ATWINC15x0

Wi-Fi Network Controller Software Design Guide

Introduction

Microchip's SmartConnect ATWINC15x0 is an IEEE® 802.11 b/g/n network controller SoC for Internet of Things (IoT) applications. It is an ideal add-on to the existing microcontroller (MCU) solutions bringing Wi-Fi and network capabilities through a SPI-to-Wi-Fi interface. The ATWINC15x0 connects to any Microchip AVR® or Microchip SMART MCU with minimal resource requirements.

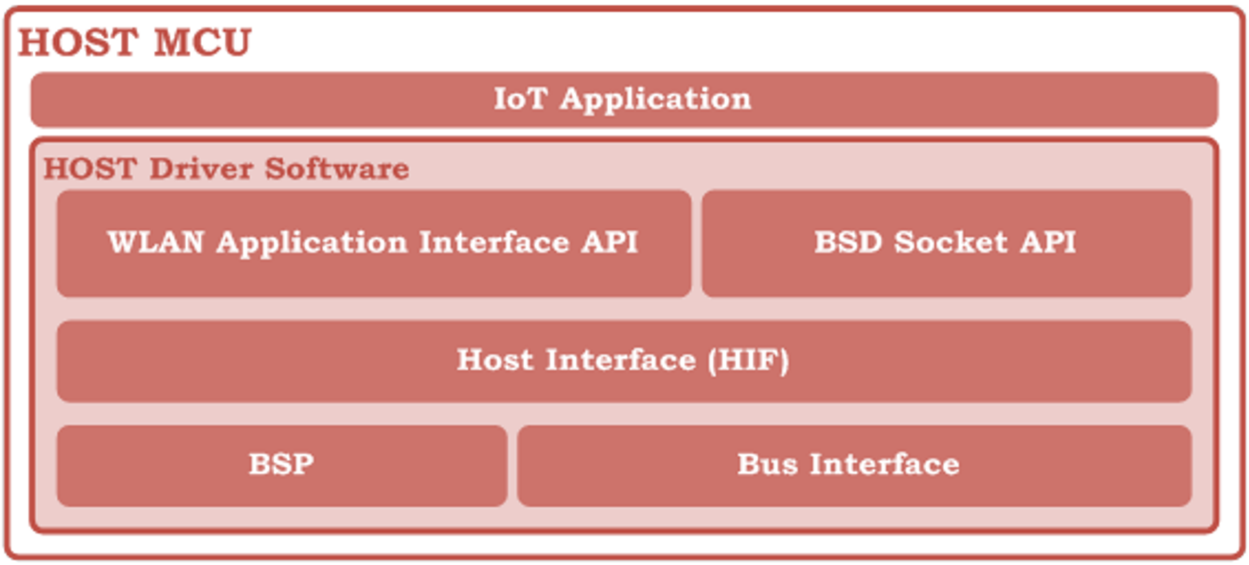
Features

- 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 Wi-Fi Network Controller (WINC) to operate with a wide range of MCUs including low-end MCUs.
- Embedded uIP 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
- 8Mb and 4Mb internal Flash memory with Over-the-Air (OTA) firmware upgrade
- Low-power consumption with different Power Save modes
- Low footprint host driver with the following capabilities:
  - Can run on 8-, 16-, and 32- bit MCU using SPI interface
  - Little and Big endian support

Table of Contents

1. Host Driver Architecture

Figure 1-1. Host Driver Software Architecture



The ATWINC15x0 host driver software is a C library, which provides the host MCU application with necessary APIs to perform necessary WLAN and socket operations. The figure above 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.

1.1 WLAN API

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

This includes the following services:

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

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

1.2 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 the WINC sockets and BSD sockets.

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

The detailed description of the socket operations is provided in **Section 6 "WINC Socket Programming"**.

1.3 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 `m2m_hif.h` file.

The detailed description of the HIF design is provided in **Section 15 “Host Interface Protocol”**.

1.4 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, and so on).

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

1.5 Serial Bus Interface

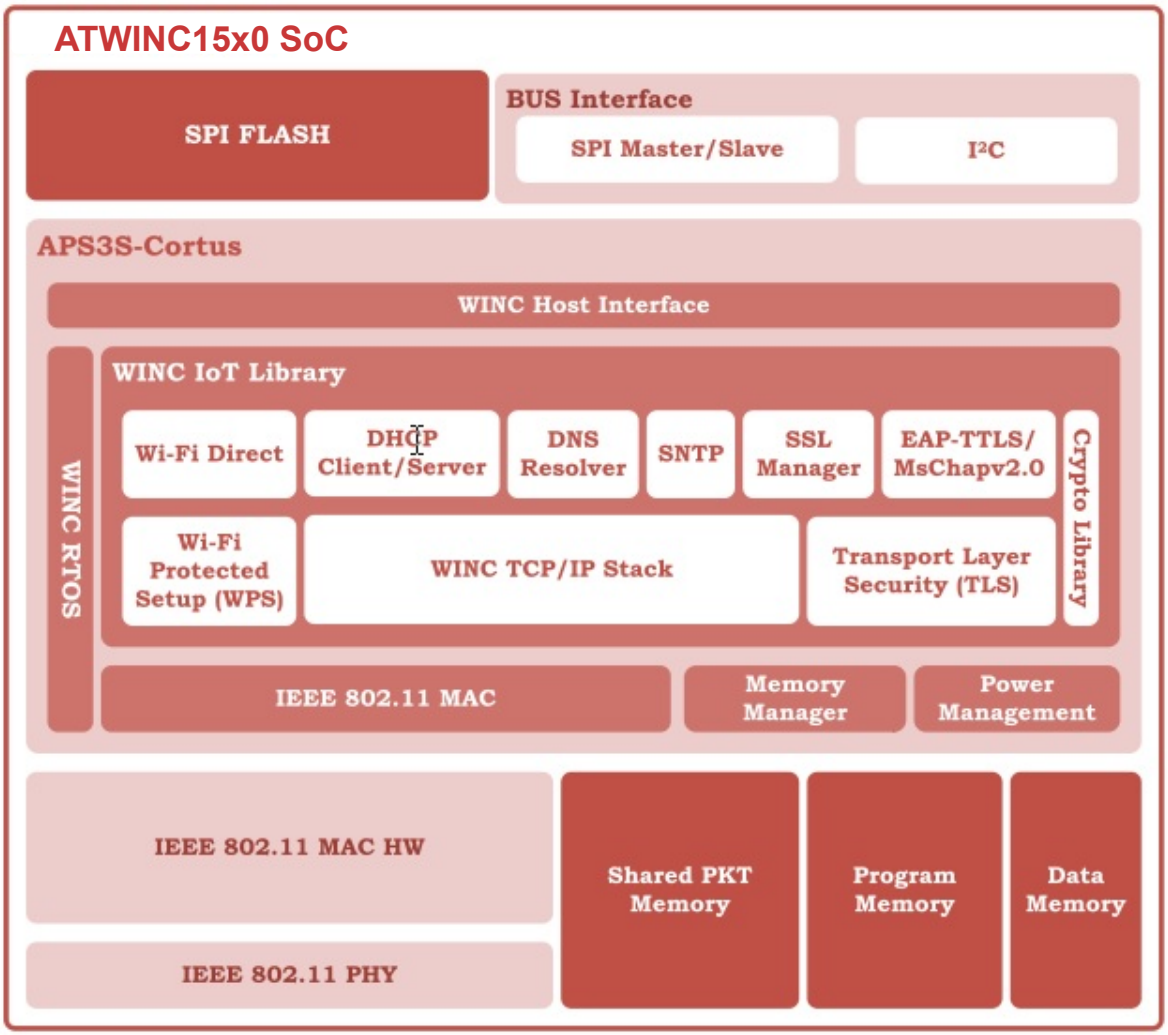
The Serial Bus Interface module abstracts the hardware associated with implementing the bus between the Host and the WINC. The serial bus interface abstracts I2C, SPI, or UART bus (Currently, host driver supports only SPI bus interface). The basic bus access operations (Read and Write) are implemented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

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

2. ATWINC15x0 System Architecture

The following figure shows the ATWINC15x0 system architecture. In addition to its built-in Wi-Fi IEEE-802.11 physical layer and RF front end, the WINC ASIC contains 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. ATWINC15x0 System Architecture



2.1 Bus Interface

Hardware logic for the supported bus types for the ATWINC15x0 communications.

Note: SPI is currently the Bus Interface supported by the Host Driver.

2.2 Nonvolatile Storage

The ATWINC1510 has an integrated 8Mb and the ATWINC15x0 has an integrated 4Mb serial Flash inside the WINC package (SIP). This stores the WINC firmware image and can store a second image to support OTA. It also stores information used by the WINC firmware in the run-time.

The detailed description of the serial Flash is provided in **Section 13 "WINC Serial Flash Memory"**.

2.3 CPU

The SoC contains an APS3S-Cortus 32-bit CPU running at 40MHz 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 ATWINC15x0 firmware code.

2.6 Data Memory

64KB RAM is provided for the ATWINC15x0 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 the 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 Save 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 the ATWINC15x0 CPU. The ATWINC15x0 RTOS provides semaphores and timer functionality.

2.12 WINC IoT Library

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

2.12.1 WINC TCP/IP STACK

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

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 is a customized version of the original uIP implementation which has several enhancements to boost TCP and UDP throughput.

2.12.2 DHCP CLIENT/SERVER

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

The WINC firmware provides an instance of a DHCP server that automatically starts when the 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.

2.12.3 DNS RESOLVER

The 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.

2.12.4 SNTP

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

2.12.5 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.

2.12.6 TRANSPORT LAYER SECURITY

For TLS implementation, refer to **Section 7 “Transport Layer Security (TLS)”** for details.

2.12.7 WI-FI PROTECTED SETUP

For WPS protocol implementation, refer to **Section 10.3 “Wi-Fi Protected Setup (WPS)”** for details.

2.12.8 WI-FI DIRECT

For Wi-Fi Direct protocol implementation, refer to **Section 9 “Wi-Fi Direct P2P Mode”** for details.

2.12.9 CRYPTO LIBRARY

The Crypto Library contains a set of cryptographic algorithms used by the 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 the WINC firmware.

- BSP Initialization
- WINC Host Driver Initialization
- Socket Layer Initialization
- Call WINC Driver Entry Point

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

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 the corresponding host MCU control GPIOs.
- Initializes the host MCU GPIO which connects to the WINC interrupt line. It configures the GPIO as an interrupt source to the host MCU. During runtime, the WINC interrupts the host to notify the application of events and data pending inside the 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 and SPI peripheral. The compilation flag `CONF_WINC_USE_SPI` must be enabled in `conf_winc.h` (bus interfaces `CONF_WINC_USE_UART` and `CONF_WINC_USE_I2C` are currently not supported).
- 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.

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

3.3 Socket Layer Initialization

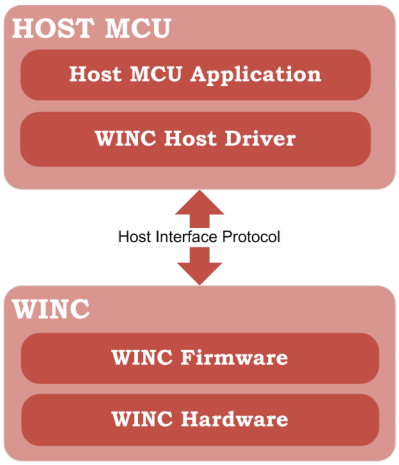
Socket layer initialization is carried out by calling the `socketInit` API. It must be called prior to any socket activity. For more information about socket initialization and programming, refer to **Section 6.2 "WINC Sockets API"**.

[1] Refer to the ATWINC15x0-MR210xB Data Sheet (DS70005304) for more information about the hardware power-up/down sequence.

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 as described in **Section 15 "Host Interface (HIF) Protocol"**. The HIF protocol defines how to serialize and de-serialize API requests and response callbacks over the serial bus interface SPI (I2C and UART are currently not supported).

Figure 3-1. WINC System Architecture



The 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 is 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 the WINC firmware to the host MCU on a First-Come First-Served (FCFS) manner. For instance, the host MCU application has two open sockets: socket 1 and socket 2. If the WINC receives socket 1 data followed by socket 2 data, then HIF delivers socket data in two HIF protocol messages in the order in which it is 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 the WINC firmware. The host MCU application is a big state machine which processes received data and events when the WINC driver calls the event callback function(s). To receive event callbacks, the host MCU application is required to

[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.

call the m2m_wifi_handle_events API to let the host driver retrieve and process the pending events from the WINC firmware. It is recommended to call this function if any of the following events occur:

- The host MCU application polls the API in main loop or a dedicated task.
- When the host MCU receives an interrupt from the WINC firmware.

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

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

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

Note:

- 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.
- When the host MCU is polling m2m_wifi_handle_events, the API checks for pending unhandled interrupt from the WINC. If no interrupt is pending, it returns immediately. If an interrupt is pending, m2m_wifi_handle_events sequentially reads all the pending HIF message 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 the previous sections.

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

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

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

    /*intilize the 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

The WINC firmware offers a set of configurable parameters that control its behavior. There is a set of APIs provided to the 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 uses 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 the WINC system to UTC time to ensure proper validity check of the X509 certificate expiration date. Since the WINC does not contain a built-in real-time clock (RTC), there are two ways to obtain UTC time:

- Using the internal SNTP client – this 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, for example, `time-c.nist.gov`. The SNTP client uses a default update cycle of one day.
- From the host MCU RTC – if the host MCU has a RTC, the application may disable the SNTP client by calling `m2m_wifi_enable_sntp(0)` (by passing zero as the argument) after the WINC initialization. The application provisions the WINC system time by calling `m2m_wifi_set_system_time` API.

4.1.2 Firmware and Driver Version

During start-up, 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.

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

The version parameters provided are as follows:

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

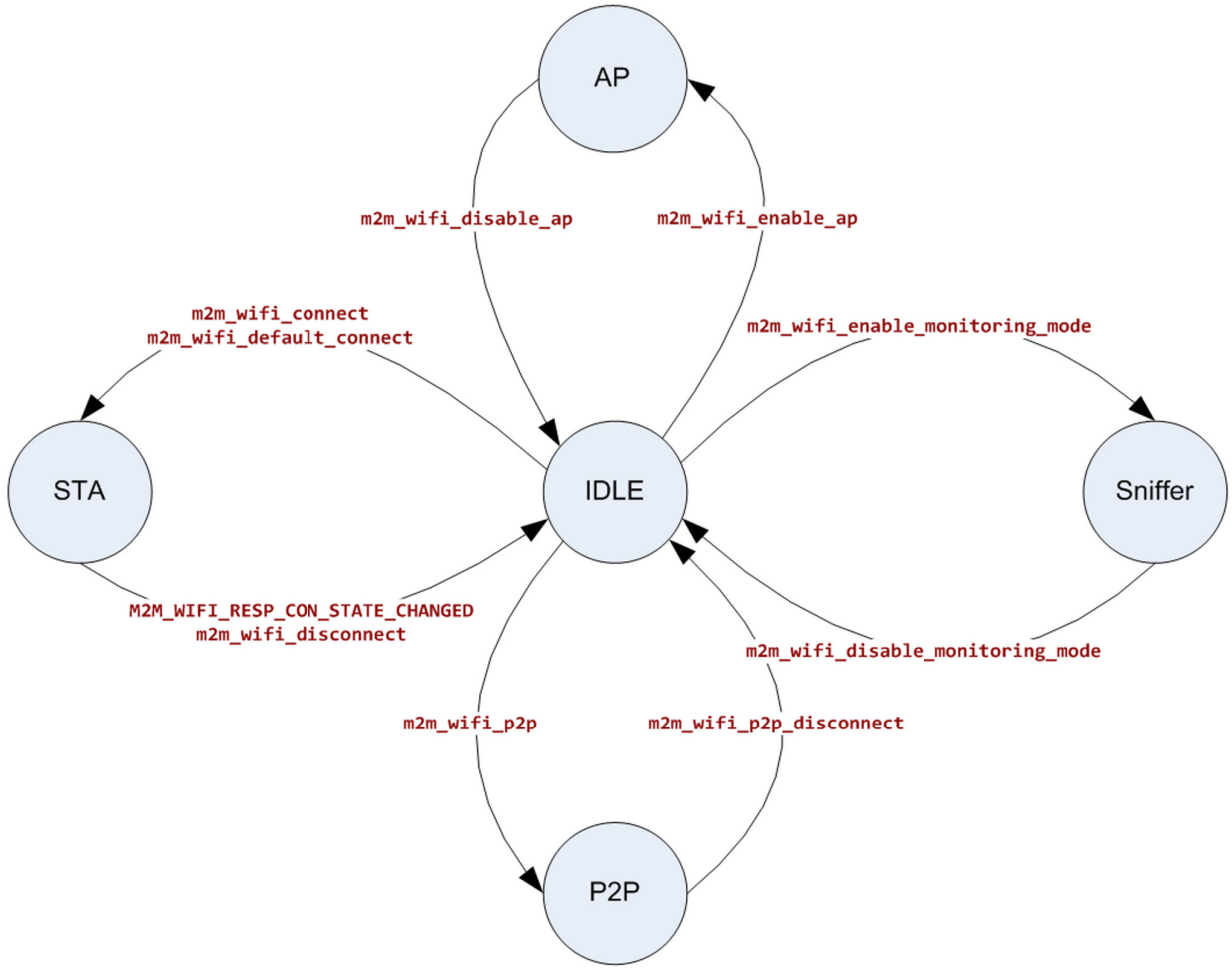
4.2 WINC Modes of Operation

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, the WINC remains in Idle mode waiting for any command to change the mode or to update the configuration parameters. In this mode, the WINC enters into Power Save mode which disables the IEEE 802.11 radio and all unneeded peripherals and suspends the WINC CPU. If the WINC receives any configuration commands from the host MCU, it updates the configuration, sends back the response to the host MCU, and then returns to the Power Save mode.

4.2.2 Wi-Fi Station Mode

The 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. The 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. The WINC firmware ignores mode change requests while in this mode until the WINC exits the mode.

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

For more information about this mode, refer to **Section 5 "Wi-Fi Station Mode"**.

4.2.3 Wi-Fi Direct (P2P) Mode

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

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

For more information about this mode, refer to **Section 9 "Wi-Fi Direct P2P Mode"**.

4.2.4 Wi-Fi Hotspot (AP) Mode

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

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

For more information about this mode, refer to **Section 9 "Wi-Fi AP Mode"**.

4.2.5 Sniffer (Monitoring) Mode

Note: Sniffer mode is not supported in the firmware version 19.5.x.

In this mode, the 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. The 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. The WINC firmware ignores mode change requests while in this mode until the WINC exits the mode.

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

For more information about this mode, refer to **Section 8 "Wi-Fi Sniffer Mode"**.

4.3 Network Parameters

4.3.1 Wi-Fi MAC Address

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

- Assignment from the host MCU – this method occurs when the host MCU application calls the `m2m_wifi_set_mac_address` API after initialization using `m2m_wifi_init` API.
- Assignment from the WINC OTP (One-Time-Programmable) memory – the 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 programmatically sets a different MAC address after initialization using the API `m2m_wifi_set_mac_address`.

Note:

- OTP MAC address is programmed in the WINC OTP memory at the time of manufacturing.
- Use `m2m_wifi_get_otp_mac_address` API to check if there is a valid programmed MAC address in the 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 the WINC firmware regardless from whether it is assigned from the host MCU or from the WINC OTP.
- For more details on API, refer to the Atmel Software Framework for ATWINC1500 (Wi-Fi).

4.3.2 Device Name

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

Note: If no device name is provided, the default device name is ATWINC15x0_ee:ff where *ee* and *ff* are the last 4 bytes of the WINC MAC address in hexadecimal notation, for example, *aa:bb:cc:dd:ee:ff*.

4.3.3 IP Address

The WINC firmware uses the embedded DHCP client to automatically obtain an IP configuration after a successful Wi-Fi connection. After the IP configuration is obtained, the host MCU application is notified by the asynchronous event `M2M_WIFI_REQ_DHCP_CONF`.

Alternatively, the host MCU application can set a static IP configuration by calling the `m2m_wifi_set_static_ip` API. Before setting a static IP address, it is recommended to disable DHCP using the API `m2m_wifi_enable_dhcp(0)` and then set the static IP as shown below.

```
In Main(), disable dhcp after m2m_wifi_init as shown below
/* 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)
    {}
}
m2m_wifi_enable_dhcp(0);

Set Static IP when WINC is connected to AP as shown below.
static void wifi_cb(uint8_t u8MsgType, void *pvMsg)
{
    switch (u8MsgType) {
    case M2M_WIFI_RESP_CON_STATE_CHANGED:
    {
        tstrM2mWifiStateChanged *pstrWifiState = (tstrM2mWifiStateChanged *)pvMsg;
        if (pstrWifiState->u8CurrState == M2M_WIFI_CONNECTED){

            printf("Wi-Fi connected\r\n");

            tstrM2MIPConfig ip_client;
            ip_client.u32StaticIP = _htonl(0xc0a80167);       // Provide the required Static
IP
            ip_client.u32DNS = _htonl(0xc0a80101);            // Provide DNS server details
            ip_client.u32SubnetMask = _htonl(0xFFFFFF00);     // Provide the SubnetMask for
the currently connected AP
            ip_client.u32Gateway = _htonl(0xc0a80101);        // Provide the GAteway IP for
the AP
            printf("Wi-Fi setting static ip\r\n");
            m2m_wifi_set_static_ip(&ip_client);
        }
    }
}
```

4.4 Power Save Modes

The WINC firmware supports multiple Power Save 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.

The WINC supports the following Power Save modes:

- M2M_PS_MANUAL
- M2M_PS_DEEP_AUTOMATIC
- M2M_PS_AUTOMATIC (deprecated, not be used in new implementations)
- M2M_PS_H_AUTOMATIC (deprecated, not be used in new implementations)

Note: `M2M_PS_DEEP_AUTOMATIC` mode recommended for most applications.

4.4.1 M2M_PS_MANUAL

This is a fully host-driven Power Save mode.

- The WINC sleeps when the host uses the `m2m_wifi_request_sleep` API. During this period, the host MCU can also sleep for extended durations.
- The WINC wakes up when the host MCU application requests services from the WINC by calling any host driver API function, for example, Wi-Fi or socket operation.

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

For an active Wi-Fi connection, the AP may decide to drop the connection if the WINC is absent because it sleeps for long time duration. If connection is dropped, the WINC detects the disconnection on the next wake-up cycle and notifies the host to reconnect to the AP again. To maintain an active Wi-Fi connection for extended durations, the host MCU application must periodically wake-up the WINC in order to send a keep-alive Wi-Fi frame to the AP. The host must carefully choose the sleep period 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 also fits applications which periodically send notifications with a very long spacing between notifications. Careful power planning is required when using this mode. If the host MCU decides to Sleep for a longer period, it may use `M2M_PS_MANUAL` or may power off the WINC[3]. The advantage of this mode compared to powering off the WINC is that `M2M_PS_MANUAL` saves the time required for the WINC firmware to boot since the firmware is always loaded in the WINC memory. The real advantage and disadvantage depend on the nature of the application. In some applications, the sleep duration can be long enough to be a power-efficient decision to power off the WINC and then power it on again and reconnect to the AP when the host MCU wakes up. In other situations, a latency-sensitive application may choose to use `M2M_PS_MANUAL` to avoid the WINC firmware boot latency on the expense of slightly increased power consumption.

During the WINC Sleep mode, the WINC in `M2M_PS_MANUAL` mode saves more power than `M2M_PS_DEEP_AUTOMATIC` mode. In `M2M_PS_MANUAL` mode, the WINC skips beacon monitoring whereas in `M2M_PS_DEEP_AUTOMATIC` mode, it wakes up to receive beacons. The comparison also includes the effect of the host MCU sleep duration: if the host MCU sleeps for a longer period, the Wi-Fi connection may frequently drop and the power advantage of the `M2M_PS_MANUAL` mode is lost due to the

[3] Refer to the ATWINC15x0-MR210xB Data Sheet (DS70005304) for more information about the hardware power-up/down sequence.

power consumed in the Wi-Fi reconnection. In contrast, the `M2M_PS_DEEP_AUTOMATIC` mode can keep the Wi-Fi connection for long durations at the expense of waking up the WINC to monitor the AP beacon.

4.4.2 M2M_PS_AUTOMATIC

This mode is deprecated and kept for backward compatibility and development reasons. It is not recommended to use in new implementations.

4.4.3 M2M_PS_H_AUTOMATIC

This mode is deprecated and kept for backward compatibility and development reasons. It is not recommended to use in new implementations.

4.4.4 M2M_PS_DEEP_AUTOMATIC

This mode implements the Wi-Fi standard power-saving method in the ATWINC15x0 module, which sleeps and periodically wakes up to monitor AP beacons. The AP is required to buffer data while stations are in Power Save mode and transmit data later when stations wake-up. The AP periodically transmits a beacon frame to synchronize with a network every beacon period. A station, which is in Power Save mode, periodically wakes up to receive the beacon and monitor the signaling information included in the beacon. The beacon conveys information to the station about unicast data, which belongs to the station and currently buffered inside the AP while the station is Sleep mode. The beacon also provides information to the station when the AP is going to send broadcast/multicast data.

In this mode, the ATWINC15x0 module enters into Sleep state by turning off the IEEE 802.11 radio, MAC, and system clock. Prior Sleep mode, the ATWINC15x0 programs a hardware timer (running on an internal low-power oscillator) with a sleep period determined by the WINC firmware power management module.

Any of the following events can wake-up the ATWINC15x0 module from Sleep state:

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

4.5 Configuring Listen Interval and DTIM Monitoring

The WINC allows the host MCU application to tweak system power consumption by configuring beacon monitoring parameters. The AP periodically sends beacons every *DTIM period* (for example, 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*. Listen interval tells AP how many beacons periods the station can sleep before it wakes up to receive data buffer in AP.

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

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

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

[4] The wake-up sequence is internally handled in the WINC host driver by the `hif_chip_wake` API. Refer to **Section 15 "Host Interface Protocol"** for more information.

5. Wi-Fi Station Mode

This chapter provides information about the WINC Wi-Fi Station (STA) mode as described in **Section 4.2.2 "Wi-Fi Station Mode"**. The STA mode involves a scan operation; association to an AP using parameters (SSID and credentials) provided by the host MCU or using AP parameters stored in the WINC nonvolatile 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, 13 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, 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, the WINC sends probe request Wi-Fi frames and waits for the current Wi-Fi channel to receive probe response frames from nearby APs before it switches to the next channel. Increasing the scan wait time has a positive effect on the number of access points detected during scan. However, it has a negative effect on the power consumption and overall scan duration. The WINC firmware default scan wait time is optimized to provide the tradeoff between the power consumption and scan accuracy. The WINC firmware provides flexible configuration options to the host MCU application to increase the scan time. Refer to the `m2m_wifi_set_scan_options` for more details.

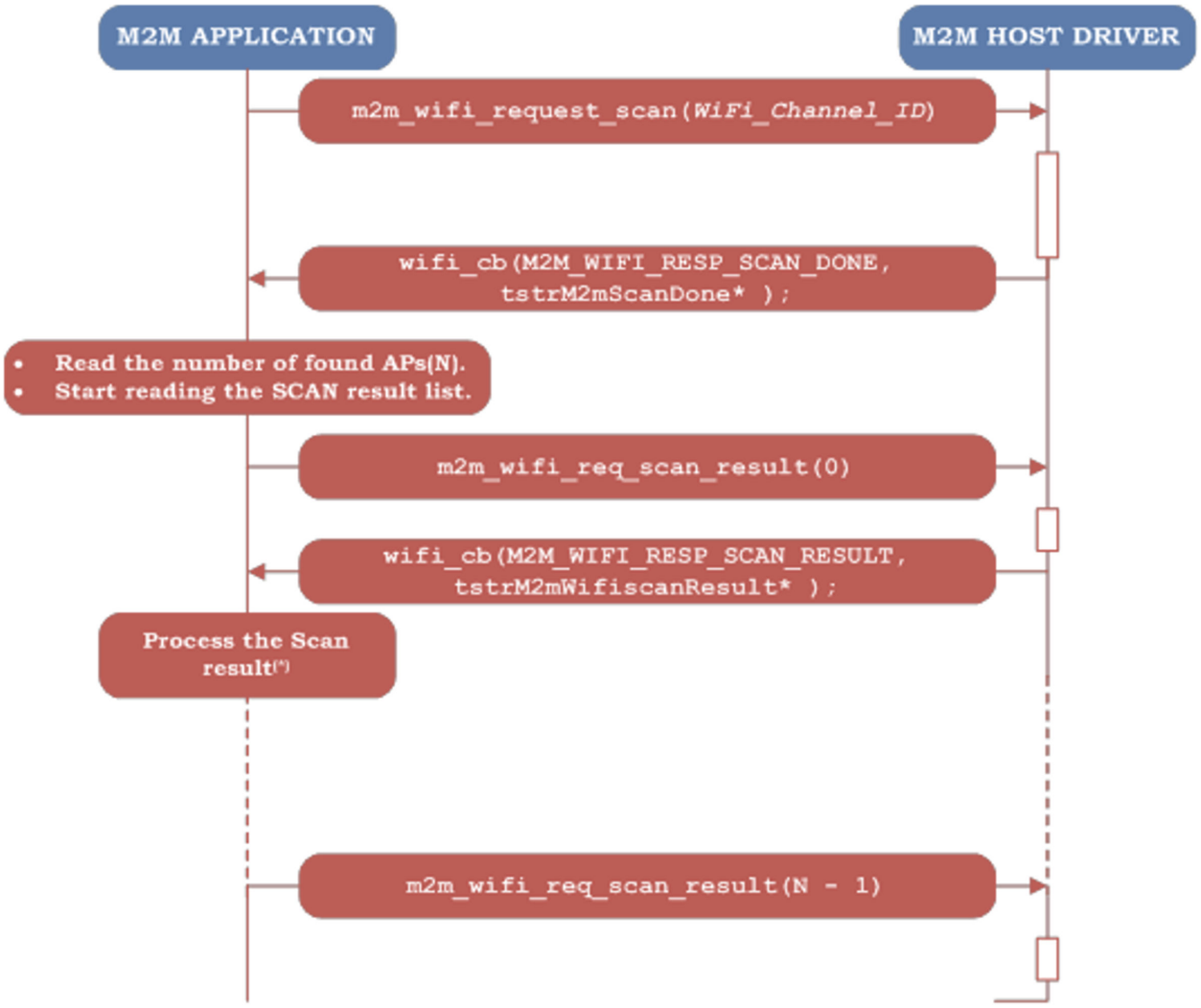
5.2 Wi-Fi Scan

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 which can be set by calling `m2m_wifi_set_scan_options(tstrM2MScanOption* ptstrM2MScanOption)`. For instance, if the host MCU application requests to scan all channels, the scan time is equal to *NoOfChannels (13)* * *`ptstrM2MScanOption->u8NumOfSlot`* * *`ptstrM2MScanOption->u8SlotTime`*.

The scan operation is illustrated in the following figure.

