ATWINC3400 Wi-Fi/BLE Network Controller - Software Design Guide

User Guide

Introduction

Atmel® SmartConnect ATWINC3400 SoC is an IEEE® 802.11 b/g/n and Bluetooth® Smart BLE network controller for applications in the Internet-of-Things. It is an ideal add-on to existing MCU solutions bringing Wi-Fi® and network capabilities through an SPI-to-Wi-Fi interface. The ATWINC3400 connects to any Atmel AVR® or Atmel | SMART™ MCU with minimal resource requirements.

Features

- Wi-Fi IEEE® 802.11 b/g/n STA and AP modes
- Wi-Fi Protected Setup (WPS)
- Discovery and provisioning via Smartphone using BLE
- Support of WEP and WPA/WPA2 personal security
- Embedded network stack protocols to offload work from the host MCU. This allows operation with a wide range of MCUs including low end MCUs.
- Embedded TCP/IP stack with BSD-style socket API
- Embedded network protocols
  - DHCP client/server
  - DNS resolver client
  - SNTP client for UTC time synchronization
- Embedded TLS security abstracted behind BSD-style socket API
- HTTP Server for optional provisioning using AP mode
- Ultra-low cost IEEE 802.11 b/g/n RF/PH/MAC SoC
- Ultra-low power Bluetooth SMART (BLE 4.0) SoC with Integrated MCU, Transceiver, Modem, MAC, PA, TR Switch, and Power Management Unit
- Fast boot from on-chip Boot ROM
- 8Mb internal Flash memory
- Low power consumption with different power saving modes
- SPI, I^2C, and UART support
- Low footprint host driver with the following capabilities:
  - Can run on 8, 16, and 32 bit MCU
  - Little and Big endian support
  - Consumes about 8KB of code memory and 1KB of data memory on host MCU

Table of Contents

1 Icon Key Identifiers

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

2 Glossary

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

3 References

[R01] Atmel-42640-Getting-Started-Guide-for-ATWINC3400WiFi-using-SAMD21-Xplained-Pro_UserGuide.

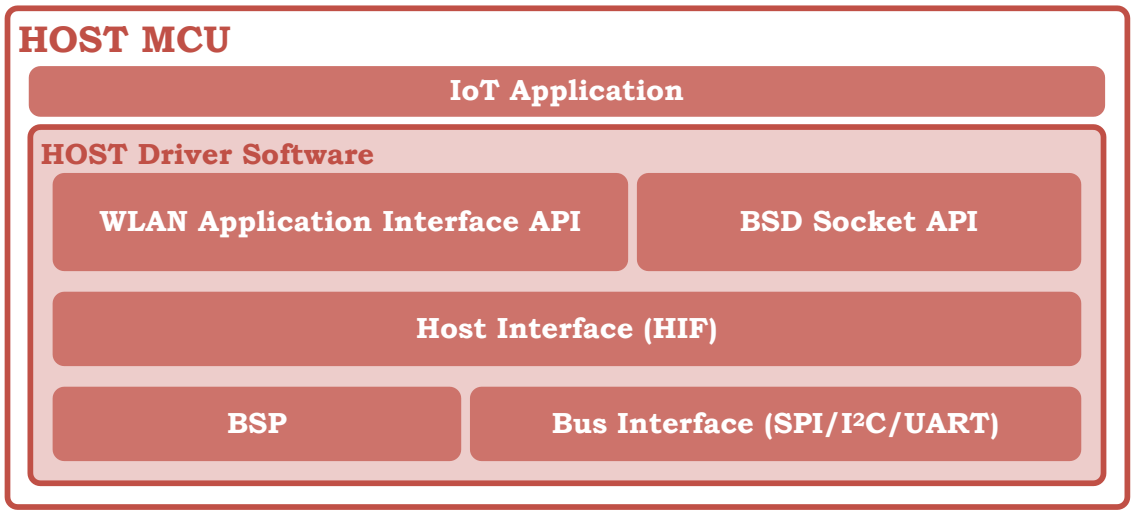
[R02] Atmel-42639-Software-Programming-Guide-for-ATWINC3400-WiFi-using-SAMD21-Xplained-Pro_UserGuide.

[R03] Atmel-42535-ATWINC3400-MR210-IEEE80211bgn-Link-Ctlr-with-Integrated-Low-Energy-Bluetooth40_Datasheet

[R04] Atmel-42683-ATWINC3400-BLE-WiFi-Scan-and-Connect-Services-Guide_UserGuide

4 Host Driver Architecture

Figure 4-1. Host Driver Software Architecture



ATWINC host driver software is a C library which provides the host MCU application with necessary APIs to perform necessary WLAN, BLE, and socket operations. Figure 4-1 shows the architecture of the ATWINC host driver software which runs on the host MCU. The components of the host driver are described in the following sub-sections.

4.1 WLAN API

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

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

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

4.2 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 ATWINC sockets and BSD sockets. Please refer to the C in this document (the API reference manual) along with the provided socket code examples will guide you to understand similarities and differences between ATWINC and BSD sockets.

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

The detailed description of the socket operations is provided in Chapter 9: ATWINC Socket Programming.

4.3 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 file: **`m2m_hif.h`**.

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

4.4 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, etc.).

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

4.5 Serial Bus Interface

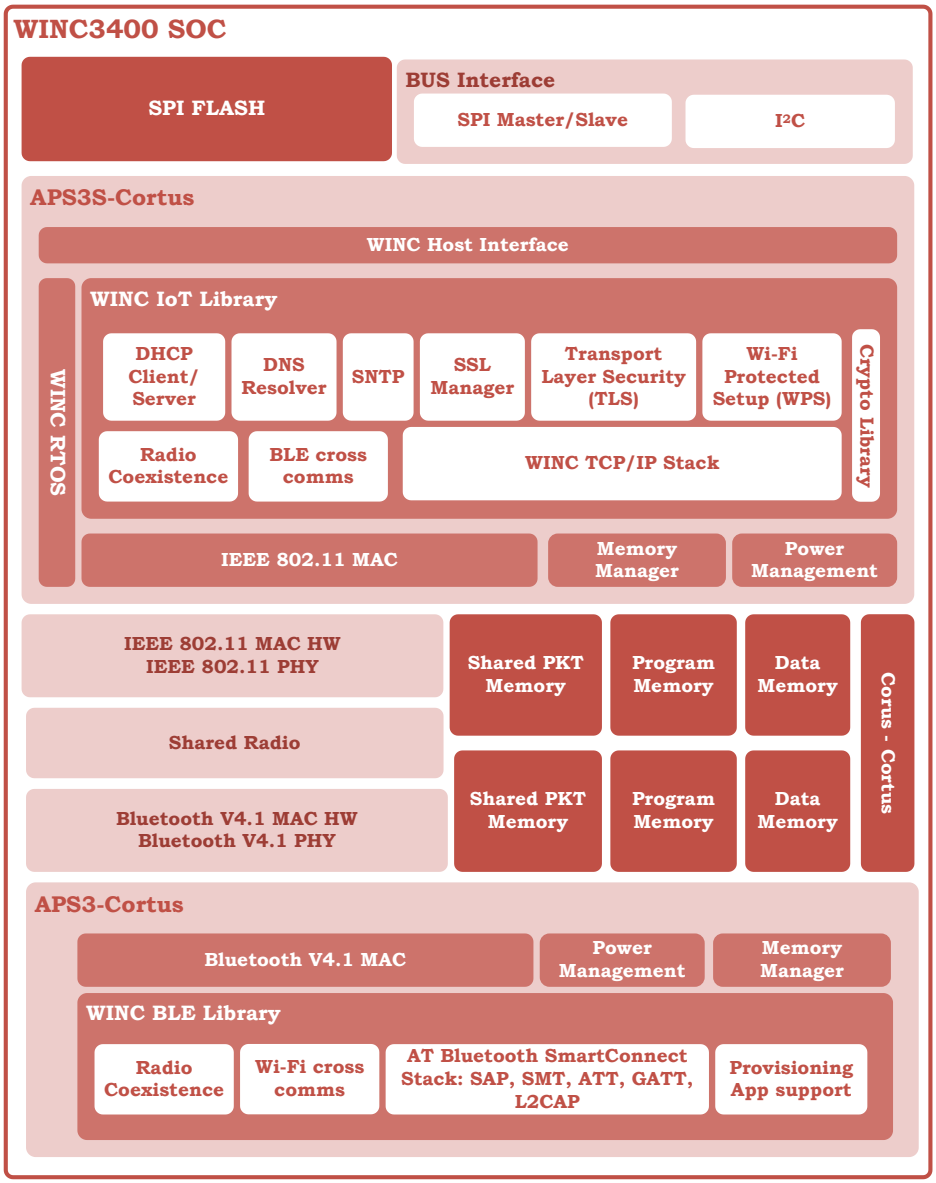
The Serial Bus Interface module abstracts the hardware associated with implementing the bus between the Host and the WINC. The serial bus interface abstracts I^2C, SPI, or UART bus interface. The basic bus access operations (Read and Write) are implemented in this module as appropriate for the interface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

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

5 WINC System Architecture

Figure 5-1 shows the WINC system architecture. The Soc contains two 32-bit CPUs: an APS3S-Cortus for Wi-Fi and an APS3-Cortus for BLE. In addition it has separate built-in Wi-Fi IEEE-802.11 and BLE 4.0 physical layers sharing a single final RF front end. The firmware for Wi-Fi comprises the Wi-Fi IEEE-802.11 MAC layer and embedded protocol stacks which offload the host MCU. The firmware for BLE implements the entrie BLE stack and privides an application supporting a profile defined for device provisioning. The components of the system are described in the following sub-sections.

Figure 5-1. WINC System Architecture



5.1 Bus Interface

Hardware logic for the supported bus types for WINC communications.

5.2 Non-volatile Storage

The SoC has an integrated 8Mb serial flash inside the WINC package (SIP). This stores both the WINC Wi-Fi firmware image and the BLE firmware image. It also stores information used by WINC firmware in the run-time.

The detailed description of the serial flash is provided in Chapter 14: ATWINC Serial Flash Memory.

5.3 CPUs

The SoC contains two 32-bit CPUs:

- An APS3S-Cortus 32-bit CPU running at 40MHz clock speed, which executes the embedded WINC Wi-Fi firmware
- An APS3-Cortus 32-bit CPU running at 26MHz clock speed, which executes the embedded WINC BLE firmware

5.4 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.

