStatus	roll-back image valid status
uint32 u32OtaRollbackImagFirmwareVer	Roll-back image version (ex 18.0.3)
uint32 u32OtaControlSecCrc	CRC for the control structure to ensure validity

typedef struct tstrOtaInitHdr

OTA Image Header.

Data Field	Definition
uint32 u32OtaMagicVal	Magic value kept in the OTA image after the sha256 Digest buffer to define the Start of OTA Header
uint32 u32OtaPayloadSzie	The Total OTA image payload size, include the sha256 key size

typedef struct tstrOtaUpdateInfo

OTA Update Information.

See also:

tenuWPSTrigger

Data Field	Definition
uint32 u8NcfUpgradeVersion	NCF OTA Upgrade Version
uint32 u8NcfCurrentVersion	NCF OTA Current firmware version
uint32 u8NcdUpgradeVersion	NCD (host) upgraded version (if the u8NcdRe-quiredUpgrade == true)
uint8 u8NcdRequiredUpgrade	NCD Required upgrade to the above version
uint8 u8DownloadUrlOffset	Download URL offset in the received packet
uint8 u8DownloadUrlSize	Download URL size in the received packet
uint8 __PAD8__	Padding bytes for forcing 4-byte alignment

typedef struct tstrOtaUpdateStatusResp

OTA Update status response Information.

Data Field	Definition
uint8 u8OtaUpdateStatusType	Status type tenuOtaUpdateStatusType
uint8 u8OtaUpdateStatus	OTA_SUCCESS OTA_ERR_WORKING_IMAGE_LOAD_FAIL OTA_ERR_INVAILD_CONTROL_SEC M2M_ERR_OTA_SWITCH_FAIL M2M_ERR_OTA_START_UPDATE_FAIL M2M_ERR_OTA_ROLLBACK_FAIL M2M_ERR_OTA_INVAILD_FLASH_SIZE M2M_ERR_OTA_INVAILD_ARG
uint8 _PAD16_[2]	Padding bytes for forcing 4-byte alignment

F.1.3 Function

- **m2m_ota_init**
  - **NMI_API sint8 m2m_ota_init (tpfOtaUpdateCb pfOtaUpdateCb, tpfOtaNotifCb pfOtaNotifCb)**

Synchronous initialization function for the OTA layer by registering the update callback.

The notification callback is not supported at the current version. Calling this API is a MUST for all the OTA API's.

Parameters:

in	*pfOtaUpdateCb*	OTA Update callback function
in	*pfOtaNotifCb*	OTA notify callback function

Returns:

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

- **m2m_ota_notif_set_url**
  - **NMI_API sint8 m2m_ota_notif_set_url (uint8 * u8Url)**

Set the OTA notification server URL, the functions need to be called before any check for update.

Parameters:

in	*u8Url*	OTA notification server URL Calling m2m_ota_init is required to be called before this function. Notification Server is not supported in the current version (function is not implemented).

Warning:

Notification Server is not supported in the current version (function is not implemented).

See also:

m2m_ota_init

Returns:

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

- **m2m_ota_notif_check_for_update**
  - **NMI_API sint8 m2m_ota_notif_check_for_update (void)**

Synchronous function to check for the OTA update using the Notification Server.

Function is not implemented (not supported at the current version).

Warning:

Function is not implemented (not supported at the current version).

See also:

m2m_ota_init

m2m_ota_notif_set_url

Returns:

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

- **m2m_ota_notif_sched**
  - **NMI_API sint8 m2m_ota_notif_sched (uint32 u32Period)**

Schedule OTA notification server check for update request after specific number of days.

Parameters:

in	*u32Period*	Period in days

See also:

m2m_ota_init

m2m_ota_notif_check_for_update

m2m_ota_notif_set_url

Returns:

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

- **m2m_ota_start_update**

– **NMI_API sint8 m2m_ota_start_update (uint8 * u8DownloadUrl)**

Request OTA start update using the downloaded URL, the OTA module will download the OTA image and ensure integrity of the image, and update the validity of the image in control structure. Switching to that image requires calling m2m_ota_switch_firmware API. As a prerequisite **m2m_ota_init** should be before using m2m_ota_start().

Parameters:

in	*u8DownloadUrl*	The download firmware URL, you get it from device info according to the application server

Warning:

Calling this API does not guarantee OTA WINC image update, It depends on the connection with the download server and the validity of the image If the API response fails this may invalidate the roll-back image if it was previously valid, since the WINC does not have any internal memory except the flash roll-back image location to validate the downloaded image from.

See also:

m2m_ota_init

tpfOtaUpdateCb

Returns:

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

Example

The example shows an example of how the OTA image update is carried out.

```
static void OtaUpdateCb(uint8 u8OtaUpdateStatusType ,uint8 u8OtaUpdateStatus)
{
    if(u8OtaUpdateStatusType == DL_STATUS) {
        if(u8OtaUpdateStatus == OTA_STATUS_SUCSESS) {
            //switch to the upgraded firmware
            m2m_ota_switch_firmware();
        }
    }
    else if(u8OtaUpdateStatusType == SW_STATUS) {
        if(u8OtaUpdateStatus == OTA_STATUS_SUCSESS) {
            M2M_INFO("Now OTA successfully done");
            //start the host SW upgrade then system reset is required (Reinitialize the driver)
        }
    }
}
void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    case M2M_WIFI_REQ_DHCP_CONF:
    {
        //after successfully connection, start the over air upgrade
        m2m_ota_start_update(OTA_URL);
    }
    break;
    default:
    break;
}
int main (void)
{
    tstrWifiInitParam param;
    tstr1xAuthCredentials gstrCred1x    = AUTH_CREDENTIALS;
    nm_bsp_init();

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

    //intilize the WINC Driver
    ret = m2m_wifi_init(&param);
    if (M2M_SUCCESS != ret)
    {
        M2M_ERR("Driver Init Failed <%d>\n",ret);
        while(1);
    }
    //intilize the ota module
    m2m_ota_init(OtaUpdateCb,NULL);
    //connect to AP that provide connection to the OTA server
    m2m_wifi_default_connect();

    while(1)
    {

        //Handle the app state machine plus the WINC event handler
        while(m2m_wifi_handle_events(NULL) != M2M_SUCCESS) {

        }

    }
}
```

- **m2m_ota_rollback**
  - **NMI_API sint8 m2m_ota_rollback (void)**

Request OTA Roll-back to the old (other) WINC image, the WINC firmware will check the validity of the Roll-back image and switch to it if it is valid.

If the API response is success, system restart is required (re-initialize the driver with hardware rest) update the host driver version may be required if it is does match the minimum version supported by the WINC firmware.

See also:

m2m_ota_init

m2m_ota_start_update

Returns:

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

- **m2m_ota_switch_firmware**
  - **NMI_API sint8 m2m_ota_switch_firmware (void)**

Switch to the upgraded firmware, that API will update the control structure working image to the upgraded image take effect will be on the next system restart.

Warning:

It is important to note that if the API succeeds, system restart is required (re-initializing the driver with hardware reset) updating the host driver version may be required if it does not match the minimum driver version supported by the WINC's firmware.

See also:

m2m_ota_init

m2m_ota_start_update

Returns:

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

- **m2m_wifi_download_mode**
  - **NMI_API void m2m_wifi_download_mode (void)**

Synchronous download mode function that prepares the WINC firmware to enter the download mode. The WINC firmware is prepared for download, through initializations for the WINC driver including bus initializations and interrupt enabling, it also halts the chip, to allow for the firmware downloads. When in the download mode the WINC is ready for firmware download or certificate download. Firmware can be downloaded through a number of interfaces, UART, I^{2}C, and SPI.

Returns:

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

- **m2m_wifi_init**
  - **NMI_API sint8 m2m_wifi_init (tstrWifiInitParam *pWifiInitParam)**

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

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

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

M2M_WIFI_RESP_CON_STATE_CHANGED

M2M_WIFI_RESP_CONN_INFO

M2M_WIFI_REQ_DHCP_CONF

M2M_WIFI_REQ_WPS

M2M_WIFI_RESP_IP_CONFLICT

M2M_WIFI_RESP_SCAN_DONE

M2M_WIFI_RESP_SCAN_RESULT

M2M_WIFI_RESP_CURRENT_RSSI

M2M_WIFI_RESP_CLIENT_INFO

M2M_WIFI_RESP_PROVISION_INFO

M2M_WIFI_RESP_DEFAULT_CONNECT

Example:

In case bypass mode is defined:

M2M_WIFI_RESP_ETHERNET_RX_PACKET

In case Monitoring mode is used:

M2M_WIFI_RESP_WIFI_RX_PACKET

Any application using the WINC 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 WIFI layer call back function, monitoring mode call back and **tstrEthInitParam** structure containing bypass mode parameters.

Precondition:

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

Warning:

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

See also:

m2m_wifi_deinit

tenuM2mStaCmd

Returns:

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

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

Synchronous de-initialization function to the WINC1500 driver. Deinitializes the host interface and frees any resources used by the M2M_WIFI layer. This function must be called in the application closing phase, to ensure that all resources have been correctly released. No arguments are expected to be passed in.

Parameters:

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

Returns:

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

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

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

Precondition:

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

Warning:

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

Returns:

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

- **m2m_wifi_default_connect**
  - **NMI_API sint8 m2m_wifi_default_connect (void)**

Asynchronous Wi-Fi connection function. An application calling this function will cause the firmware to correspondingly connect to the last successfully connected AP from the cached connections. A failure to connect will result in a response of **M2M_WIFI_RESP_DEFAULT_CONNECT** indicating the connection error as defined in the structure **tstrM2MDefaultConnResp**.

Possible errors are:

The connection list is empty **M2M_DEFAULT_CONN_EMPTY_LIST** or a mismatch for the saved AP name **M2M_DEFAULT_CONN_SCAN_MISMATCH**.

The only difference between this function and **m2m_wifi_connect**, is the connection parameters. Connection using this function is expected to connect to cached connection parameters.

Precondition:

Prior to connecting, the WINC driver should have been successfully initialized by calling the **m2m_wifi_init** function.

See also:

m2m_wifi_connect

Warning:

This function must be called in station mode only. It's important to note that successful completion of a call to **m2m_wifi_default_connect()** does not guarantee success of the WIFI connection, and a negative return value indicates only locally-detected errors.