Figure 5-1. Wi-Fi Scan Operation



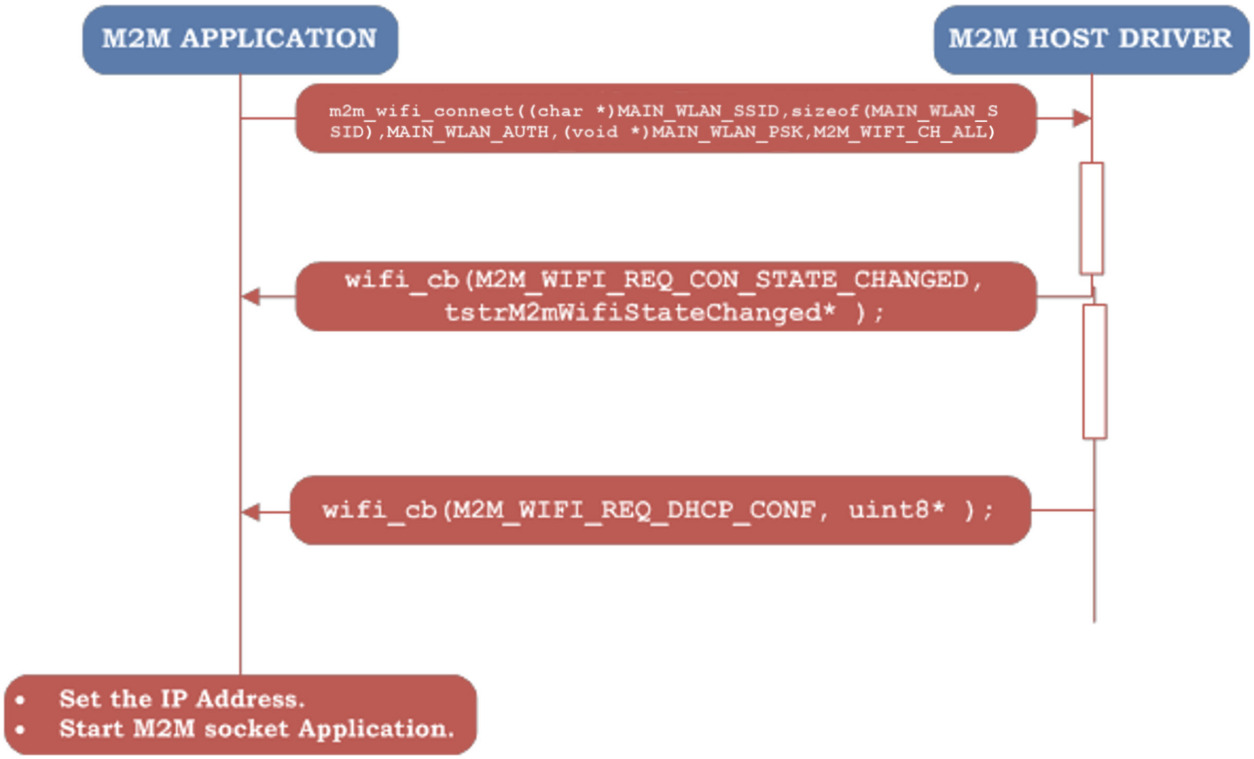
5.3 On Demand Wi-Fi Connection

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

Note: 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 nonvolatile storage attached to the host MCU.

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

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



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 profile provisioned into the WINC serial Flash via the provisioning method described in **Section 10 "Provisioning"**. Alternatively, a connection profile is created and stored in the WINC serial Flash when the host MCU application successfully connects to an AP using the `m2m_wifi_connect` API as described in **Section 5.3 "On Demand Wi-Fi Connection"**. If there is no cached profile or if a connection cannot be established with any of the cached profile, 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.

Note: 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 the following figure.

Figure 5-3. Wi-Fi Default Connection

5.5 Wi-Fi Security

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

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

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

Note: The reader must have a basic understanding of the following topics before reading this chapter:

- BSD sockets
- TCP
- UDP
- Internet protocols

6.1.1 WINC Socket Types

The WINC sockets API provides two types of sockets:

- Datagram sockets (connectionless sockets) – uses the UDP protocol
- Stream sockets (connection-oriented sockets) – uses the TCP protocol

6.1.2 Socket Properties

Each WINC socket is identified by a unique combination of the following:

- Socket ID – a unique identifier for each socket. This is the return value of the socket API.
- Local socket address – a combination of the WINC IP address and port number assigned by the WINC firmware for the socket.
- Protocol – transport layer protocol, either TCP or UDP.
- 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 which is described in the succeeding section.

Note: TCP port 53 and UDP port 53 represent two different sockets.

6.1.3 Limitations

- The WINC sockets API support up to 7 TCP sockets and 4 UDP sockets.
- The WINC sockets API support only IPv4. It does not support IPv6.

6.2 WINC Sockets API

6.2.1 API Prerequisites

- C header file `socket.h` – this includes all the necessary socket API function declarations. When using any WINC sockets API as described in the following sections, the host MCU application must include the `socket.h` header file.
- Initialization – the WINC socket API initializes once before calling any sockets API function. This is done using the `socketInit` API described in **Section 6.2.3 “Socket API Functions”**.

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 the WINC asynchronous APIs are described in **Section 3.4.1 "Asynchronous Events"**.

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 a connection time-out occurred. The socket callback function provides the necessary information to determine the connection status.

6.2.3 Socket API Functions

The WINC sockets API provide the following functions.

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 must 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 only occurs 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.

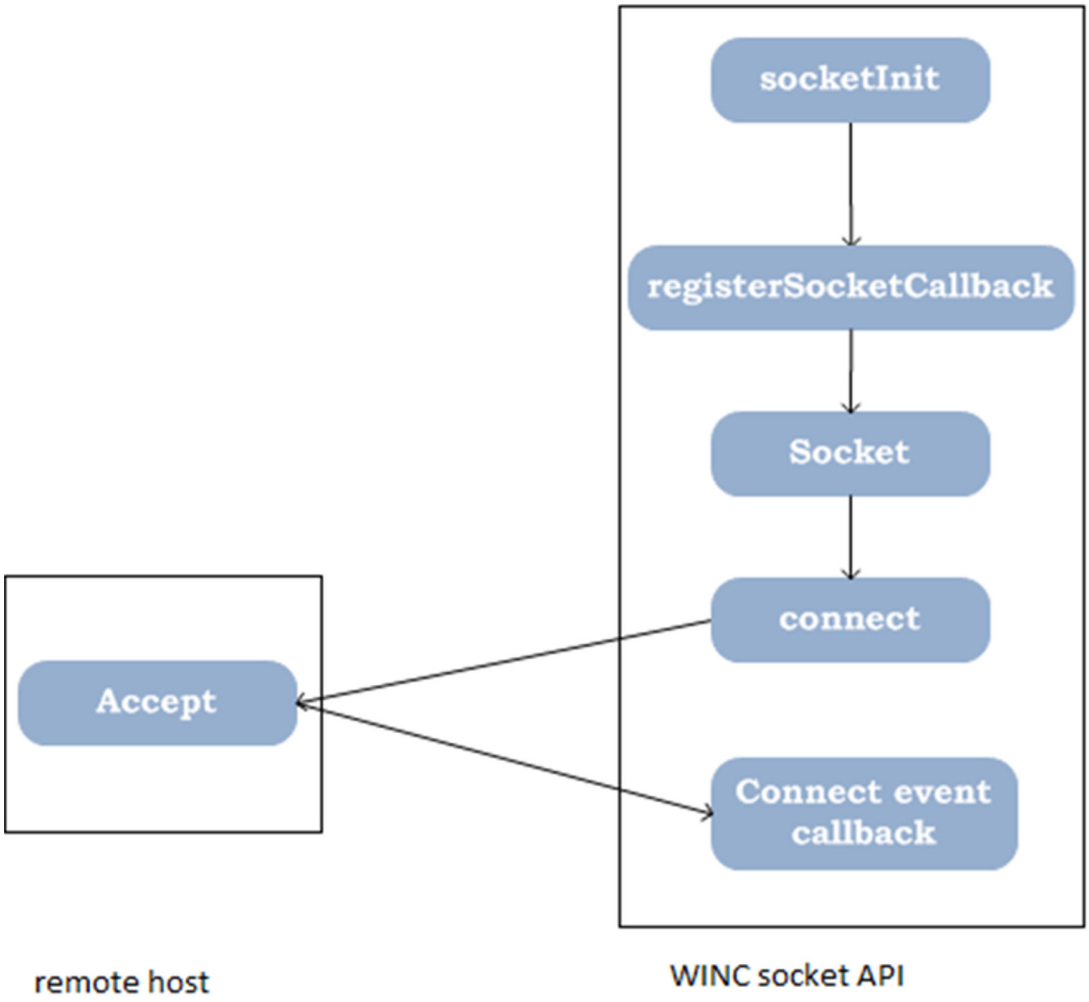
The `connect` function results in a `SOCKET_MSG_CONNECT` sent to the socket event handler callback upon completion. The connect event is sent when the TCP server accepts the connection or, if no remote host response is received, after a time-out interval of approximately 30 seconds.

Note: The `SOCKET_MSG_CONNECT` event callback provides a `tstrSocketConnectMsg` containing an error code. The error code value indicates:

- zero value to indicate the successful connection or
- negative value to indicate an error due to a time-out condition or if `connect` is used with UDP socket.

The following figure 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. It is used to associate a socket with an address structure (port number and IP address).

The bind function call results to a `SOCKET_MSG_BIND` event sent to the socket callback handler with the bind status. Calls to `listen`, `send`, `sendto`, `recv`, and `recvfrom` functions must 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 accepts 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.

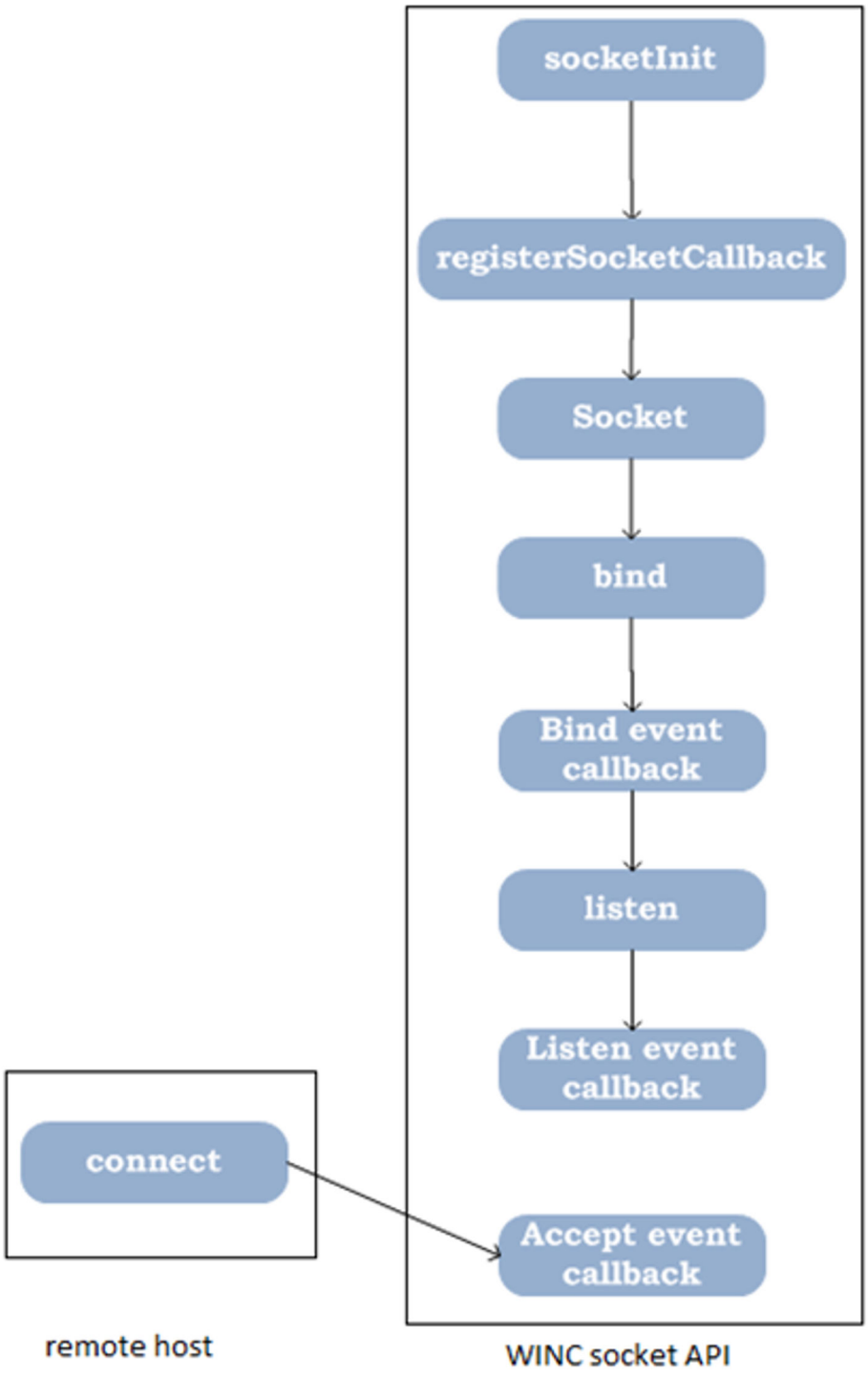
When a remote peer establishes a connection, a `SOCKET_MSG_ACCEPT` event notification is sent to the application.

6.2.3.7 accept

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

Note: The `listen` API implicitly accepts the TCP remote peer connections request.

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 contains 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 recommended way is to use `sendto` API, where the destination address is defined. However, it is possible to use `send` API instead of `sendto` API. For this at least one successful call must be made to `sendto` API prior to the consecutive calls of `send` function. This ensures that the destination address is saved in the WINC firmware.

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

The `SOCKET_MSG_SEND` event callback returns 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, and 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 must have the received data.

The received data size indicates the status as follows:

- Positive – data is received
- Zero – socket connection is terminated
- Negative – 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, the received data is 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 must match with a call to the `close` function. In addition, sockets that are accepted on a server socket port must 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 as follows:

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

Note: Disabling send/sendto callbacks using setsockopts is recommended in high throughput applications.

6.2.3.13 gethostbyname

The gethostbyname function is used to resolve a host name (for example, URL) to a host IP address via the Domain Name System (DNS). This is limited only for IPv4 addresses. The operation depends on the configuration of a DNS server IP address and access to the DNS hierarchy through the internet.

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

Note: An IP returns a zero value to indicate an error (for example, 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

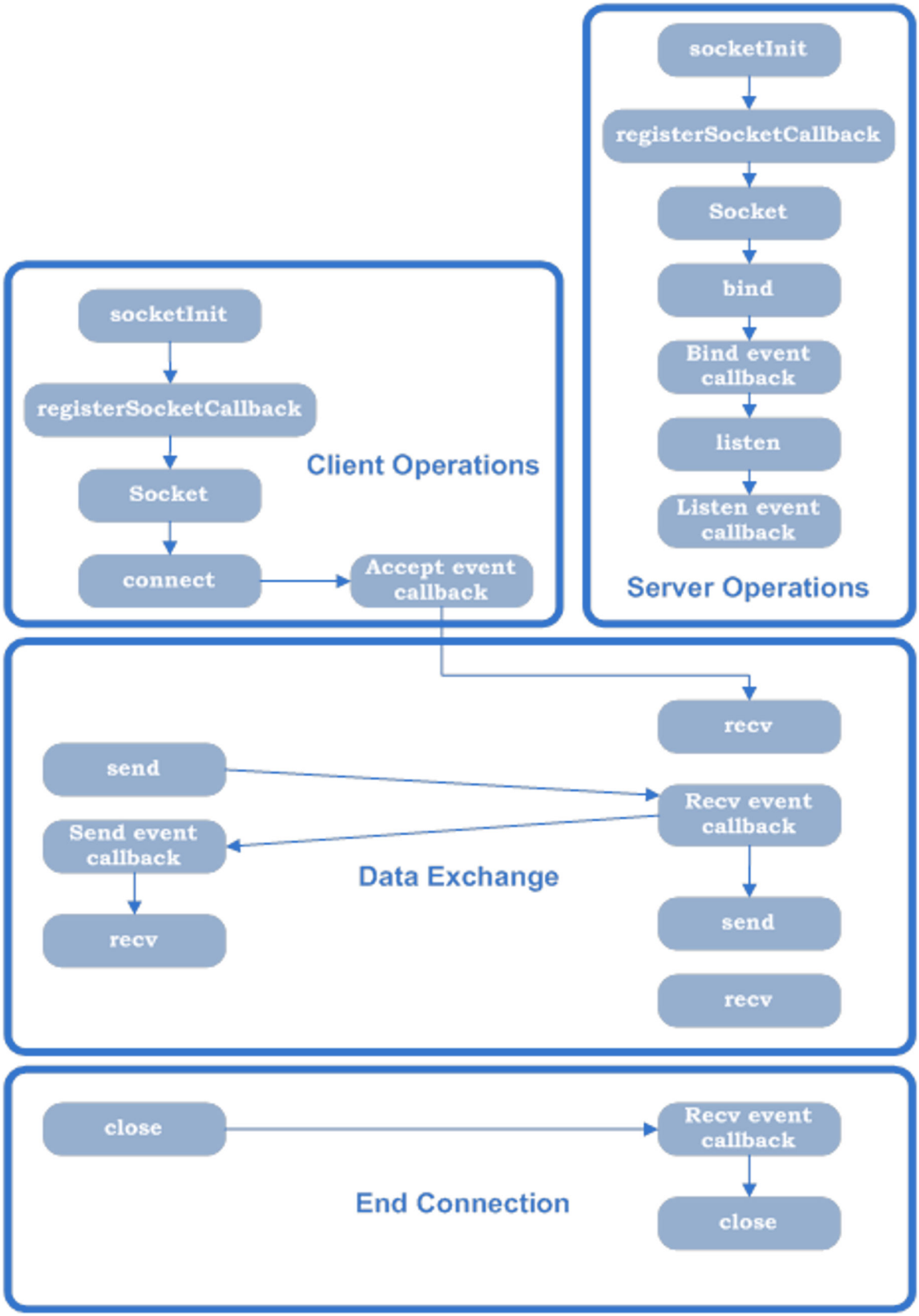
BSD API	WINC API	WINC API Type	Server/ Client	TCP/UDP	Brief
socket	socket	Synchronous	Both	both	Creates a new socket.
connect	connect	Asynchronous	Client	TCP	Initializes a TCP connection request to a remote server.
bind	bind	Asynchronous	Server	both	Binds a socket to an address (address/port).
listen	listen	Asynchronous	Server	TCP	Allows 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	Receives packet.
recvfrom	recvfrom	Asynchronous	Both	Both	Receives packet.
read		Not supported			

BSD API	WINC API	WINC API Type	Server/ Client	TCP/UDP	Brief
close	close	Synchronous	Both	Both	Terminates the TCP connection and release system resources.
gethostbyname	gethostbyname	Asynchronous	Both	Both	Gets the IP address of a 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	Converts 2 byte integer from the host representation to the Network byte order representation (and vice versa).
htonl/ntohl21	_htonl/_ntohl	Synchronous	Both	Both	Converts 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.

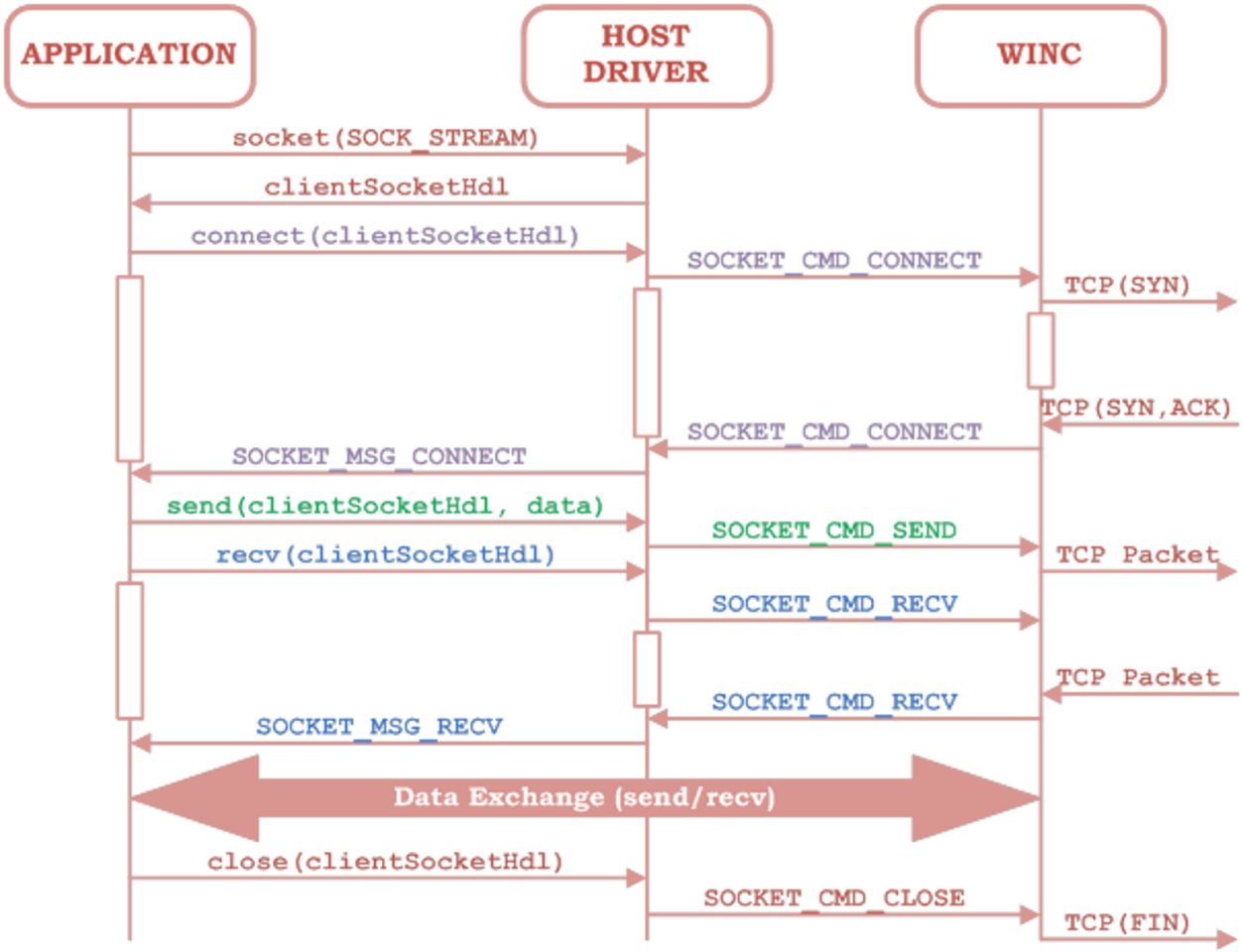
Figure 6-3. Typical Socket Connection Flow



6.3.1 TCP Client Operation

The following figure shows the flow for transferring data with a TCP client.

Figure 6-4. TCP Client Sequence Diagram

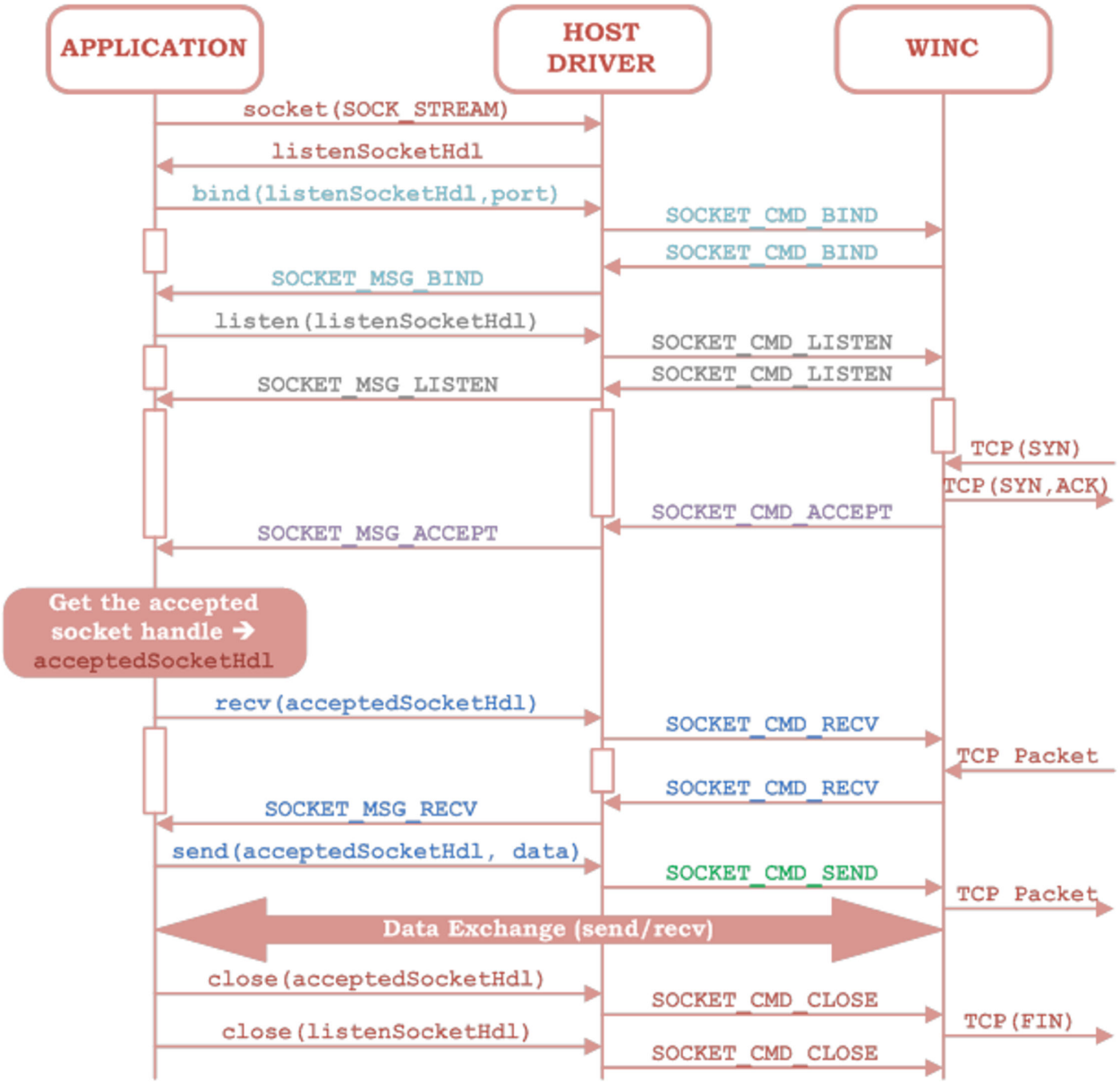


Note:

1. The host application must register a socket notification callback function. The function must be of `tpfAppSocketCb` type 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 the figure above. If only the server URL is known, then the application must resolve the server URL first calling the `gethostbyname` API.

6.3.2 TCP Server Operation

Figure 6-5. TCP Server Sequence Diagram

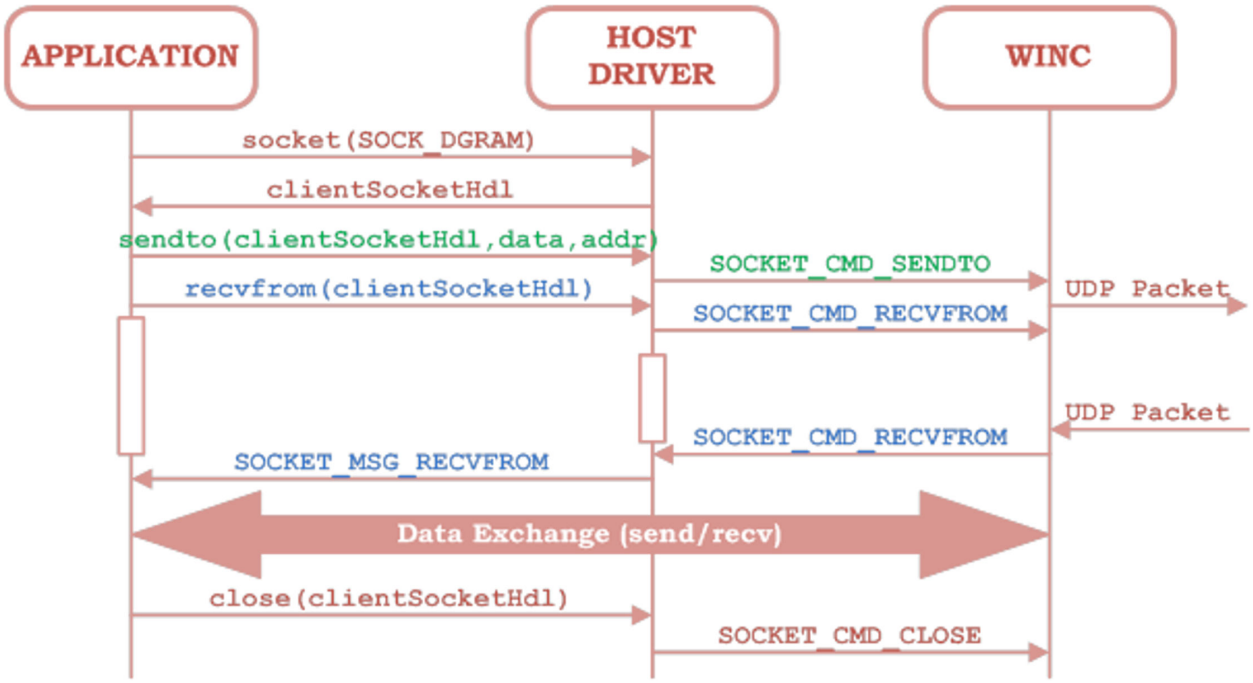


Note: The host application must register a socket notification callback function. The function must be of type `tpfAppSocketCb` and must handle socket event notifications appropriately.

6.3.3 UDP Client Operation

The following figure shows the flow for transferring data with a UDP client.

Figure 6-6. UDP Client Sequence Diagram



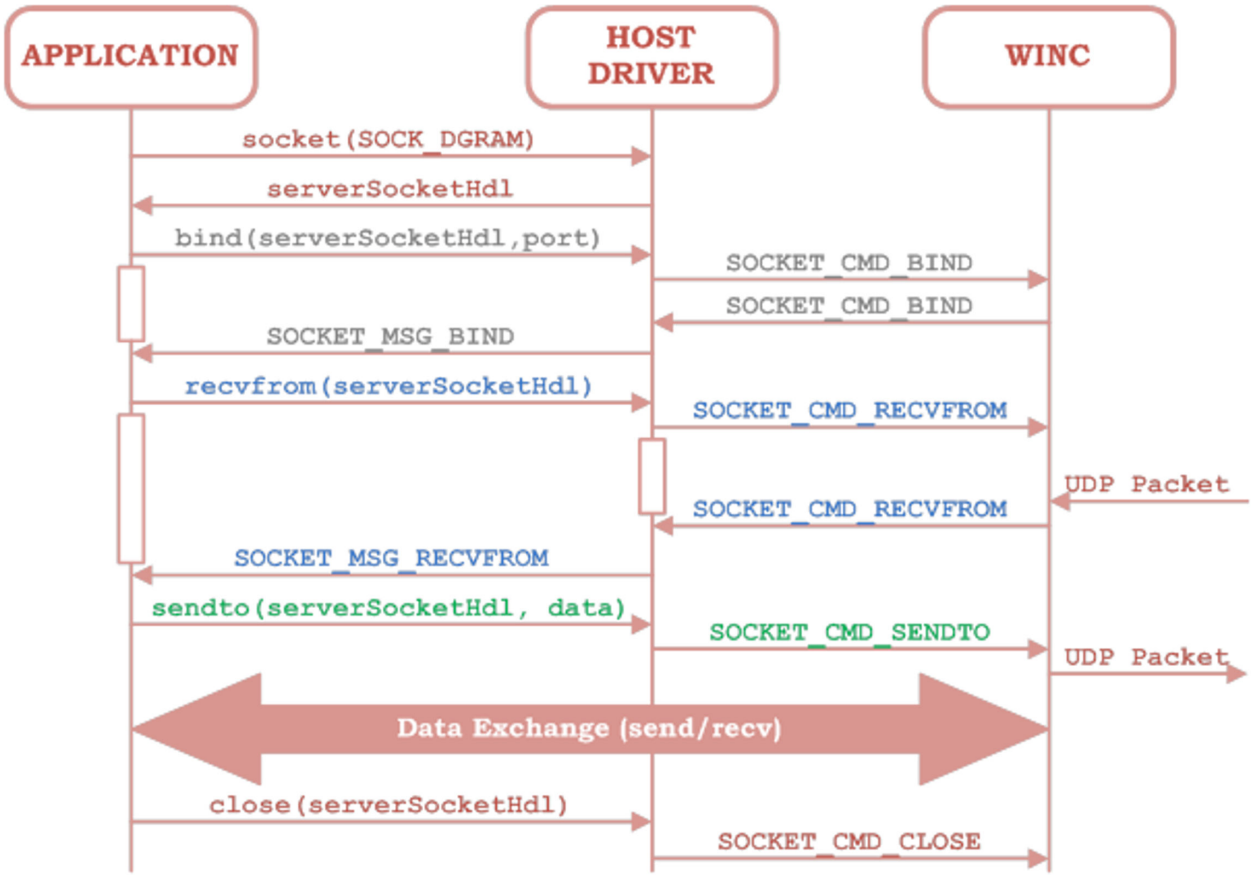
Note:

1. The first send message must be performed with the `sendto` API with the destination address specified.
2. If further messages are sent to the same address, the `send` API can also be used. Refer to **Section 6.2.3.8 "send"** for more details.