5.5 Bluetooth BLE V4.0 MAC Hardware

The BLE Medium Access Controller (MAC) encodes and decodes HCI packets, constructs baseband data packages, schedules frames and manages and monitors connection status, slot usage, data flow, routing, segmentation, and buffer control.

The core performs Link Control Layer management supporting the main BLE states, including advertising and connection.

5.6 Program Memory

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

292KB Instruction RAM is provided for execution of the WINC BLE firmware code.

5.7 Data Memory

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

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

5.8 Shared Packet Memory

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

32KB memory is provided for BLE TX/RX packet management. It is shared between the BLE MAC hardware and the CPU. This memory is managed by the BLE Memory Manager SW component.

5.9 IEEE 802.11 MAC Firmware

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

5.10 Bluetooth V2.1 MAC Firmware

The BLE subsystem implements all the critical real-time functions required for full compliance with Specification of the Bluetooth System, v4.1, Bluetooth SIG. The firmware implements an integrated Bluetooth Low Energy stack which is Bluetooth V4.1 compliant. The firmware supports access to the GAP, SMP, ATT, GATT client / server and L2CAP service layer protocols. In the ATWINC3400 these services are used by a built-in application for WI-Fi provisioning.

5.11 Memory Managers

The memory managers on both the Wi-Fi and BLE sides are responsible for the allocation and de-allocation of memory chunks in both shared packet memory and data memory.

5.12 Power Managements

The Power Management modules on both the Wi-Fi and BLE sides are responsible for handling different power saving modes supported by the ATWINC and coordinating these modes with the Wi-Fi and BLE transceiver.

5.13 ATWINC RTOS

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

5.14 ATWINC IoT Library

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

- **ATWINC TCP/IP STACK**

  The ATWINC TCP/IP is an IPv4.0 stack based on the µIP TCP/IP stack (pronounced micro-IP).

  µIP 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 ATWINC TCP/IP stack adds to the original µIP implementation several enhancements to boost TCP and UDP throughput.

- **DHCP CLIENT/SERVER**

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

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

- **DNS RESOLVER**

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

- **SNTP**

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

- **TRANSPORT LAYER SECURITY**

  For TLS implementation, see Chapter 10: Transport Layer Security (TLS) for details.

- **WI-FI PROTECTED SETUP**

  For WPS protocol implementation, see Section 12.3: Wi-Fi Protected Setup (WPS) for details.

- **CRYPTO LIBRARY**

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

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

6 ATWINC Initialization and Simple Application

After powering up the ATWINC device, a set of synchronous initialization sequences must be executed for the correct operation of the Wi-Fi functions and BLE functions. This chapter aims to explain the different steps required during the initialization phase of the system. The host MCU does not communicate directly with the BLE function but controls it through messages to the Wi-Fi MCU. This allows a single interface and driver to be used to communicate to the ATWINC device.

After initialization, the host MCU application is required to call the ATWINC driver entry point to handle events from ATWINC firmware.

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

WARNING Failure to complete any of the initialization steps will result in failure in ATWINC startup.

6.1 BSP Initialization

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

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

6.2 ATWINC Host Driver Initialization

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

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

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

6.3 Socket Layer Initialization

Socket layer initialization is carried out by calling the **socketInit** API. It must be called prior to any socket activity. Refer to Section 9.2.1 for more information about socket initialization and programming.

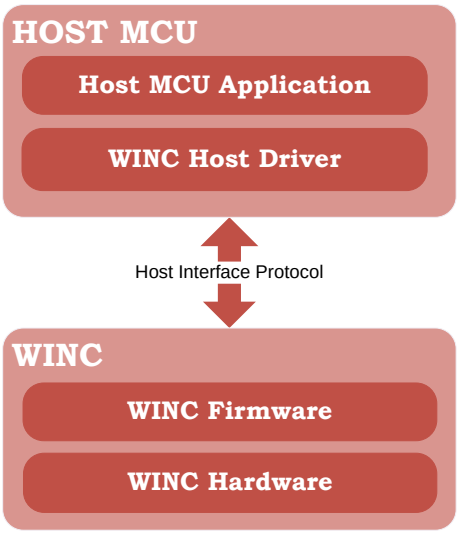
6.4 ATWINC Event Handling

The ATWINC host driver API allows the host MCU application to interact with the ATWINC firmware. To facilitate interaction, the ATWINC driver implements the *Host Interface (HIF) Protocol* described in Chapter 16:

[1] Refer to ATWINC3400 datasheet [R02] for more information about ATWINC hardware reset sequence.

Host Interface Protocol. The HIF protocol defines how to serialize and de-serializes API requests and response callbacks over the serial bus interface: I²C, UART, or SPI.

Figure 6-1. ATWINC System Architecture



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

6.4.1 Asynchronous Events

Some ATWINC host driver APIs are synchronous function calls, where the result is ready by the return of the function. However, most ATWINC 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 ATWINC 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 ATWINC 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 ATWINC firmware to the host MCU in FCFS manner. For instance, suppose that host MCU application has two open sockets; socket 1 and socket 2. If ATWINC receives socket 1 data followed by socket 2 data, then HIF shall deliver socket data in two HIF protocol messages in the order they were received. HIF does not allow reading socket 2 data before socket 1 data.

6.4.2 Interrupt Handling

The HIF interrupts the host MCU when one or more events are pending in ATWINC firmware. The host MCU application is a big state machine which processes received data and events when ATWINC driver calls the event callback function(s). In order to receive event callbacks, the host MCU application is required to call the

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

m2m_wifi_handle_events API to let the host driver retrieve and process the pending events from the ATWINC firmware. It is recommended to call this function either:

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

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

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

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

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

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

6.5 Code Example

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

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

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

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

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

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

7 ATWINC Configuration

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

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

7.1 Device Parameters

7.1.1 System Time

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

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

7.1.2 Firmware and HIF Version

During startup, the host driver requests the firmware version through `m2m_wifi_get_firmware_version` API which returns the structure `tstrM2mRev` containing the version number and host interface (HIF) level of the current ATWINC firmware.

WARNING If the HIF level of the current driver is not equal to the HIF level of the current ATWINC firmware, the driver initialization will fail.

The version parameters provided are:

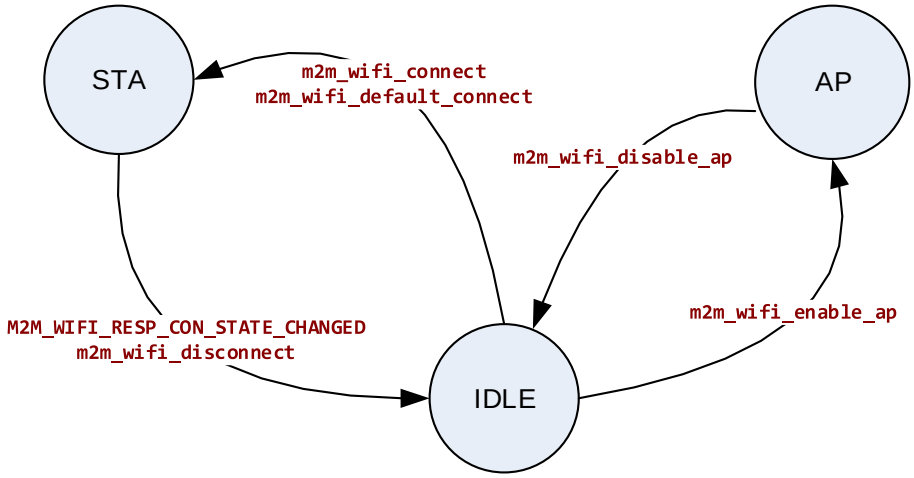
- `M2M_HIF_LEVEL`: Host Interface Level for driver/firmware compatibility
- `M2M_FIRMWARE_VERSION_MAJOR_NO`: Firmware Major release version number
- `M2M_FIRMWARE_VERSION_MINOR_NO`: Firmware Minor release version number
- `M2M_FIRMWARE_VERSION_PATCH_NO`: Firmware Patch release version number

7.2 ATWINC Modes of Operation

The ATWINC firmware supports the following modes of operation:

- Idle Mode
- Wi-Fi STA Mode
- Wi-Fi Hotspot (AP)

Figure 7-1. ATWINC Modes of Operation



7.2.1 Idle Mode

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

7.2.2 Wi-Fi Station Mode

ATWINC enters station (STA) mode when the host MCU requests connection to an AP using the **m2m_wifi_connect** or **m2m_wifi_default_connect** APIs. ATWINC 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. ATWINC firmware ignores mode change requests while in this mode until ATWINC exits the mode.

TIPS

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

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

7.2.3 Wi-Fi Hotspot (AP) Mode

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

TIPS

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

For more information about this mode, refer to Chapter 11: Wi-Fi AP Mode.

7.3 Network Parameters

7.3.1 Wi-Fi MAC Address

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

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

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

For more details, refer to description of the following APIs in the WINC3400_IoT_SW_APIs.chm that was supplied in the WINC3400_IoT_REL software package.

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

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

7.3.2 IP Address

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

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

7.4 Power Saving Parameters

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

7.4.1 Power Saving Modes

ATWINC firmware supports multiple power saving modes which provide flexibility to the host MCU application to tweak the system power consumption. The host MCU can configure the ATWINC power saving policy using

the **m2m_wifi_set_sleep_mode** and **m2m_wifi_set_lsn_int** APIs. ATWINC supports the following power saving modes:

- **M2M_PS_MANUAL**
- **M2M_PS_AUTOMATIC**
- **M2M_PS_H_AUTOMATIC**
- **M2M_PS_DEEP_AUTOMATIC**

TIPS **M2M_PS_DEEP_AUTOMATIC** mode recommended for most applications.

7.4.1.1 M2M_PS_MANUAL

This is a fully host-driven power saving mode.

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

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

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

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

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

[3] Refer to ATWINC datasheet in [R02] for hardware power off sequence.