Returns:

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

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

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

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

Parameters:

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

Precondition:

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

Warning:

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

See also:

tuniM2MWifiAuth

tstr1xAuthCredentials

tstrM2mWifiWepParams

Returns:

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

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

Precondition:

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

Warning:

This function must be called in station mode only.

See also:

m2m_wifi_connect

m2m_wifi_default_connect

Returns:

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

- **m2m_wifi_start_provision_mode**
  - **NMI_API sint8 m2m_wifi_start_provision_mode (tstrM2MAPConfig *pstrAPConfig, char *pcHttpServerDomainName, uint8 bEnableHttpRedirect)**

Asynchronous Wi-Fi provisioning function, which starts the WINC HTTP PROVISIONING mode. The function triggers the WINC to activate the Wi-Fi AP (HOTSPOT) mode with the passed configuration parameters and then starts the HTTP Provision WEB Server. The provisioning status is returned in an event **M2M_WIFI_RESP_PROVISION_INFO**.

Parameters:

in	*pstrAPConfig*	AP configuration parameters as defined in **tstrM2MAPConfig** configuration structure. A NULL value passed in, will result in a negative error M2M_ERR_FAIL.
in	*pcHttpServerDomainName*	Domain name of the HTTP Provision server which others will use to load the provisioning Home page. For example "wincconf.net".
in	*bEnableHttpRedirect*	A flag to enable/disable the HTTP redirect feature. Possible values are: 0:Disable, 1:Enable

Precondition:

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

Warning:

DO NOT use ".local" in the pcHttpServerDomainName.

See also:

tpfAppWifiCb

m2m_wifi_init

Returns:

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

Example

The example demonstrates a code snippet for how provisioning is triggered and the response event received accordingly.

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


void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    switch(u8WiFiEvent)
    {
    case M2M_WIFI_RESP_PROVISION_INFO:
        {
            tstrM2MProvisionInfo    *pstrProvInfo = (tstrM2MProvisionInfo*)pvMsg;
            if(pstrProvInfo->u8Status == M2M_SUCCESS)
            {
                m2m_wifi_connect((char*)pstrProvInfo->au8SSID,
(uint8)strlen(pstrProvInfo-> au8SSID), pstrProvInfo->u8SecType,
                        pstrProvInfo->au8Password, M2M_WIFI_CH_ALL);

                printf("PROV SSID : %s\n",pstrProvInfo->au8SSID);
                printf("PROV PSK  : %s\n",pstrProvInfo->au8Password);
            }
            else
            {
                printf("(ERR) Provisioning Failed\n");
            }
        }
        break;

        default:
        break;
    }
}

int main()
{
    tstrWifiInitParam   param;

    param.pfAppWifiCb   = wifi_event_cb;
    if(!m2m_wifi_init(&param))
    {
        tstrM2MAPConfig     apConfig;
        uint8               bEnableRedirect = 1;

        strcpy(apConfig.au8SSID, "WINC_SSID");
        apConfig.u8ListenChannel    = 1;
        apConfig.u8SecType          = M2M_WIFI_SEC_OPEN;
        apConfig.u8SsidHide         = 0;

        // IP Address
        apConfig.au8DHCPServerIP[0] = 192;
        apConfig.au8DHCPServerIP[1] = 168;
        apConfig.au8DHCPServerIP[2] = 1;
        apConfig.au8DHCPServerIP[0] = 1;

        m2m_wifi_start_provision_mode(&apConfig, "atmelwincconf.com", bEnableRedirect);

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

```

- **m2m_wifi_stop_provision_mode**
  - **NMI_API sint8 m2m_wifi_stop_provision_mode (void)**

Synchronous provision termination function which stops the provision mode if it is active.

Precondition:

An active provisioning session must be active before it is terminated through this function.

See also:

m2m_wifi_start_provision_mode

Returns:

The function returns ZERO for success and a negative value otherwise.

- **m2m_wifi_get_connection_info**
  - **NMI_API sint8 m2m_wifi_get_connection_info (void)**

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

Precondition:

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

Warning:

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

See also:

tpfAppWifiCb

m2m_wifi_init

M2M_WIFI_RESP_CONN_INFO

tstrM2MConnInfo

Returns:

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

Example:

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

```
#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;

    case M2M_WIFI_REQ_DHCP_CONF:
        {
            // Get the current AP information.
            m2m_wifi_get_connection_info();
        }
        break;
    default:
        break;
    }
}

int main()
{
    tstrWifiInitParam   param;

    param.pfAppWifiCb   = wifi_event_cb;
    if(!m2m_wifi_init(&param))
    {
        // connect to the default AP
        m2m_wifi_default_connect();

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

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

Synchronous MAC address assigning to the NMC1500. It is used for non-production SW. Assign MAC address to the WINC device.

Parameters:

in	*au8MacAddress*	MAC Address to be provisioned to the WINC

Returns:

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

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

Asynchronous WPS triggering function. This function is called for the WINC 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 startup. Registering the callback is done through passing it to the **m2m_wifi_init**.
- The event **M2M_WIFI_REQ_WPS** must be handled in the callback to receive the WPS status
- The WINC 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_events(NULL);
        }
    }
}
```

- **m2m_wifi_wps_disable**
  - **NMI_API sint8 m2m_wifi_wps_disable (void)**

Disable the NMC1500 WPS operation.

Returns:

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

- **m2m_wifi_p2p**
  - **NMI_API sint8 m2m_wifi_p2p (uint8 u8Channel)**

Asynchronous Wi-Fi direct (P2P) enabling mode function. The WINC supports P2P in device listening mode ONLY (intent is ZERO). The WINC P2P implementation does not support P2P GO (Group Owner) mode. Active P2P devices (e.g. phones) could find the WINC in the search list. When a device is connected to WINC, a Wi-Fi notification event **M2M_WIFI_RESP_CON_STATE_CHANGED** is triggered. After a short while, the DHCP IP Address is obtained and an event **M2M_WIFI_REQ_DHCP_CONF** is triggered. Refer to the code examples for a more illustrative example.

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.

Precondition:

- A Wi-Fi notification callback of type tpfAppWifiCb MUST be implemented and registered at initialization. Registering the callback is done through passing it to the **m2m_wifi_init**.
- The events **M2M_WIFI_RESP_CON_STATE_CHANGED** and **M2M_WIFI_REQ_DHCP_CONF** 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

M2M_WIFI_REQ_DHCP_CONF

tstrM2mWifiStateChanged

Returns:

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

Example:

The code snippet shown an example of how the p2p mode operates.

```
#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("Wi-Fi State :: %s :: ErrCode %d\n", pstrWifiState->u8CurrState?
"CONNECTED":"DISCONNECTED",pstrWifiState->u8ErrCode);

            // Do something
        }
        break;

    case M2M_WIFI_REQ_DHCP_CONF:
        {
            uint8   *pu8IPAddress = (uint8*)pvMsg;

            printf("P2P IP Address \"%u.%u.%u.%u\"\n",pu8IPAddress[0],pu8IPAd-
dress[1],pu8IPAddress[2],pu8IPAddress[3]);
        }
        break;

    default:
        break;
    }
}

int main()
{
    tstrWifiInitParam   param;

    param.pfAppWifiCb   = wifi_event_cb;
    if(!m2m_wifi_init(&param))
    {
        // Trigger P2P
        m2m_wifi_p2p(1);

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

```

- **m2m_wifi_p2p_disconnect**
  - **NMI_API sint8 m2m_wifi_p2p_disconnect (void)**

Disable the NMC1500 device Wi-Fi direct mode (P2P).

Precondition:

The p2p mode must have be enabled and active before a disconnect can be called.

See also:

m2m_wifi_p2p

Returns:

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

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

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

Only 1 STA could be associated at a time.

Open and WEP are the only supported security types.

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 event **M2M_WIFI_REQ_DHCP_CONF** must be handled in the callback
- The **m2m_wifi_handle_events** MUST be called to receive the responses in the callback

See also:

tpfAppWifiCb

tenuM2mSecType

m2m_wifi_init

M2M_WIFI_REQ_DHCP_CONF

tstrM2mWifiStateChanged

tstrM2MAPConfig

Returns:

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

Example:

The code snippet demonstrates how the AP mode is enabled after the driver is initialized in the application's main function and the handling of the event **M2M_WIFI_REQ_DHCP_CONF**, to indicate successful connection.

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

void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    switch(u8WiFiEvent)
    {
    case M2M_WIFI_REQ_DHCP_CONF:
        {
            uint8   *pu8IPAddress = (uint8*)pvMsg;

            printf("Associated STA has IP Address \"%u.%u.%u.%u\"\n",pu8IPAd-
dress[0],pu8IPAddress[1],pu8IPAddress[2],pu8IPAddress[3]);
        }
        break;

    default:
        break;
    }
}

int main()
{
    tstrWifiInitParam   param;

    param.pfAppWifiCb   = wifi_event_cb;
    if(!m2m_wifi_init(&param))
    {
        tstrM2MAPConfig     apConfig;

        strcpy(apConfig.au8SSID, "WINC_SSID");
        apConfig.u8ListenChannel    = 1;
        apConfig.u8SecType          = M2M_WIFI_SEC_OPEN;
        apConfig.u8SsidHide         = 0;

        // IP Address
        apConfig.au8DHCPServerIP[0] = 192;
        apConfig.au8DHCPServerIP[1] = 168;
        apConfig.au8DHCPServerIP[2] = 1;
        apConfig.au8DHCPServerIP[0] = 1;

        // Trigger AP
        m2m_wifi_enable_ap(&apConfig);

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

```

- **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_set_static_ip**
  - **NMI_API sint8 m2m_wifi_set_static_ip (tstrM2MIPConfig *pstrStaticIPConf)**

Synchronous static IP Address configuration function. To be removed in up-coming releases.

Parameters:

in	*pstrStaticIPConf*	Pointer to a structure holding the static IP Configurations (IP, Gateway, subnet mask, and DNS address)

Warning:

This function should not be used. DHCP configuration is requested automatically after successful Wi-Fi connection is established. It is a legacy API and will be removed from the interface.