6.3.4 UDP Server Operation

The following figure shows the flow for transferring data after establishing a UDP server.

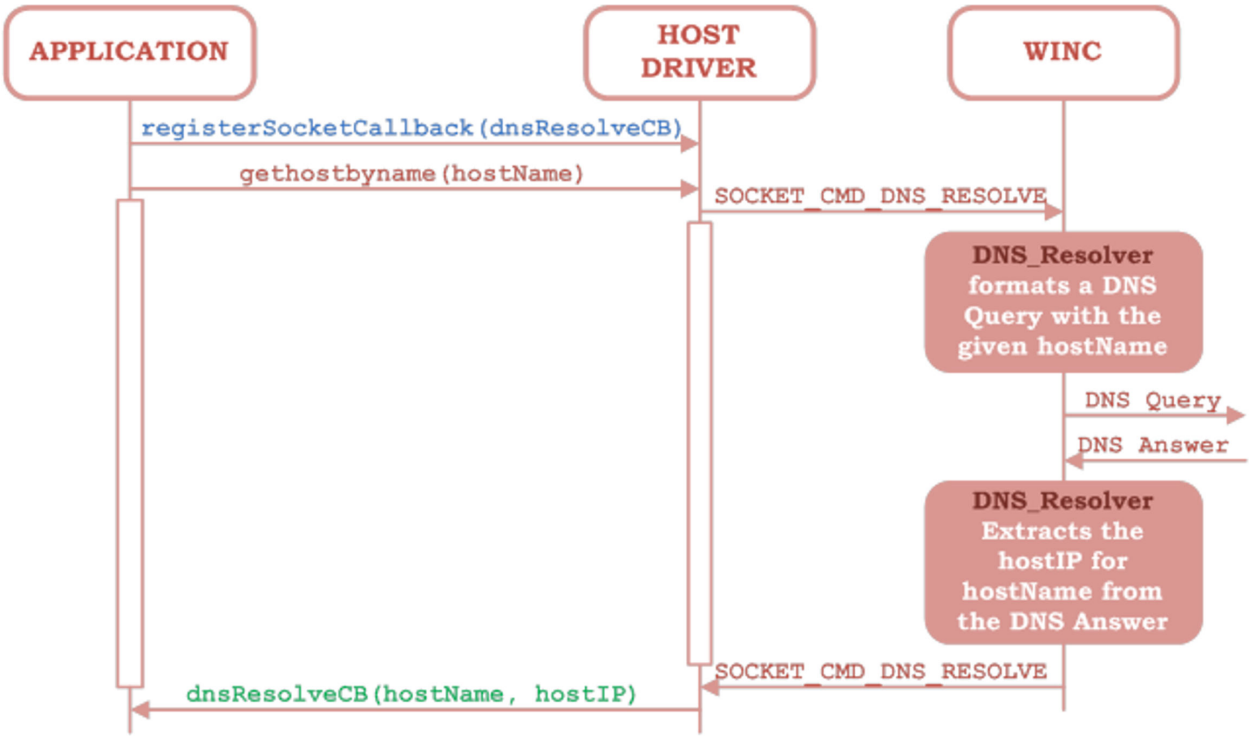
Figure 6-7. UDP Server Sequence Diagram



6.3.5 DNS Host Name Resolution

The following figure shows the flow of DNS host name resolution.

Figure 6-8. DNS Resolution Sequence



Note:

1. The host application requests to resolve hostname (for example, http://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` is called with the hostIP.
4. If an error occurs or if the DNS request encounters a time-out, 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 examples for different socket applications. For additional socket code examples, refer to the ATWINC15x0 Software Programming Guide (DS70005305).

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.
```

```
                uint8    acSendBuffer[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 (TLS) layer sits on top of TCP and provides security services including privacy, authenticity, and message integrity. Various security methods are available with TLS in the WINC firmware.

7.1 TLS Overview

The ATWIN15x0 features an embedded low-memory footprint TLS protocol stack bundled within the WINC firmware.

It features the following functionality:

- Supports TLS versions TLS1.0, TLS1.1 and TLS1.2.
- Supports TLS client operation with TLS client authentication.
- Supports TLS server mode.
- A simple application interface to the TLS stack. The TLS functionality is abstracted by the ATWINC15x0 socket interface, hiding the implementation complexity from the application developer and minimizing the effort to port existing plain TCP code to TLS.

7.2 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 can use the TLS in the same way as the TCP (non-TLS) client and server. The main difference between the TLS sockets and the regular TCP sockets is that the application sets the `SOCKET_FLAGS_SSL` while creating the TLS client and server listening sockets. The detailed sequence of TLS connection establishment is described in the following figure.

Note:

- For proper TLS Client operation, ensure that both `SOCKET_FLAGS_SSL` flag and the correct port number is set in the TLS client application. For instance, an HTTP client application uses no flag 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 in `connect` API to port 433.
- For proper TLS server operation, ensure that both `SOCKET_FLAGS_SSL` flag and the correct port number is set in the TLS server application. For instance, an HTTP server application uses no flag when calling `socket` API function and `bind` to port 80. The same application source code becomes an HTTPS server application, if you use the flag `SOCKET_FLAGS_SSL` and change the port number in `bind` API to port 443.

Figure 7-1. TLS Client Application Connection Establishment

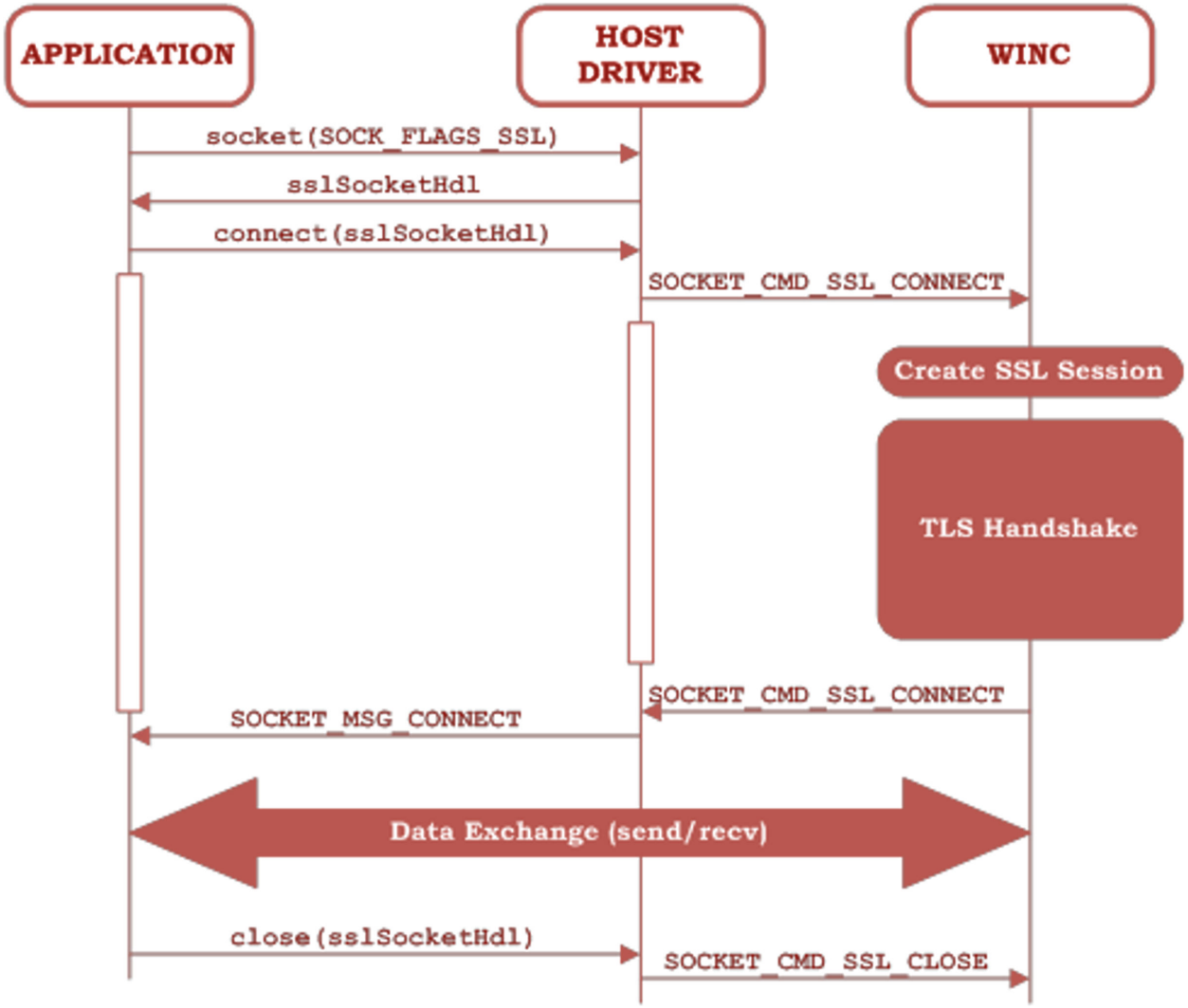
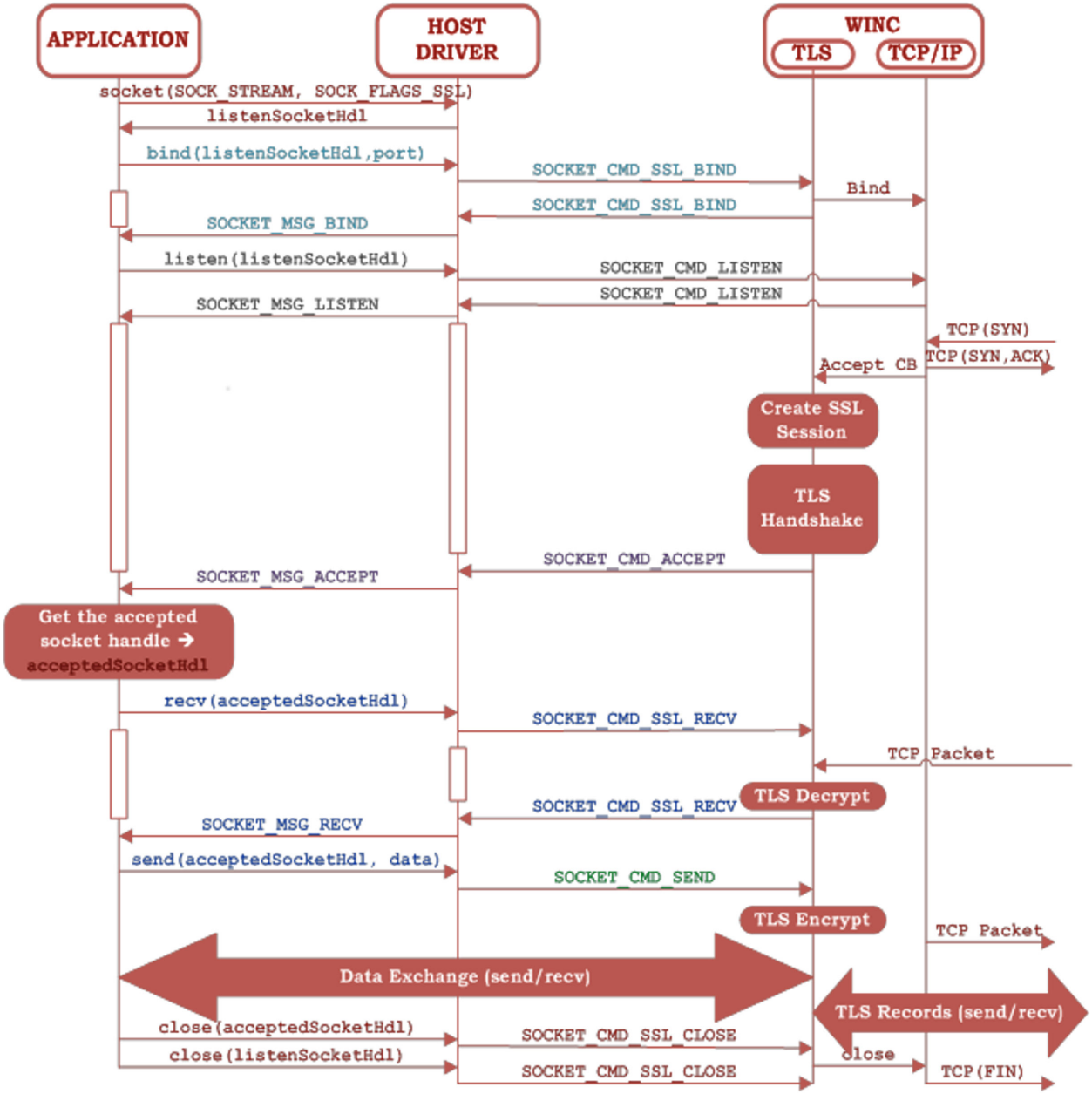


Figure 7-2. TLS Server Application Connection Establishment



7.3 Server Certificate Installation

7.3.1 Technical Background

7.3.1.1 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 may 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.

7.3.1.2 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.

7.3.2 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 ATWINC15x0. If this is not done, the TLS connection to the server is locally aborted by the WINC.
- The root certificate must be in **DER** format. If it is not provided in **DER** format, it must be converted before installation. Refer to **Section 17 "How to Generate Certificates"** 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
```

7.4 WINC TLS Limitations

7.4.1 Concurrent Connections

Only 2 TLS concurrent connections are allowed.

7.4.2 TLS Supported Ciphers

The ATWINC15x0 supports the following cipher suites (for both client and server modes).

- TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_RSA_WITH_AES_128_GCM_SHA256
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA256

The ATWINC15x0 also optionally support the following ECC cipher suites.

- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ ECDSA _WITH_AES_128_CBC_SHA256

7.4.3 Supported Hash Algorithms

The current implementation (WINC firmware version 19.5.2 onwards) supports the following hash algorithms:

- MD5
- SHA-1
- SHA256
- SHA384

- SHA512
- RSA 4096

7.4.4 TLS Certificate Constraints

For TLS server and TLS client authentication, the ATWINC15x0 can accept the following certificate types:

- RSA certificates with key size no more than 2048 bits
- ECDSA certificates only for NIST P256 EC Curve (secp256r1); conditionally supported

7.4.5 ECC Cipher Suite

The ATWINC15x0 TLS library features support of ECC cipher suites. Although, the ATWINC15x0 device does not contain a built-in hardware accelerator for ECC math, the WINC TLS library leverages the ECC math from the host MCU. To perform the ECC computations needed by the ECC ciphers, an ECC hardware accelerator (or software library) on the host MCU is mandatory.

The WINC TLS initializes with the ECC cipher suites disabled by default. The host MCU application can enable the ciphers via the API `sslSetActiveCipherSuites`.

7.5 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 the WINC Access Point (AP) mode and describes how to setup this mode and configure its parameters.

8.2 Setting the WINC AP Mode

Set the WINC AP mode configuration parameters using the `tstrM2MAPConfig` structure.

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

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

For more details on API, refer to the Atmel Software Framework for ATWINC1500 (Wi-Fi).

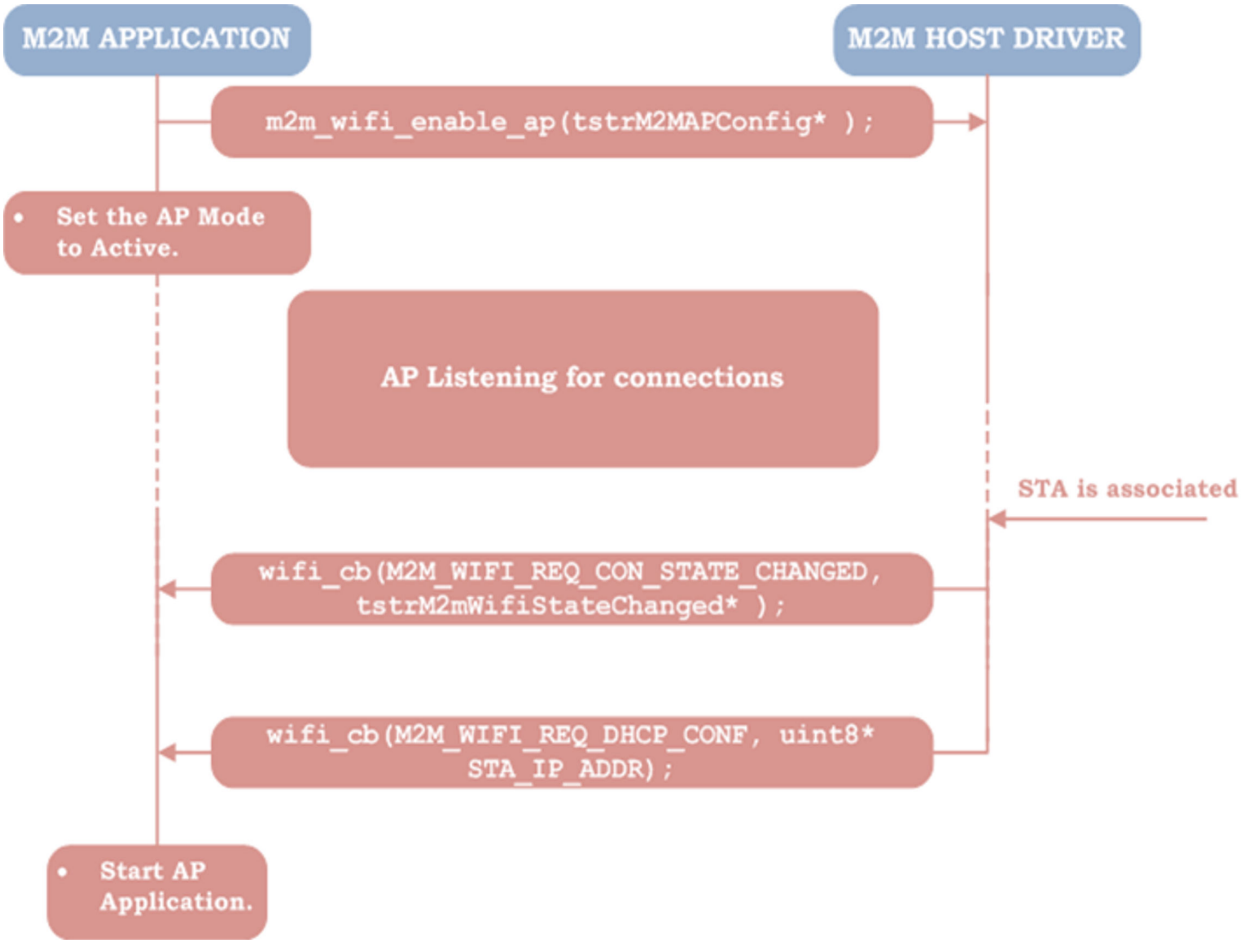
8.3 Limitations

- The AP can only support a single associated station. Further connection attempts are rejected.
- The WINC supports WPA2 security feature starting from the firmware version 19.5.x.
- Concurrency (simultaneous STA/P2P and AP mode) is not supported. Prior to activating the AP mode, the host MCU application must disable the mode that is currently running.

8.4 Sequence Diagram

Once AP mode is established, data interface does not exist before a station associates to the AP; therefore, the application needs to wait until it receives a notification via an event callback. This process is shown in the following figure.

Figure 8-1. WINC AP Mode Establishment



8.5 AP Mode Code Example

The following example shows how to configure the 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

The Wi-Fi Direct or Peer-to-Peer (P2P) allows two wireless devices to discover each other, negotiate on which device acts 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

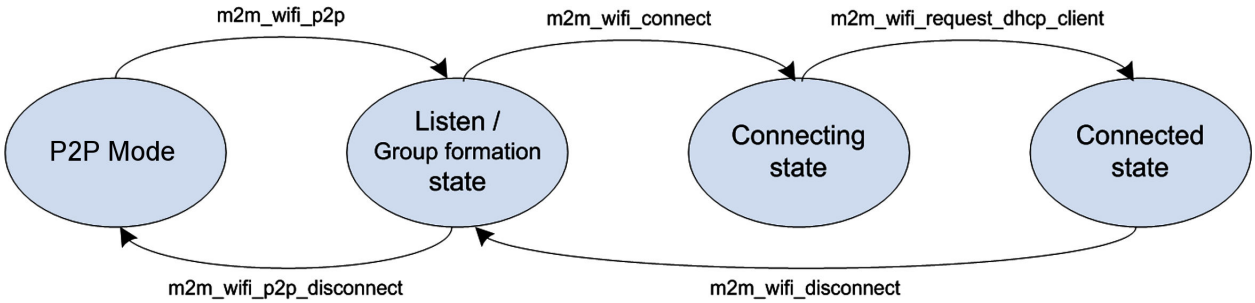
- P2P Client mode is supported
- P2P Device Discovery
- P2P Listen state

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 mode is disabled during P2P mode.
- The 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



The 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 is explained in the following sections.

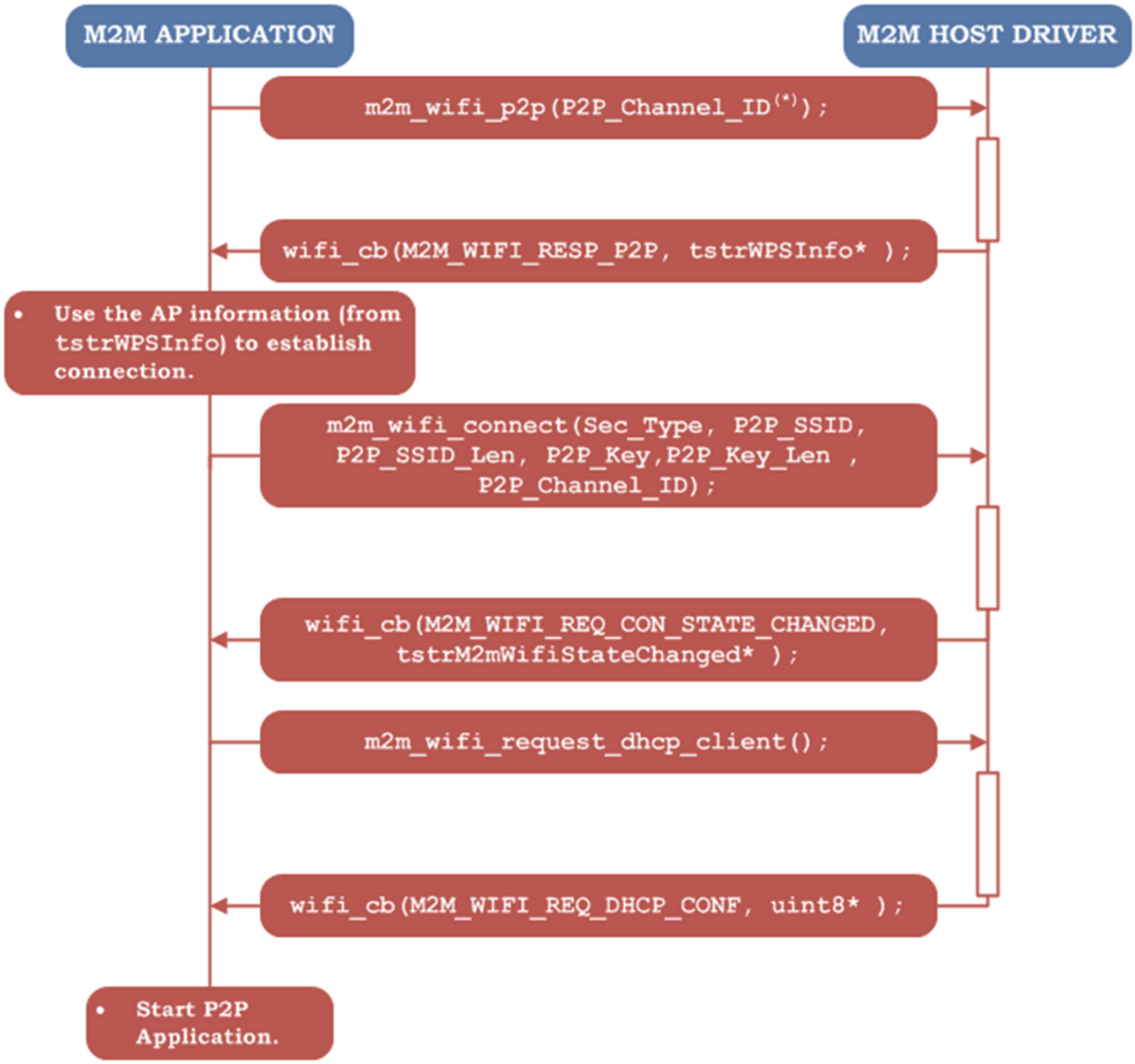
9.5 WINC P2P Listen State

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 initiates group owner (GO) negotiation and the WINC device always declines to become the group owner. Assuming the peer device takes the GO role, the WINC performs 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



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 event does 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 needs to connect. The entry of this information is facilitated through the following provisioning steps.

The current ATWINC15x0 software supports two methods of provisioning:

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

10.1 Limitations

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

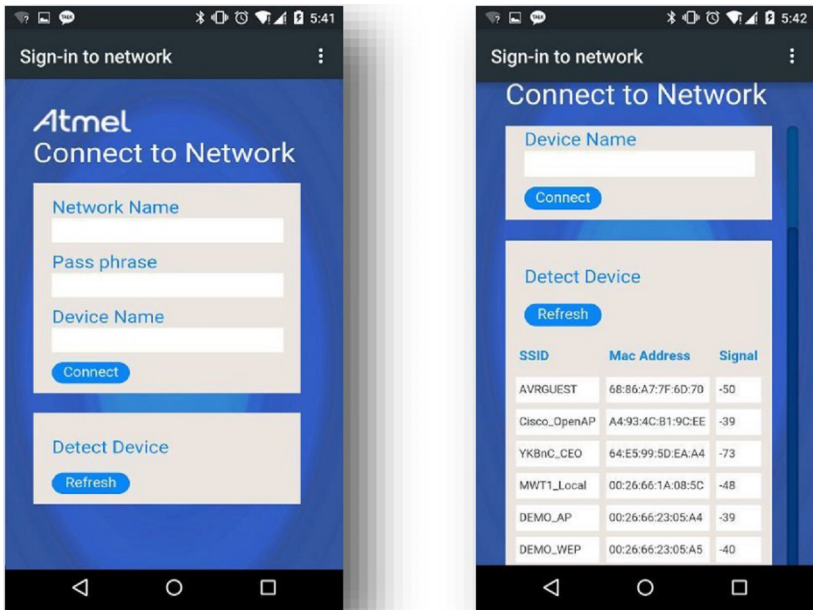
- The WINC AP limitations are applicable to the Provisioning mode. Refer to **Section 8.3 "Limitations"** for a list of AP mode limitations.
- Provisioning uses AP mode with open security. No Wi-Fi security nor application level security (for example, TLS) is used; therefore, the AP credentials entered 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 the 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, and so on) is instructed to connect to the WINC HTTP server. Once connected, the desired configuration can be entered.

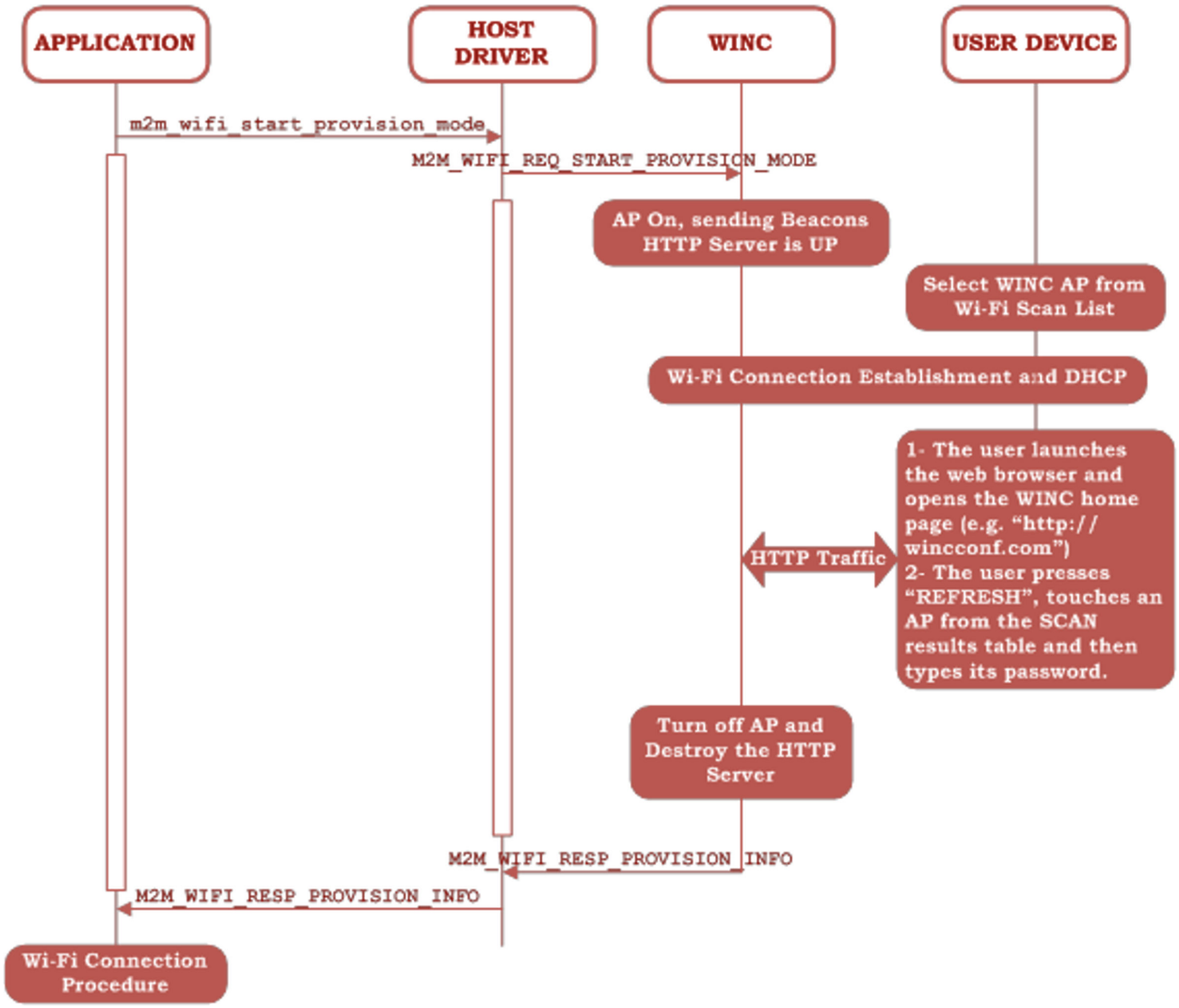
The HTTP Provisioning home page is as shown in the following figure.

Figure 10-1. WINC HTTP Provisioning Page



10.2.1 Provisioning Control Flow

Figure 10-2. HTTP Provisioning Sequence Diagram



The preceding figure 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 smartphone 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 bit (`bEnableHttpRedirect`) is set in `m2m_wifi_start_provision_mode` API, then all http traffic (http://URL) from the associated device (Phone, PC, and so on) are redirected to the WINC HTTP Provisioning home page. Some phones display a notification message "sign in to Wi-Fi networks?" which, when accepted, automatically loads the WINC home pages. The WINC home page, as shown in Figure 10.1, appears 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 automatically appears in the "Network Name" text box.

8. The user must then enter the correct AP passphrase (for WPA/WPA2 personal security) in the “Pass Phrase” text box. If the desired AP uses open security, (M2M_WIFI_SEC_OPEN) then the Pass Phrase field is left empty.
9. A WINC device name may be optionally configured, if desired, by the user in the “Device Name” text box.
10. Then user should press **Connect**.

The WINC turns 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, set the redirect flag when calling the API `m2m_wifi_start_provision_mode`. See the below code example for further 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

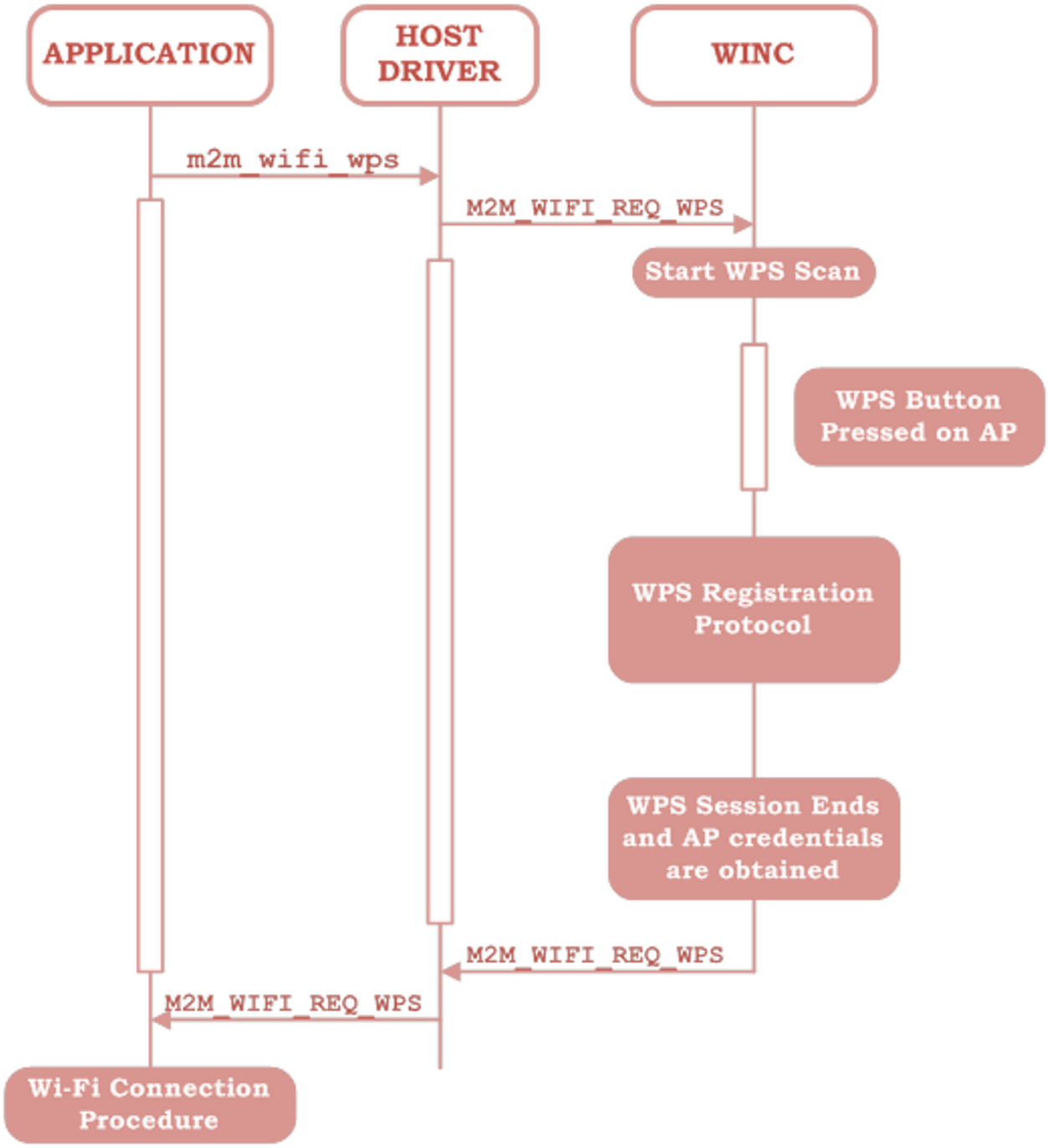
There are two authentication methods that can be used with WPS:

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 ATWINC15x0 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. Since the 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 ATWINC15x0 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 the following figure.

10.3.2 WPS Control Flow

Figure 10-3. WPS Operation for Push Button Trigger



10.3.3 WPS Limitations

- WPS is used to transfer the WPA/WPA2 key only; other security types are not supported.
- The WPS standard rejects the session (WPS response fail) if the WPS button pressed on more than one AP in the same proximity, and the application can try again after couple of minutes.
- If no WPS button is pressed on the AP, the WPS scan will time-out 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.

10.3.4 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

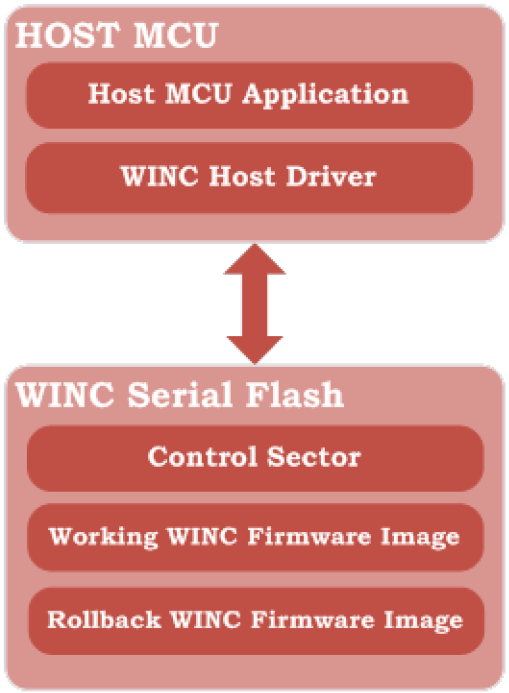
11.1 Overview

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

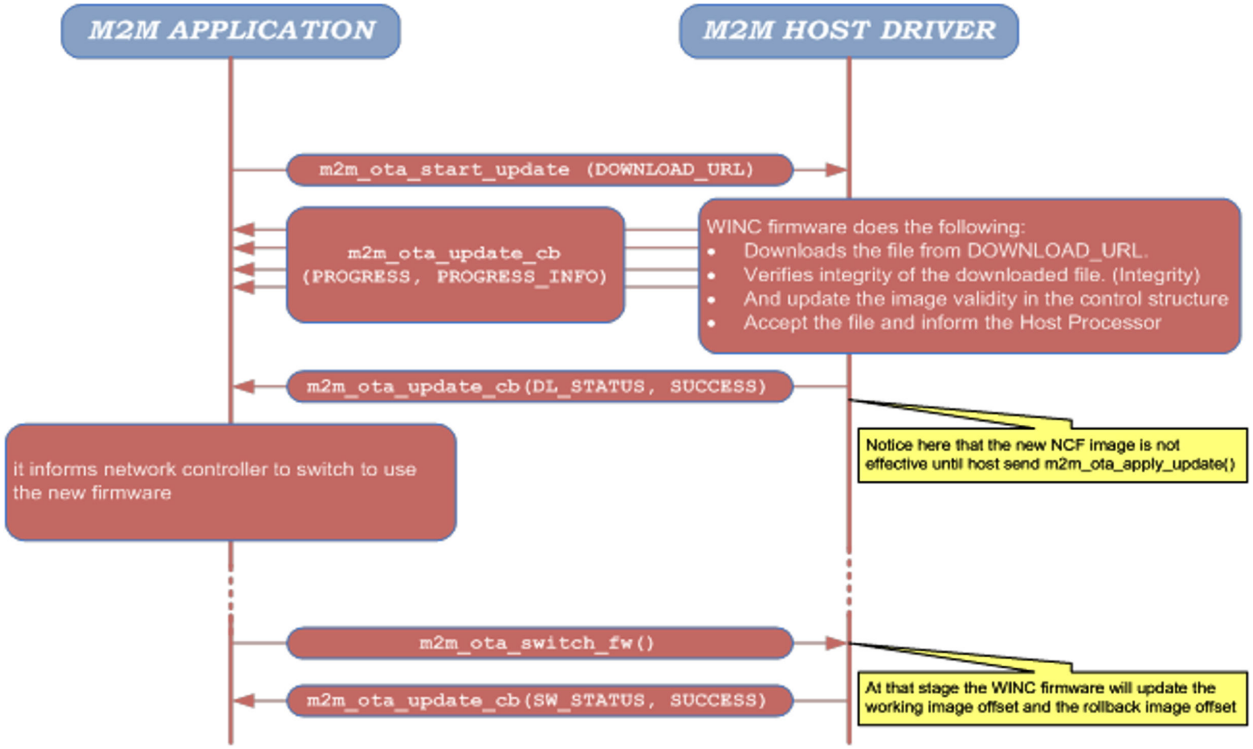
The WINC serial Flash can store two copies of the firmware image: a working image and a rollback image. Upon first-time boot, the working image is the factory image and the rollback image is invalid (empty). The WINC has insufficient internal memory to save the whole image in RAM during an OTA upgrade; therefore, 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



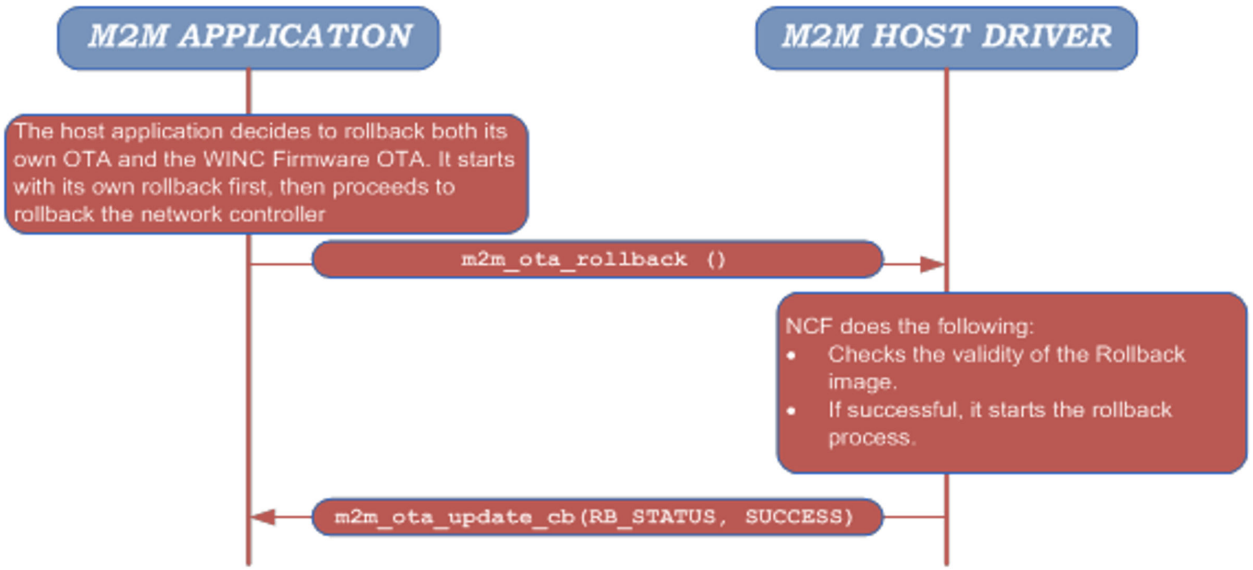
11.3 OTA Download Sequence Diagram

Figure 11-2. OTA Image Download and Install



11.4 OTA Firmware Rollback

Figure 11-3. OTA Image Rollback Sequence



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.

11.6 OTA Code Example