7.4.1.2 M2M_PS_AUTOMATIC

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

7.4.1.3 M2M_PS_H_AUTOMATIC

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

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

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

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

7.4.1.4 M2M_PS_DEEP_AUTOMATIC

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

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

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

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

7.4.2 Configuring Listen Interval and DTIM Monitoring

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

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

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

TIPS

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

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

8 Wi-Fi Station Mode

This chapter provides information about ATWINC Wi-Fi station (STA) mode described in Section 7.2.2. Wi-Fi station mode involves scan operation; association to an AP using parameters (SSID and credentials) provided by host MCU or using AP parameters stored in ATWINC non-volatile storage (default connection). This chapter also provides information about supported security modes along with code examples.

8.1 Scan Configuration Parameters

8.1.1 Scan Region

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

8.1.2 Scan Options

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

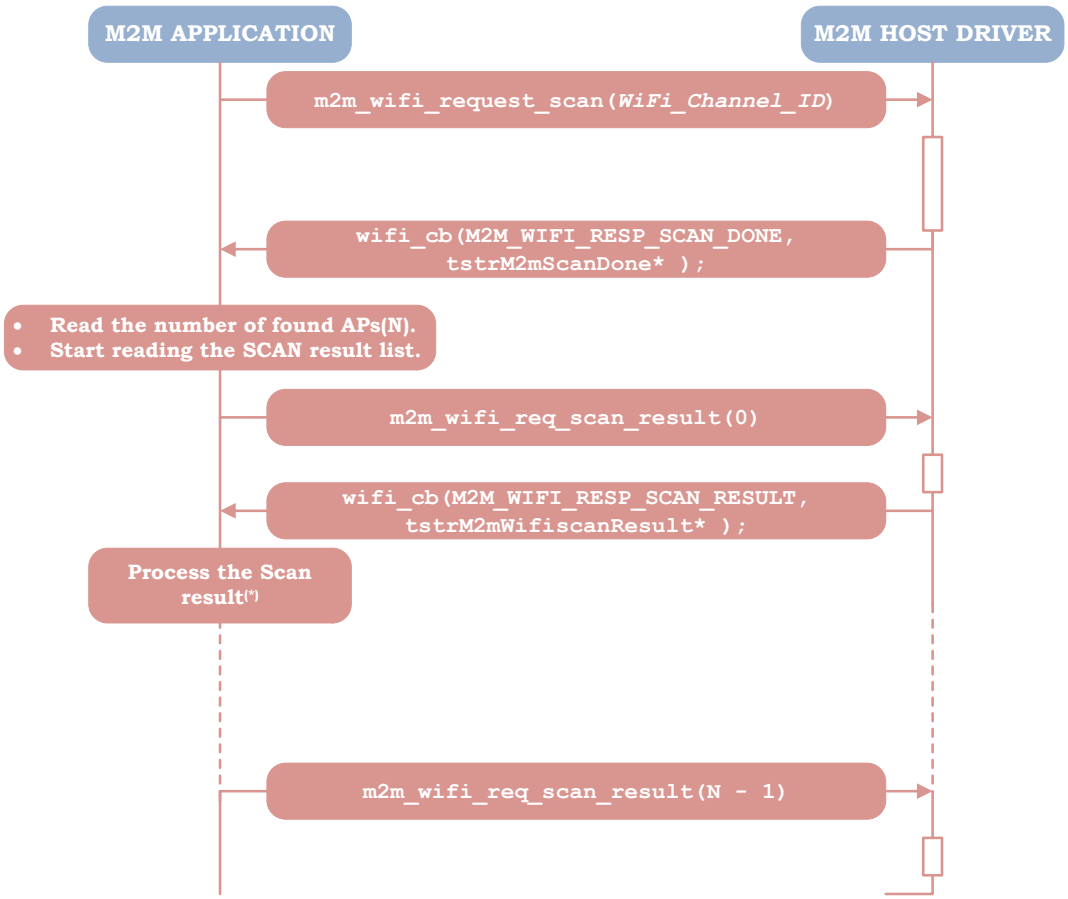
8.2 Wi-Fi Scan

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

The scan response time depends on the scan options. For instance, if the host MCU application requests to scan all channels, the scan time will be equal to N**`oOfChannels (14) * M2M_SCAN_MIN_NUM_SLOTS* M2M_SCAN_MIN_SLOT_TIME`** (refer to the WINC3400_IoT_SW_APIs.chm that was supplied in the WINC3400_IoT_REL software package on how to customize the scan parameters).

The scan operation is illustrated in Figure 8-1.

Figure 8-1. Wi-Fi Scan Operation



8.3 On Demand Wi-Fi Connection

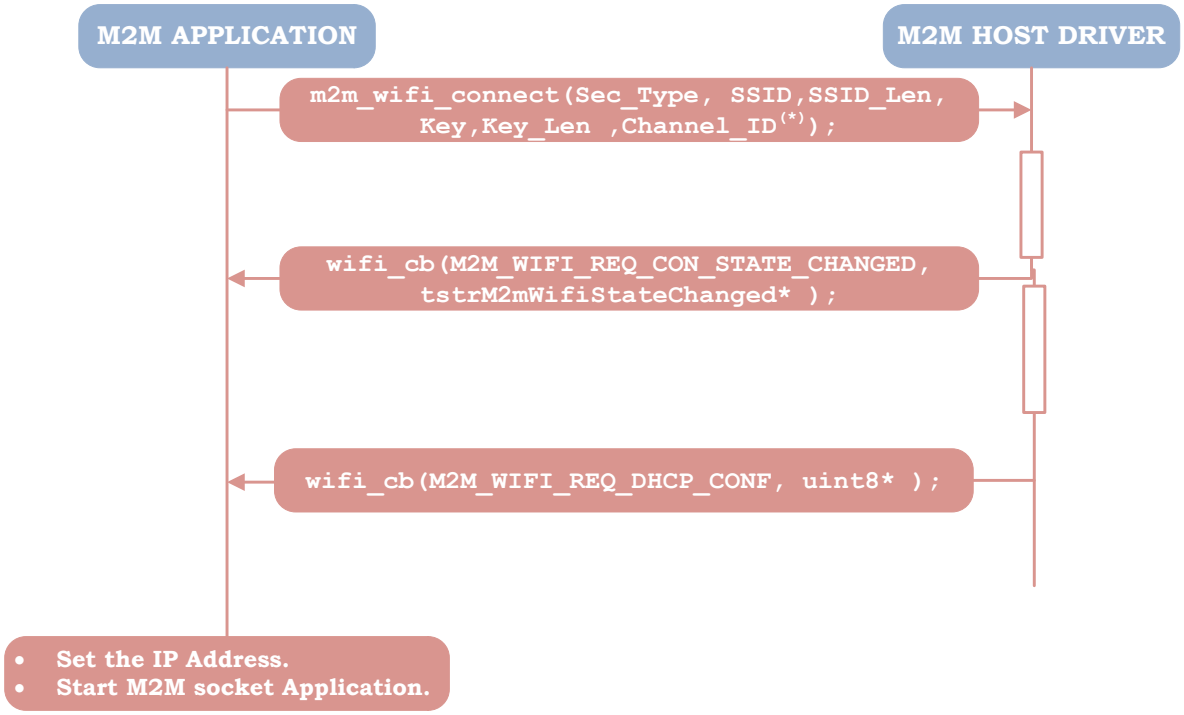
The host MCU application may establish a Wi-Fi connection on demand if all the required connection parameters (SSID, security credentials, etc.) are known to the application. To start a Wi-Fi connection on demand, the application shall call the API **m2m_wifi_connect**.

TIPS

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

The Wi-Fi on demand connection operation is described in Figure 8-2.

Figure 8-2. On Demand Wi-Fi Connection



8.4 Default Connection

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

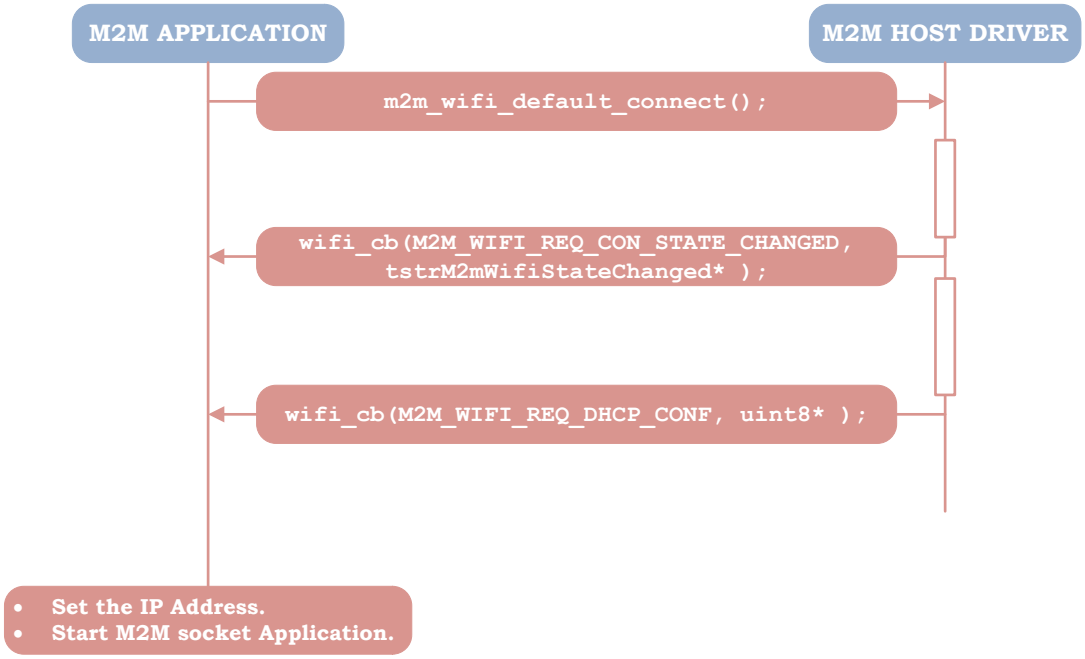
Default connection relies on the connection profiles provisioned into ATWINC serial flash via the provisioning method described in Chapter 12: Provisioning. Alternatively, connection profiles are created and stored in ATWINC serial flash when the host MCU application successfully connects once to an AP using the **m2m_wifi_connect** API described in Section 8.3. If there are no cached profiles or if a connection cannot be established with any of the cached profiles, an event of type **M2M_WIFI_RESP_DEFAULT_CONNECT** is delivered to the host driver indicating failure.

Upon successful default connection, the host application can read the current Wi-Fi connection status information by calling the **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, and IP address obtained by DHCP.