See also:

tstrM2MIPConfig

Returns:

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

- **m2m_wifi_request_dhcp_client**
  - **NMI_API sint8 m2m_wifi_request_dhcp_client (void)**

Starts the DHCP client operation (DHCP requested by the firmware automatically in STA/AP/P2P mode).

Warning:

This function should not be used. DHCP configuration is requested automatically after successful Wi-Fi connection is established.

Returns:

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

- **m2m_wifi_request_dhcp_server**
  - **NMI_API sint8 m2m_wifi_request_dhcp_server (uint8 *addr)**

Legacy API should be removed from the interface in up-coming releases. (DHCP requested by the firmware automatically in STA/AP/P2P mode).

Warning:

This function should not be used. DHCP server is started automatically when enabling the AP mode. It is a legacy API and will be removed.

Returns:

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

- **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.

- **m2m_wifi_set_scan_region**
  - **NMI_API sint8 m2m_wifi_set_scan_region (uint8 ScanRegion)**

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

Parameters:

in	*ScanRegion;*	ASIA = 14 NORTH_AMERICA = 11

See also:

tenuM2mScanCh

m2m_wifi_request_scan

Returns:

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

- **m2m_wifi_request_scan**
  - **NMI_API sint8 m2m_wifi_request_scan (uint8 ch)**

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

Parameters:

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

Warning:

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

Precondition:

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

See also:

M2M_WIFI_RESP_SCAN_DONE

M2M_WIFI_RESP_SCAN_RESULT

tpfAppWifiCb

tstrM2mWifiscanResult

tenuM2mScanCh

m2m_wifi_init

m2m_wifi_handle_events

m2m_wifi_req_scan_result

Returns:

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

Example:

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

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

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

```

- **m2m_wifi_get_num_ap_found**
  - **NMI_API uint8 m2m_wifi_get_num_ap_found (void)**

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

Precondition:

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

Warning:

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

See also:

m2m_wifi_request_scan

M2M_WIFI_RESP_SCAN_DONE

M2M_WIFI_RESP_SCAN_RESULT

Returns:

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

Example:

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

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

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

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

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

Parameters:

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

See also:

tstrM2mWifiscanResult

m2m_wifi_get_num_ap_found

m2m_wifi_request_scan

Precondition:

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

Warning:

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

Returns:

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

Example:

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

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

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

```

- **m2m_wifi_req_curr_rssi**
  - NMI_API sint8 m2m_wifi_req_curr_rssi (void)

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

Precondition:

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

Returns:

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

Example:

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

```
#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_CURRENT_RSSI:
        {
            sint8   *rssi = (sint8*)pvMsg;
            M2M_INFO("ch rssi %d\n",*rssi);
        }
        break;
    default:
        break;
    }
}

int main()
{
    tstrWifiInitParam   param;

    param.pfAppWifiCb   = wifi_event_cb;
    if(!m2m_wifi_init(&param))
    {
        // Scan all channels
        m2m_wifi_req_curr_rssi();

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

```

- **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:

in	*pu8MacAddr*	Output MAC address buffer of 6 bytes size. Valid only if *pu8Valid=1.
in	*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/socket function.

See also:

m2m_wifi_get_mac_address

Returns:

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

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

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

Parameters:

in	*pu8MacAddr*	Output MAC address buffer of 6 bytes size.

Precondition:

m2m_wifi_init required to be called before any WIFI/socket function.

See also:

m2m_wifi_get_otp_mac_address

Returns:

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

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

Synchronous power-save mode setting function for the NMC1500.

Parameters:

in	*PsTyp*	Desired power saving mode. Supported types are defined in **tenuPowerSaveModes**.
in	*BcastEn*	Broadcast reception enable flag. If it is 1, the WINC1500 must be awake each DTIM beacon for receiving broadcast traffic. If it is 0, the WINC1500 will not wakeup at the DTIM beacon, but its wakeup depends only on the configured Listen Interval.

Warning:

The function called once after initialization.

See also:

tenuPowerSaveModes

m2m_wifi_get_sleep_mode

Returns:

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

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

Synchronous power save request function, which requests from the NMC1500 device to sleep in the mode previously set for a specific time. This function should be used in the M2M_PS_MANUAL Power save mode (only).

Parameters:

in	*u32SlpReqTime*	Request Sleep in ms

Warning:

The function should be called in M2M_PS_MANUAL power save only.

See also:

tenuPowerSaveModes

m2m_wifi_set_sleep_mode

Returns:

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

- **m2m_wifi_get_sleep_mode**
  - **NMI_API uint8 m2m_wifi_get_sleep_mode (void)**

Synchronous power save mode retrieval function.

See also:

tenuPowerSaveModes

m2m_wifi_set_sleep_mode

Returns:

The current operating power saving mode.

- **m2m_wifi_req_client_ctrl**
  - **NMI_API sint8 m2m_wifi_req_client_ctrl (uint8 cmd)**

Asynchronous command sending function to the PS Client (An NMC1500 board running the ps_firmware) if the PS client send any commands it will be received through the **M2M_WIFI_RESP_CLIENT_INFO** event.

Parameters:

in	*cmd*	Control command sent from PS Server to PS Client (command values defined by the application)

Precondition:

m2m_wifi_req_server_init should be called first.

Warning:

This mode is not supported in the current release.

See also:

m2m_wifi_req_server_init

M2M_WIFI_RESP_CLIENT_INFO

Returns:

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

- **m2m_wifi_req_server_init**
  - **NMI_API sint8 m2m_wifi_req_server_init (uint8 ch)**

Synchronous function to initialize the PS Server. The WINC1500 supports non secure communication with another WINC1500, (SERVER/CLIENT) through one byte command (probe request and probe response) without any connection setup. The server mode can't be used with any other modes (STA/P2P/AP).

Parameters:

in	*ch*	Server listening channel

Warning:

This mode is not supported in the current release.

See also:

m2m_wifi_req_client_ctrl

Returns:

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

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

Set the WINC1500 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.

- **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.

- **m2m_wifi_enable_monitoring_mode**
  - **NMI_API sint8 m2m_wifi_enable_monitoring_mode (tstrM2MWifiMonitorModeCtrl *pstrMtrCtrl, uint8 *pu8PayloadBuffer, uint16 u16BufferSize, uint16 u16DataOffset)**

Asynchronous Wi-Fi monitoring enable mode (Promiscuous mode) function. This function enables the monitoring mode, which starts transmission of the packets based on the filter information passed in as a parameter. All packets that meet the filtering criteria are passed to the application layer, to be handled by the assigned monitoring callback function. The monitoring callback function must be implemented before starting the monitoring mode, in-order to handle the packets received. Registering of the implemented callback function is through the callback pointer tpfAppMonCb in the tstrWifiInitParam structure. Passed to m2m_wifi_init function at initialization.

Parameters:

in	*pstrMtrCtrl*	Pointer to tstrM2MWifiMonitorModeCtrl structure holding the Filtering parameters
in	*pu8PayloadBuffer*	Pointer to a Buffer allocated by the application. The buffer SHALL hold the Data field of the WIFI RX Packet (Or a part from it). If it is set to NULL, the WIFI data payload will be discarded by the monitoring driver.
in	*u16BufferSize*	The total size of the pu8PayloadBuffer in bytes
in	*u16DataOffset*	Starting offset in the DATA FIELD of the received WIFI packet. The application may be interested in reading specific information from the received packet. It must assign the offset to the starting position of it relative to the DATA payload start. Example, if the SSID is needed to be read from a PROBE REQ packet, the u16Offset MUST be set to 0.

Warning:

This mode available as sniffer ONLY, you cannot be connected in any modes (Station, Access Point, or P2P).

See also:

tstrM2MWifiMonitorModeCtrl

tstrM2MWifiRxPacketInfo

tstrWifiInitParam

m2m_wifi_disable_monitoring_mode

Returns:

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

Example

The example demonstrates the main function where-by the monitoring enable function is called after the initialization of the driver and the packets are handled in the callback function.

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

//Declare receive buffer
uint8 gmgmt[1600];

//Callback functions
void wifi_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    ;
}
void wifi_monitoring_cb(tstrM2MWifiRxPacketInfo *pstrWifiRxPacket, uint8 *pu8Payload, uint16 u16PayloadSize)
{
    if((NULL != pstrWifiRxPacket) && (0 != u16PayloadSize)) {
        if(MANAGEMENT == pstrWifiRxPacket->u8FrameType) {
            M2M_INFO("***# MGMT PACKET #***\n");
        } else if(DATA_BASICTYPE == pstrWifiRxPacket->u8FrameType) {
            M2M_INFO("***# DATA PACKET #***\n");
        } else if(CONTROL == pstrWifiRxPacket->u8FrameType) {
            M2M_INFO("***# CONTROL PACKET #***\n");
        }
    }
}

int main()
{
    //Register wifi_monitoring_cb
    tstrWifiInitParam param;
    param.pfAppWifiCb = wifi_cb;
    param.pfAppMonCb  = wifi_monitoring_cb;

    nm_bsp_init();

    if(!m2m_wifi_init(&param)) {
        //Enable Monitor Mode with filter to receive all data frames on channel 1
        tstrM2MWifiMonitorModeCtrl  strMonitorCtrl = {0};
        strMonitorCtrl.u8ChannelID      = 1;
        strMonitorCtrl.u8FrameType      = DATA_BASICTYPE;
        strMonitorCtrl.u8FrameSubtype   = M2M_WIFI_FRAME_SUB_TYPE_ANY; //Receive any sub-type of data frame
        m2m_wifi_enable_monitoring_mode(&strMonitorCtrl, gmgmt, sizeof(gmgmt), 0);