```
/*!<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) {}
    }
}
```

Note: For more details on example codes, refer to the ATWINC15x0 Software Programming Guide (DS70005305).

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, set the `IP_DROP_MEMBERSHIP` option for the required socket accompanied with the multicast address.

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

12.3 Multicast Socket Code Example

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, notifying 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 4Mb capacity in the ATWINC1500 and 8Mb capacity in the ATWINC1510. The Flash memory is used to store:

- User configuration
- Firmware
- Connection Profiles

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

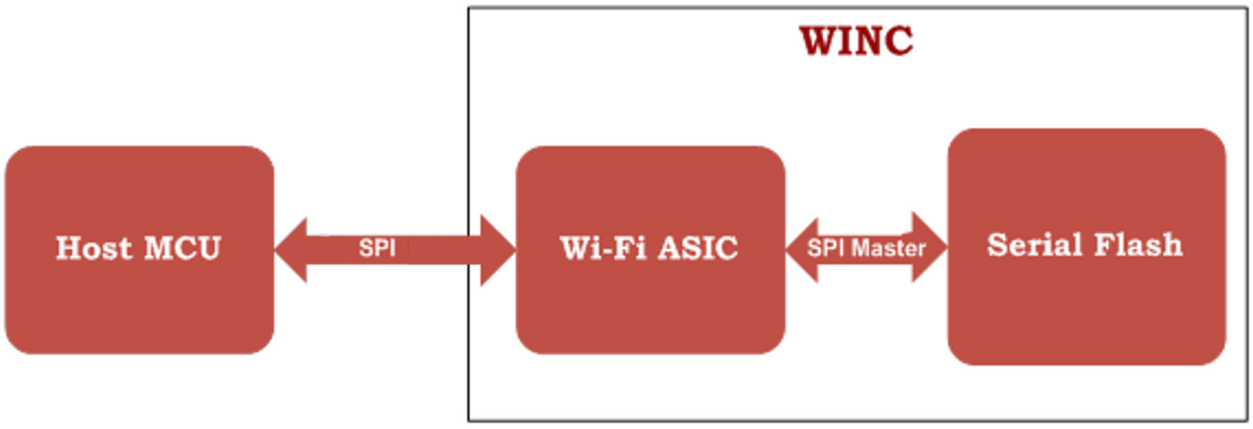
A minimum of 4Mb Flash is required for OTA feature in order to store both working and rollback images.

The Flash memory can be read, written 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 the host and the WINC processor.

13.2 Accessing to Serial Flash

- The host has transparent access to the serial (SPI) Flash through the WINC SPI Master.
- The host can program the serial (SPI) Flash without the 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 the WINC's mode to Download mode first before attempting 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 is successfully completed.

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 the size of the 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 the 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.

Prior writing to any sector, erase this sector first. If some data needs to be changed within a sector, it is advised to read the sector first, modify the data and 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 is written after it returns success by reading data again and compare it with the original.

13.3.1 Flash Read, Erase, and Write Code Examples

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

int main()
{
    uint8    au8FlashContent[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

Note: Sniffer mode is not supported in the firmware version 19.5.x.

14.1 Overview

In Sniffer mode, the ATWINC15x0 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 returns 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, and so on).
- 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 transmits it on the air as it is.

Note: The ATWINC15x0 must be disconnected before activating this mode.

14.2 Sniffer (Monitoring) Mode APIs

- Use the following API function to enable WINC Sniffer mode
  - ```
    sint8 m2m_wifi_enable_monitoring_mode(tstrM2MWifiMonitorModeCtrl   *pstrMtrCtrl,
        uint8 *pu8PayloadBuffer,
        uint16 u16BufferSize,
        uint16 u16DataOffset
       );
    ```
- Use the following API function to disable WINC Sniffer mode
  - ```
    sint8 m2m_wifi_disable_monitoring_mode(void);
    ```
- Use the following API function to send packets in WINC Sniffer mode
  - ```
    sint8 m2m_wifi_send_wlan_pkt(uint8 *pu8WlanPacket,
      uint16 u16WlanHeaderLength,
      uint16 u16WlanPktSize
     );
    ```

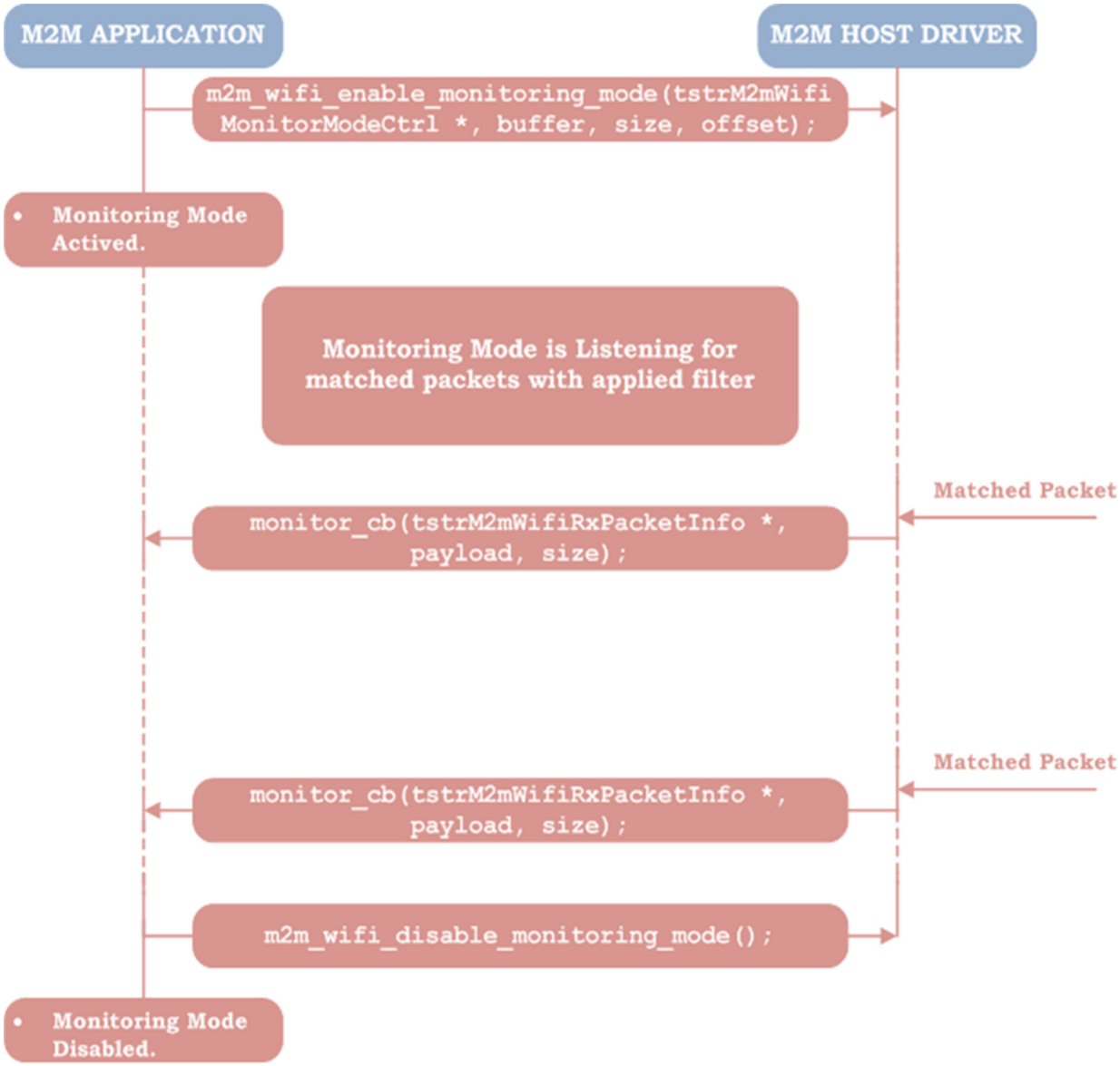
14.3 Monitoring Parameters

To enable monitoring, the application needs to populate a `tstrM2MWifiMonitorModeCtrl` structure type to specify which frames are captured. The other parameters define a buffer into which the captured data is placed. Once monitoring is active, the monitoring call back is 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

The following figure shows the Monitoring mode sequence, which includes enabling/disabling the mode and receiving matched packets.
    ```

Figure 14-1. Monitoring Mode Sequence Diagram



14.5 Code Example

```
#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;
}
```

Note: For more details on example codes, refer to the ATWINC15x0 Software Programming Guide (DS70005305).

15. Host Interface (HIF) Protocol

Communication between the user application and the WINC device is facilitated by the driver software. This driver implements the Host Interface (HIF) 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 is established and similar to BSD socket definitions.

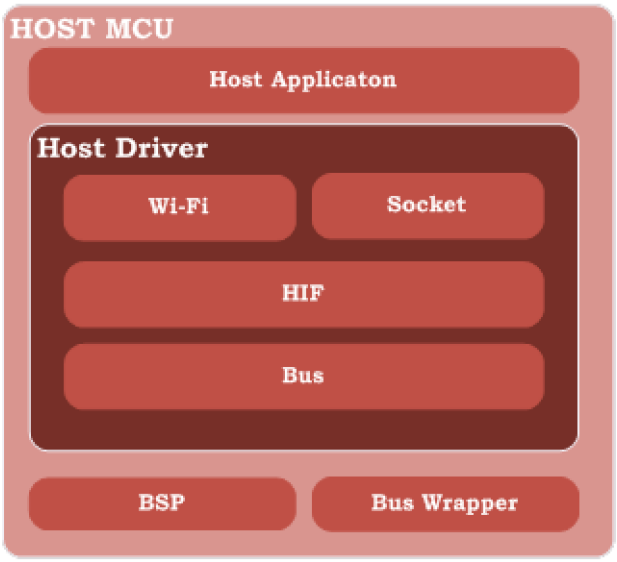
The host driver implements services asynchronously. This means that when the application calls an API to request a service action, the call is non-blocking and returns immediately, often before the action is completed. Where appropriate notification that an action has completed is provided in a subsequent message from the WINC device to the host which is delivered to the application via a callback function. In general, 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.

Note: Dealing with HIF messages in the 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 to notify that a new request is posted. Upon receipt, the WINC firmware parses the request and starts the required operation.

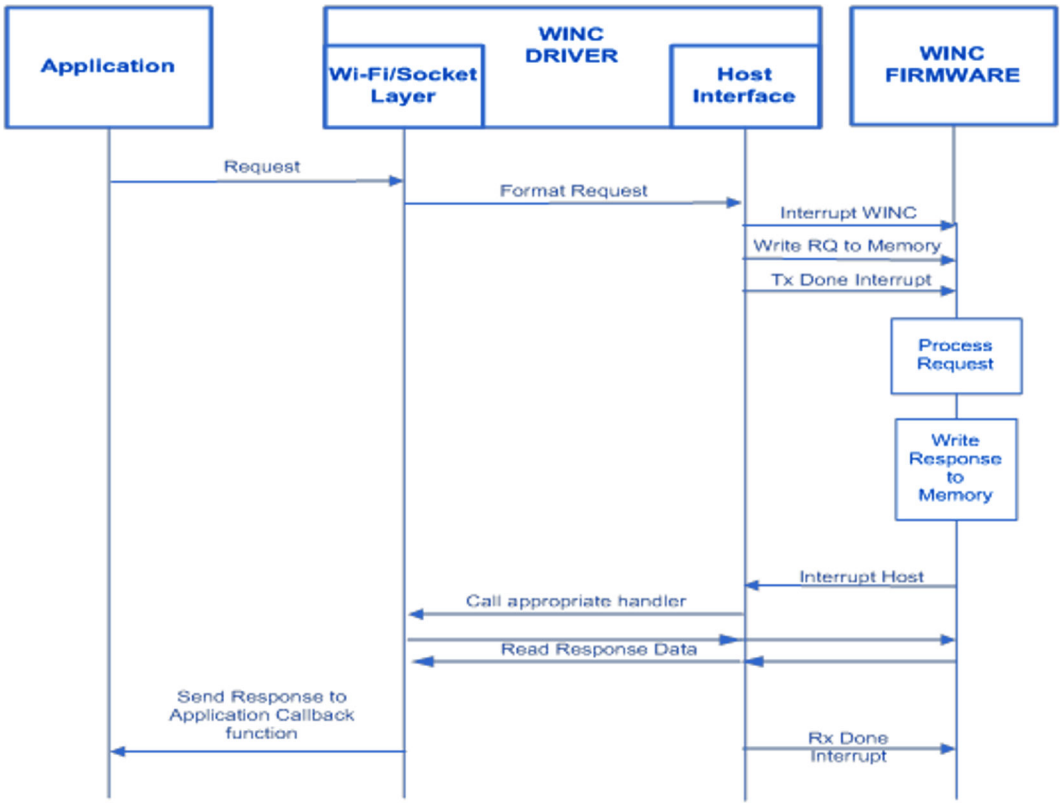
Figure 15-1. WINC Driver Layers



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 the firmware driver at host and the WINC firmware.

The Request/Response sequence between the host and the WINC chip is shown in the following figure.

Figure 15-2. The Request/Response Sequence Diagram



15.1 Transfer Sequence Between the HIF Layer and the 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).

15.1.1 Frame Transmit

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

Figure 15-3. HIF Frame Transmit to WINC

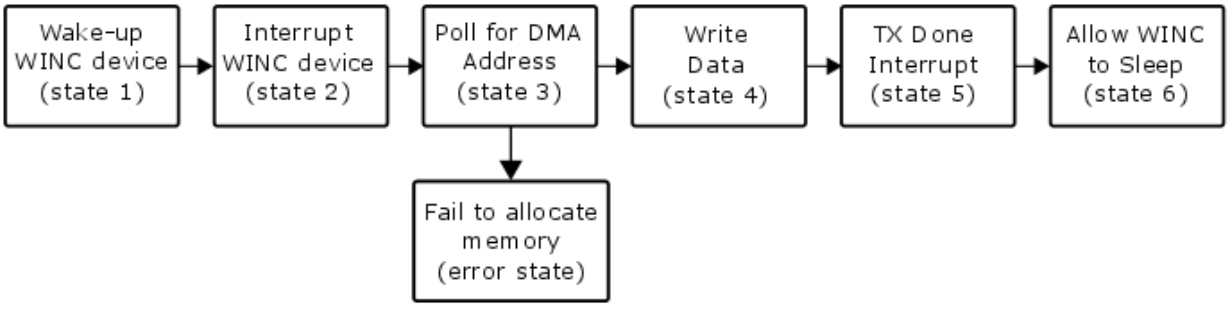


Table 15-1. Steps in HIF Frame Transmit to WINC

Step	Description
Step (1) Wake-up the WINC device	Wake-up the device to be able to receive the 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	Send a notification that writing the data is completed by setting BIT [1] of WIFI_HOST_RCV_CTRL_3 register.
Step (6) Allow the WINC device to Sleep	Allow the WINC device to enter Sleep mode again (if it wishes).

15.1.2 Frame Receive

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

Figure 15-4. HIF Frame Receive from WINC to Host

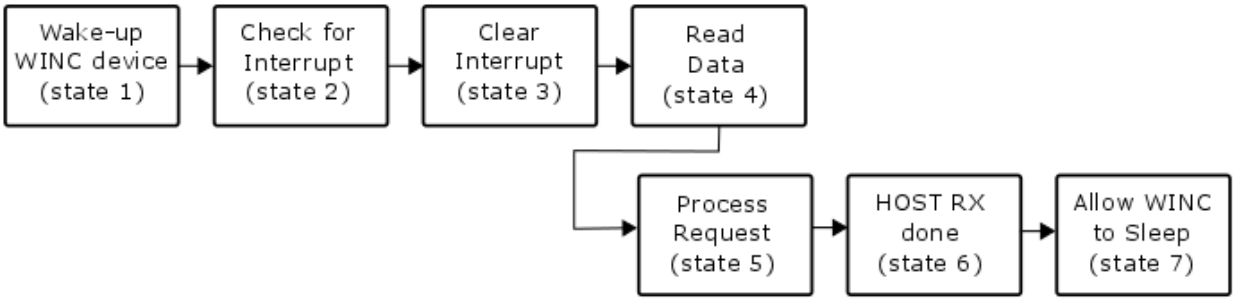


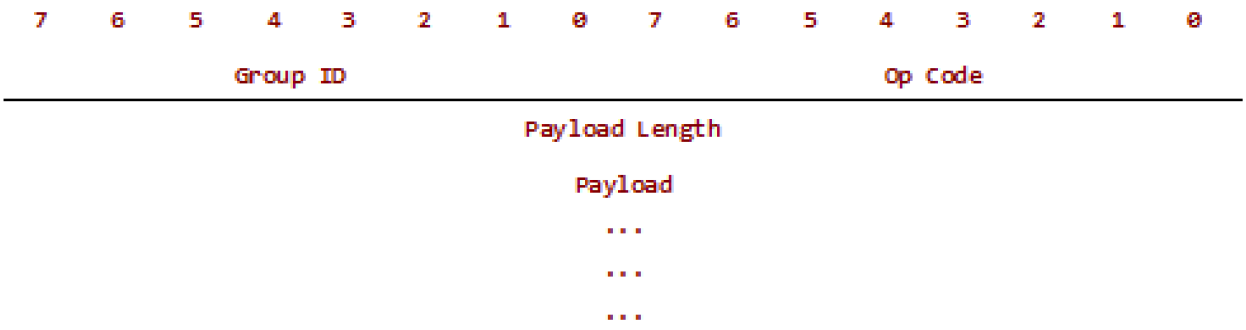
Table 15-2. Steps in HIF Frame Receive from WINC to Host

Step	Description
Step (1) Wake-up the WINC device	Wake-up 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 interrupt is 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.

Step	Description
	Read data block with size obtained from WIFI_HOST_RCV_CTRL_0 register BIT [13] <-> BIT [2].
Step (5) Process Request	Parse the HIF header at the start of the data and forward the data to the appropriate registered Callback function.
Step (6) HOST RX Done	Raise an interrupt for the chip to free the memory holding the data by setting BIT [1] of WIFI_HOST_RCV_CTRL_0 register. Enable host interrupt reception again.
Step (7) Allow the WINC device to Sleep	Allow the WINC device to enter Sleep mode again (if it wishes).

15.2 HIF Message Header Structure

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



- The Group ID (8-bit) – a group ID is the category of the message. Valid categories are enumerated in `tenuM2mReqGroup`.
- 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.
  Note: Refer to the `m2m_types.h` for the full list of commands.
- Payload Length (16-bit) – the payload length is shown in bytes (does not include header).