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

The Wi-Fi default connection operation is described in Figure 8-3.

Figure 8-3. Wi-Fi Default Connection



8.5 Wi-Fi Security

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

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

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

8.6 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)
          {
          }
     }
}
```

9 ATWINC Socket Programming

9.1 Overview

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

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

9.1.1 ATWINC Socket Types

The ATWINC sockets API provides two types of sockets:

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

9.1.2 Socket Properties

Each ATWINC socket is identified by a unique combination of:

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

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

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

9.1.3 Limitations

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

9.2 ATWINC Sockets API

9.2.1 API Prerequisites

- **The C header file** "`socket.h`": Includes all the necessary socket API function declarations. When using any ATWINC sockets API described in the following sections, the host MCU application should to include the socket.h header file.
- **Initialization**: The ATWINC socket API shall be initialized once before calling any sockets API function. This is done by using the "`socketInit`" API described in the Section 9.2.3.

9.2.2 Non-blocking Asynchronous Socket APIs

Most ATWINC socket APIs are asynchronous function calls that do not block the host MCU application. The behavior of ATWINC asynchronous APIs is described in Section 6.4.1.

For example, the host MCU application can register an application-defined socket event callback function using the ATWINC 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 ATWINC socket API to call the registered socket callback when the connection is established or if there was a connection timeout. The socket callback function provides the necessary information to determine if the connection was successful or not.

9.2.3 Socket API Functions

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

9.2.3.1 socketInit

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

9.2.3.2 registerSocketCallback

The **registerSocketCallback** function allows the host MCU application to provide the ATWINC 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 ATWINC socket API whenever a socket event occurs. Within this handler, the host MCU application should provide an application-defined logic that handles the events of interest.