        while(1) {
            m2m_wifi_handle_events(NULL);
        }
    }
    return 0;

```

- **m2m_wifi_disable_monitoring_mode**
  - **NMI_API sint8 m2m_wifi_disable_monitoring_mode (void)**

Synchronous function to disable Wi-Fi monitoring mode (Promiscuous mode). Expected to be called, if the enable monitoring mode is set, but if it was called without enabling no negative impact will reside.

See also:

m2m_wifi_enable_monitoring_mode

Returns:

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

- **m2m_wifi_send_wlan_pkt**
  - **NMI_API sint8 m2m_wifi_send_wlan_pkt (uint8 *pu8WlanPacket, uint16 u16WlanHeaderLength, uint16 u16WlanPktSize)**

Synchronous function to transmit a WIFI RAW packet while the implementation of this packet is left to the application developer.

Parameters:

in	*pu8WlanPacket*	Pointer to a buffer holding the whole WIFI frame
in	*u16WlanHeaderLength*	The size of the WIFI packet header ONLY
in	*u16WlanPktSize*	The size of the whole bytes in packet

Precondition:

Enable monitoring mode first using **m2m_wifi_enable_monitoring_mode**.

Note:

Packets are user's responsibility.

Warning:

This function available in monitoring mode ONLY.

See also:

m2m_wifi_enable_monitoring_mode

m2m_wifi_disable_monitoring_mode

Returns:

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

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

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

Parameters:

in	*pu8Packet*	Pointer to a buffer holding the whole Ethernet frame
in	*u16PacketSize*	The size of the whole bytes in packet

Note:

Packets are the user's responsibility.

Warning:

This function available in Bypass mode ONLY. Make sure that firmware version built with macro ETH_MODE.

Returns:

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

- **m2m_wifi_enable_sntp**
  - **NMI_API sint8 m2m_wifi_enable_sntp (uint8 bEnable)**

Synchronous function to Enable/Disable the native SNTP client in the m2m firmware. The SNTP is enabled by default at start-up. The SNTP client at firmware is used to sync the system clock to the UTC time from a well-known time servers (e.g. "time-c.nist.gov"). The SNTP client uses a default update cycle of 1 day. The UTC is important for checking the expiration date of X509 certificates used while establishing TLS (Transport Layer Security) connections. It is highly recommended to use it if there is no other means to get the UTC time. If there is a RTC on the host MCU, the SNTP could be disabled and the host should set the system time to the firmware using the m2m_wifi_set_system_time function.

Parameters:

in	*bEnable*	Enabling/Disabling flag '0': disable SNTP '1': enable SNTP

See also:

m2m_wifi_set_sytem_time

Returns:

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

- **m2m_wifi_set_sytem_time**
  - **NMI_API sint8 m2m_wifi_set_sytem_time (uint32 u32UTCSeconds)**

Synchronous function for setting the system time in time/date format **(uint32)**.

The **tstrSystemTime** structure can be used as a reference to the time values that should be set and pass its value as **uint32**.

Parameters:

in	*u32RTCSeconds*	UTC value in seconds

Note:

If there is an RTC on the host MCU, the SNTP could be disabled and the host should set the system time to the firmware using the API **m2m_wifi_set_sytem_time**.

See also:

m2m_wifi_enable_sntp

tstrSystemTime

Returns:

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

- **m2m_wifi_set_cust_InfoElement**
  - **NMI_API sint8 m2m_wifi_set_cust_InfoElement (uint8 *pau8M2mCustInfoElement)**

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

According to the information element layout shown below, if it is required to set new data for the information elements, pass in the buffer with the information according to the sizes and ordering defined bellow. 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 byte.

- Used in Access Point Mode

See also:

m2m_wifi_enable_sntp

tstrSystemTime

Returns:

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

Example

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