15.3 HIF Layer APIs

The interface between the application and the driver is 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
- hif_set_sleep_mode
- hif_get_sleep_mode

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 clockless register access. It sets bit '1' of register 0x01 and sets the value of WAKE_REG register to WAKE_VALUE.
- sint8 hif_chip_sleep (void) – this function enables Sleep mode for the 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 (for example, M2M_WIFI, M2M_HIF, M2M_OTA and so on.). A callback is registered by upper layers to receive specific events of a specific message group.
- sint8 hif_isr (void) – this is the host interface interrupt service routine. It handles interrupts generated by the WINC chip and parses the HIF header to call back the appropriate handler.
- sint8 hif_receive (uint32 u32Addr, uint8 *pu8Buf, uint16 u16Sz, uint8 is Done) – 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 is typically extracted from an earlier part of a transaction.
- sint8 hif_send (uint8 u8Gid, uint8 u8Opcode, uint8 *pu8CtrlBuf, uint16 u16CtrlBufSize, uint8 *pu8DataBuf, uint16 u16DataSize, uint16 16DataOffset) – this function causes the host driver to send data to the WINC chip. The WINC chip must be prepared for reception according to the flow described in the previous section.
- void hif_set_sleep_mode (uint8 u8Pstype) – this function is used to set the Sleep mode of the HIF layer.
- uint8 hif_get_sleep_mode (void) – this function return the Sleep mode of the HIF layer.

15.4 Scan Code Example

The following code example illustrates the Request/Response flow on a Wi-Fi Scan request.

Note: For more details on example codes, refer to the ATWINC15x0 Software Programming Guide (DS70005305).

- 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 the 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);
        }
    }
}
```

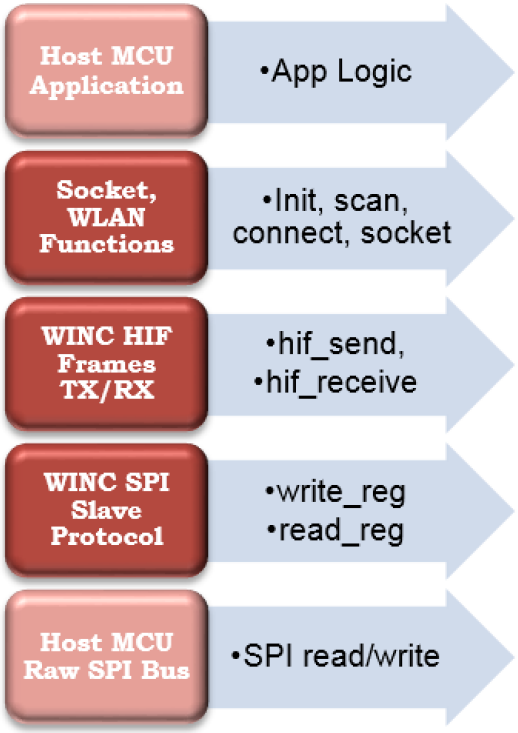
16. WINC SPI Protocol

The WINC main interface is SPI. The WINC device employs a protocol to allow exchange of formatted binary messages between the WINC firmware and the 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 – the WINC SPI Slave protocol, which allows the host MCU application to perform register/ memory read and write operation in the ATWINC15x0 device using raw SPI data exchange.
- Layer 2 – the host MCU application uses the register and memory read and write capabilities to exchange the host interface frames with the WINC firmware. It also provides asynchronous callback from the WINC firmware to the host MCU through interrupts and the host interface RX frames. For more information on this layer, refer to **Section 15 "Host Interface (HIF) Protocol"**.
- Layer 3 – allows the host MCU application to exchange high level messages (for example, Wi-Fi scan, socket connection, or TCP data received) with the WINC firmware to employ in the host MCU application logic.

Figure 16-1. WINC SPI Protocol Layers



16.1 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

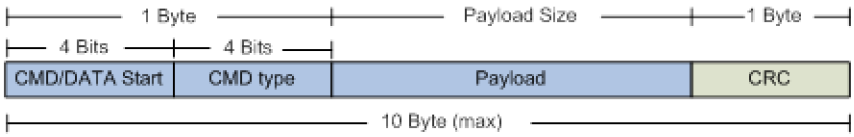
In the case of Command and Data messages, the last byte is used as data integrity check.

The format of Command and Response and Data frames are 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 sends a response for each data packet.
- For a RD transaction (*Master is receiving data packets*), the Master does not send a response. If there is an error, the Master requests a retransmission on the lost data packet.
- Protection of commands and data packets by CRC is optional.

16.1.1 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:

Table 16-1. Frame Format Fields

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 **Type A (length 3)** ***1- DMA single-word RD*** *Param: Read Address:* *Payload bytes:* B0: ADDRESS[23:16]

Field	Size	Description
		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 **Type B (length 5)** ***1- DMA RD Transaction*** *Params:* DMA Start Address: 3 bytes DMA count: 2 bytes *Payload bytes:* B0: ADDRESS[23:16]

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

Field	Size	Description
		B1: ADDRESS[15:8] B2: ADDRESS[7:0] B3: COUNT[23:16] B4: COUNT[15:8] B5: COUNT[7:0] ***3- Internal register WR**** *Params:* Offset address: 3 bytes Write data: 3 bytes * “clocked or clockless registers” *Payload bytes:* B0: OFFSET-ADDR[15:8] B1: OFFSET-ADDR [7:0] B2: DATA[31:24] B3: DATA [23:16] B4: DATA [15:8] B5: DATA [7:0] **Type D (length 7)** ***1- DMA single-word WR*** *Params:* 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’

Field	Size	Description
		bits 1-7: 7 bit CRC value computed using polynomial G(x) = X^7 + X^3 + 1 with seed value: 0x7F

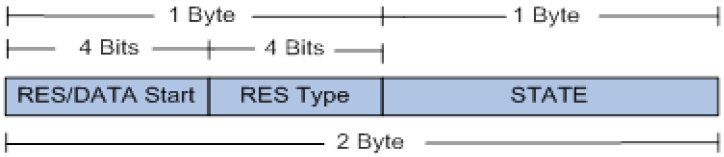
The following table summarizes the different commands according to the payload type (DMA address = 3 bytes):

Table 16-2. Commands in Payload

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

16.1.2 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:

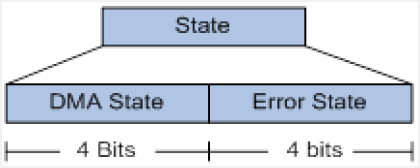
- B0-B3: Error state
- B4-B7: DMA state

States that may be indicated are:

- DMA state:
  - DMA ready for any transaction
  - DMA engine is busy
- Error state:
  - No error

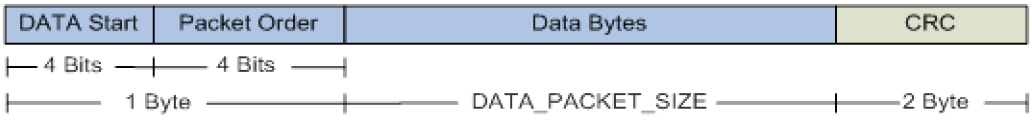
– Unsupported command
– Receiving unexpected data packet
– Command CRC7 error

Table 16-3. Response Format

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

16.1.3 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 the

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

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

Where,

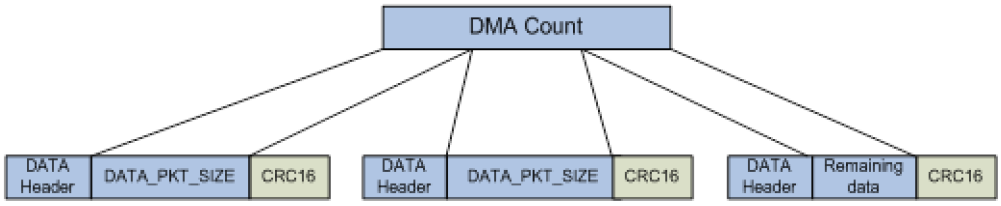
1.. n-1 = length of the DATA_PACKET_SIZE

n = frame length

This is illustrated below.

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

- If DMA count > DATA_PACKET_SIZE:



- If remaining data < DATA_PACKET_SIZE, the last data packet is:
  "DATA_Header + remaining data + optional CRC16 ", that is no padding.

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

Table 16-4. Frame Field

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

16.1.4 Error Recovery Mechanism

Table 16-5. Error Recovery Mechanism

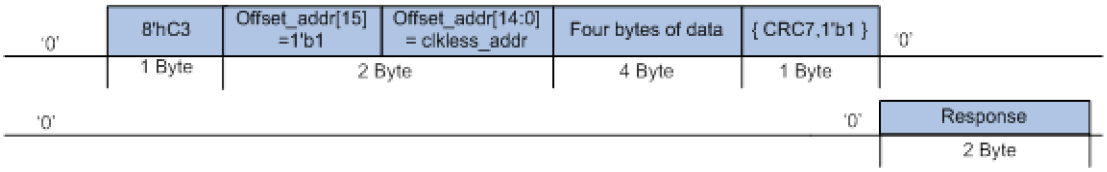
Error Type	Recovery Mechanism
Master	
CRC error in command	1. Error response received from Slave. 2. Retransmit the command.
CRC error in received data	1. Issue a repeat command for the data packet that has a CRC error. 2. Slave sends a response to the previous command. 3. Slave keeps the start DMA address of the previous data packet, so it can retransmit it. 4. Receive the data packet again.
No response is received from Slave	• Synchronization is lost between the Master and Slave. • The worst case is when Slave is in receiving data state. • Solution: The Master must wait for max DATA_PACKET_SIZE period then generate a Soft Reset command.
Unexpected response	Retransmit the command.
TX/RX Data count error	Retransmit the command.
No response to Soft Reset command	• Transmit all ones until Master receives a response of all ones from the Slave. • Then deactivate the output data line.
Slave	
Unsupported command	• Send response with error. • Returns to command monitor state.
Receive command CRC error	• Send response with error. • Wait for command retransmission.
Received data CRC error	• Send response with error. • Wait for retransmission of the data packet.
Internal general error	• The Master must do a Soft Reset on the Slave.
TX/RX Data count error	• Only the Master can detect this error. • Slave operates with the data count received until the count finishes or the Master terminates the transaction. • In both cases, the Master can retry the command from the start.
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 must activate the output data line.

Error Type	Recovery Mechanism
	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 Notes	• The Slave must monitor the received line for command reception at any time. • When a CMD start is detected, the Slave receives 8 bytes then return again to the command reception state. • When the Slave is transmitting data, it must also monitor for command reception. • When the Slave is receiving data, it monitors for command reception between the data packets. • Issuing a Soft Reset command is detected in all cases.

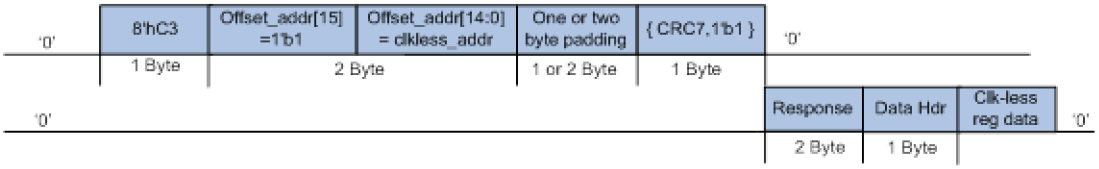
16.1.5 Clockless Registers Access

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 must be ‘`1`’ to differentiate between the Clockless and Clocked access mode.

For Clockless register **write**: - the protocol Master must wait for the response as shown here:



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

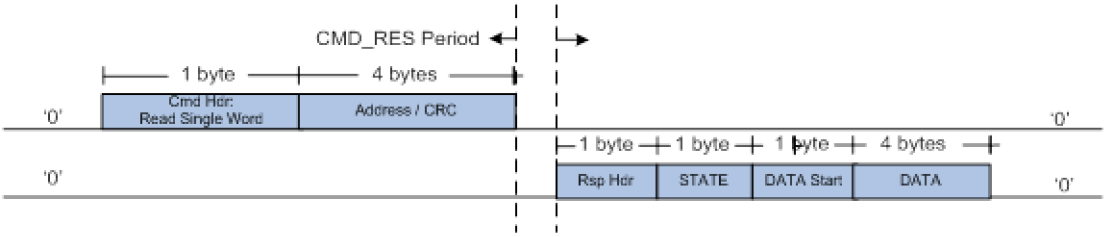


16.2 Message Flow for Basic Transactions

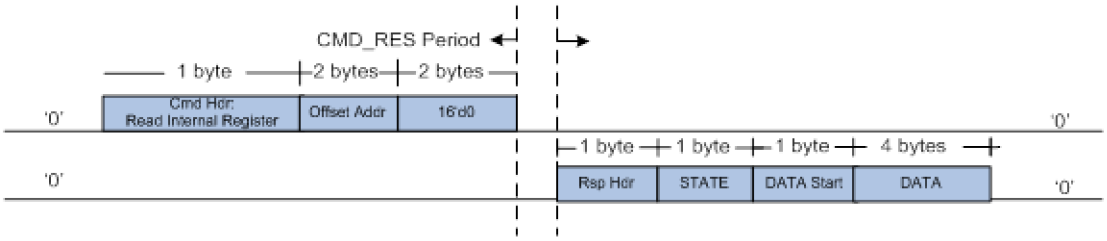
This section shows the essential message exchanges and timings associated with the following commands:

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

16.2.1 Read Single Word



16.2.2 Read Internal Register (for clockless registers)



16.2.3 Read Block

Normal transaction:

Master – issues a DMA read transaction and waits for a response.

Slave – sends a response after CMD_RES_PERIOD.

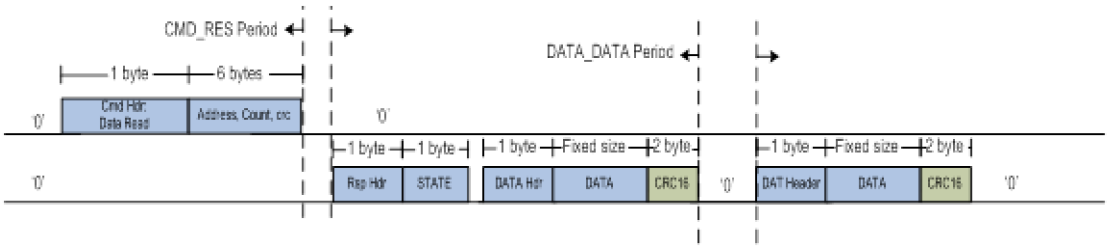
Master – waits for a data packet start.

Slave – sends the data packets, separated by DATA_DATA_PERIOD[1] 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 until the count ends.

Master – receives 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:



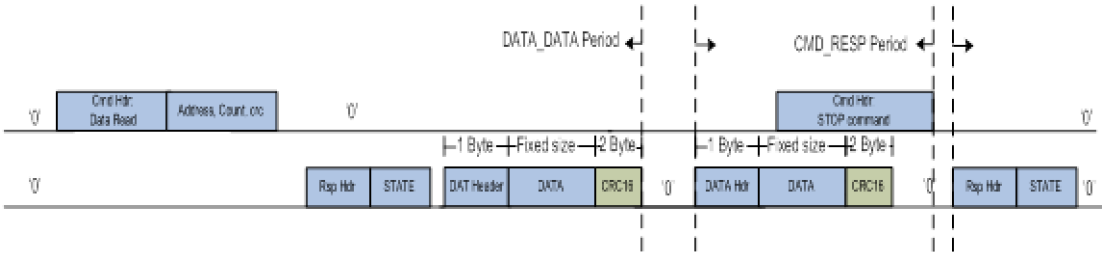
Termination command is issued:

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

Master – monitors for RES_START after CMD_RESP_PERIOD.

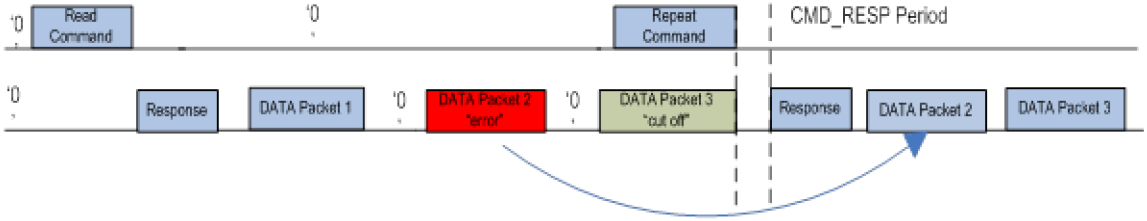
Slave – cuts off the current running data packet if there is any.

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



Repeat command is issued:

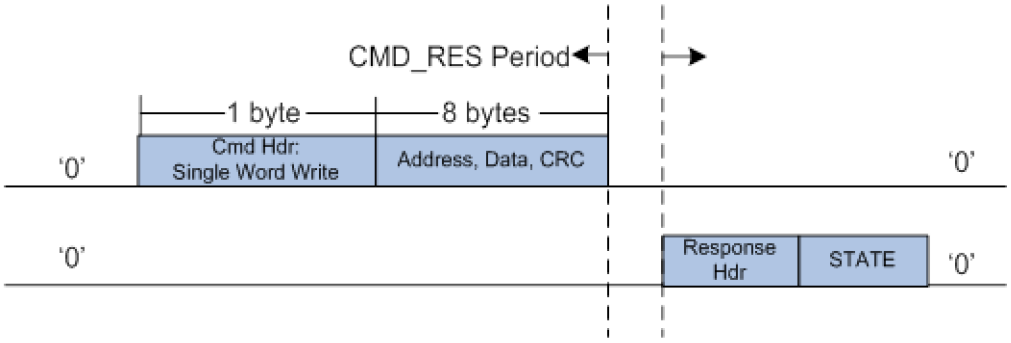
1. Master – can issue a repeat command at any time during the transaction.
2. Master – monitors for RES_START after CMD_RESP_PERIOD.
3. Slave – cuts off the current running data packet, if any.
4. Slave – responds to the repeat command after CMD_RESP_PERIOD from the end of the repeat command packet.
5. Slave – sends the data packet again that has an error then continues the transaction as normal.



[1] The period between the data packets is “DATA_DATA_PERIOD + DMA access time.” The Master monitors for DATA_START directly after DATA_DATA_PERIOD.

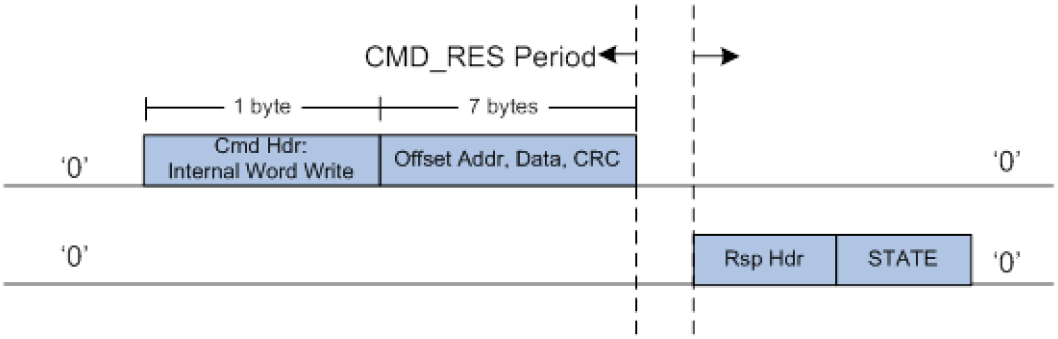
16.2.4 Write Single Word

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



16.2.5 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.



16.2.6 Write Block

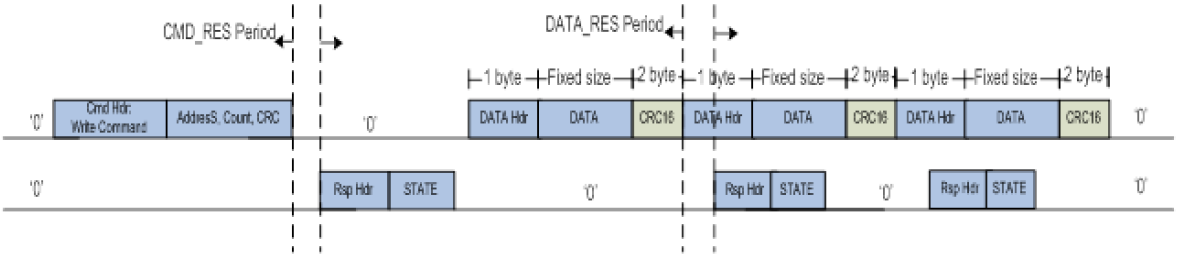
- **Case 1: Master waits for a command response:**
  - 1.1. Master – issues a DMA write command and waits for a response.
  - 1.2. Slave – sends response after CMD_RES_PERIOD.
  - 1.3. Master – sends the data packets after receiving response.
  - 1.4. Slave – sends a response packet for each data packet received after DATA_RES_PERIOD.
  - 1.5. 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

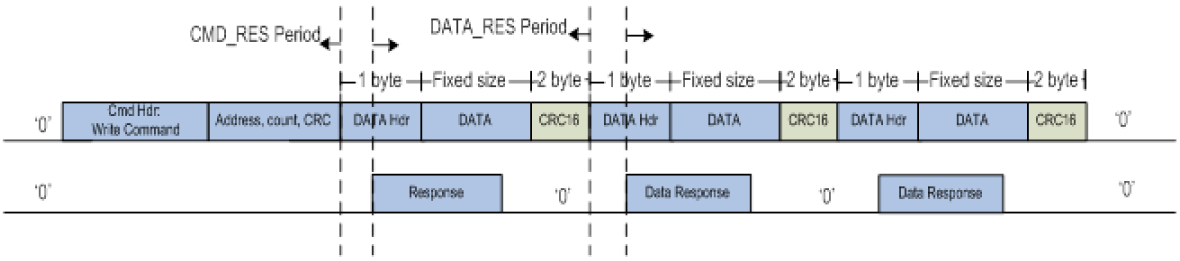
    The Master must monitor for RES_START after CMD_RES_PERIOD

    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:**
  - 2.1. Master – sends the data packets directly after the command but it still monitors for a command response after CMD_RESP_PERIOD.
  - 2.2. Master – retransmits the data packets if there is an error in the command.



16.3 SPI Level Protocol Example

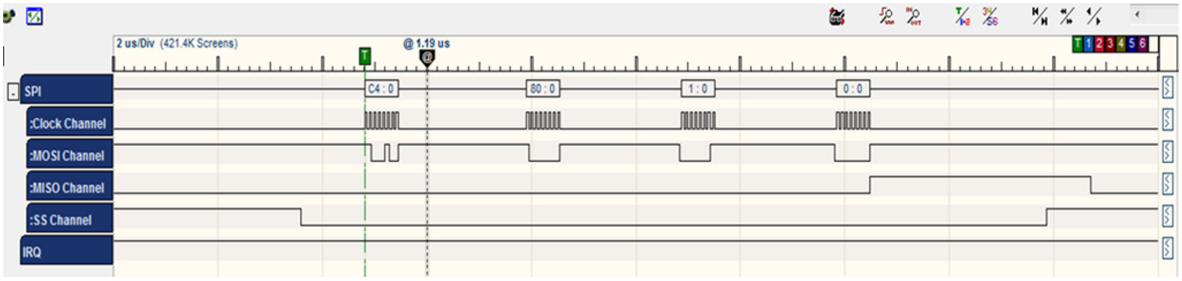
To illustrate how the WINC SPI protocol works, the SPI bytes from the scan request example are dumped and the sequence is described below.

16.3.1 TX (Send Request)

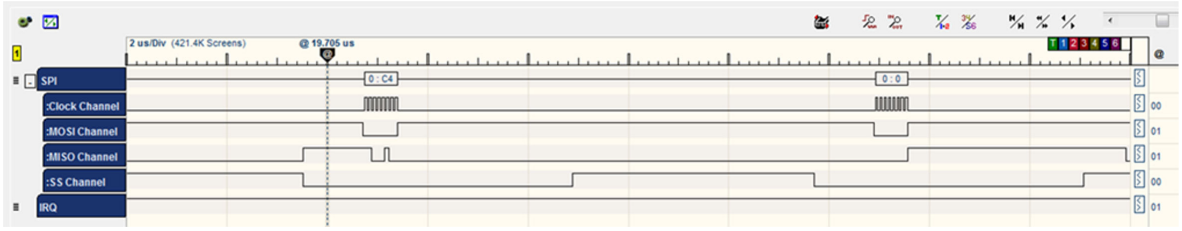
1. First step in `hif_send() API` is to wake-up the chip.

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

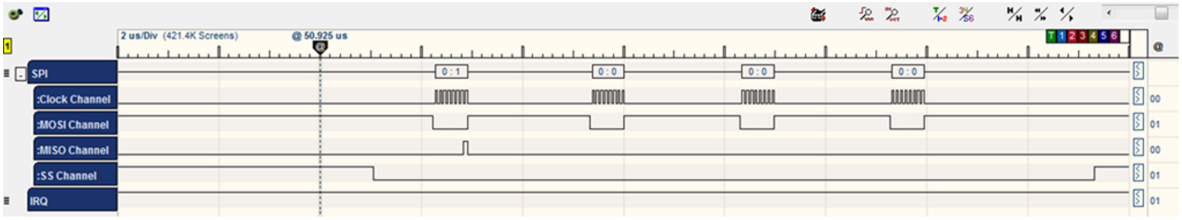
```
Command         CMD_INTERNAL_READ:    0xC4    /* internal register read */
            BYTE [0] = CMD_INTERNAL_READ
               BYTE [1] = address >> 8;            /* address = 0x01 */
            BYTE [1] |= (1 << 7);                /* clockless register */
            BYTE [2] = address;
  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



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

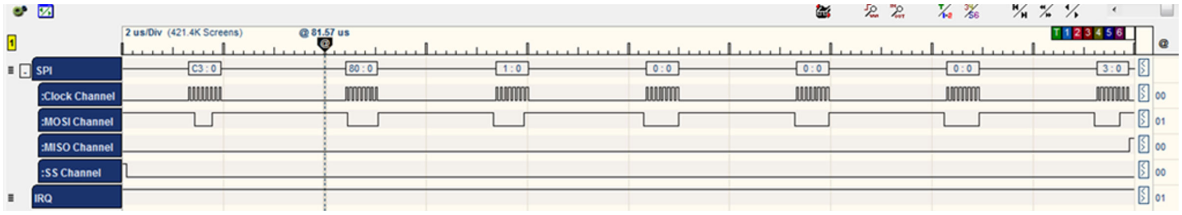


3. The WINC chip sends the value of the register 0x01 which equals 0x01.

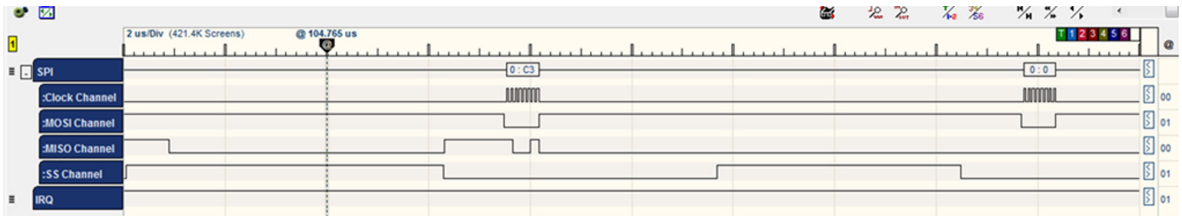



```
Command    CMD_INTERNAL_WRITE:    C3          /*     internal register write */
        BYTE [0] = CMD_INTERNAL_WRITE
        BYTE [1] = address >> 8;               /*     address = 0x01          */
        BYTE [1] |= (1 << 7);               /*     clockless register     */
        BYTE [2] = address;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

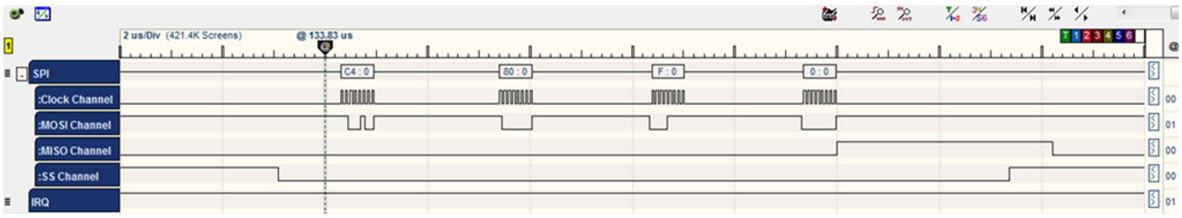
```
        BYTE [5] = u32data >> 8;
        BYTE [6] = u32data;
```



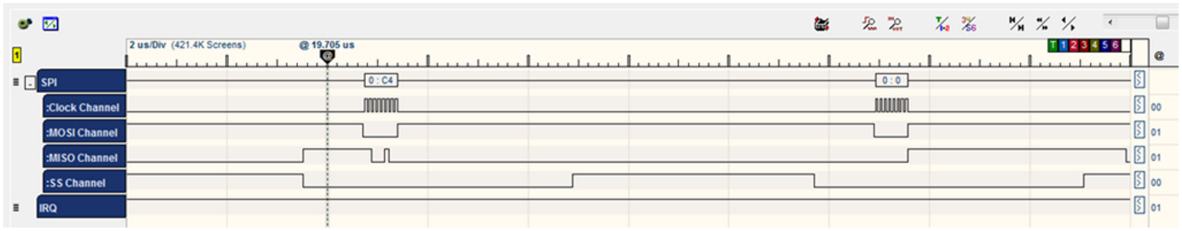
4. The WINC acknowledges the command by sending two bytes [C3] [0].



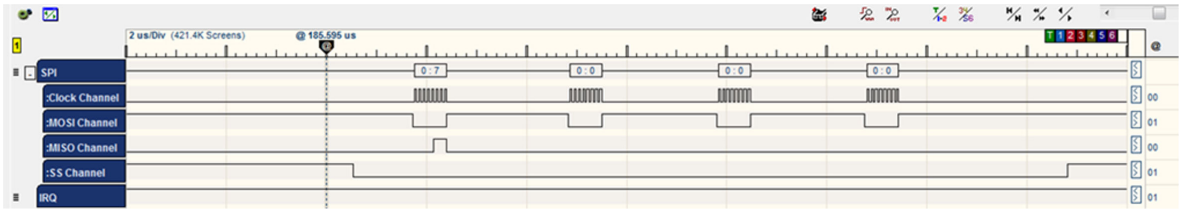

```
Command    CMD_INTERNAL_READ:    0xC4    /*     internal register read    */
        BYTE [0] = CMD_INTERNAL_READ
        BYTE [1] = address >> 8;            /*     address = 0x0F          */
        BYTE [1] |= (1 << 7);         /*     clockless register     */
        BYTE [2] = address;
     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



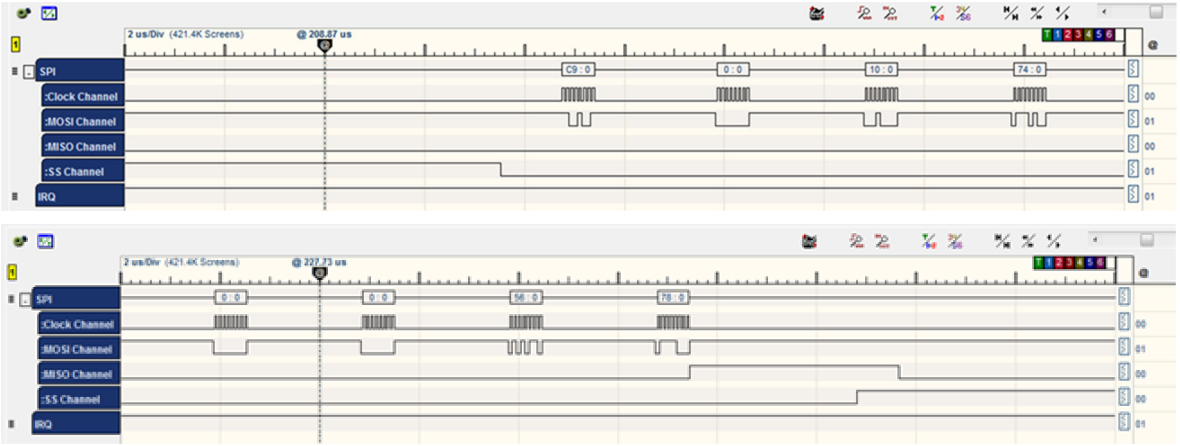
5. The WINC acknowledges the command by sending three bytes [C4] [0] [F3].