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

9.2.3.3 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.

9.2.3.4 connect

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

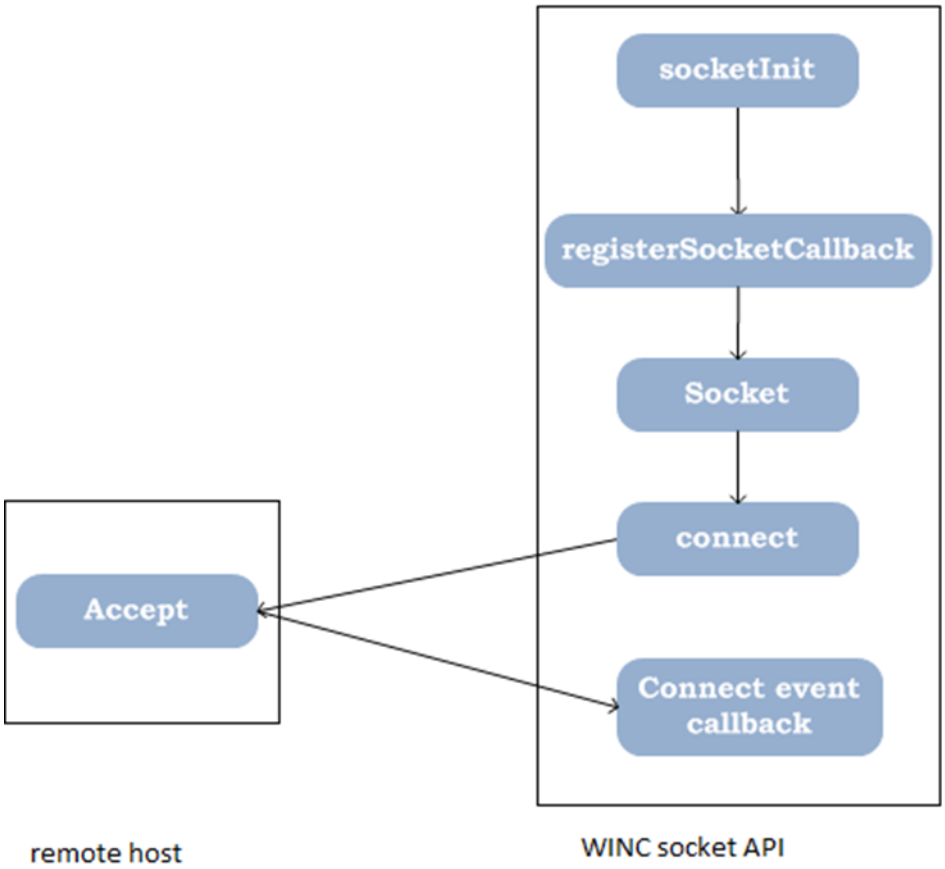
The **connect** function will result in a **SOCKET_MSG_CONNECT** sent to the socket event handler callback upon completion. The connect event will be sent when the TCP server accepts the connection or, if no remote host response is received, after a timeout interval of approximately 30 seconds.

TIPS

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

Figure 9-1 shows the ATWINC socket API connect to remote server host.

Figure 9-1. TCP Client API Call Sequence



9.2.3.5 bind

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

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

9.2.3.6 listen

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

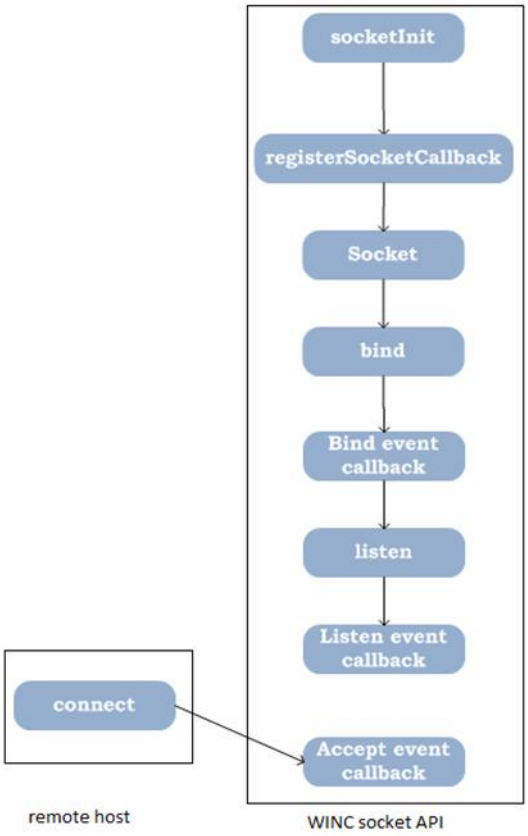
When a remote peer establishes a connection, a **SOCKET_MSG_ACCEPT** event notification is sent to the application.

9.2.3.7 accept

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

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

Figure 9-2. TCP Server API Call Sequence



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

9.2.3.8 send

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

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

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

9.2.3.9 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.

9.2.3.10 recv / recvfrom

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

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

The received data size indicates the status:

- Positive: Data received.
- Zero: Socket connection is terminated.
- Negative value: This indicates an error.

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

9.2.3.11 close

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

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

9.2.3.12 setsockopt

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

The options supported are:

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

TIPS Disabling send/sendto callbacks using **setsockopts** is recommended in high throughput applications.

9.2.3.13 gethostbyname

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

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

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

9.2.4 Summary

Table 9-1 summarizes the ATWINC socket API and shows its compatibility with BSD socket APIs:

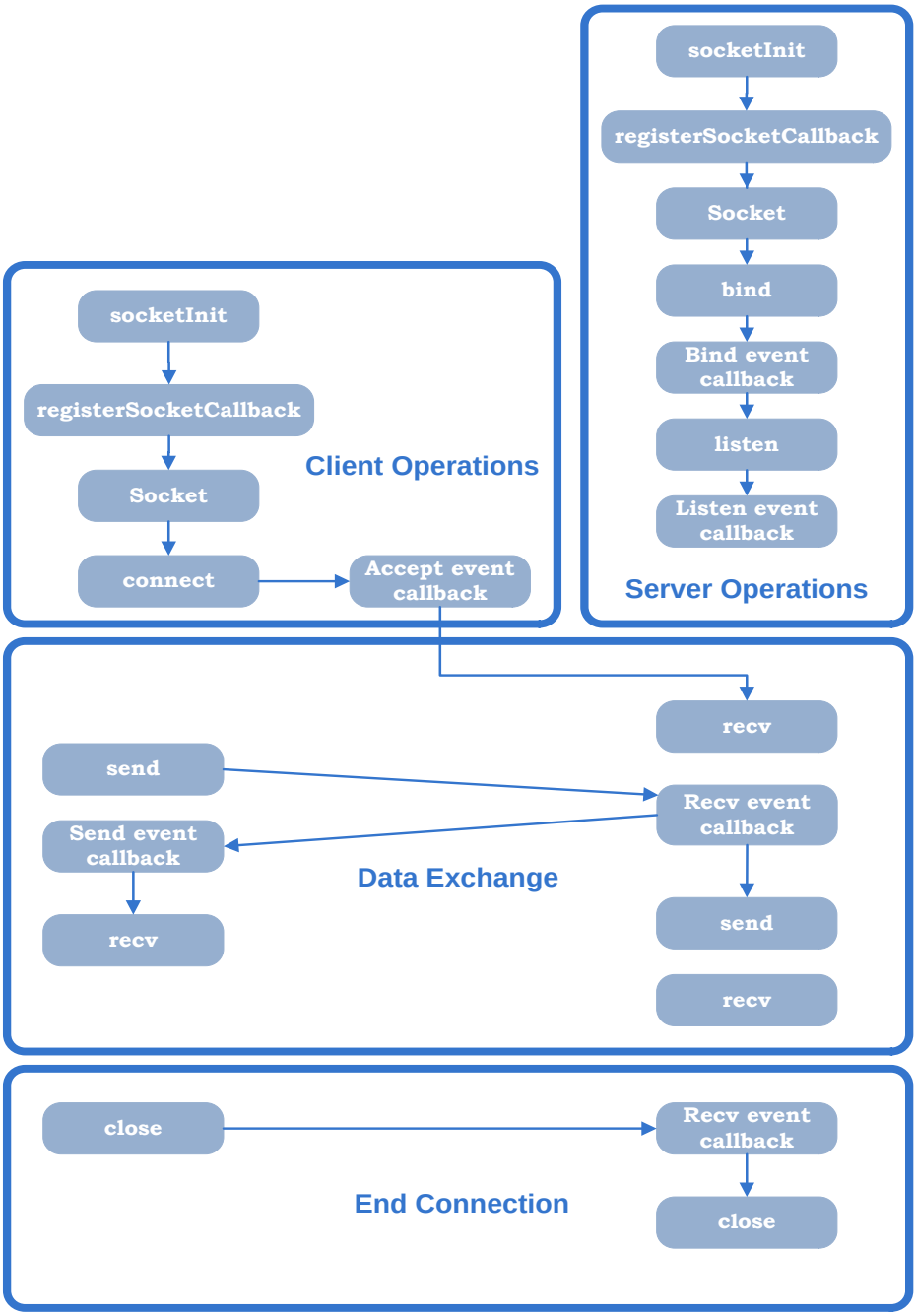
Table 9-1. ATWINC Socket API Summary

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

9.3 Socket Connection Flow