```
char elementData[21];
static char state = 0; // To Add, Append, and Delete
if(0 == state) { //Add 3 IEs
    state = 1;
    //Total Number of Bytes
    elementData[0]=12;
    //First IE
    elementData[1]=200; elementData[2]=1; elementData[3]='A';
    //Second IE
    elementData[4]=201; elementData[5]=2; elementData[6]='B'; elementData[7]='C';
    //Third IE
    elementData[8]=202; elementData[9]=3; elementData[10]='D'; elementData[11]=0; elementData[12]='F';
} else if(1 == state) {
    //Append 2 IEs to others, Notice that we keep old data in array starting with\n
    //element 13 and total number of bytes increased to 20
    state = 2;
    //Total Number of Bytes
    elementData[0]=20;
    //Fourth IE
    elementData[13]=203; elementData[14]=1; elementData[15]='G';
    //Fifth IE
    elementData[16]=204; elementData[17]=3; elementData[18]='X'; elementData[19]=5; elementData[20]='Z';
} else if(2 == state) {  //Delete All IEs
    state = 0;
    //Total Number of Bytes
    elementData[0]=0;
}
m2m_wifi_set_cust_InfoElement(elementData);
```

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

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

Parameters:

in	*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.

Warning:

This function is available in bypass mode ONLY. Make sure that firmware version built with the macro ETH_MODE.

Returns:

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

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

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

Parameters:

in	pvBuffer	Pointer to Buffer to receive data. NULL pointer causes a negative error M2M_ERR_FAIL.
in	u16BufferLen	Length of data to be received. Maximum length of data should not exceed the size defined by TCP/IP defined as **SOCKET_BUFFER_MAX_LENGTH**.

Warning:

This function is available in the bypass mode ONLY. Make sure that firmware version is built with macro ETH_MODE.

Returns:

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

F.2 BSP

This module contains NMC1500 BSP APIs declarations.

F.2.1 Defines

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

typedef void(* tpfNmBspIsr) (void)

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

F.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

F.2.3 Function

- **nm_bsp_init**
  - **sint8 nm_bsp_init (void)**

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

Note:

Implementation of this function is host dependent.

Warning:

Missing use will lead to failure in driver initialization.

Returns:

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

- **nm_bsp_deinit**

sint8 nm_bsp_deinit (void)

De-initialization for BSP (Board Support Package).

Precondition:

Initialize **nm_bsp_init** first.

Note:

Implementation of this function is host dependent.

Warning:

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

See also:

nm_bsp_init

Returns:

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

- **nm_bsp_reset**

void nm_bsp_reset (void)

Resetting NMC1500 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 WINC 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 WINC 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.

F.3 Socket

BSD compatible socket interface between the host layer and the network protocol stacks in the firmware. These functions are used by the host application to send or receive packets and to do other socket operations.

F.3.1 Defines

The following list of macros are used to define constants used throughout the socket layer.

Defines	Definition	Value
#define HOSTNAME_MAX_SIZE	Maximum allowed size for a host domain name passed to the function **gethostbyname**. command value. Used with the setsocketopt function.	**64**
#define SOCKET_BUFFER_MAX_LENGTH	Maximum allowed size for a socket data buffer. Used with **send** socket function to ensure that the buffer sent is within the allowed range.	**1400**
#define AF_INET	The AF_INET is the address family used for IPv4. An IPv4 transport address is specified with the **sockaddr_in** structure. (It is the only supported type for the current implementation.)	**2**
#define SOCK_STREAM	One of the IPv4 supported socket types for reliable connection-oriented stream connection. Passed to the socket function for the **socket** creation operation.	**1**
#define SOCK_DGRAM	One of the IPv4 supported socket types for unreliable connectionless datagram connection. Passed to the socket function for the **socket** creation operation.	**2**
#define SOCKET_FLAGS_SSL	This flag shall be passed to the socket API for SSL session.	**0x01**
#define TCP_SOCK_MAX	Maximum number of simultaneous TCP sockets.	**7**
#define UDP_SOCK_MAX	Maximum number of simultaneous UDP sockets.	**4**
#define MAX_SOCKET	Maximum number of Sockets.	**(TCP_SOCK_MAX + UDP_SOCK_MAX)**
#define SOL_SOCKET	Socket option. Used with the setsocketopt function.	**1**
#define SO_SET_UDP_SEND_CALLBACK	Socket option used by the application to enable/disable the use of UDP send callbacks. Used with the setsocketopt function.	**0x00**
#define IP_ADD_MEMBERSHIP	Set Socket Option Add Membership command value. Used with the setsocketopt function.	**0x01**
#define IP_DROP_MEMBERSHIP	Set Socket Option Drop Membership.	**0x02**

F.3.2 Error Codes

The following list of macros are used to define the possible error codes returned as a result of a call to a socket function. Errors are listed in numerical order with the error macro name.

Defines	Definition	Value
#define SOCK_ERR_NO_ERROR	Successful socket operation.	**0**
#define SOCK_ERR_INVALID_ADDRESS	Socket address is invalid. The socket operation cannot be completed successfully without specifying a specific address. For example: **bind** is called without specifying a port number.	**-1**

Defines	Definition	Value
#define SOCK_ERR_ADDR_AL-READY_IN_USE	Socket operation cannot bind on the given address. With socket operations, only one IP address per socket is permitted. Any attempt for a new socket to bind with an IP address already bound to an-other open socket, will return the following error code. States that **bind** operation failed.	**-2**
#define SOCK_ERR_MAX_TCP_SOCK	Exceeded the maximum number of TCP sockets. A maximum number of TCP sockets opened simulta-neously is defined through **TCP_SOCK_MAX**. It is not permitted to exceed that number at **socket** cre-ation. Identifies that socket operation failed.	**-3**
#define SOCK_ERR_MAX_UDP_SOCK	Exceeded the maximum number of UDP sockets. A maximum number of UDP sockets opened simulta-neously is defined through **UDP_SOCK_MAX**. It is not permitted to exceed that number at socket cre-ation. Identifies that **socket** operation failed.	**-4**
#define SOCK_ERR_INVA-LID_ARG	An invalid argument is passed to a function.	**-6**
#define SOCK_ERR_MAX_LIS-TEN_SOCK	Exceeded the maximum number of TCP passive listening sockets. Identifies that **listen** operation failed.	**-7**
#define SOCK_ERR_INVALID -9	The requested socket operation is not valid in the current socket state. For example: **accept** is called on a TCP socket before **bind** or **listen**.	
#define SOCK_ERR_ADDR_IS_RE-QUIRED	Destination address is required. Failure to provide the socket address required for the socket opera-tion to be completed. It is generated as an error to the **sendto** function when the address required to send the data to, is not known.	**-11**
#define SOCK_ERR_CONN_ABORTED	The socket is closed by the peer. The local socket is closed also.	**-12**
#define SOCK_ERR_	The socket pending operation has timed out.	**TIMEOUT -13**
#define SOCK_ERR_BUFFER_FULL	No buffer space available to be used for the re-quested socket operation.	**-14**
#define _htonl(m)	Convert a 4-byte integer from the host representa-tion to the Network byte order representation.	**(uint32)(((uint32)(m << 24)) | ((uint32)((m & 0x0000FF00) << 8)) | ((uint32)((m & 0x00FF0000) >> 8)) | ((uint32)(m >> 24)))**
#define _htons	Convert a 2-byte integer (short) from the host rep-resentation to the Network byte order representa-tion.	**(A) (uint16)((((uint16) (A)) << 8) | (((uint16) (A)) >> 8))**
#define _ntohl	Convert a 4-byte integer from the Network byte or-der representation to the host representation.	**_htonl**
#define _ntohs	Convert a 2-byte integer from the Network byte or-der representation to the host representation.	**_htons**

F.3.3 Enumeration/Typedef

F.3.3.1 Asynchronous Events

Specific enumeration used for asynchronous operations.

typedef void(* tpfAppSocketCb) (SOCKET sock, uint8 u8Msg, void *pvMsg)

The main socket application callback function. Applications register their main socket application callback through this function by calling **registerSocketCallback**. In response to events received, the following callback function is called to handle the corresponding asynchronous function called. Example: **bind**, **connect** ... etc.

Parameters:

in	*sock*	Socket ID for the callback. The socket callback function is called whenever a new event is received in response to socket operations.
in	*u8Msg*	Socket event type. Possible values are: SOCKET_MSG_BIND SOCKET_MSG_LISTEN SOCKET_MSG_ACCEPT SOCKET_MSG_CONNECT SOCKET_MSG_RECV SOCKET_MSG_SEND SOCKET_MSG_SENDTO SOCKET_MSG_RECVFROM
in	*pvMsg*	Pointer to message structure. Existing types are: tstrSocketBindMsg tstrSocketListenMsg tstrSocketAcceptMsg tstrSocketConnectMsg tstrSocketRecvMsg

See also:

tenuSocketCallbackMsgType

tstrSocketRecvMsg

tstrSocketConnectMsg

tstrSocketAcceptMsg

tstrSocketListenMsg

tstrSocketBindMsg

typedef void(* tpfAppResolveCb) (uint8 *pu8DomainName, uint32 u32ServerIP)

DNS resolution callback function.

Applications requiring DNS resolution should register their callback through this function by calling **registerSocketCallback**. The following callback is triggered in response to asynchronous call to the **gethostbyname** function (DNS Resolution callback).

Parameters:

in	*pu8DomainName*	Domain name of the host.
in	*u32ServerIP*	Server IPv4 address encoded in NW byte order format. If it is Zero, then the DNS resolution failed.

typedef struct tstrSocketAcceptMsg

Socket accept status.

Socket accept information is returned through this structure in response to the asynchronous call to the **accept** function. This structure together with the event **SOCKET_MSG_ACCEPT** are passed-in parameters to the callback function.

Data Filed	Definition
SOCKET sock	On a successful accept operation, the return information is the socket ID for the accepted connection with the remote peer. Otherwise a negative error code is returned to indicate failure of the accept operation.
struct sockaddr_in strAddr	Socket address structure for the remote peer.

typedef struct tstrSocketAcceptMsg

Socket accept status.

Socket accept information is returned through this structure in response to the asynchronous call to the **accept** function. This structure together with the event **SOCKET_MSG_ACCEPT** are passed-in parameters to the callback function.

Data Filed	Definition
sint8 status	The result of the bind operation. Holding a value of ZERO for a successful bind or otherwise a negative error code corresponding to the type of error.

typedef struct tstrSocketConnectMsg

Socket accept status.

Socket accept information is returned through this structure in response to the asynchronous call to the **accept** function. This structure together with the event **SOCKET_MSG_ACCEPT** are passed-in parameters to the callback function.

Data Filed	Definition
SOCKET sock	Socket ID referring to the socket passed to the connect function call.

Data Filed	Definition
sint8 s8Error	Connect error code. Holding a value of ZERO for a successful connect or otherwise a negative error code corresponding to the type of error.

typedef struct tstrSocketListenMsg

Socket listen status.

Socket listen information is returned through this structure in response to the asynchronous call to the **listen** function. This structure together with the event **SOCKET_MSG_LISTEN** are passed-in parameters to the callback function.

Data Filed	Definition
sint8 status	Holding a value of ZERO for a successful listen or otherwise a negative error code corresponding to the type of error.

typedef struct tstrSocketRecvMsg

Socket recv status.

Socket receive information is returned through this structure in response to the asynchronous call to the **recv** or **recvfrom** socket functions. This structure together with the events **SOCKET_MSG_RECV** or SOCKET_MSG_RECVFROM are passed-in parameters to the callback function.

Remarks:

In case the received data from the remote peer is larger than the USER buffer size defined during the asynchronous call to the **recv** function, the data is delivered to the user in a number of consecutive chunks according to the USER Buffer size. a negative or zero buffer size indicates an error with the following code: **SOCK_ERR_NO_ERROR**: Socket connection closed **SOCK_ERR_CONN_ABORTED**: Socket connection aborted: Socket receive timed out.

Data Filed	Definition
uint8* pu8Buffer	Pointer to the USER buffer (passed to recv and recvfrom function) containing the re-ceived data chunk.
sint16 s16BufferSize	The received data chunk size. Holds a negative value if there is a receive error or ZERO on success upon reception of close socket message.
uint16 u16RemainingSize	The number of bytes remaining in the current recv operation.
struct sockaddr_in strRe-moteAddr	Socket address structure for the remote peer. It is valid for **SOCKET_MSG_RECVFROM** event.

enum tenuSocketCallbackMsgType

Asynchronous APIs, make use of callback functions, in-order to return back the results once the corresponding socket operation is completed. Hence resuming the normal execution of the application code while the socket operation returns the results. Callback functions expect event messages to be passed in, in-order to identify the operation they're returning the results for. The following enum identifies the type of events that are received in the callback function.

Application Use: In order for application developers to handle the pending events from the network controller through the callback functions. A function call must be made to the function **m2m_wifi_handle_events** at least once for each socket operation.

See also:

bind

listen

accept

connect

send

recv

Enumeration Types	
SOCKET_MSG_BIND	Bind socket event
SOCKET_MSG_LISTEN	Listen socket event
SOCKET_MSG_DNS_RESOLVE	DNS Resolution event
SOCKET_MSG_ACCEPT	Accept socket event
SOCKET_MSG_CONNECT	Connect socket event
SOCKET_MSG_RECV	Receive socket event
SOCKET_MSG_SEND	Send socket event
SOCKET_MSG_SENDTO	sendto socket event
SOCKET_MSG_RECVFROM	Recvfrom socket event

F.3.3.2 Typedef

typedef struct in_addr

IPv4 address representation.

This structure is used as a placeholder for IPV4 address in other structures.

See also:

sockaddr_in

Data Filed	Definition
uint32 s_addr	Network Byte Order representation of the IPv4 address. For example, the address "192.168.0.10" is represented as 0x0A00A8C0.

typedef struct sockaddr

Generic socket address structure.

See also:

sockaddr_in

Data Filed	Definition
uint16 sa_family	Socket address family
uint8 sa_data[14]	Maximum size of all the different socket address structures

typedef struct sockaddr_in

Socket address structure for IPV4 addresses. Used to specify socket address information to which to connect to. Can be cast to **sockaddr** structure.

See also:

sockaddr_in

Data Filed	Definition
uint16 sin_family	Specifies the address family (AF). Members of AF_INET address family are IPv4 addresses. Hence, the only supported value for this is AF_INET.
uint16 sin_port	Port number of the socket. Network sockets are identified by a pair of IP addresses and port number. It must be set in the Network Byte Order format, _htons (e.g. _htons(80)). Can NOT have zero value.
in_addr sin_addr	IP Address of the socket. The IP address is of type in_addr structure. Can be set to "0" to accept any IP address for server operation. non zero otherwise.
uint8 sin_zero[8]	Padding to make structure the same size as sockaddr

F.3.4 Function

- **socketInit**
  - **NMI_API void socketInit (void)**

The function performs the necessary initializations for the socket library through the following steps:

1. A check made by the global variable gbSocketInit, ensuring that initialization for sockets is performed only once, in-order to prevent resetting the socket instances already created in the global socket array (gastrSockets).
2. Zero initializations to the global socket array (gastrSockets), which holds the list of TCP sockets.
3. Register the socket HIF (Host Interface) callback function through the call to the hif_register_cb function. This facilitates handling all of the socket related functions received through interrupts from the firmware.

Remarks:

This initialization function must be invoked before any socket operation is performed. No error codes from this initialization function since the socket array is statically allocated based in the maximum number of sockets **MAX_SOCKET** based on the systems capability.

Example

This example demonstrates the use of the socketinit for socket initialization for an mqtt chat application.

```
tstrWifiInitParam param;
int8_t ret;
char topic[strlen(MAIN_CHAT_TOPIC) + MAIN_CHAT_USER_NAME_SIZE + 1];

//Initialize the board.
system_init();
…
…
// Initialize socket interface.
socketInit();
registerSocketCallback(socket_event_handler, socket_resolve_handler);

// Connect to router.
m2m_wifi_connect((char *)MAIN_WLAN_SSID, sizeof(MAIN_WLAN_SSID),
         MAIN_WLAN_AUTH, (char *)MAIN_WLAN_PSK, M2M_WIFI_CH_ALL);
```

- **registerSocketCallback**
  - **NMI_API void registerSocketCallback (tpfAppSocketCb socket_cb, tpfAppResolveCb resolve_cb)**

Register two callback functions one for asynchronous socket events and the other one for DNS callback registering function. The registered callback functions are used to retrieve information in response to the asynchronous socket functions called.

Parameters:

In	*tpfAppSocketCb*	Assignment of callback function to the global callback **tpfAppSocketCb** gpfAppSocketCb. Delivers socket messages to the host application. In response to the asynchronous function calls, such as **bind**, **listen**, …
In	*tpfAppResolveCb*	Assignment of callback function to the global callback **tpfAppResolveCb** gpfAppResolveCb. Used for DNS resolving functionalities. The DNS resolving technique is determined by the application registering the callback. NULL is assigned when, DNS resolution is not required.

Remarks:

If any of the socket functionalities is not to be used, NULL is passed in as a parameter. It must be invoked after socketInit and before other socket layer operations.

Example:

This example demonstrates the use of the registerSocketCallback to register a socket callback function with DNS resolution CB set to null for a simple UDP server example.