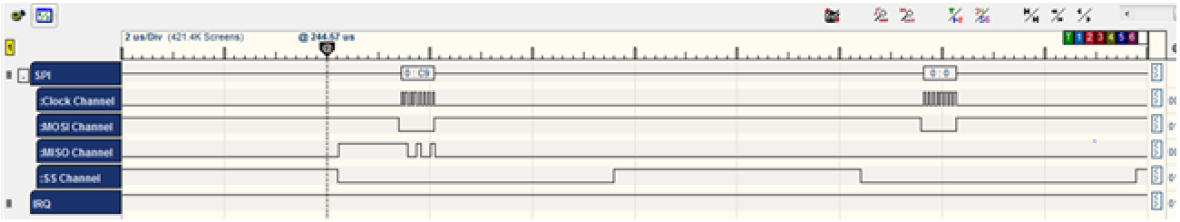
6. The WINC chip sends the value of the register 0x01 which equals 0x07.




```
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;
```



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

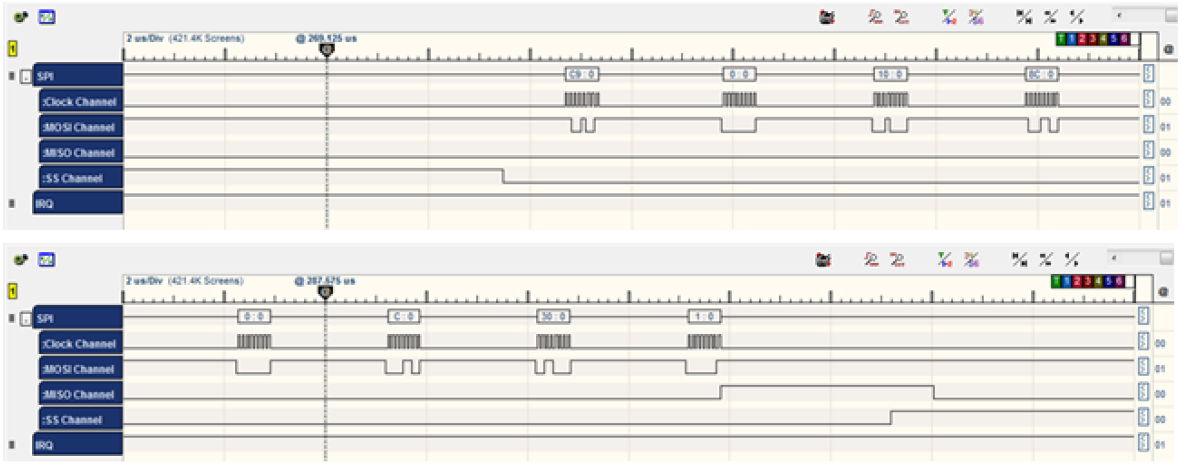


8. At this point, HIF finishes executing the clockless wake-up of the WINC chip.
9. The HIF layer prepares and sets the HIF layer header to NMI_STATE_REG register (4 byte or 8 byte header describing the packet to be sent).
10. Set bit '1' of WIFI_HOST_RCV_CTRL_2 register to raise an interrupt to the chip.

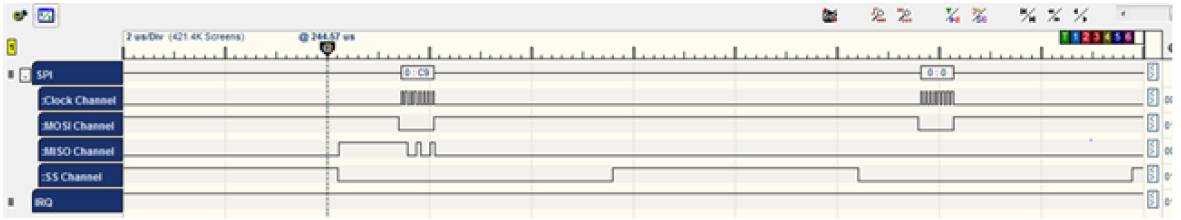
```
sint8 hif_send(uint8 u8Gid,uint8 u8Opcode,uint8 *pu8CtrlBuf,uint16 u16CtrlBufSize,
               uint8 *pu8DataBuf,uint16 u16DataSize, uint16 u16DataOffset)
{
    volatile tstrHifHdr    strHif;
    volatile uint32 reg;
    strHif.u8Opcode    = u8Opcode&(~NBIT7);
    strHif.u8Gid        = u8Gid;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

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

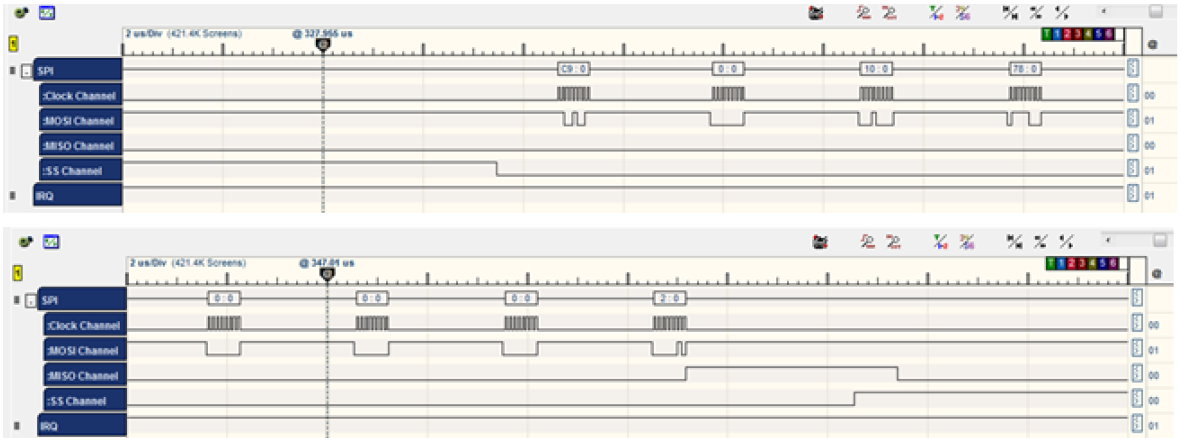
```
Command    CMD_SINGLE_WRITE:0XC9        /* single word write */
        BYTE [0] = CMD_SINGLE_WRITE
        BYTE [1] = address >> 16;        /* NMI_STATE_REG address = 0x108c */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = u32data >> 24;        /* Data = 0x000C3001 */
        BYTE [5] = u32data >> 16;        /* 0x0C is the length and equals 12  */
        BYTE [6] = u32data >> 8;         /* 0x30 is the Opcode =
M2M_WIFI_REQ_SET_SCAN_REGION */
        BYTE [7] = u32data;           /* 0x01 is the Group ID = M2M_REQ_GRP_WIFI  */
```



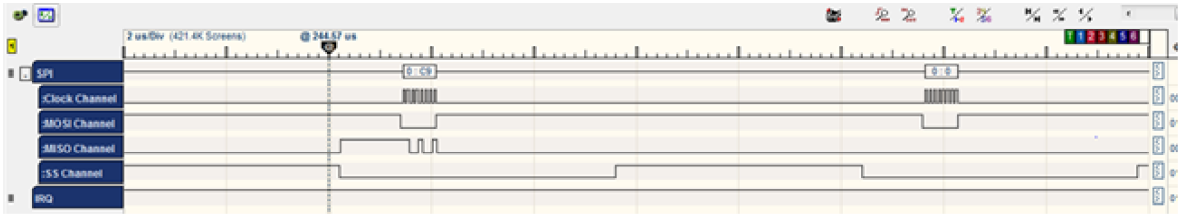
11. The WINC acknowledges the command by sending two bytes [C9] [0].




```
Command    CMD_SINGLE_WRITE:0XC9          /*      single word write */
        BYTE [0] = CMD_SINGLE_WRITE
        BYTE [1] = address >> 16;         /*      WIFI_HOST_RCV_CTRL_2address = 0x1078*/
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = u32data >> 24;          /*      Data = 0x02 */
        BYTE [5] = u32data >> 16;
        BYTE [6] = u32data >> 8;
        BYTE [7] = u32data;
```



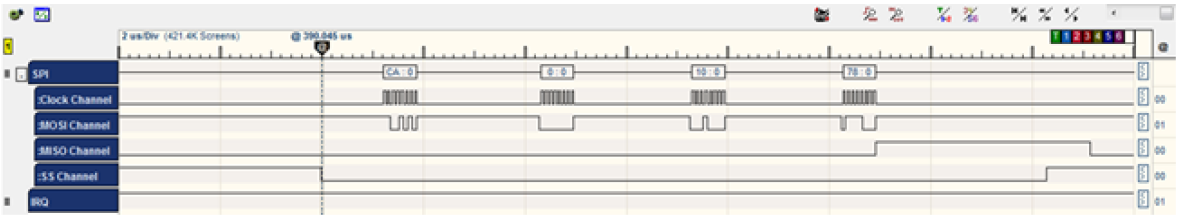
12. The WINC acknowledges the command by sending two bytes [C9] [0].



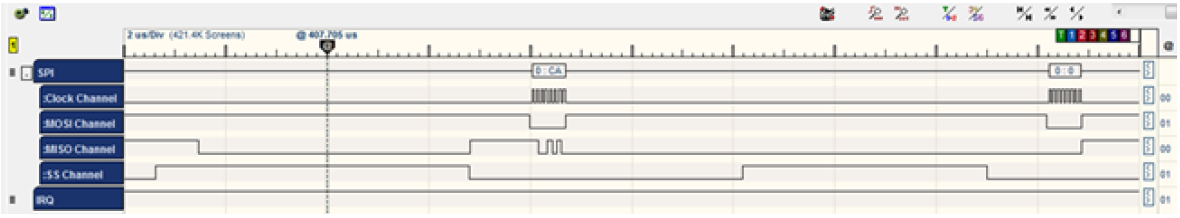
13. Then HIF polls for DMA address.

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

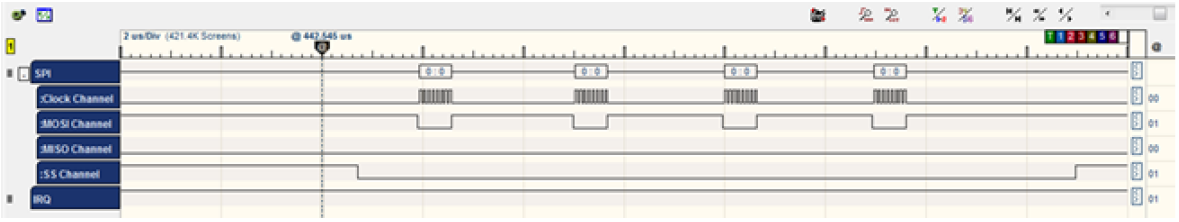
```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_2 address = 0x1078 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



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

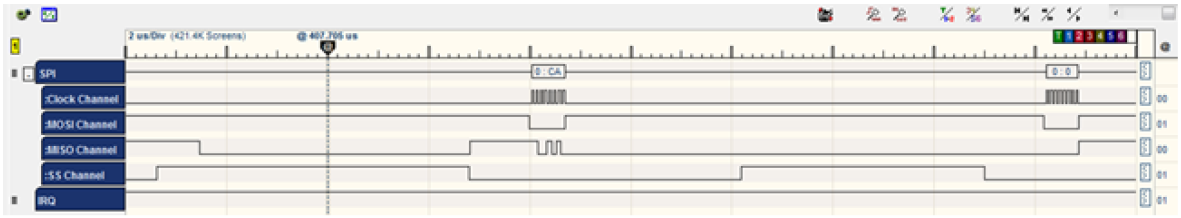


15. The WINC chip sends the value of the register 0x1078, which equals 0x00.

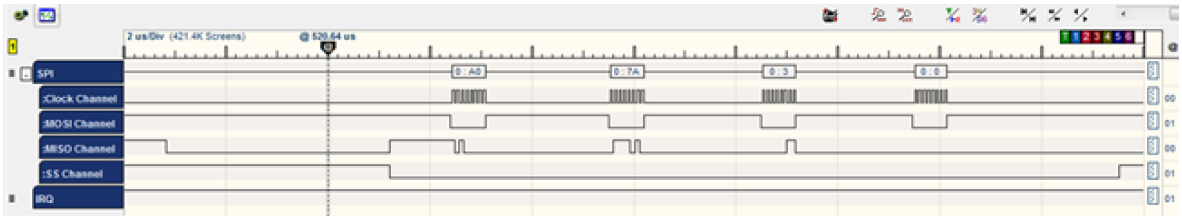



```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;          /* address = 0x1504 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```

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



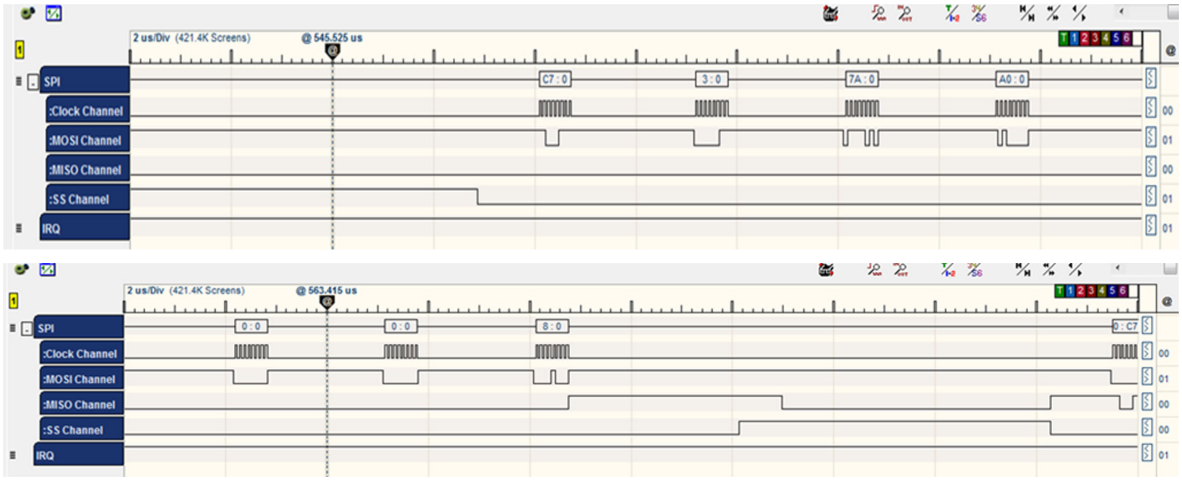
17. The WINC chip sends the value of the register 0x1504, which equals 0x037AA0.



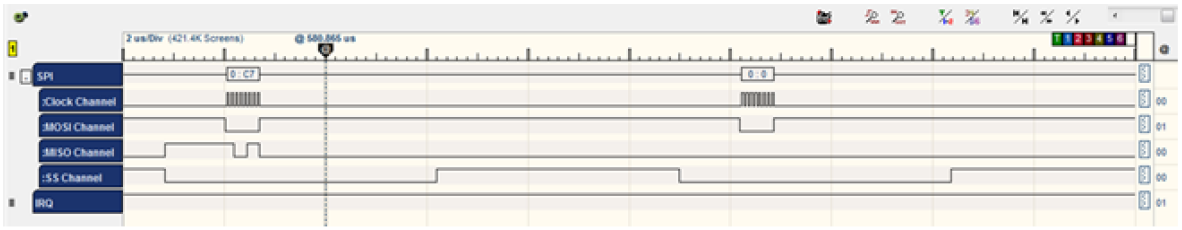
18. The WINC writes the HIF header to the DMA memory address.

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

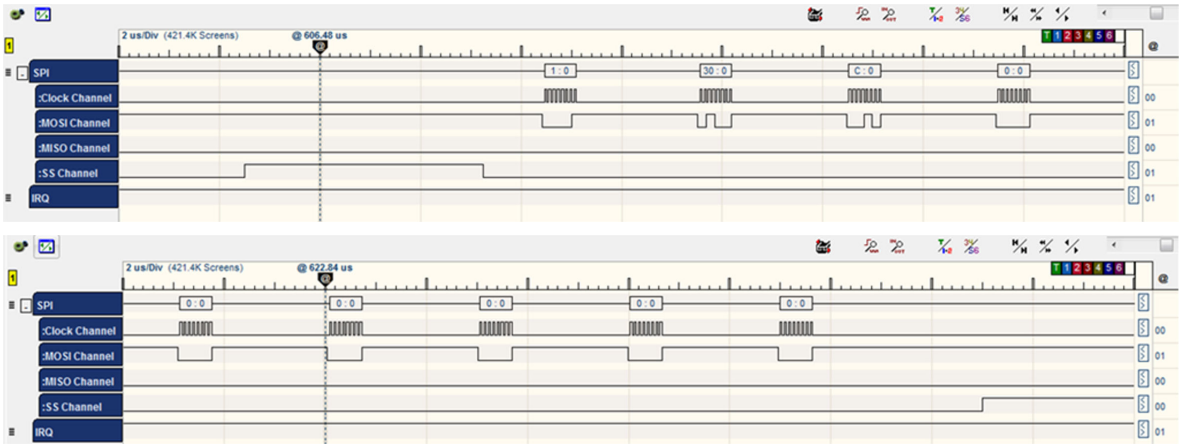
```
Command    CMD_DMA_EXT_WRITE:    0xC7         /* DMA extended write */
        BYTE [0] = CMD_DMA_EXT_WRITE
        BYTE [1] = address >> 16;             /* address = 0x037AA0 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = size >> 16;             /* size = 0x08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



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



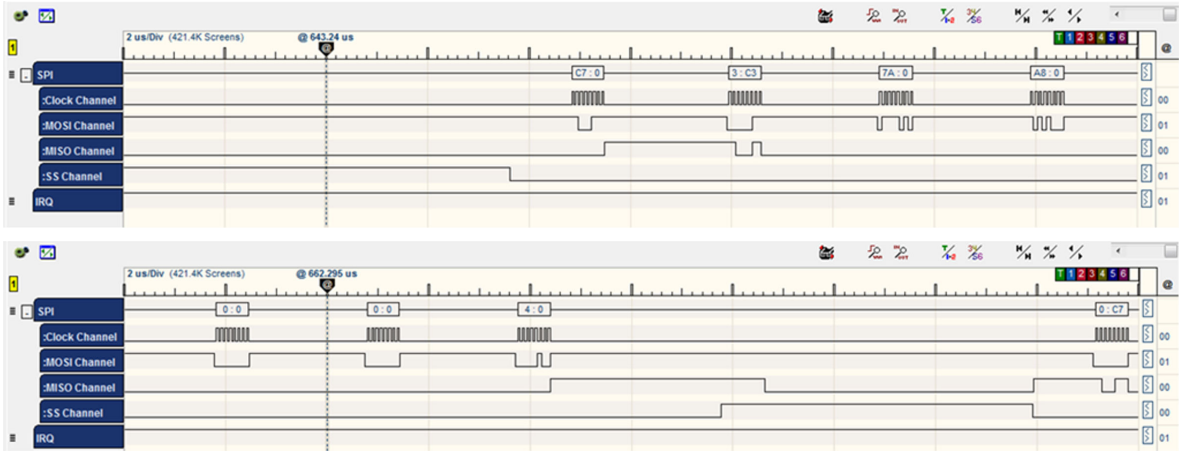
20. The HIF layer writes the data.



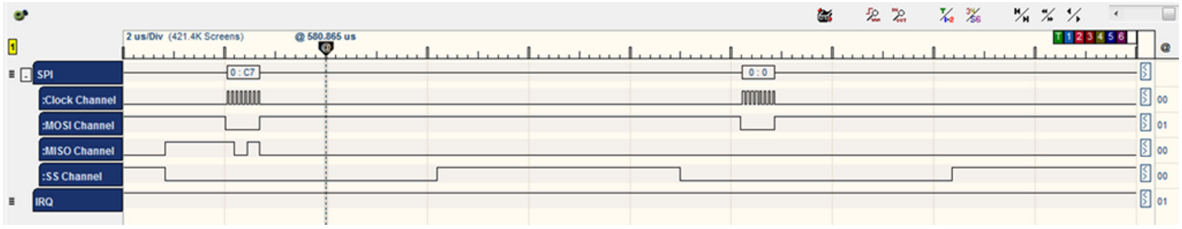
21. The HIF writes the Control Buffer data (part of the framing of the request).

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

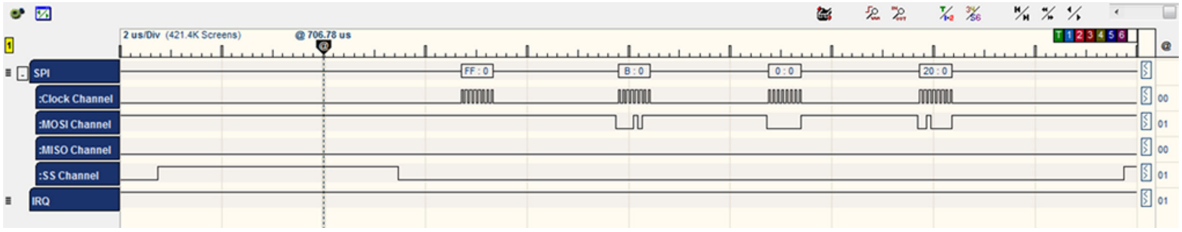
```
Command    CMD_DMA_EXT_WRITE:    0xC7           /* DMA extended write */
        BYTE [0] = CMD_DMA_EXT_WRITE
        BYTE [1] = address >> 16;            /* address = 0x037AA8 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = size >> 16;               /* size = 0x04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



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



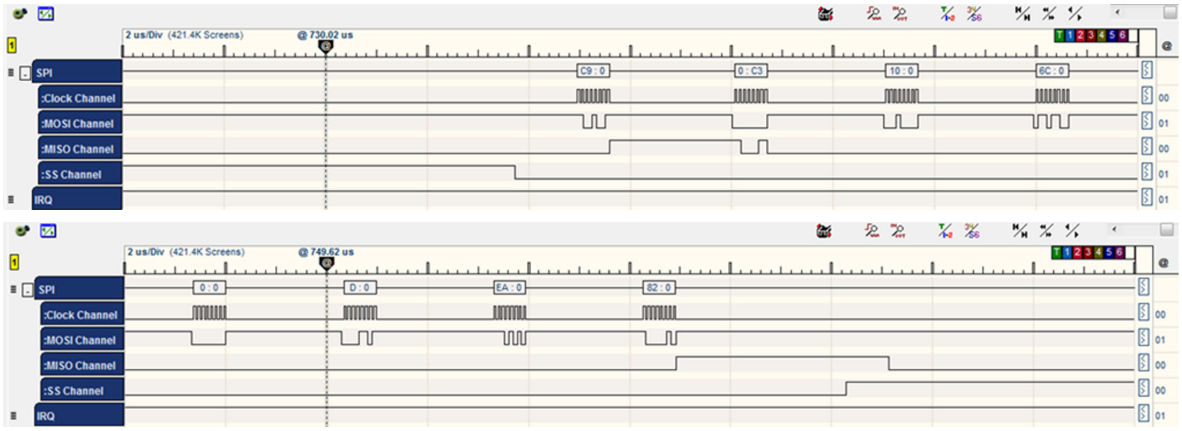
23. The HIF layer writes the data.



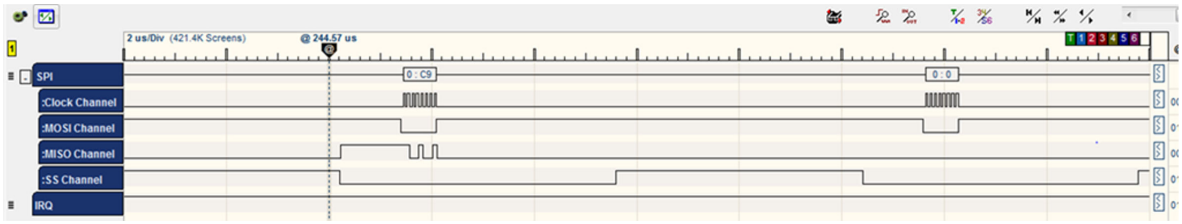
24. The HIF finished writing the request data to memory and is going to interrupt the chip notifying that host TX is done.

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

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



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

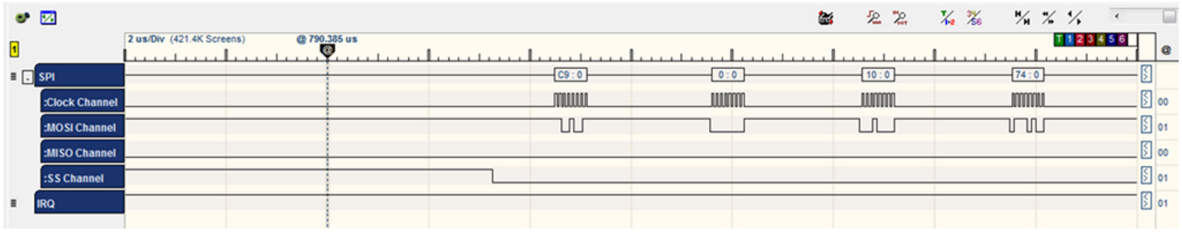


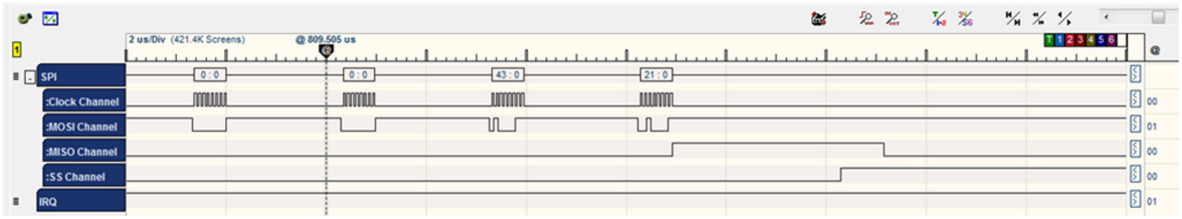
26. The HIF layer allows the chip to enter Sleep mode again.

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

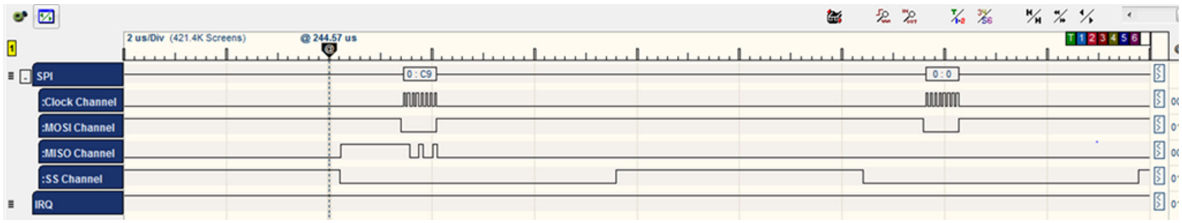
```
    }
}
```

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

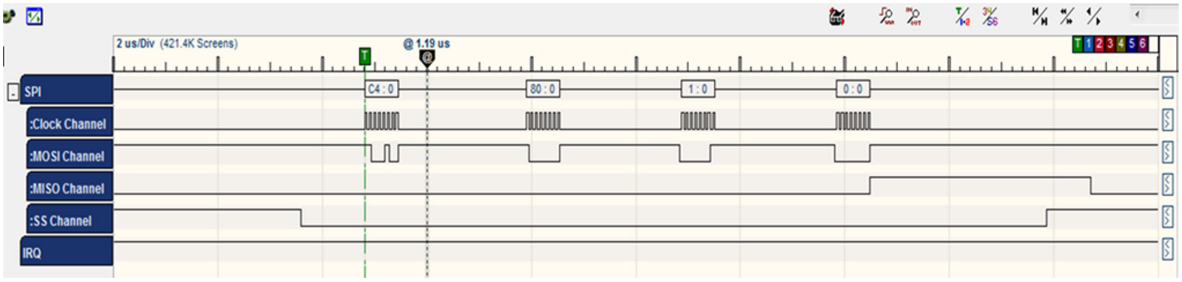




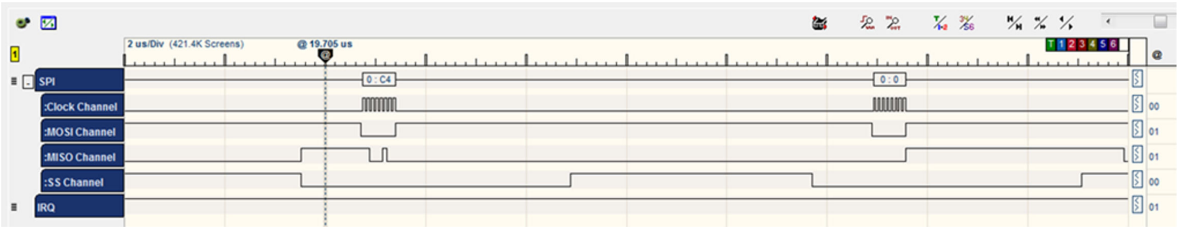
27. The WINC acknowledges the command by sending two bytes [C9] [0].



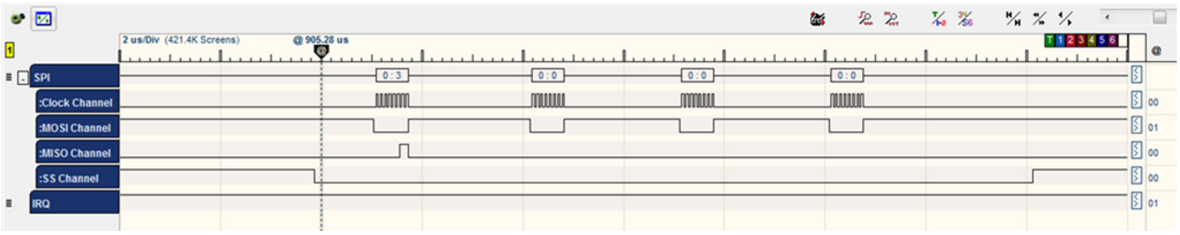

```
Command     CMD_INTERNAL_READ:    0xC4     /* internal register read     */
        BYTE [0] = CMD_INTERNAL_READ
        BYTE [1] = address >> 8;          /* address = 0x01     */
        BYTE [1] |= (1 << 7);             /* clockless register     */
        BYTE [2] = address;
        BYTE [3] =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



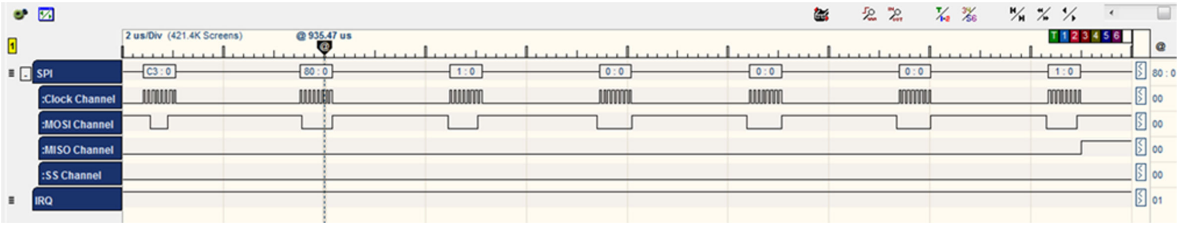
28. The WINC acknowledges the command by sending three bytes [C4] [0] [F3].