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

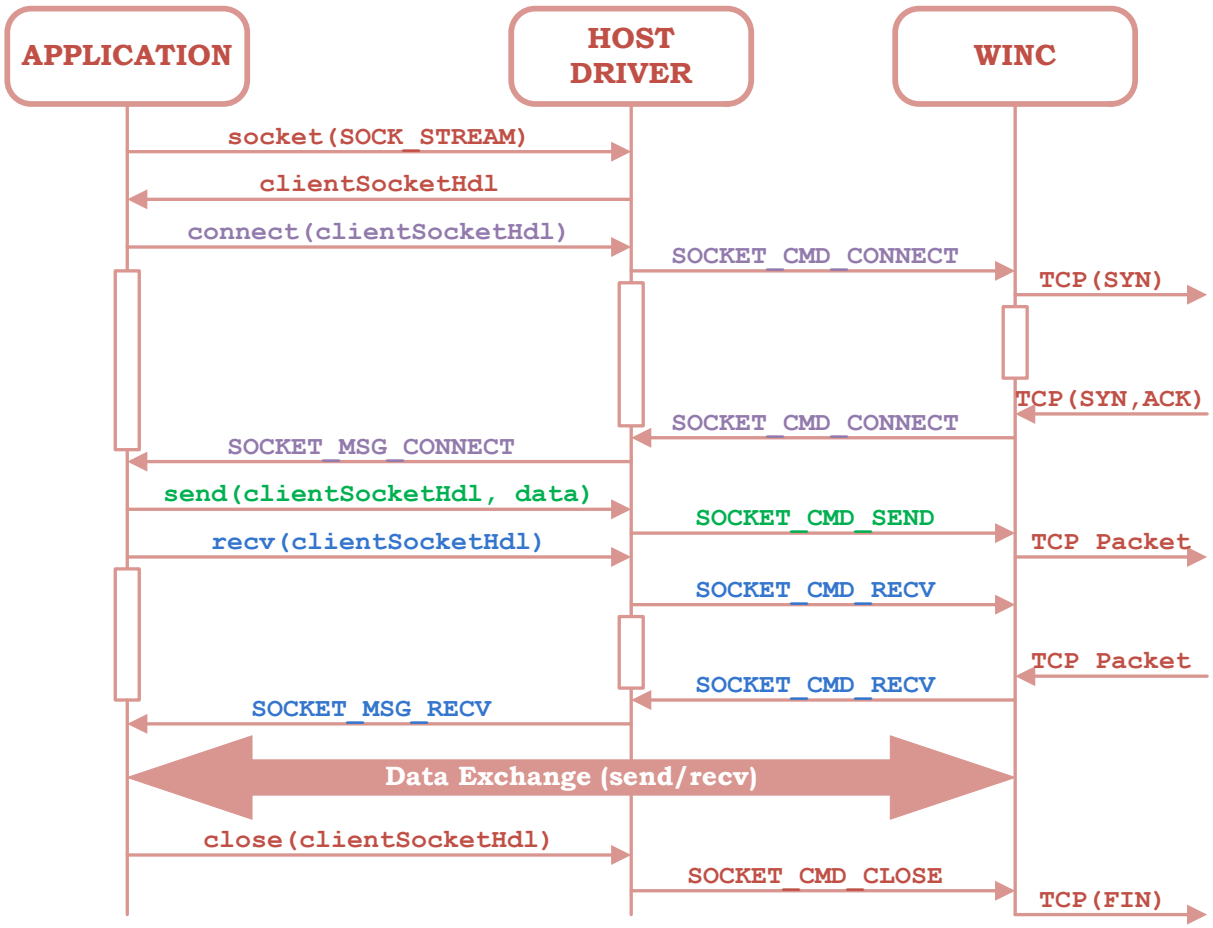
Figure 9-3. Typical Socket Connection Flow



9.3.1 TCP Client Operation

Figure 9-4 shows the message flow for transferring data with a TCP client.

Figure 9-4. TCP Client Sequence Diagram

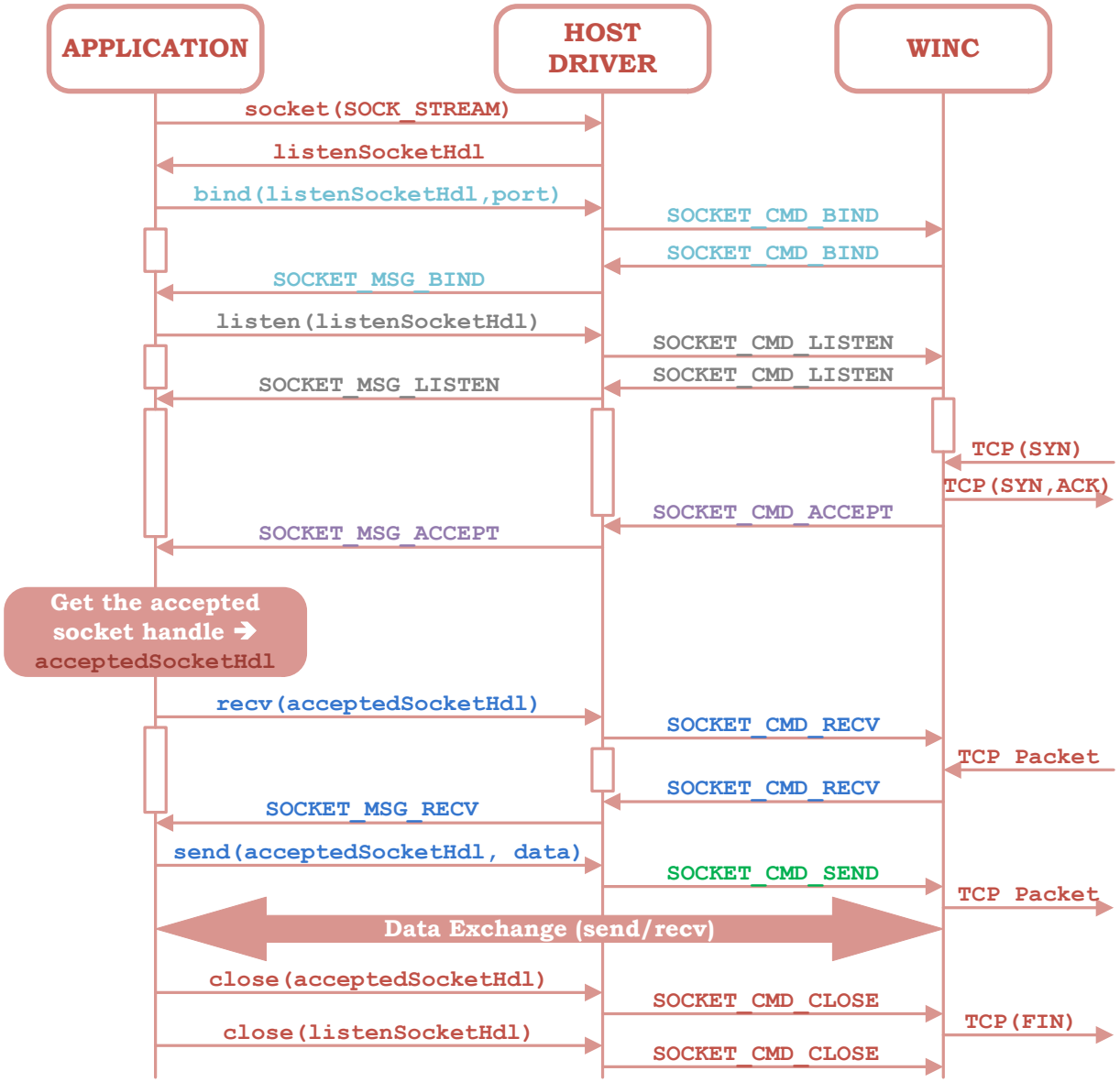


Notes: 1. The host application must register a socket notification callback function. The function must be of type **tpfAppSocketCb** and must handle socket event notifications appropriately.

2. If the client knows the IP of the server, it may call **connect** directly as shown in Figure 9-4. If only the server URL is known, then the application should resolve the server URL first calling the **gethostbyname** API.

9.3.2 TCP Server Operation

Figure 9-5. TCP Server Sequence Diagram

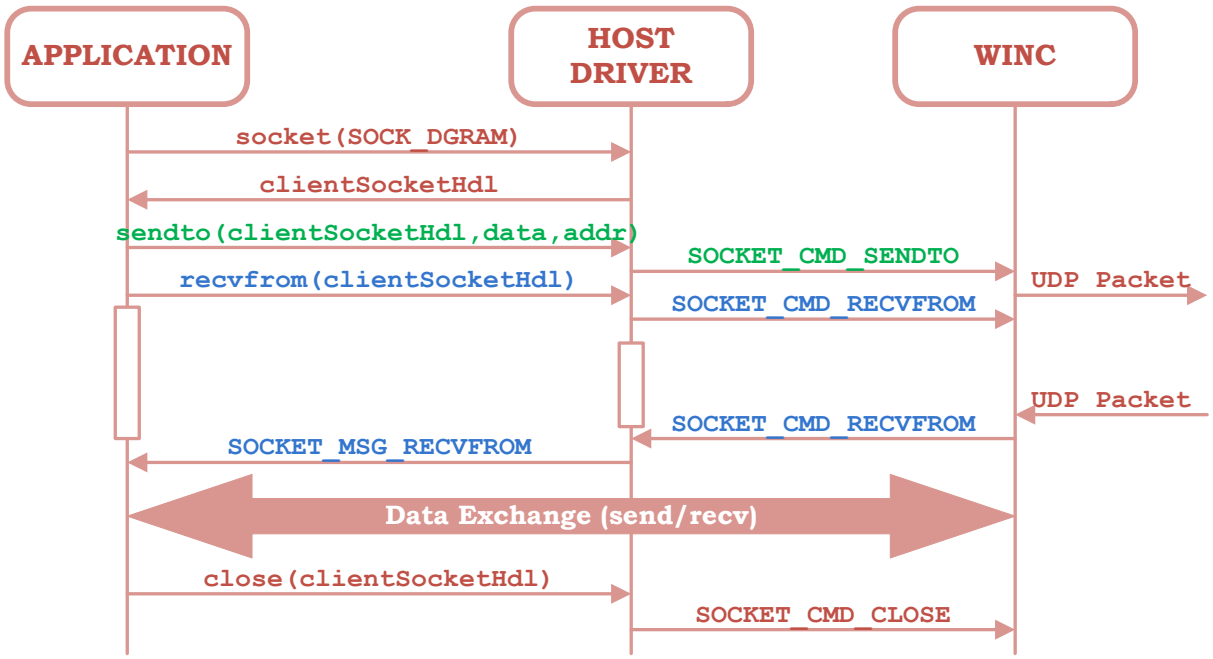


Note: The host application must register a socket notification callback function. The function must be of type: **`tpfAppSocketCb`** and must handle socket event notifications appropriately.

9.3.3 UDP Client Operation

Figure 9-6 shows the message flow for transferring data with a UDP client.

Figure 9-6. UDP Client Sequence Diagram

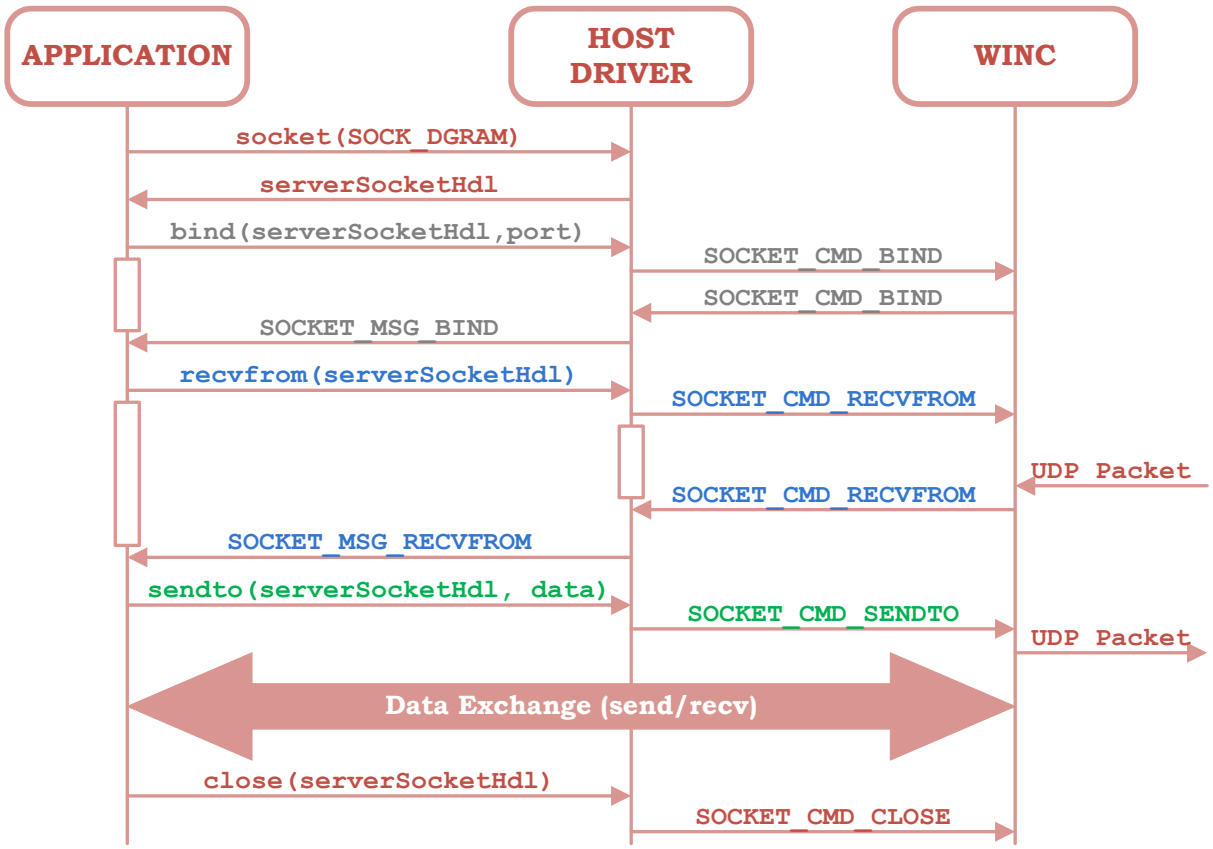


Notes:
1. The first send message must be performed with the **sendto** API with the destination address specified.
2. If further messages are to be sent to the same address, the **send** API can be used.
3. **recv** can be used instead of **recvfrom**.

9.3.4 UDP Server Operation

Figure 9-7 shows the message flow for transferring data after establishing a UDP server.

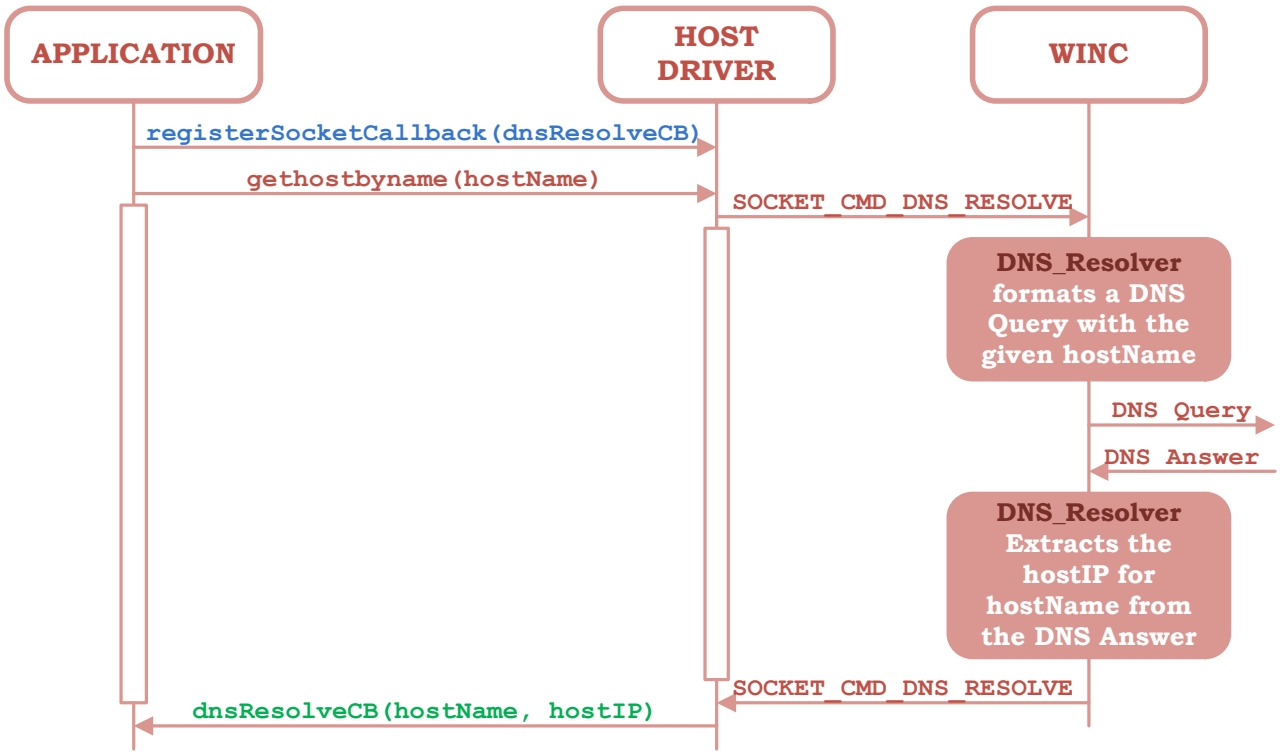
Figure 9-7. UDP Server Sequence Diagram



9.3.5 DNS Host Name Resolution

Figure 9-8 shows the message flow for resolving a URL to and IP address.

Figure 9-8. DNS Resolution Sequence



Notes:
1. The host application requests to resolve hostname (e.g. www.foobar.com), by calling the function **gethostbyname**.
2. Before calling the **gethostbyname**, the application must register a DNS response callback function using the function **registerSocketCallback**.
3. After the ATWINC DNS_Resolver module obtains the IP address (hostIP) corresponding to the given HostName, the **dnsResolveCB** will be called with the hostIP.
4. If an error occurs or if the DNS request encounters a timeout, the **dnsResolveCB** is called with IP address value zero indicating a failure to resolve the domain name.

9.4 Example Code

This section provides code samples for different socket applications. For additional socket code example, refer to [R03] ATWINC3400 Software Programming Guide.

9.4.1 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);
}
```