```
tstrWifiInitParam param;
int8_t ret;
struct sockaddr_in addr;

// Initialize the board
system_init();

…
…

// Initialize socket address structure.
addr.sin_family = AF_INET;
addr.sin_port = _htons(MAIN_WIFI_M2M_SERVER_PORT);
addr.sin_addr.s_addr = _htonl(MAIN_WIFI_M2M_SERVER_IP);

// 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) {
                /* infinite loop in case of error in initialization*/
        }
}

// Initialize socket module
socketInit();
registerSocketCallback(socket_cb, NULL);

// Connect to router.
m2m_wifi_connect((char *)MAIN_WLAN_SSID, sizeof(MAIN_WLAN_SSID), MAIN_WLAN_AUTH,
                 (char *)MAIN_WLAN_PSK, M2M_WIFI_CH_ALL);
```

- **socket**
  - **NMI_API SOCKET socket (uint16 u16Domain, uint8 u8Type, uint8 u8Flags)**

Synchronous socket allocation function based on the specified socket type. Created sockets are non-blocking and their possible types are either TCP or a UDP sockets. The maximum allowed number of TCP sockets is [**TCP_SOCK_MAX**] sockets while the maximum number of UDP sockets that can be created simultaneously is [**UDP_SOCK_MAX**] sockets.

Parameters:

In	*u16Domain*	Socket family. The only allowed value is **AF_INET** (IPv4.0) for TCP/UDP sockets.
In	*u8Type*	Socket type allowed values are: • SOCK_STREAM (TCP) • SOCK_DGRAM (UDP)
In	*u8Flags*	Used to specify the socket creation flags. It shall be set to zero for normal TCP/UDP sockets. If could be **SOCKET_FLAGS_SSL** if the socket is used for SSL session. The use of the flag **SOCKET_FLAGS_SSL** has no meaning in case of UDP sockets.

Precondition:

The **socketInit** function must be called once at the beginning of the application to initialize the socket handler before any call to the socket function can be made.

See also:

connect, bind, listen, accept, recv, recvfrom, send, sendto, close, setsockopt, getsockopt

Returns:

On successful socket creation, a non-blocking socket type is created and a socket ID is returned.

In case of failure the function returns a negative value, identifying one of the socket error codes defined. For example: **SOCK_ERR_INVALID** for invalid argument or **SOCK_ERR_MAX_TCP_SOCK** if the number of TCP allocated sockets exceeds the number of available sockets.

Remarks:

The socket function must be called prior to any other related socket functions "e.g. send, recv, close ... etc."

Example

This example demonstrates the use of the socket function to allocate the socket, returning the socket handler to be used for other socket operations. Socket creation is dependent on the socket type.

UDP example

```
SOCKET UdpServerSocket = -1;
UdpServerSocket = socket(AF_INET, SOCK_DGRAM, 0);
```

TCP example

```
static SOCKET tcp_client_socket = -1;
tcp_client_socket = socket(AF_INET, SOCK_STREAM, 0));
```

- **bind**
    - **NMI_API sint8 bind (SOCKET sock, struct sockaddr *pstrAddr, uint8 u8AddrLen)**

Asynchronous bind function associates the provided address and local port to the socket. The function can be used with both TCP and UDP sockets it's mandatory to call the "bind" function before starting any UDP or TCP server operation. Upon socket bind completion, the application will receive a [**SOCKET_MSG_BIND**] message in the socket callback.

Parameters:

In	*sock*	Socket ID, must hold a non-negative value. A negative value will return a socket error **SOCK_ERR_INVALID_ARG**. Indicating that an invalid argument is passed in.
In	*pstrAddr*	Pointer to socket address structure "sockaddr_in" **sockaddr_in**.
In	*u8AddrLen*	Size of the given socket address structure in bytes.

Precondition:

The socket function must be called to allocate a socket before passing the socket ID to the bind function.

See also:

socket, connect, listen, accept, recv, recvfrom, send, sendto

Returns:

The function returns ZERO for successful operations and a negative value otherwise. The possible error values are:

SOCK_ERR_NO_ERROR Indicating that the operation was successful.

SOCK_ERR_INVALID_ARG Indicating passing invalid arguments such as negative socket ID or NULL socket address structure.

SOCK_ERR_INVALID Indicate socket bind failure.

Example

This example demonstrates the call of the bind socket operation after a successful socket operation.

```
struct sockaddr_in  addr;
SOCKET udpServerSocket =-1;
int ret = -1;

if(udpServerSocket == -1)
{
        udpServerSocket = socket(AF_INET,SOCK_DGRAM,0);
        if(udpServerSocket >= 0)
        {
                addr.sin_family     = AF_INET;
            addr.sin_port           = _htons(UDP_SERVER_PORT);
            addr.sin_addr.s_addr    = 0;
            ret = bind(udpServerSocket,(struct sockaddr*)&addr,sizeof(addr));

            if(ret == 0) {
                        printf("Bind success!\n");
                } else {
                        printf("Bind Failed. Error code = %d\n",ret);
                 close(udpServerSocket);
                }
        }
        else
        {
                printf("UDP Server Socket Creation Failed\n");
                return;
        }
}
```

- **listen**
  - **NMI_API sint8 listen (SOCKET sock, uint8 backlog)**

After successful socket binding to an IP address and port on the system, start listening on a passive socket for incoming connections. The socket must be bound on a local port or the listen operation fails. Upon the call to the asynchronous listen function, response is received through the event **SOCKET_MSG_BIND** in the socket callback. A successful listen means the TCP server operation is active. If a connection is accepted, then the application socket callback function is notified with the new connected socket through the event **SOCKET_MSG_ACCEPT**. Hence there is no need to call the **accept** function after calling **listen**.

After a connection is accepted, the user is then required to call the **recv** to receive any packets transmitted by the remote host or to receive notification of socket connection termination.

Parameters:

In	*sock*	Socket ID, must hold a non-negative value. A negative value will return a socket error **SOCK_ERR_INVALID_ARG**. Indicating that an invalid argument is passed in.

In	*backlog*	Not used by the current implementation.

Precondition:

The bind function must be called to assign the port number and IP address to the socket before the listen operation.

See also:

bind, accept, recv, recvfrom, send, sendto

Returns:

The function returns ZERO for successful operations and a negative value otherwise. The possible error values are:

SOCK_ERR_NO_ERROR Indicating that the operation was successful.

SOCK_ERR_INVALID_ARG Indicating passing invalid arguments such as negative socket ID.

SOCK_ERR_INVALID Indicate socket listen failure.

Example

This example demonstrates the call of the listen socket operation after a successful socket operation.

```
static void TCP_Socketcallback(SOCKET sock, uint8 u8Msg, void * pvMsg)
{
        int ret =-1;
        switch(u8Msg)
        {
                case SOCKET_MSG_BIND:
                {
                        tstrSocketBindMsg   *pstrBind = (tstrSocketBindMsg*)pvMsg;
                        if(pstrBind != NULL)
                        {
                                if(pstrBind->status == 0) {
                                        ret = listen(sock, 0);
                                        if(ret <0) {
                                                printf("Listen failure! Error = %d\n",ret);
                                        }
                                } else {
                                        M2M_ERR("bind Failure!\n");
                                        close(sock);
                                }
                        }
                }
                break;

                case SOCKET_MSG_LISTEN:
                {
                        tstrSocketListenMsg *pstrListen = (tstrSocketListenMsg*)pvMsg;
                        if(pstrListen != NULL)
                        {
                                if(pstrListen->status == 0) {
                                        ret = accept(sock,NULL,0);
                                } else {
                                        M2M_ERR("listen Failure!\n");
                                        close(sock);
                                }
                        }
                }
                break;

                case SOCKET_MSG_ACCEPT:
```

```
                {
                        tstrSocketAcceptMsg *pstrAccept = (tstrSocketAcceptMsg*)pvMsg;
                        if(pstrAccept->sock >= 0) {
                                TcpNotificationSocket = pstrAccept->sock;
                                recv(pstrAccept->sock, gau8RxBuffer, sizeof(gau8RxBuffer),
                                        TEST_RECV_TIMEOUT);
                        } else {
                                M2M_ERR("accept failure\n");
                        }
                }
                break;

                default:
                        break;
        }
}
```

- **accept**
  - **NMI_API sint8 accept (SOCKET sock, struct sockaddr *addr, uint8 *addrlen)**

The function is deprecated. An empty deceleration is used to prevent errors when legacy application code is used.

Parameters:

In	*sock*	Socket ID, must hold a non-negative value. A negative value will return a socket error **SOCK_ERR_INVALID_ARG**. Indicating that an invalid argument is passed in.
In	*addr*	Not used in the current implementation
In	*addrlen*	Not used in the current implementation

Returns:

The function returns ZERO for successful operations and a negative value otherwise. The possible error values are:

SOCK_ERR_NO_ERROR Indicating that the operation was successful.

SOCK_ERR_INVALID_ARG Indicating passing invalid arguments such as negative socket ID.

- **connect**
  - **NMI_API sint8 connect (SOCKET sock, struct sockaddr *pstrAddr, uint8 u8AddrLen)**

Establishes a TCP connection with a remote server. The asynchronous connect function must be called after receiving a valid socket ID from the **socket** function. The application socket callback function is notified of a successful new socket connection through the event **SOCKET_MSG_CONNECT**. A successful connect means the TCP session is active. The application is then required to make a call to the **recv** to receive any packets transmitted by the remote server, unless the application is interrupted by a notification of socket connection termination.

Parameters:

In	*sock*	Socket ID, must hold a non-negative value. A negative value will return a socket error **SOCK_ERR_INVALID_ARG**. Indicating that an invalid argument is passed in.
In	*pstrAddr*	Address of the remote server
In	*pstrAddr*	Pointer to socket address structure "sockaddr_in" **sockaddr_in**

In	*u8AddrLen*	Size of the given socket address structure in bytes. Not currently used, implemented for BSD compatibility only.

Precondition:

The socket function must be called to allocate a TCP socket before passing the socket ID to the bind function. If the socket is not bound, you do NOT have to call bind before the "connect" function.

See also:

socket, recv, send, close

Returns:

The function returns ZERO for successful operations and a negative value otherwise. The possible error values are:

SOCK_ERR_NO_ERROR Indicating that the operation was successful.

SOCK_ERR_INVALID_ARG Indicating passing invalid arguments such as negative socket ID or NULL socket address structure.

SOCK_ERR_INVALID Indicate socket connect failure.

Example

The example demonstrates a TCP application, showing how the asynchronous call to the connect function is made through the main function and how the callback function handles the **SOCKET_MSG_CONNECT** event.

UDP example