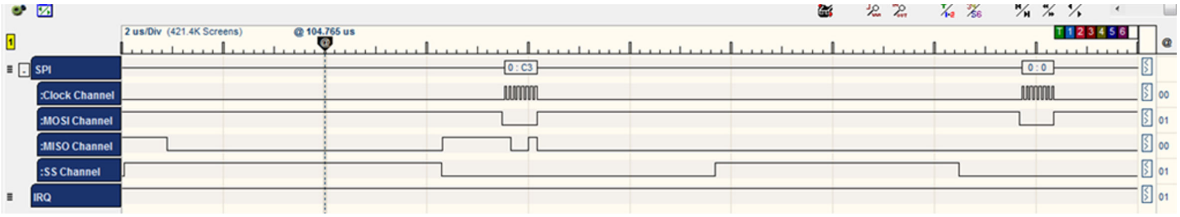
29. The WINC chip sends the value of the register 0x01 which equals 0x03.




```
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;
```



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



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

16.3.2 RX (Receive Response)

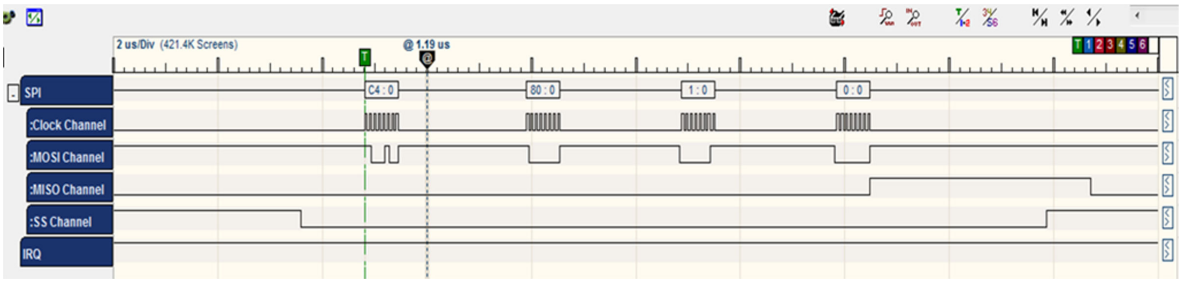
After finishing the required operation (scan Wi-Fi), the WINC interrupts the host to notify the processing of the request. The host handles this interrupt to receive the response.

1. First step in `hif_isr ( )` is to wake-up the 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);
```

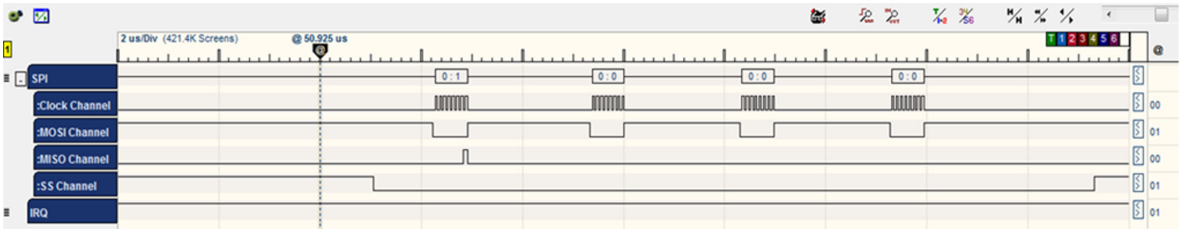
```
    return ret;
}
```

```
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;
```

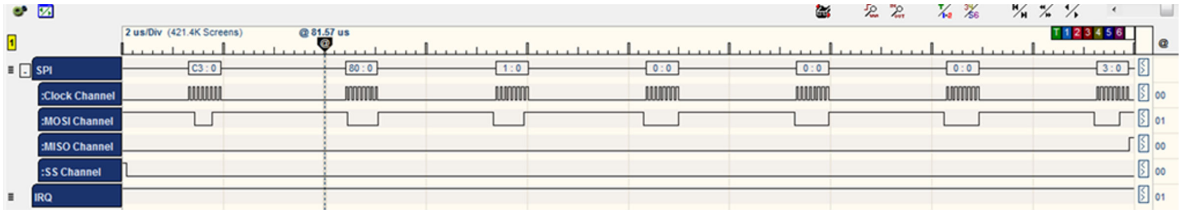


2. The WINC acknowledges the command by sending three bytes [C4] [0] [F3].

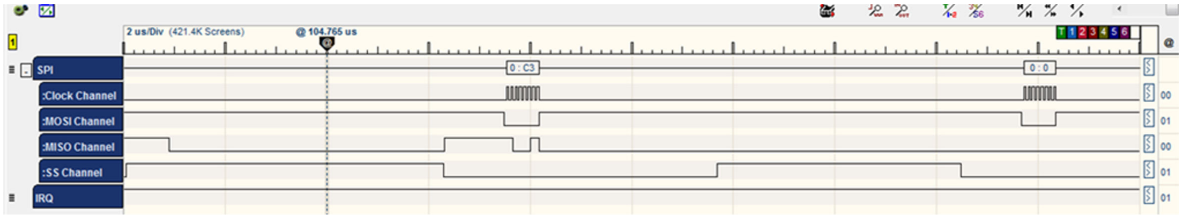
3. The WINC chip sends the value of the register 0x01 which equals 0x01.



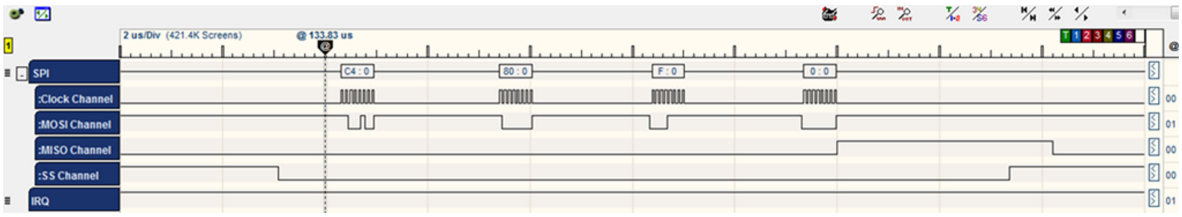

```
Command    CMD_INTERNAL_WRITE:    C3          /*     internal register write */
        BYTE [0] = CMD_INTERNAL_WRITE
        BYTE [1] = address >> 8;               /*     address = 0x01         */
        BYTE [1] |= (1 << 7);               /*     clockless register     */
        BYTE [2] = address;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



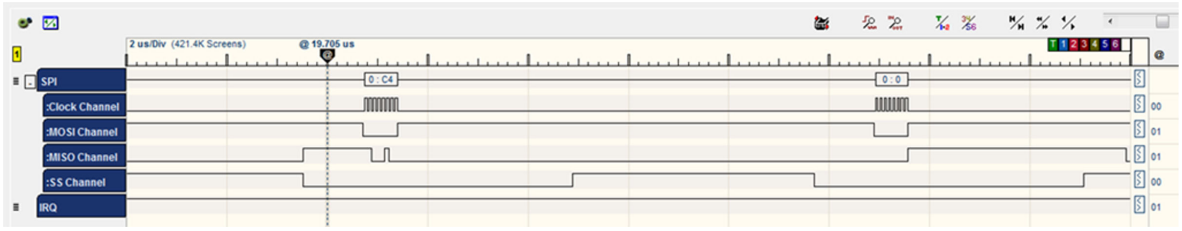
4. The WINC acknowledges the command by sending two bytes [C3] [0].



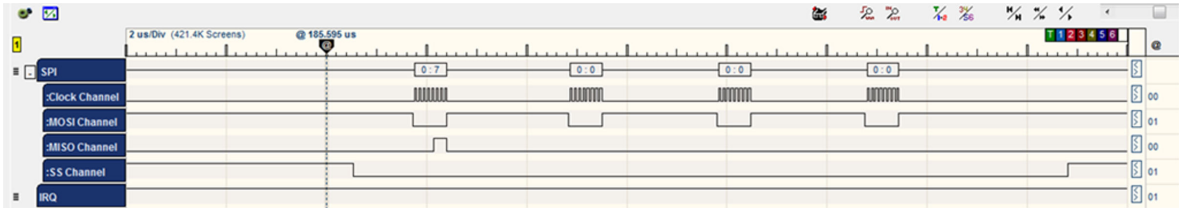

```
Command    CMD_INTERNAL_READ:    0xC4          /*     internal register read    */
        BYTE [0] = CMD_INTERNAL_READ
        BYTE [1] = address >> 8;             /*     address = 0x0F          */
        BYTE [1] |= (1 << 7);                /*     clockless register      */
        BYTE [2] =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



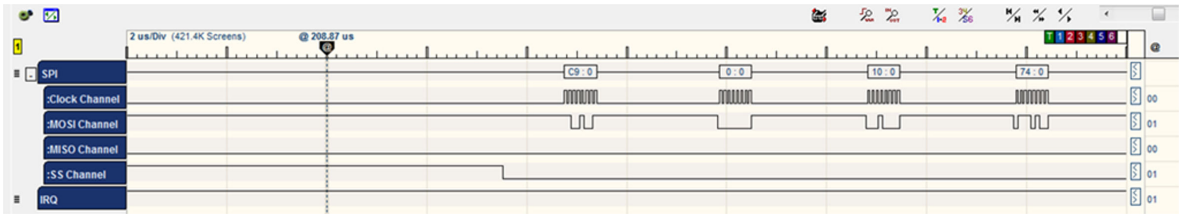
5. The WINC acknowledges the command by sending three bytes [C4] [0] [F3].

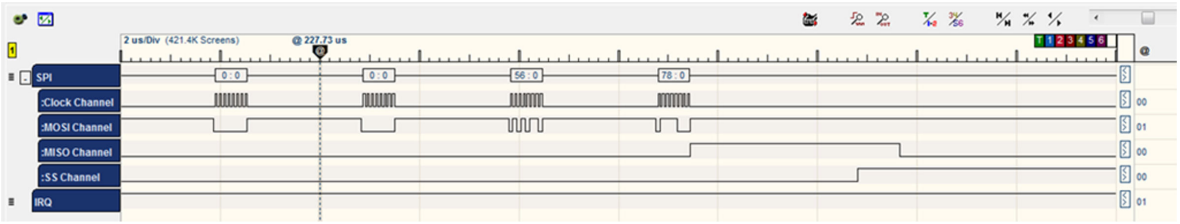


6. Then WINC chip sends the value of the register 0x01 which equals 0x07.

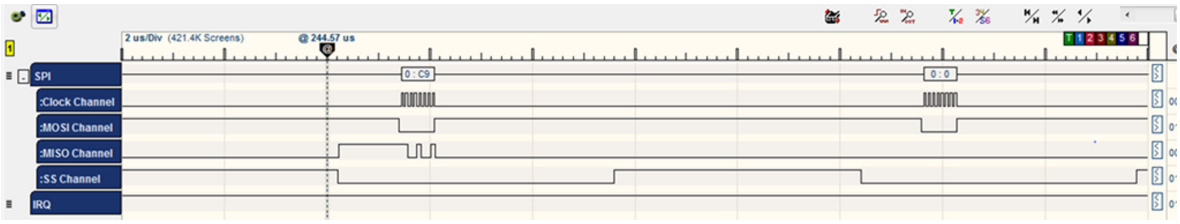



```
Command    CMD_SINGLE_WRITE:0XC9              /* single word write */
        BYTE [0] = CMD_SINGLE_WRITE
        BYTE [1] = address >> 16;         /* WAKE_REG address = 0x1074 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = u32data >> 24;         /* WAKE_VALUE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





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

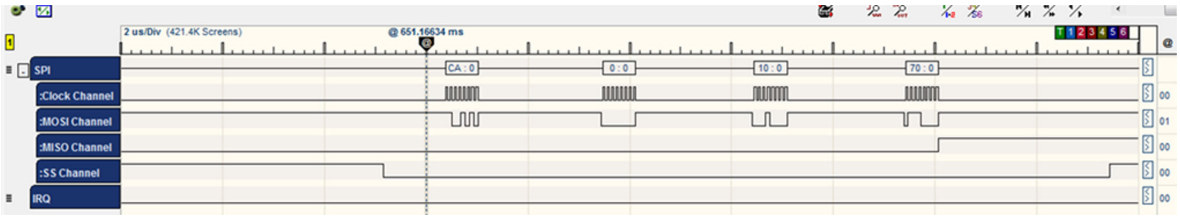


8. Read register WIFI_HOST_RCV_CTRL_0 to check if there is a new interrupt, and clear it.

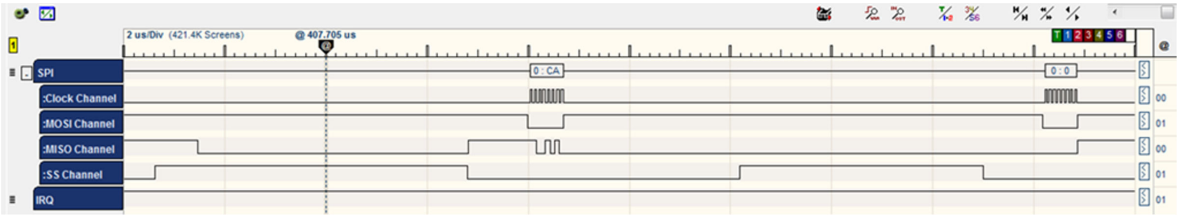
```
static sint8 hif_isr(void)
{
    sint8 ret ;
    uint32 reg;
    volatile tstrHifHdr strHif;

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

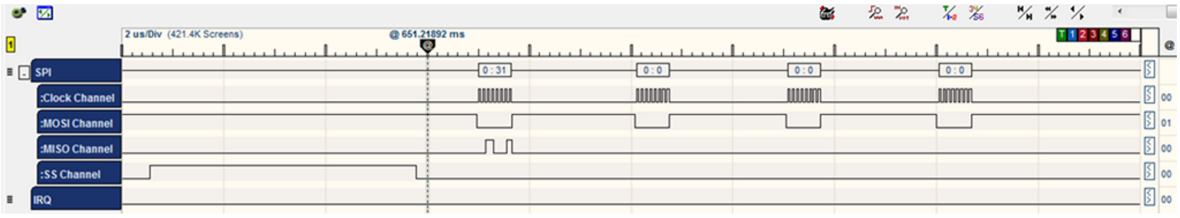
```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read         */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



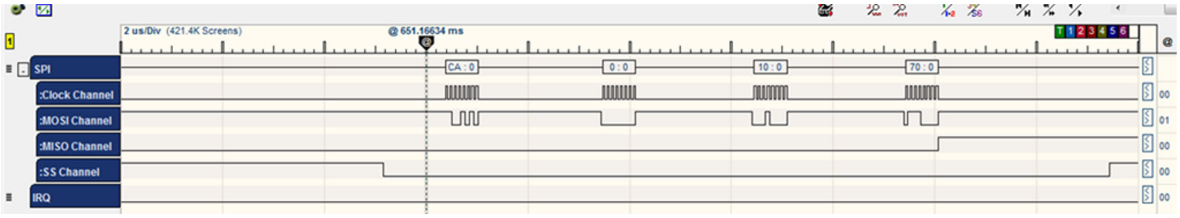
9. The WINC acknowledges the command by sending three bytes [CA] [0] [F3].



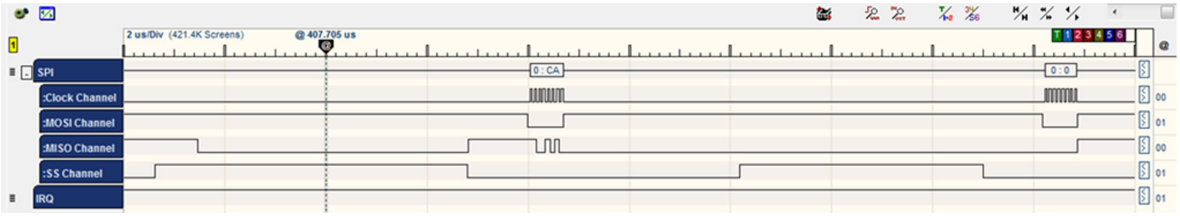
10. The WINC chip sends the value of the register 0x1070 which equals 0x31.



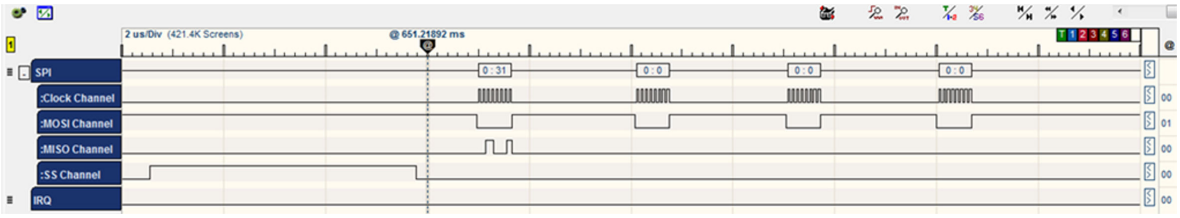

```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read         */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



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

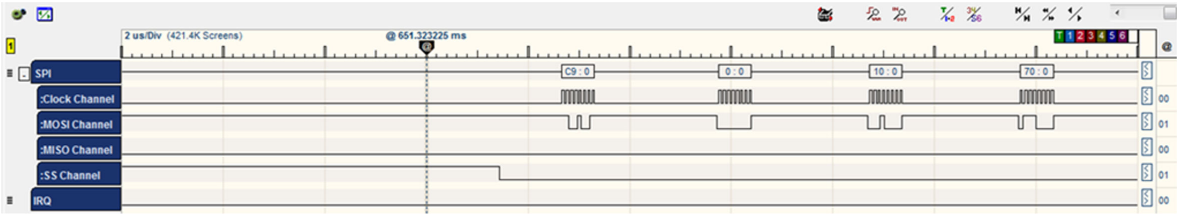


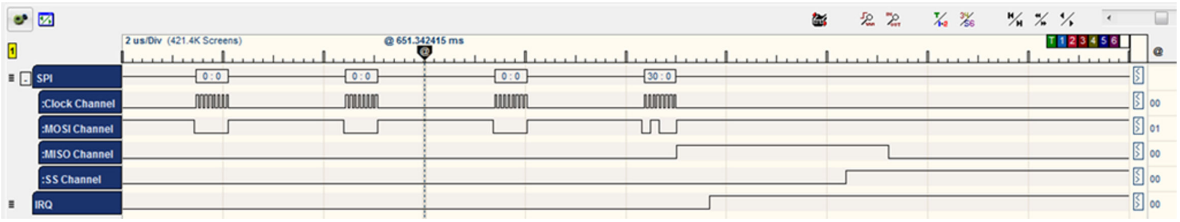
12. The WINC chip sends the value of the register 0x1070 which equals 0x31.



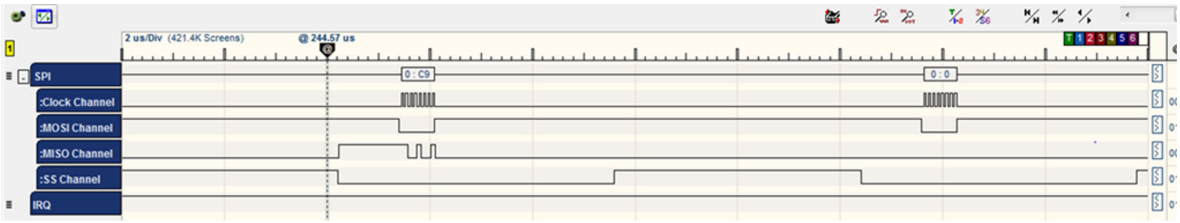
13. Clear the WINC Interrupt.

```
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;
```





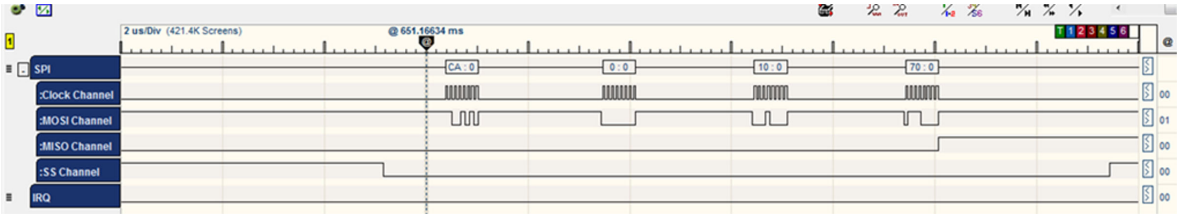
14. The chip acknowledges the command by sending two bytes [C9] [0].



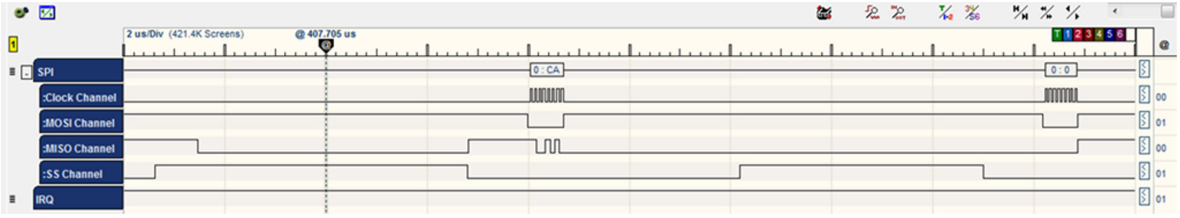
15. The HIF reads the data size.

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

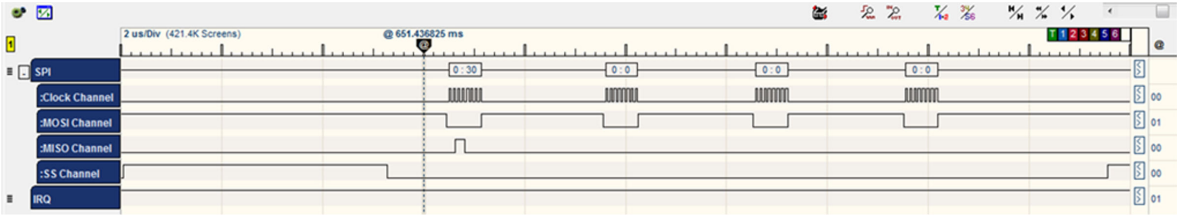
```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read          */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



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



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

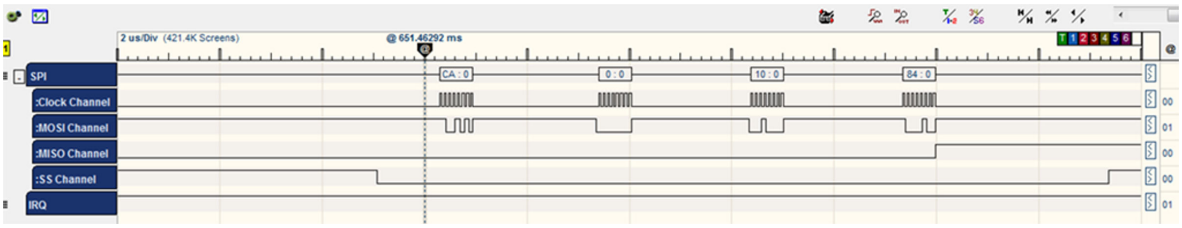


18. The HIF reads hif header address.

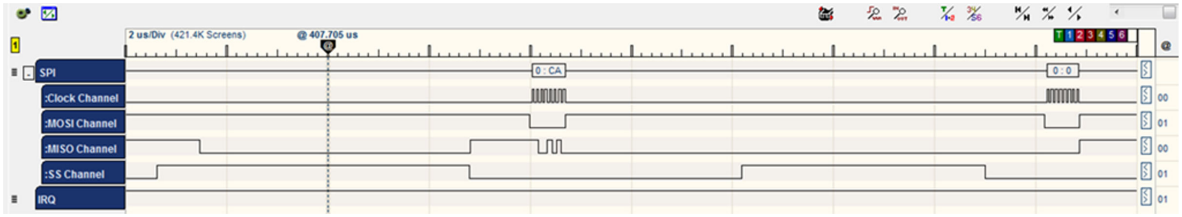
```
/** start bus transfer**/
ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_1, &address);
```

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

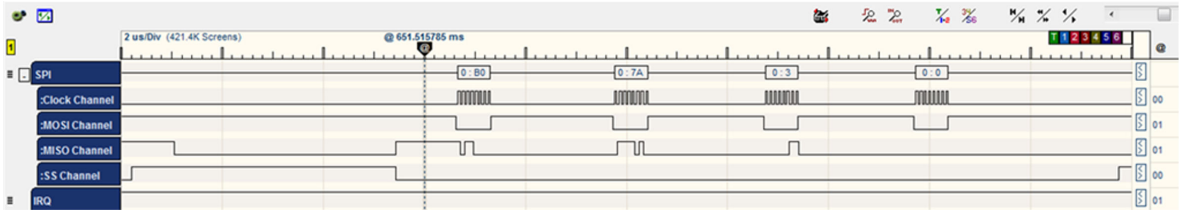
```
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



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



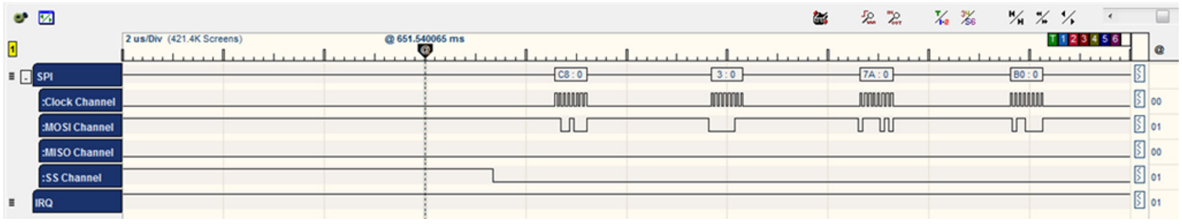
20. The WINC chip sends the value of the register 0x1078 which equals 0x037AB0.

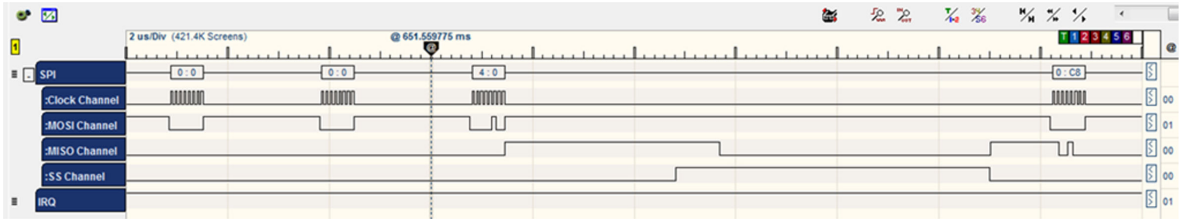


21. The HIF reads the hif header data (as a block).

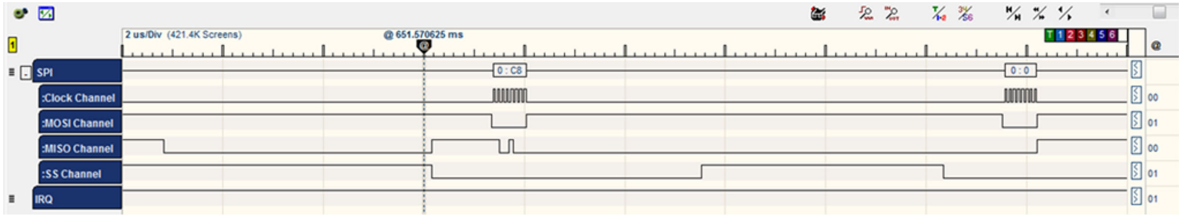
```
ret = nm_read_block(address, (uint8*)&strHif, sizeof(tstrHifHdr));
```

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

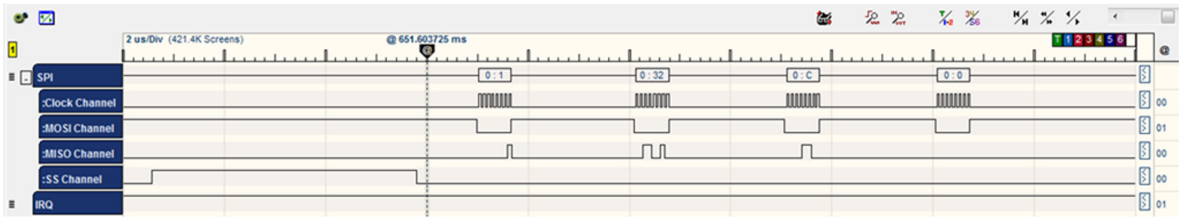




22. The WINC acknowledges the command by sending three bytes [C8] [0] [F3].



23. The WINC sends the data block (four bytes).



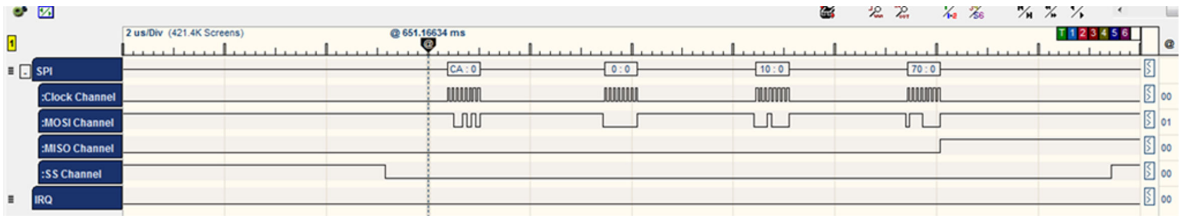
24. The HIF calls the appropriate handler according to the hif header received which tries to receive the Response data payload.
Note: `hif_receive ( )` obtains additional data.

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

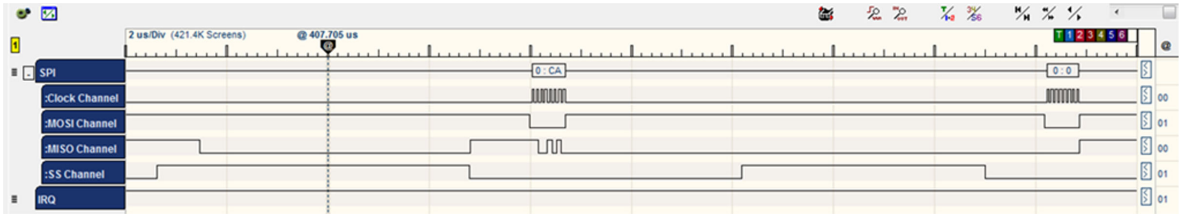
    ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_0,&reg);
    size = (uint16)((reg >> 2) & 0xfff);
    ret = nm_read_reg_with_ret(WIFI_HOST_RCV_CTRL_1,&address);
    /* Receive the payload */
    ret = nm_read_block(u32Addr, pu8Buf, u16Sz);

}
```

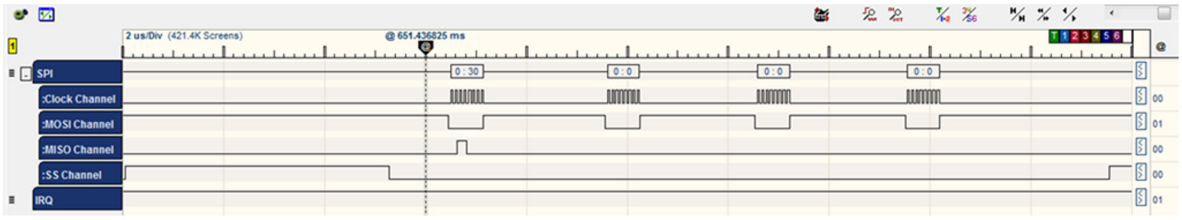
```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read         */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;        /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



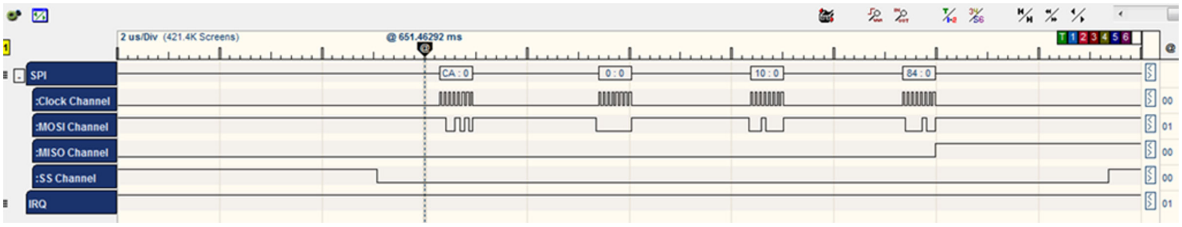
25. The WINC acknowledges the command by sending three bytes [CA] [0] [F3].



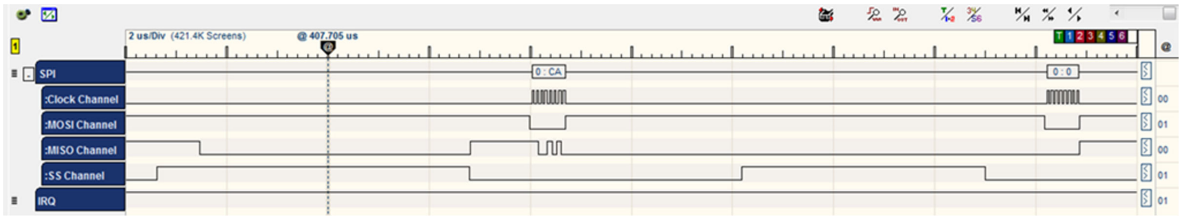
26. The WINC chip sends the value of the register 0x1070 which equals 0x30.



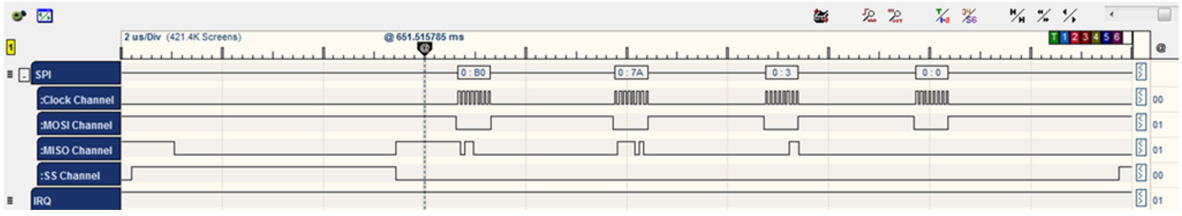

```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read         */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_1 address = 0x1084 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



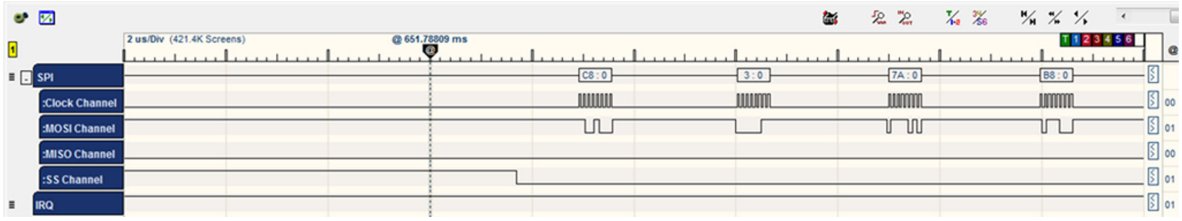
27. The WINC acknowledges the command by sending three bytes [CA] [0] [F3].