9.4.2 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));
        }
}
```

9.4.3 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);
    }
}
```

9.4.4 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));
    }
}
```

10 Transport Layer Security (TLS)

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

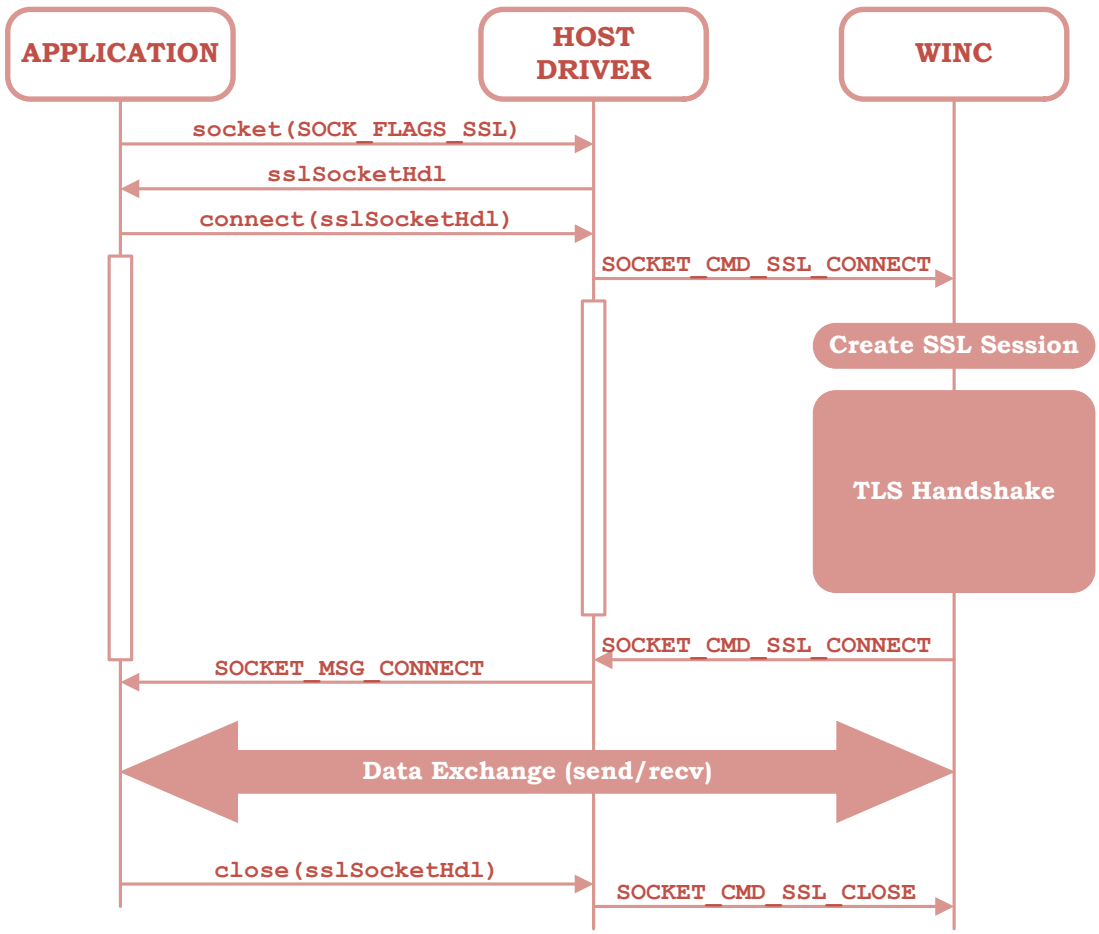
INFO ATWINC implements the Transport Layer Security protocol TLS v1.0, client mode.

10.1 TLS Connection Establishment

From the application's point of view, the TLS functionality is wrapped behind the socket APIs. This hides the complexity of TLS from the application which could use the TLS in the same fashion as the TCP client. The main difference between TLS sockets and regular TCP sockets is that the application sets the **SOCKET_FLAGS_SSL** while creating the TLS client socket. The detailed sequence of TLS connection establishment is described in Figure 10-1.

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

Figure 10-1. TLS Connection Establishment



10.2 Server Certificate Installation

10.2.1 Technical Background

10.2.1.1 Public Key Infrastructure

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

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

10.2.1.2 TLS Server Authentication

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

10.2.2 Adding a Certificate to the ATWINC Trusted Root Certificate Store

- Before connecting to a TLS Server, the root certificate of the server must be installed on the ATWINC3400. If this is not done, the TLS Connection to the server is aborted locally by ATWINC.
- The root certificate must be in `DER` format. If it is not provided in `DER` format, it must be converted before installation. See Appendix A for certificate formats and conversion methods.
- To install the certificate, execute **root_certificate_downloader.exe** with the following syntax:

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

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

10.3 ATWINC TLS Limitations

10.3.1 Modes of Operation

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

10.3.2 Concurrent Connections

Only two TLS concurrent connections are allowed.

10.3.3 Supported Cipher Suites

The current implementation is limited to the following cipher suites:

- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_256_CBC_SHA256

10.3.4 Supported Hash Algorithms

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

10.4 SSL Client Code Example

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

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

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

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

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

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

     if(u32ServerIP != 0)
     {
          sslSocketHdl = socket(AF_INET,SOCK_STREAM,u8Flags);
          if(sslSocketHdl >= 0)
```

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

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

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

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

11 Wi-Fi AP Mode

11.1 Overview

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

11.2 Setting ATWINC AP Mode

ATWINC AP mode configuration parameters should be set first by using the **tstrM2MAPConfig** structure.

There are two functions to enable/disable AP mode:

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

For more information about structure and APIs, refer to the API reference in the WINC3400_IoT_SW_APIs.chm that was supplied in the WINC3400_IoT_REL software package.

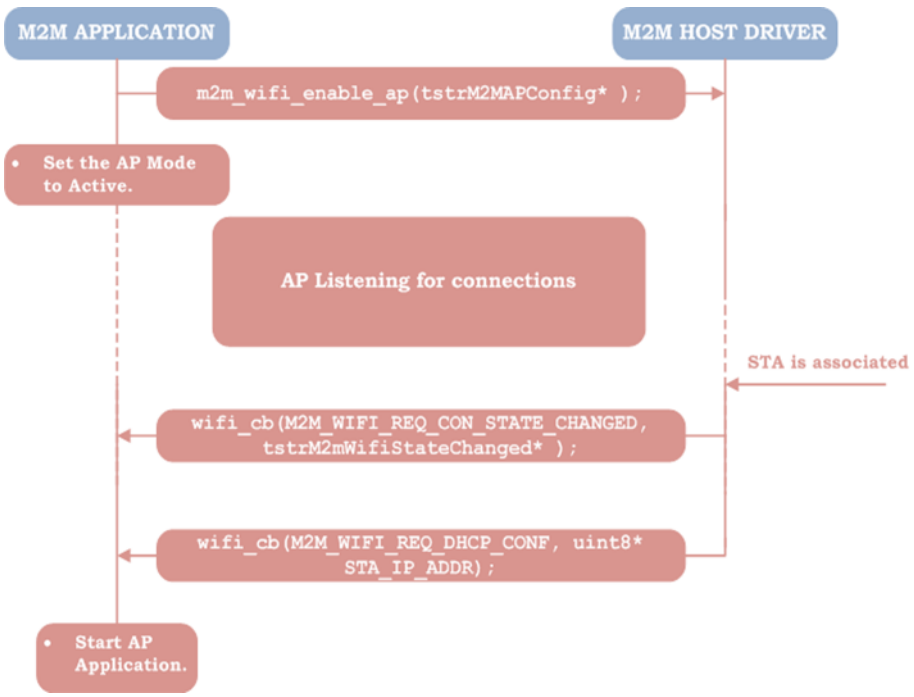
11.3 Limitations

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

11.4 Sequence Diagram

Once the AP mode has been established, no data interface exists until after a station associates to the AP. Therefore the application needs to wait until it receives a notification via an event callback. This process is shown in Figure 11-1.