```
struct sockaddr_in  Serv_Addr;
SOCKET TcpClientSocket =-1;
int ret = -1

TcpClientSocket = socket(AF_INET, SOCK_STREAM, 0);
Serv_Addr.sin_family = AF_INET;
Serv_Addr.sin_port = _htons(1234);
Serv_Addr.sin_addr.s_addr = inet_addr(SERVER);
printf("Connected to server via socket %u\n",TcpClientSocket);

do
{
    ret = connect(TcpClientSocket,(sockaddr_in*)&Serv_Addr,sizeof(Serv_Addr));
    if(ret != 0) {
              printf("Connection Error\n");
    } else {
              printf("Connection successful.\n");
              break;
    }
}while(1);
```

TCP example

```
if(u8Msg == SOCKET_MSG_CONNECT)
{
       tstrSocketConnectMsg    *pstrConnect = (tstrSocketConnectMsg*)pvMsg;
       if(pstrConnect->s8Error == 0)
       {
              uint8   acBuffer[GROWL_MSG_SIZE];
              uint16  u16MsgSize;

              printf("Connect success!\n");
```

```
			u16MsgSize = FormatMsg(u8ClientID, acBuffer);
			send(sock, acBuffer, u16MsgSize, 0);
			recv(pstrNotification->Socket, (void*)au8Msg, GROWL_DESCRIPTION_MAX_LENGTH,
					GROWL_RX_TIMEOUT);
			u8Retry = GROWL_CONNECT_RETRY;
		}
		else
		{
			M2M_DBG("Connection Failed, Error: %d\n",pstrConnect->s8Error");
			close(pstrNotification->Socket);
		}
}
```

- **recv**
  - **NMI_API sint16 recv (SOCKET sock, void *pvRecvBuf, uint16 u16BufLen, uint32 u32Timeoutmsec)**

An asynchronous receive function, used to retrieve data from a TCP stream. Before calling the recv function, a successful socket connection status must have been received through any of the two socket events [**SOCKET_MSG_CONNECT**] or [**SOCKET_MSG_ACCEPT**], from the socket callback. Hence, indicating that the socket is already connected to a remote host. The application receives the required data in response to this asynchronous call through the reception of the event **SOCKET_MSG_RECV** in the socket callback.

Receiving the **SOCKET_MSG_RECV** message in the callback with zero or negative buffer length indicates the following:

SOCK_ERR_NO_ERROR: Socket connection closed.

SOCK_ERR_CONN_ABORTED: Socket connection aborted.

SOCK_ERR_TIMEOUT: Socket receive timed out the application code is expected to close the socket through the call to the close function upon the appearance of the above mentioned errors.

Parameters:

In	*sock*	Socket ID, must hold a non-negative value. A negative value will return a socket error **SOCK_ERR_INVALID_ARG**. Indicating that an invalid argument is passed in.
In	*pvRecvBuf*	Pointer to a buffer that will hold the received data. The buffer is used in the recv callback to deliver the received data to the caller. The buffer must be resident in memory (heap or global buffer).
In	*u16BufLen*	The buffer size in bytes
In	*u32Timeoutmse*	Timeout for the recv function in milliseconds. If the value is set to ZERO, the timeout will be set to infinite (the recv function waits forever). If the timeout period is elapsed with no data received, the socket will get a timeout error.

Precondition:

The socket function must be called to allocate a TCP socket before passing the socket ID to the recv function.

The socket in a connected state is expected to receive data through the socket interface.

See also:

socket, connect, bind, listen, recvfrom, close

Returns:

The function returns ZERO for successful operations and a negative value otherwise. The possible error values are:

SOCK_ERR_NO_ERROR Indicating that the operation was successful.

SOCK_ERR_INVALID_ARG Indicating passing invalid arguments such as negative socket ID or NULL receive buffer.

SOCK_ERR_BUFFER_FULL Indicate socket receive failure.

Example

The example demonstrates a code snippet for the calling of the recv function in the socket callback upon notification of accept or connect events, and the parsing of the received data when the **SOCKET_MSG_RECV** event is received.

```
switch(u8Msg)
{
        case SOCKET_MSG_ACCEPT:
        {
                tstrSocketAcceptMsg *pstrAccept = (tstrSocketAcceptMsg*)pvMsg;

                if(pstrAccept->sock >= 0) {
                        recv(pstrAccept->sock, gau8RxBuffer, sizeof(gau8RxBuffer),
                TEST_RECV_TIMEOUT);
        } else {
                        M2M_ERR("accept\n");
        }
        }
        break;

        case SOCKET_MSG_RECV:
        {
                tstrSocketRecvMsg   *pstrRx = (tstrSocketRecvMsg*)pvMsg;
        if(pstrRx->s16BufferSize > 0) {
                        recv(sock,gau8RxBuffer,sizeof(gau8RxBuffer),TEST_RECV_TIMEOUT);
                } else {
                        printf("Socet recv Error: %d\n",pstrRx->s16BufferSize);
                        close(sock);
        }
        }
        break;

        default:
                break;
        }
}
```

- **recvfrom**
  - **NMI_API sint16 recvfrom (SOCKET sock, void *pvRecvBuf, uint16 u16BufLen, uint32 u32Timeoutmsec)**

Receives data from a UDP socket.

The asynchronous recvfrom function is used to retrieve data from a UDP socket. The socket must already be bound to a local port before a call to the recvfrom function is made (i.e. message **SOCKET_MSG_BIND** is received with successful status in the socket callback).

Upon calling the recvfrom function with a successful return code, the application is expected to receive a notification in the socket callback whenever a message is received through the **SOCKET_MSG_RECVFROM** event.

Receiving the **SOCKET_MSG_RECVFROM** message in the callback with zero, indicates that the socket is closed. Whereby a negative buffer length indicates one of the socket error codes such as socket timeout error.

The recvfrom callback can also be used to show the IP address of the remote host that sent the frame by using the "strRemoteAddr" element in the **tstrSocketRecvMsg** structure.

(Refer to the code example.)

Parameters:

In	*sock*	Socket ID, must hold a non-negative value. A negative value will return a socket error **SOCK_ERR_INVALID_ARG**. Indicating that an invalid argument is passed in.
In	*pvRecvBuf*	Pointer to a buffer that will hold the received data. The buffer shall be used in the recv callback to deliver the received data to the caller. The buffer must be resident in memory (heap or global buffer).
In	*u16BufLen*	The buffer size in bytes
In	*u32TimeoutSeconds*	Timeout for the recv function in milliseconds. If the value is set to ZERO, the timeout will be set to infinite (the recv function waits forever).

Precondition:

The socket function must be called to allocate a TCP socket before passing the socket ID to the recv function.

The socket corresponding to the socket ID must be successfully bound to a local port through the call to a **bind** function.

See also:

socket, bind, close

Returns:

The function returns ZERO for successful operations and a negative value otherwise. The possible error values are:

SOCK_ERR_NO_ERROR Indicating that the operation was successful.

SOCK_ERR_INVALID_ARG Indicating passing invalid arguments such as negative socket ID or NULL receive buffer.

SOCK_ERR_BUFFER_FULL Indicate socket receive failure.

Example

The example demonstrates a code snippet for the calling of the recvfrom function in the socket callback upon notification of a successful bind event, and the parsing of the received data when the **SOCKET_MSG_RECVFROM** event is received.

```
switch(u8Msg)
{
        case SOCKET_MSG_BIND:
        {
                tstrSocketBindMsg   *pstrBind = (tstrSocketBindMsg*)pvMsg;
                if(pstrBind != NULL)
        {
                        if(pstrBind->status == 0) {
                                recvfrom(sock, gau8SocketTestBuffer, TEST_BUFFER_SIZE, 0);
                        } else {
                                M2M_ERR("bind\n");
                        }
                }
```

```
          }
          break;

          case SOCKET_MSG_RECVFROM:
          {
                   tstrSocketRecvMsg   *pstrRx = (tstrSocketRecvMsg*)pvMsg;
                   if(pstrRx->s16BufferSize > 0) {
                            //get the remote host address and port number
                            uint16 u16port = pstrRx->strRemoteAddr.sin_port;
                            uint32 strRemoteHostAddr = pstrRx->strRemoteAddr.sin_addr.s_addr;
                            printf("Recieved frame with size = %d.\tHost address=%x, Port number =
          %d\n\n", pstrRx->s16BufferSize,strRemoteHostAddr, u16port);