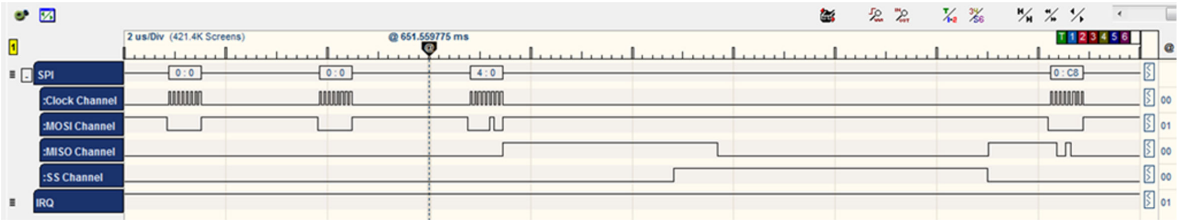


28. The WINC chip sends the value of the register 0x1078 which equals 0x037AB0.

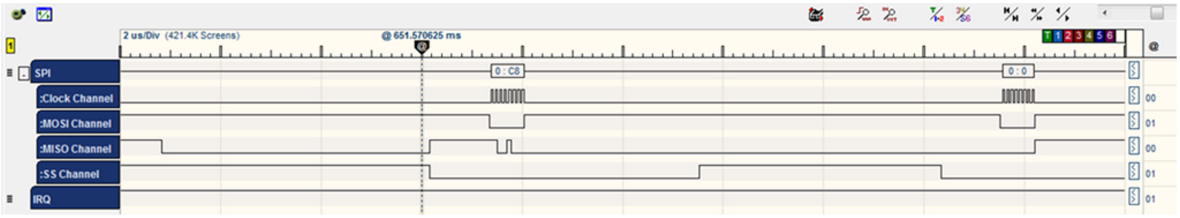



```
Command    CMD_DMA_EXT_READ:    C8        /* dma extended read */
        BYTE [0] = CMD_DMA_EXT_READ
        BYTE [1] = address >> 16;         /* address = 0x037AB8*/
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = size >> 16;
        BYTE [5] = size >>;
        BYTE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

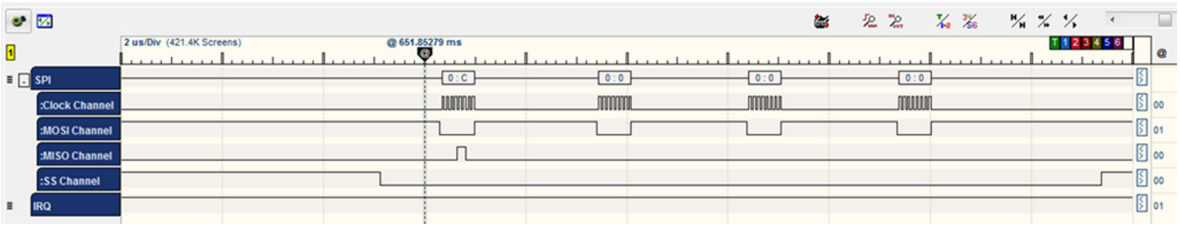




29. The WINC acknowledges the command by sending three bytes [C8] [0] [F3].



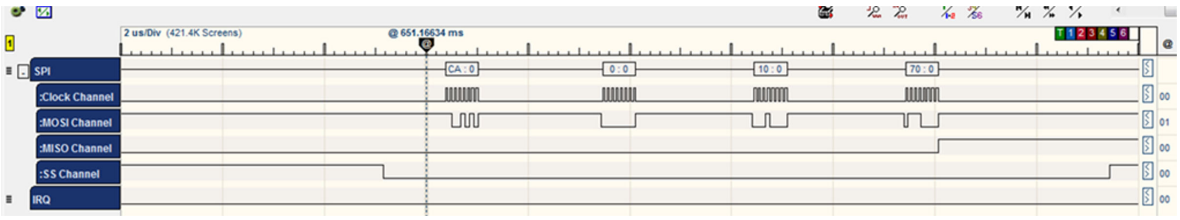
30. The WINC sends the data block (four bytes).



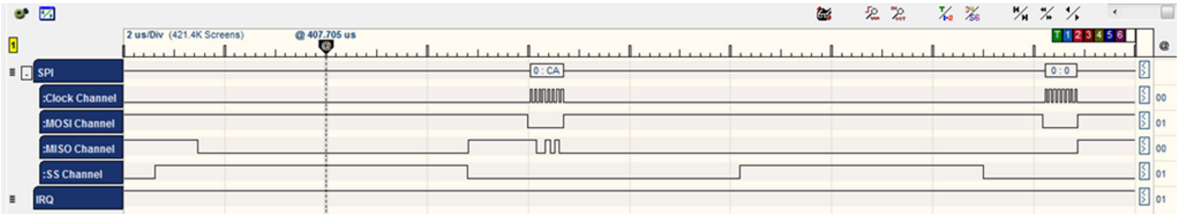
31. After the HIF layer received the response, it interrupts the chip to send the notification that the host RX is done.

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

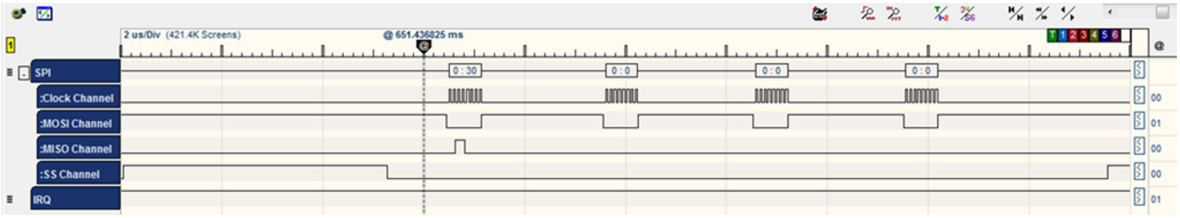
```
Command    CMD_SINGLE_READ:    0xCA        /* single word (4 bytes) read         */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



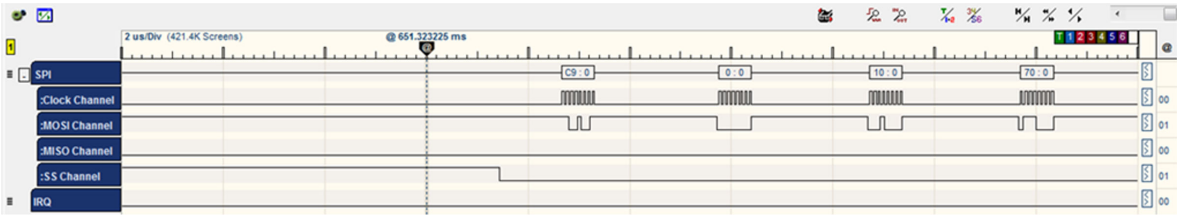
32. The WINC acknowledges the command by sending three bytes [CA] [0] [F3].

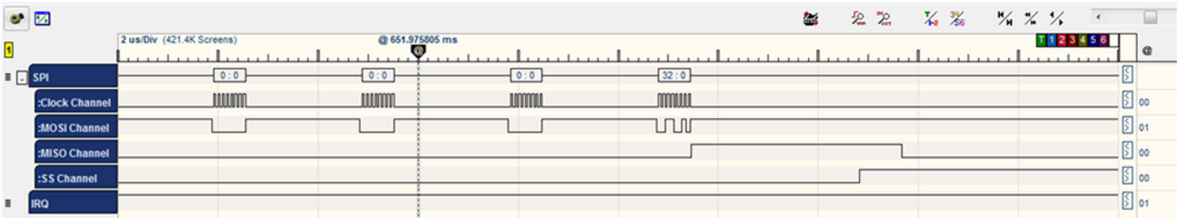


33. The WINC chip sends the value of the register 0x1070 which equals 0x30.

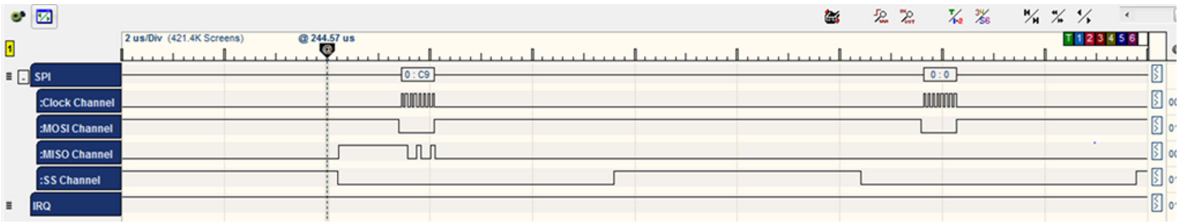



```
Command    CMD_SINGLE_WRITE:0XC9            /* single word write */
        BYTE [0] = CMD_SINGLE_WRITE
        BYTE [1] = address >> 16;        /* WIFI_HOST_RCV_CTRL_0 address = 0x1070  */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = u32data >> 24;        /* Data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





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

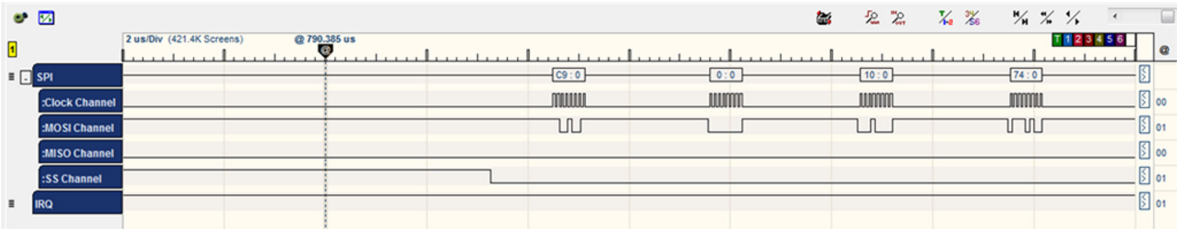


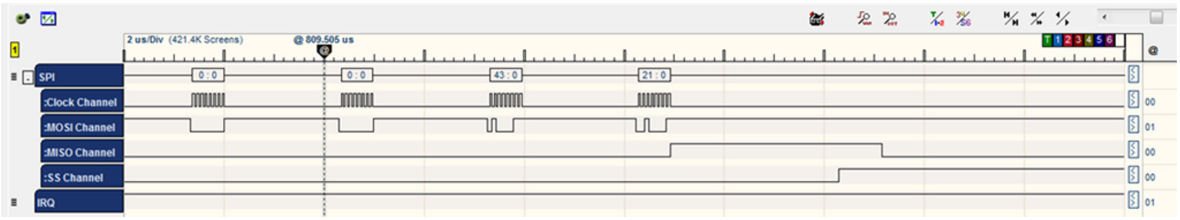
35. The HIF layer allows the chip to enter Sleep mode again.

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

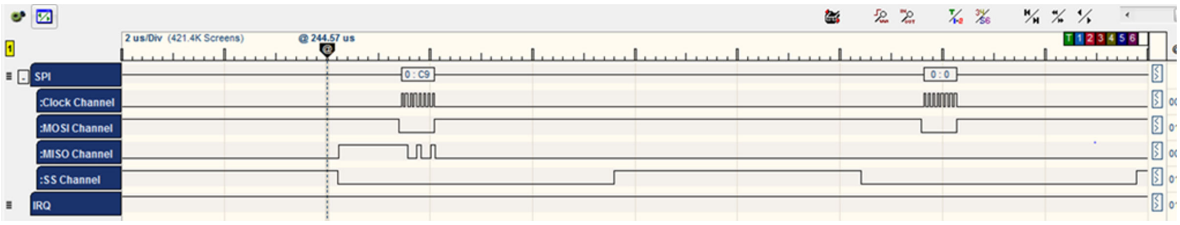
```
Command    CMD_SINGLE_WRITE:0XC9            /* single word write         */
        BYTE [0] = CMD_SINGLE_WRITE
        BYTE [1] = address >> 16;           /* WAKE_REG address = 0x1074 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = u32data >> 24;            /*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

```
        BYTE [5] = u32data >> 16;
        BYTE [6] = u32data >> 8;
        BYTE [7] = u32data;
```



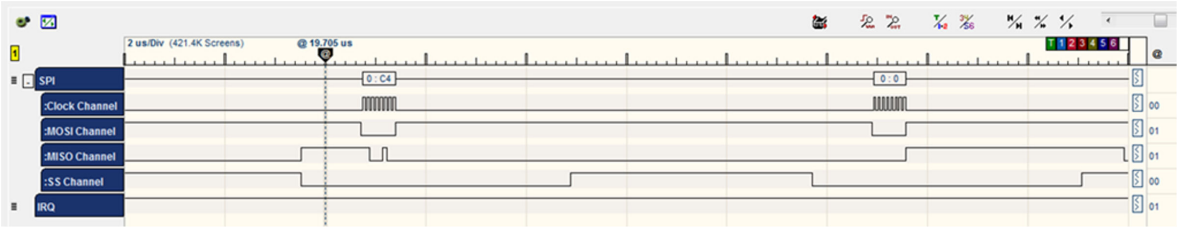


36. The WINC acknowledges the command by sending two bytes [C9] [0].

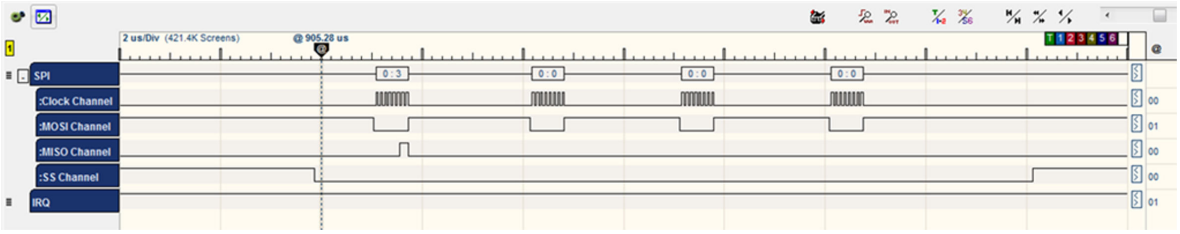



```
Command    CMD_INTERNAL_READ:    0xC4        /* internal register read     */
        BYTE [0] = CMD_INTERNAL_READ
        BYTE [1] = address >> 8;              /* address = 0x01         */
        BYTE [1] |= (1 << 7);                 /* clockless register     */
        BYTE [2] = address;
   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

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

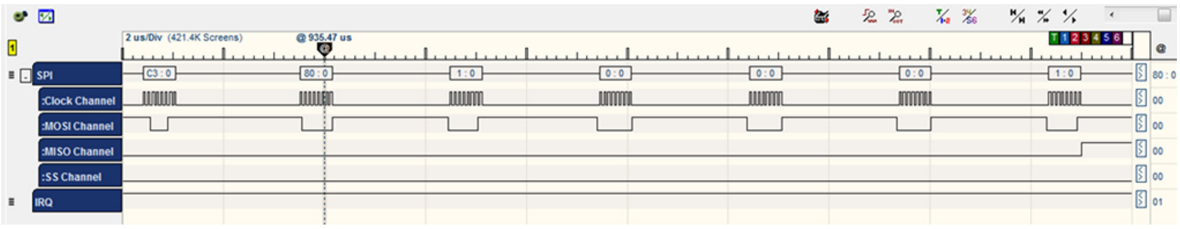


38. Then WINC chip sends the value of the register 0x01 which equals 0x03.

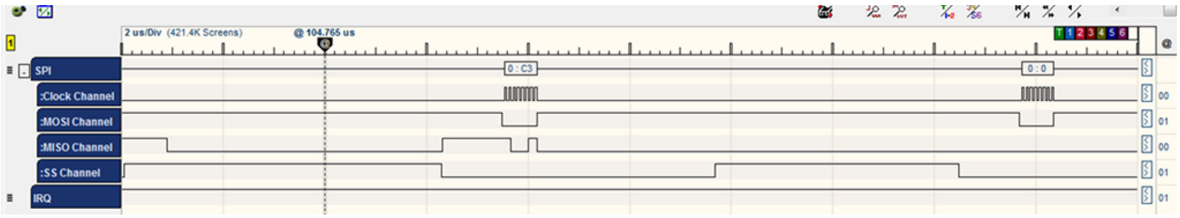



```
Command    CMD_INTERNAL_WRITE:    C3           /* internal register write      */
        BYTE [0] = CMD_INTERNAL_WRITE
        BYTE [1] = address >> 8;             /* address = 0x01      */
        BYTE [1] |= (1 << 7);                /* clockless register     */
        BYTE [2] = address;
   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

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



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



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

17. Appendix A. How to Generate Certificates

17.1 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.

17.2 Steps

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

1. Generate a key for the CA (certification authority). To generate a 4096-bit long RSA (creates a new file `CA_KEY.key` to store the random key), using the following command (CMD):

```
openssl genrsa -out CA_KEY.key 4096
```

2. Create your self-signed root CA certificate `CA_CERT.crt`; you need to provide some data for your Root certificate, using the following command (CMD):

```
openssl req -new -x509 -days 1826 -key CA_KEY.key -out CA_CERT.crt
```

3. Create the custom certificate, which is signed by the CA root certificate created earlier. First, generate the `Custom.key`, using the following command (CMD):

```
openssl genrsa -out Custom.key 4096
```

4. To generate a certificate request file (CSR) using this generated key, use the following command (CMD):

```
openssl req -new -key Custom.key -out CertReq.csr
```

5. Process the request for the certificate and get it signed by the root CA, using the following command (CMD):

```
openssl x509 -req -days 730 -in CertReq.csr -CA CA_CERT.crt -CAkey CA_KEY.key -
set_serial 01 -out CustomCert.crt
```

18. Appendix B. X.509 Certificate Format and Conversion

18.1 Introduction

The most known encodings for the X.509 digital certificates are PEM and DER formats.

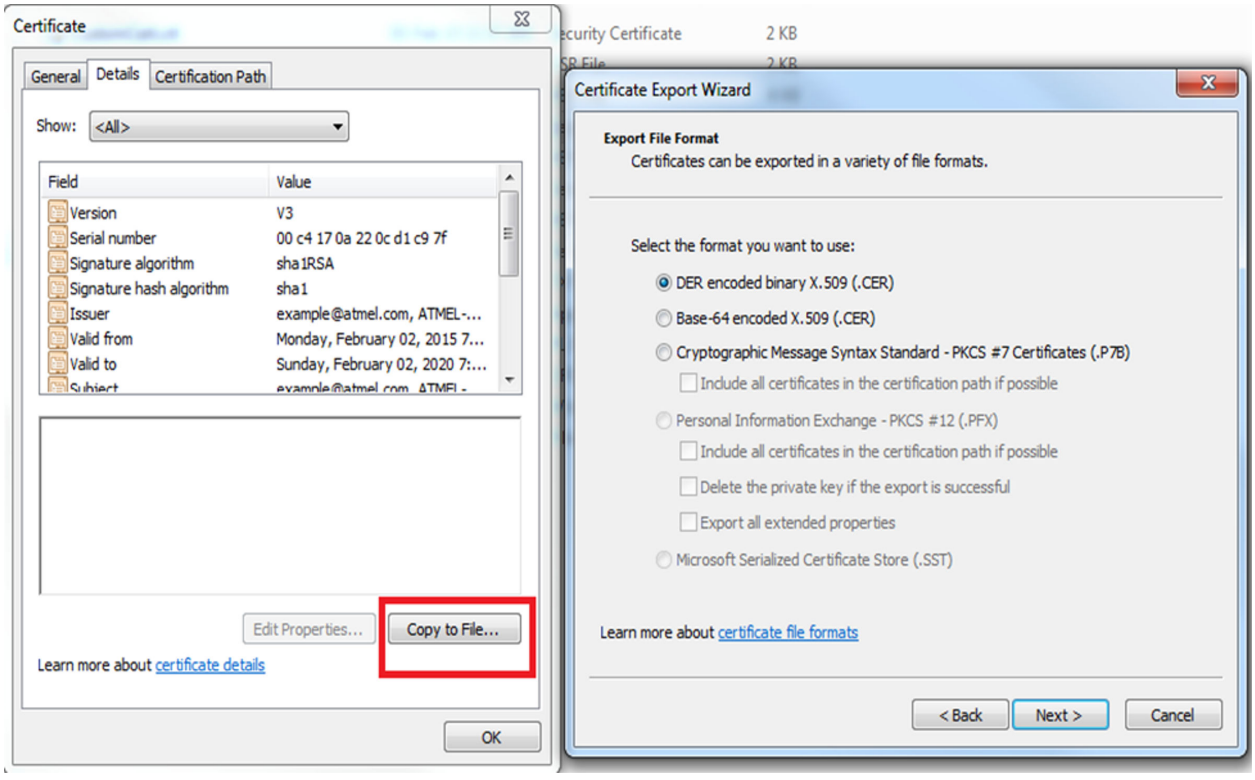
The PEM format is base64 encoding of the DER enclosed with messages "-----BEGIN CERTIFICATE-----" and "-----END CERTIFICATE-----".

18.2 Conversion Between Different Formats

The current implementation of the WINC root_certificate_downloader supports only DER format. If the certificate is not in DER format, it must be converted first. The conversion between different formats are done in several methods:

18.2.1 Using Windows

From Windows®, double click on the `.pem` certificate file and then go to the Details Tab and press “Copy to File”. Follow the Certificate Export Wizard until the **Finish** button.



18.2.2 Using OpenSSL

The OpenSSL is used for certificate conversion by the following command.

```
openssl x509 -outform der -in certificate.pem -out certificate.der
```

18.2.3 Online Conversion

There are useful online tools which provide conversion between the certificate formats, which can be found through searching online using keywords such as "OpenSSL".

19. References

The following documents can be used for further study:

- ATWINC15x0 Software Programming Guide
- ATWINC15x0-MR210xB Data Sheet

The following web page can be referred for further study on API:

- Atmel Software Framework for ATWINC1500 (Wi-Fi)

20. Document Revision History

Rev A - 05/2017

Section	Changes
Document	• Updated from Atmel to Microchip template. • Assigned a new Microchip document number. Previous version is Atmel 42420 revision B. • ISBN number added.

The Microchip Web Site

Microchip provides online support via our web site at http://www.microchip.com/. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

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

Customer Change Notification Service

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

To register, access the Microchip web site at http://www.microchip.com/. Under "Support", click on "Customer Change Notification" and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or Field Application Engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document.

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

Microchip Devices Code Protection Feature

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

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.

- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable.”

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

Legal Notice

Trademarks

ISBN: 978-1-5224-1729-3

Quality Management System Certified by DNV

ISO/TS 16949

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Fax: 480-792-7277 Technical Support: http://www.microchip.com/support Web Address: www.microchip.com **Atlanta** Duluth, GA Tel: 678-957-9614 Fax: 678-957-1455 **Austin, TX** Tel: 512-257-3370 **Boston** Westborough, MA Tel: 774-760-0087 Fax: 774-760-0088 **Chicago** Itasca, IL Tel: 630-285-0071 Fax: 630-285-0075 **Dallas** Addison, TX Tel: 972-818-7423 Fax: 972-818-2924 **Detroit** Novi, MI Tel: 248-848-4000 **Houston, TX** Tel: 281-894-5983 **Indianapolis** Noblesville, IN Tel: 317-773-8323 Fax: 317-773-5453 Tel: 317-536-2380 **Los Angeles** Mission Viejo, CA Tel: 949-462-9523 Fax: 949-462-9608 Tel: 951-273-7800 **Raleigh, NC** Tel: 919-844-7510 **New York, NY** Tel: 631-435-6000 **San Jose, CA** Tel: 408-735-9110 Tel: 408-436-4270 **Canada - Toronto** Tel: 905-695-1980 Fax: 905-695-2078	**Asia Pacific Office** Suites 3707-14, 37th Floor Tower 6, The Gateway Harbour City, Kowloon **Hong Kong** Tel: 852-2943-5100 Fax: 852-2401-3431 **Australia - Sydney** Tel: 61-2-9868-6733 Fax: 61-2-9868-6755 **China - Beijing** Tel: 86-10-8569-7000 Fax: 86-10-8528-2104 **China - Chengdu** Tel: 86-28-8665-5511 Fax: 86-28-8665-7889 **China - Chongqing** Tel: 86-23-8980-9588 Fax: 86-23-8980-9500 **China - Dongguan** Tel: 86-769-8702-9880 **China - Guangzhou** Tel: 86-20-8755-8029 **China - Hangzhou** Tel: 86-571-8792-8115 Fax: 86-571-8792-8116 **China - Hong Kong SAR** Tel: 852-2943-5100 Fax: 852-2401-3431 **China - Nanjing** Tel: 86-25-8473-2460 Fax: 86-25-8473-2470 **China - Qingdao** Tel: 86-532-8502-7355 Fax: 86-532-8502-7205 **China - Shanghai** Tel: 86-21-3326-8000 Fax: 86-21-3326-8021 **China - Shenyang** Tel: 86-24-2334-2829 Fax: 86-24-2334-2393 **China - Shenzhen** Tel: 86-755-8864-2200 Fax: 86-755-8203-1760 **China - Wuhan** Tel: 86-27-5980-5300 Fax: 86-27-5980-5118 **China - Xian** Tel: 86-29-8833-7252 Fax: 86-29-8833-7256	**China - Xiamen** Tel: 86-592-2388138 Fax: 86-592-2388130 **China - Zhuhai** Tel: 86-756-3210040 Fax: 86-756-3210049 **India - Bangalore** Tel: 91-80-3090-4444 Fax: 91-80-3090-4123 **India - New Delhi** Tel: 91-11-4160-8631 Fax: 91-11-4160-8632 **India - Pune** Tel: 91-20-3019-1500 **Japan - Osaka** Tel: 81-6-6152-7160 Fax: 81-6-6152-9310 **Japan - Tokyo** Tel: 81-3-6880- 3770 Fax: 81-3-6880-3771 **Korea - Daegu** Tel: 82-53-744-4301 Fax: 82-53-744-4302 **Korea - Seoul** Tel: 82-2-554-7200 Fax: 82-2-558-5932 or 82-2-558-5934 **Malaysia - Kuala Lumpur** Tel: 60-3-6201-9857 Fax: 60-3-6201-9859 **Malaysia - Penang** Tel: 60-4-227-8870 Fax: 60-4-227-4068 **Philippines - Manila** Tel: 63-2-634-9065 Fax: 63-2-634-9069 **Singapore** Tel: 65-6334-8870 Fax: 65-6334-8850 **Taiwan - Hsin Chu** Tel: 886-3-5778-366 Fax: 886-3-5770-955 **Taiwan - Kaohsiung** Tel: 886-7-213-7830 **Taiwan - Taipei** Tel: 886-2-2508-8600 Fax: 886-2-2508-0102 **Thailand - Bangkok** Tel: 66-2-694-1351 Fax: 66-2-694-1350	**Austria - Wels** Tel: 43-7242-2244-39 Fax: 43-7242-2244-393 **Denmark - Copenhagen** Tel: 45-4450-2828 Fax: 45-4485-2829 **Finland - Espoo** Tel: 358-9-4520-820 **France - Paris** Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79 **France - Saint Cloud** Tel: 33-1-30-60-70-00 **Germany - Garching** Tel: 49-8931-9700 **Germany - Haan** Tel: 49-2129-3766400 **Germany - Heilbronn** Tel: 49-7131-67-3636 **Germany - Karlsruhe** Tel: 49-721-625370 **Germany - Munich** Tel: 49-89-627-144-0 Fax: 49-89-627-144-44 **Germany - Rosenheim** Tel: 49-8031-354-560 **Israel - Ra'anana** Tel: 972-9-744-7705 **Italy - Milan** Tel: 39-0331-742611 Fax: 39-0331-466781 **Italy - Padova** Tel: 39-049-7625286 **Netherlands - Drunen** Tel: 31-416-690399 Fax: 31-416-690340 **Norway - Trondheim** Tel: 47-7289-7561 **Poland - Warsaw** Tel: 48-22-3325737 **Romania - Bucharest** Tel: 40-21-407-87-50 **Spain - Madrid** Tel: 34-91-708-08-90 Fax: 34-91-708-08-91 **Sweden - Gothenberg** Tel: 46-31-704-60-40 **Sweden - Stockholm** Tel: 46-8-5090-4654 **UK - Wokingham** Tel: 44-118-921-5800 Fax: 44-118-921-5820