                            ret = recvfrom(sock, gau8SocketTestBuffer,
                   sizeof(gau8SocketTestBuffer), TEST_RECV_TIMEOUT);
                   } else {
                            printf("Socet recv Error: %d\n",pstrRx->s16BufferSize);
                            ret = close(sock);
                   }
          }
          break;

          default:
                   break;
}
```

- **send**
  - **NMI_API sint16 send (SOCKET sock, void *pvSendBuffer, uint16 u16SendLength, uint16 u16Flags)**

Asynchronous sending function, used to send data on a TCP/UDP socket.

Called by the application code when there is outgoing data available required to be sent on a specific socket handler. The only difference between this function and the similar **sendto** function, is the type of socket the data is sent on and the parameters passed in. **send** function is most commonly called for sockets in a connected state. After the data is sent, the socket callback function registered using **registerSocketCallback()**, is expected to receive an event of type **SOCKET_MSG_SEND** holding information containing the number of data bytes sent.

Parameters:

In	*sock*	Socket ID, must hold a non-negative value. A negative value will return a socket error **SOCK_ERR_INVALID_ARG**. Indicating that an invalid argument is passed in.
In	*pvSendBuffer*	Pointer to a buffer holding data to be transmitted
In	*u16SendLength*	The buffer size in bytes
In	*u16Flags*	Not used in the current implementation

Precondition:

Sockets must be initialized using socketInit.

For TCP Socket:

Must use a successfully connected socket (so that the intended recipient address is known ahead of sending the data). Hence this function is expected to be called after a successful socket connect operation (in client case or accept in the server case).

For UDP Socket:

UDP sockets most commonly use **sendto** function, where the destination address is defined. However, in-order to send outgoing data using the **send** function, at least one successful call must be made to the **sendto** function prior to the consecutive calls to the **send** function, to ensure that the destination address is saved in the firmware.

Warning:

u16SendLength must not exceed **SOCKET_BUFFER_MAX_LENGTH**.

Use a valid socket identifier through a prior call to the **socket** function. Must use a valid buffer pointer. Successful completion of a call to **send()** does not guarantee delivery of the message. A negative return value indicates only locally-detected errors.

See also:

socketInit, recv, sendto, socket, connect, accept, sendto

Returns:

The function shall return **SOCK_ERR_NO_ERROR** for successful operation and a negative value (indicating the error) otherwise.

- **sendto**
  - **NMI_API sint16 sendto (SOCKET sock, void *pvSendBuffer, uint16 u16SendLength, uint16 flags, struct sockaddr *pstrDestAddr, uint8 u8AddrLen)**

Asynchronous sending function, used to send data on a UDP socket. Called by the application code when there is data required to be sent on a UDP socket handler. The application code is expected to receive data from a successful bounded socket node. The only difference between this function and the similar **send** function, is the type of socket the data is received on. This function works only with UDP sockets. After the data is sent, the socket callback function registered using **registerSocketCallback()**, is expected to receive an event of type **SOCKET_MSG_SENDTO**.

Parameters:

In	*sock*	Socket ID, must hold a non-negative value. A negative value will return a socket error **SOCK_ERR_INVALID_ARG**. Indicating that an invalid argument is passed in.
In	*pvSendBuffer*	Pointer to a buffer holding data to be transmitted. A NULL value will return a socket error **SOCK_ERR_INVALID_ARG**. Indicating that an invalid argument is passed in.
In	*u16SendLength*	The buffer size in bytes. It must not exceed **SOCKET_BUFFER_MAX_LENGTH**
In	*flags*	Not used in the current implementation
In	*pstrDestAddr*	The destination address
In	*u8AddrLen*	Destination address length in bytes. Not used in the current implementation, only included for BSD compatibility.

Precondition:

Sockets must be initialized using socketInit.

Warning:

u16SendLength must not exceed **SOCKET_BUFFER_MAX_LENGTH**.

Use a valid socket (returned from socket). A valid buffer pointer must be used (not NULL).

Successful completion of a call to **sendto()** does not guarantee delivery of the message. A negative return value indicates only locally-detected errors.

See also:

socketInit

recvfrom

sendto

socket

connect

accept

send

Returns:

The function returns **SOCK_ERR_NO_ERROR** for successful operation and a negative value (indicating the error) otherwise.

- **close**
  - **NMI_API sint8 close (SOCKET sock)**

Asynchronous close function, releases all the socket assigned resources.

Parameters:

In	*sock*	Socket ID, must hold a non-negative value. A negative value will return a socket error **SOCK_ERR_INVALID_ARG**. Indicating that an invalid argument is passed in.

Precondition:

Sockets must be initialized through the call of the socketInit function. **Close** is called only for valid socket identifiers created through the **socket** function.

Warning:

If close is called while there are still pending messages (sent or received) they will be discarded.

See also:

socketInit

socket

Returns:

The function returned **SOCK_ERR_NO_ERROR** for successful operation and a negative value (indicating the error) otherwise.

- **nmi_inet_addr**
  - **NMI_API uint32 nmi_inet_addr (char *pcIpAddr)**

Asynchronous function which returns a BSD socket compliant Internet Protocol (IPv4) socket address. This IPv4 address in the input string parameter could either be specified as a hostname, or as a numeric string representation like n.n.n.n known as the IPv4 dotted-decimal format (i.e. "192.168.10.1"). This function is used whenever an ip address needs to be set in the proper format (i.e. for the **tstrM2MIPConfig** structure).

Parameters:

In	*pcIpAddr*	A null terminated string containing the IP address in IPv4 dotted-decimal address

Returns:

Unsigned 32-bit integer representing the IP address in Network byte order (e.g. "192.168.10.1" will be expressed as 0x010AA8C0).

- **gethostbyname**
  - **NMI_API sint8 gethostbyname (uint8 *pcHostName)**

Asynchronous DNS resolving function. This function use DNS to resolve a domain name into the corresponding IP address. A call to this function will cause a DNS request to be sent and the response will be delivered to the DNS callback function registered using **registerSocketCallback()**.

Parameters:

In	*pcHostName*	NULL terminated string containing the domain name for the remote host. Its size must not exceed **HOSTNAME_MAX_SIZE**.

Warning:

Successful completion of a call to **gethostbyname()** does not guarantee success of the DNS request, a negative return value indicates only locally-detected errors, for example if the hostname parameter exceeds the allowed size **HOSTNAME_MAX_SIZE**.

See also:

registerSocketCallback

Returns:

SOCK_ERR_NO_ERROR

SOCK_ERR_INVALID_ARG

- **setsockopt**
  - **NMI_API sint8 setsockopt (SOCKET socket, uint8 u8Level, uint8 option_name, const void *option_value, uint16 u16OptionLen)**

The setsockopt() function shall set the option specified by the option_name argument, at the protocol level specified by the level argument, to the value pointed to by the option_value argument for the socket specified by the socket argument.

Possible Options:

SO_SET_UDP_SEND_CALLBACK: Enable/Disable callback messages for **sendto()**.

Since UDP is unreliable by default the user maybe interested (or not) in receiving a message of /ref **SOCKET_MSG_SENDTO** for each call of **sendto()**. Enabled if option_value equals BTRUE disabled otherwise.

IP_ADD_MEMBERSHIP: valid for UDP sockets, this option is used to receive frames sent to a multicast group. Option_value shall be a pointer to Unsigned 32 bit integer containing the Multicast ipv4 address.

IP_DROP_MEMBERSHIP: valid for UDP sockets, this option is used to stop receiving frames sent to a multicast group. Option_value shall be a pointer to Unsigned 32 bit integer containing the Multicast ipv4 address.

Possible values for s32Level: This argument is ignored.

Parameters:

In	*sock*	Socket handler
In	*level*	Protocol level always SOL_SOCKET for now
In	*option_name*	Option to be set

In	*option_value*	Pointer to user provided value
In	*option_len*	Length of the option value

Warning:

Note that sending IGMP packets to Join/Leave multicast groups is not implemented at the current moment. Calling this function will Pass/Filter packets sent to Multicast address provided in the option_value.

Returns:

The function shall return **SOCK_ERR_NO_ERROR** for successful operation and a negative value (indicating the error) otherwise.

- **getsockopt**
  - NMI_API sint8 getsockopt (SOCKET sock, uint8 u8Level, uint8 u8OptName, const void *pvOptValue, uint8 *pu8OptLen)

Get socket options. This function isn't implemented in the current release.

Parameters:

In	*sock*	Socket Identifier
In	*u8Level*	The protocol level of the option
In	*u8OptName*	The u8OptName argument specifies a single option to get
Out	*pvOptValue*	The pvOptValue argument contains pointer to a buffer containing the option value
Out	*pu8OptLen*	Option value buffer length

Returns:

The function shall return ZERO for successful operation and a negative value otherwise.

F.4 SPI Flash

F.4.1 Function

- **spi_flash_get_size**
  - **uint32 spi_flash_get_size (void)**

Returns

SPI flash size in case of success and zero in case of failure.

Note:

Returned value is in Mb (Mega Bits).

- **spi_flash_read**
  - **sint8 spi_flash_read (uint8 *pu8Buf, uint32 u32Addr, uint32 u32Sz)**

Read a specified portion of data from the SPI Flash.

Parameters:

out	*pu8Buf*	Pointer to data buffer to which the data will be copied in case of successful operation
in	*u32Addr*	Address (Offset) to read from in the SPI flash
in	*u32Sz*	Total size of data to be read in bytes

Warning:

Address (offset) plus size of data must not exceed the SPI Flash size.

No firmware is required for reading from SPI flash.

In case there is a running firmware, it is required to pause the firmware first before any attempt to access the SPI Flash to avoid any racing between host and the running firmware on the bus using **m2m_wifi_download_mode**.

Note:

This is a blocking function.

See also:

m2m_wifi_download_mode

spi_flash_get_size

Returns:

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

- **spi_flash_write**
  - **sint8 spi_flash_write (uint8 *pu8Buf, uint32 u32Offset, uint32 u32Sz)**

Write a specified portion of the data to the SPI Flash.

Parameters:

in	*pu8Buf*	Pointer to the data buffer which contains the data required to be written.
in	*u32Offset*	Address (Offset) to write at in the SPI flash.
in	*u32Sz*	Total number of data bytes.

Warning:

Address (offset) plus size of data must not exceed flash size.

No firmware is required for writing to SPI flash.

In case of there is a running firmware, it is required to pause your firmware first before any trial to access SPI flash to avoid any racing between host and running firmware on bus using **m2m_wifi_download_mode**.

Before writing to any section, it is required to erase it first.

See also:

m2m_wifi_download_mode

spi_flash_get_size, spi_flash_erase

Returns:

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

Note:

This is s blocking function

It is user's responsibility to verify that the data has been written successfully by reading the data again and comparing it to the original.

- **spi_flash_erase**
  - **sint8 spi_flash_erase (uint32 u32Offset, uint32 u32Sz)**

Erase a specified portion of SPI Flash.

Parameters:

in	*u32Offset*	Address (Offset) to erase from in the SPI flash.
in	*u32Sz*	Size of the SPI flash portion required to be erased.

See also:

m2m_wifi_download_mode

spi_flash_get_size

Warning:

Address (offset) plus size of data must not exceed the SPI flash size.

No firmware is required for writing to the SPI flash.

In case of there is a running firmware, it is required to pause the firmware first before any attempt to access the SPI flash to avoid any racing between host and the running firmware on the bus using **m2m_wifi_download_mode**.

Note:

This is a blocking function.

Returns:

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

Appendix G. Revision History

Doc Rev.	Date	Comments
42420A	03/2015	Initial document release.