Figure 11-1. ATWINC AP Mode Establishment



11.5 AP Mode Code Example

The following example shows how to configure ATWINC 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);
		}
	}
}
```

12 Provisioning

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

The ATWINC3400 software supports three methods of provisioning:

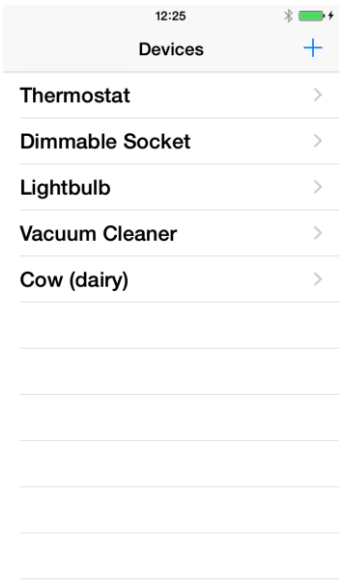
- BLE based in which a SmartPhone detects the ATWINC and uses an APP to transfer the information from the user to the ATWINC3400
- HTTP-based (browser) provisioning while ATWINC is in AP mode
- Wi-Fi Protected Setup (WPS)

12.1 BLE Provisioning

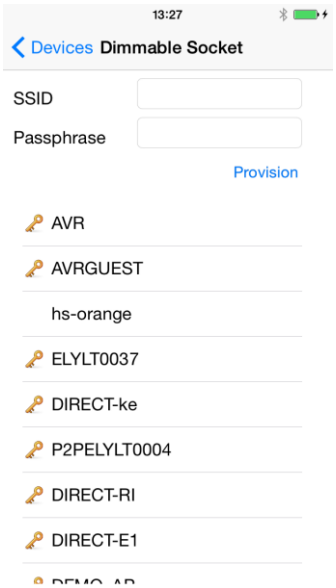
This mode of provisioning is a major feature of the ATWINC3400 and is likely to be the method of choice. It has the advantage that it is simple and intuitive for the user and does not disrupt normal SmartPhone operation during the process.

In this method the host MCU instructs the ATWINC3400 to enable the BLE provisioning mode using API **`m2m_wifi_start_ble_provision_mode`**. This causes the BLE to be activated and to start issuing BLE beacons. The beacons will be detected by a suitable SmartPhone, which will inform the user that the device is available for provisioning and provide information about the APP required to complete the process. When the user obtains/runs the APP it will request the provisioning information and transfer to the ATWINC3400. The ATWINC3400 will store the information and then connect to the specified AP, thus making itself ready for use.

The user needs to load the Atmel_IoT APP on either iOS or Andorid Smartphone. Upon launching the APP will search for ATWINC3400 based products that are available for provisioning. A list of products will be displayed using their user-friendly names:



When the user selects a product and clicks the '>' symbol, the Access Points visible to the IoT device will be shown on the screen:



The user may then select the Access Point to which he wants to connect the IoT device and enter the Passphrase at the top of the screen. After the information is entered, the provisioning information is passed to the device using BLE and then passed on to the Wi-Fi component of the ATWINC3400. The Wi-Fi component passes the credentials to the host using the callback **`M2M_WIFI_RESP_PROVISION_INFO`** and placing the information in the structure **`tstrM2MProvisionInfo`**.

Figure 12-1. BLE Provisioning Sequence Diagram

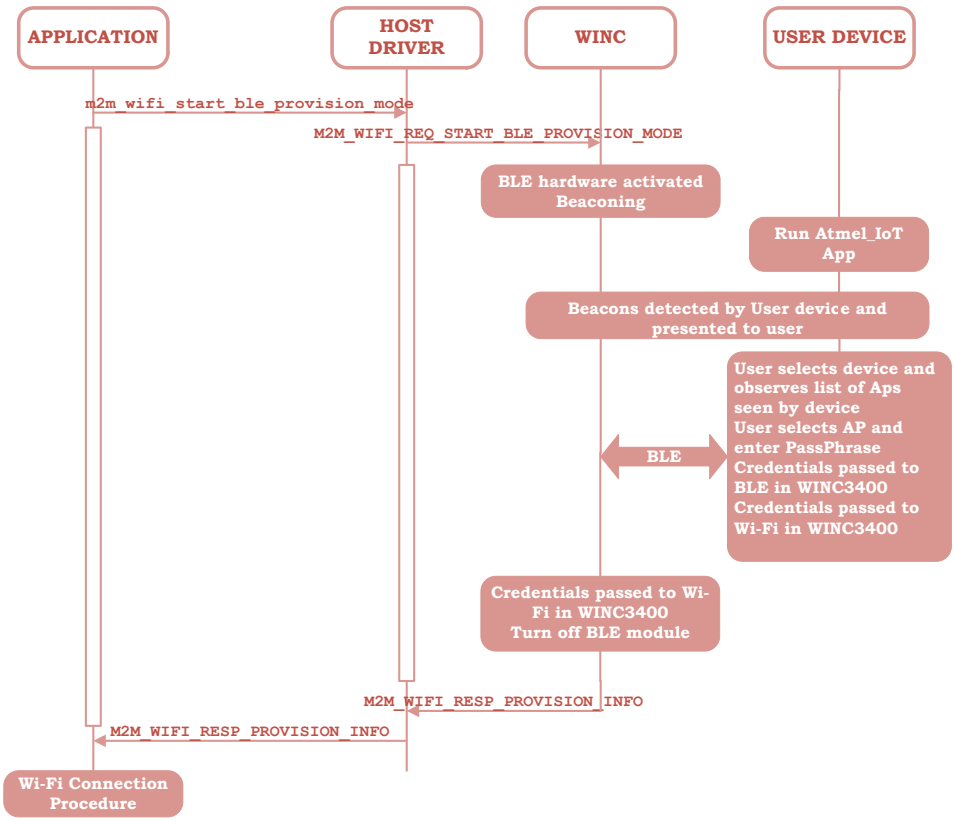


Figure 12-1 shows the provisioning operation for an ATWINC device. The detailed steps are described in the code example below.

12.1.1 BLE 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))
     {

          m2m_wifi_start_ble_provision_mode();

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

12.2 HTTP Provisioning

12.2.1 Introduction

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

12.2.2 Limitations

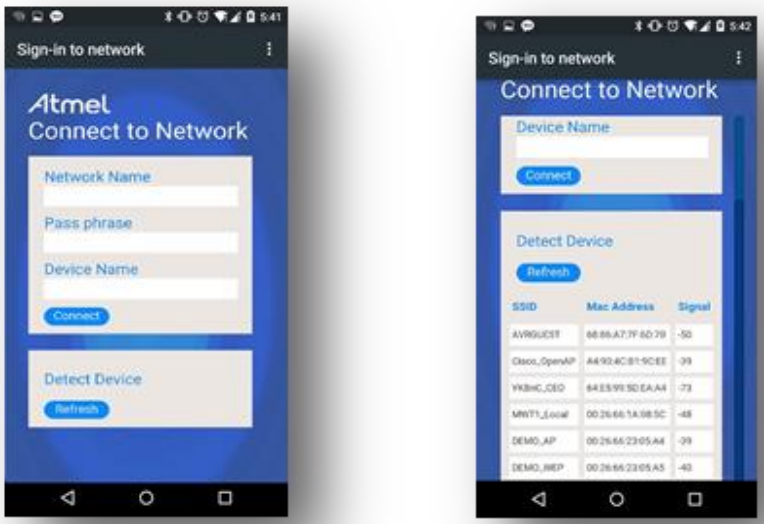
The current implementation of the HTTP Provisioning has the following limitations:

- ATWINC AP limitations apply in provisioning mode. See Section 11.3: Limitations for a list of AP mode limitations.
- Provisioning uses AP mode with open security. No Wi-Fi security nor application level security (e.g. TLS) is used and therefore the AP credentials entered by the user are sent on the clear and can be seen by eavesdroppers.
- The ATWINC Provisioning home page is a static HTML page. No server-side scripting allowed in the ATWINC 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

12.2.3 Basic Approach

The HTTP provisioning home page is as shown in Figure 12-2.

Figure 12-2. ATWINC HTTP Provisioning Page



12.2.4 Provisioning Control Flow

Figure 12-3. HTTP Provisioning Sequence Diagram

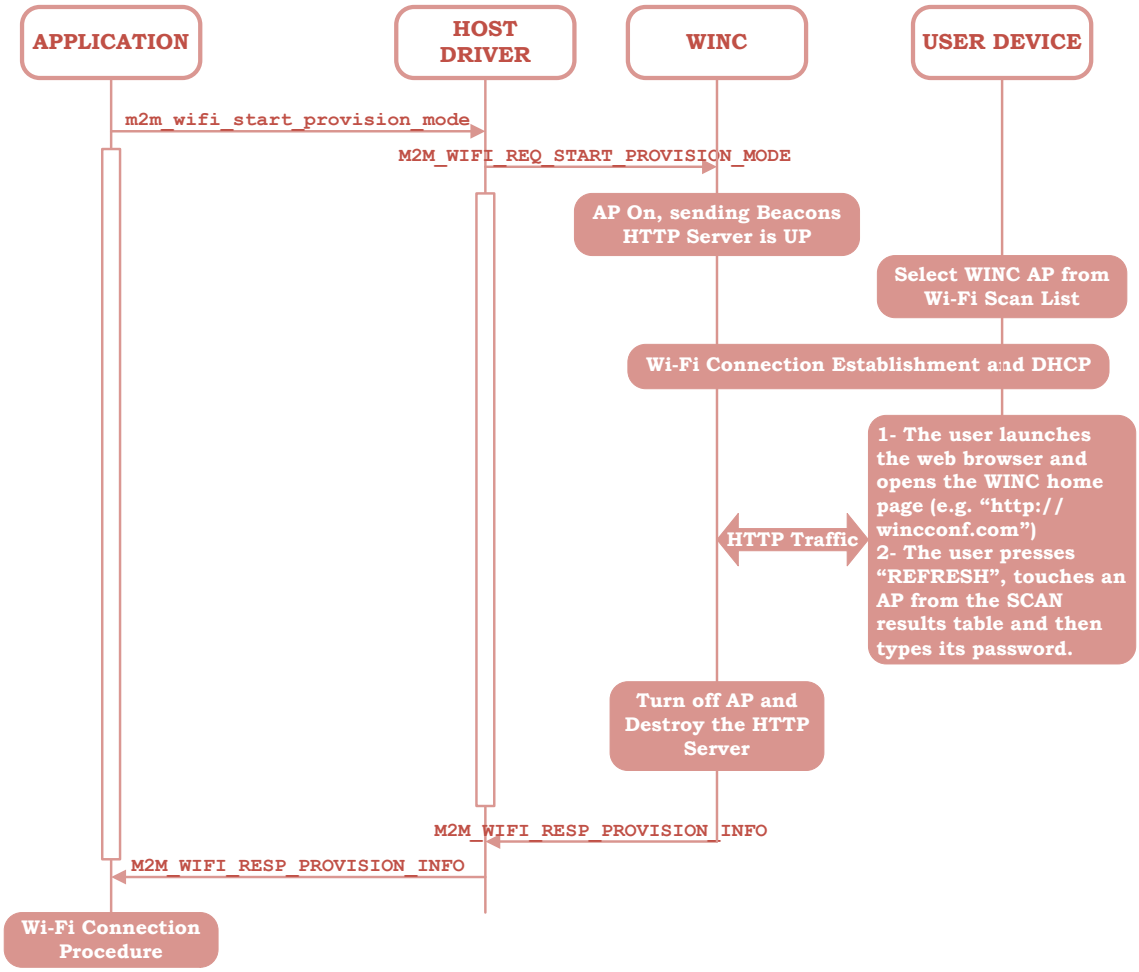


Figure 12-3 shows the provisioning operation for an ATWINC device. The detailed steps are described as follows:

1. The ATWINC device starts the HTTP Provisioning mode.
2. A user with a smart phone finds the ATWINC AP SSID in the Wi-Fi search list.
3. The user connects to the ATWINC AP.
4. The user launches the web browser and writes the ATWINC home page in the address bar.
5. If the HTTP redirect is enabled at the ATWINC, any web address the ATWINC home page will load automatically (like connecting to a public Wi-Fi hotspot). Some phones will display a notification message "sign in to Wi-Fi networks?" which, when accepted, will load the ATWINC home pages load automatically. The ATWINC home page (shown in Figure 12-2) will appear on the browser.
6. To discover the list of Wi-Fi APs in the area, the user can press "Refresh".
7. The desired AP is then selected from the search list (by one click or one touch) and its name will appear automatically in the "Network Name" text box.
8. Then the user must enter the correct AP passphrase (for WPA/WPA2 personal security) in the "Pass Phrase" text box. If the AP is not secured (Open network) the field should be left empty.
9. An ATWINC device name may be optionally configured if desired by the user in the "Device Name" text box.

10. The user presses “Connect”.

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

12.2.5 HTTP Redirect Feature

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

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

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

12.2.6 HTTP 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);
            }
        }
}
```

12.3 Wi-Fi Protected Setup (WPS)

Most modern Access Points support the 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.

12.3.1 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 ATWINC3400 by calling **`m2m_wifi_wps`** with input parameter **`WPS_PBC_TRIGGER`**.
2. **PIN method**
   The AP is always available for WPS initiation but requires proof that the user has knowledge of an 8-digit PIN, usually printed on the body of the AP. Because ATWINC 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 ATWINC device. Some APs allow the PIN to be changed through configuration. WPS is initiated on the ATWINC3400 by calling **`m2m_wifi_wps`** with input parameter **`WPS_PIN_TRIGGER`**. Given the difficulty of this approach it is not recommend for most applications.

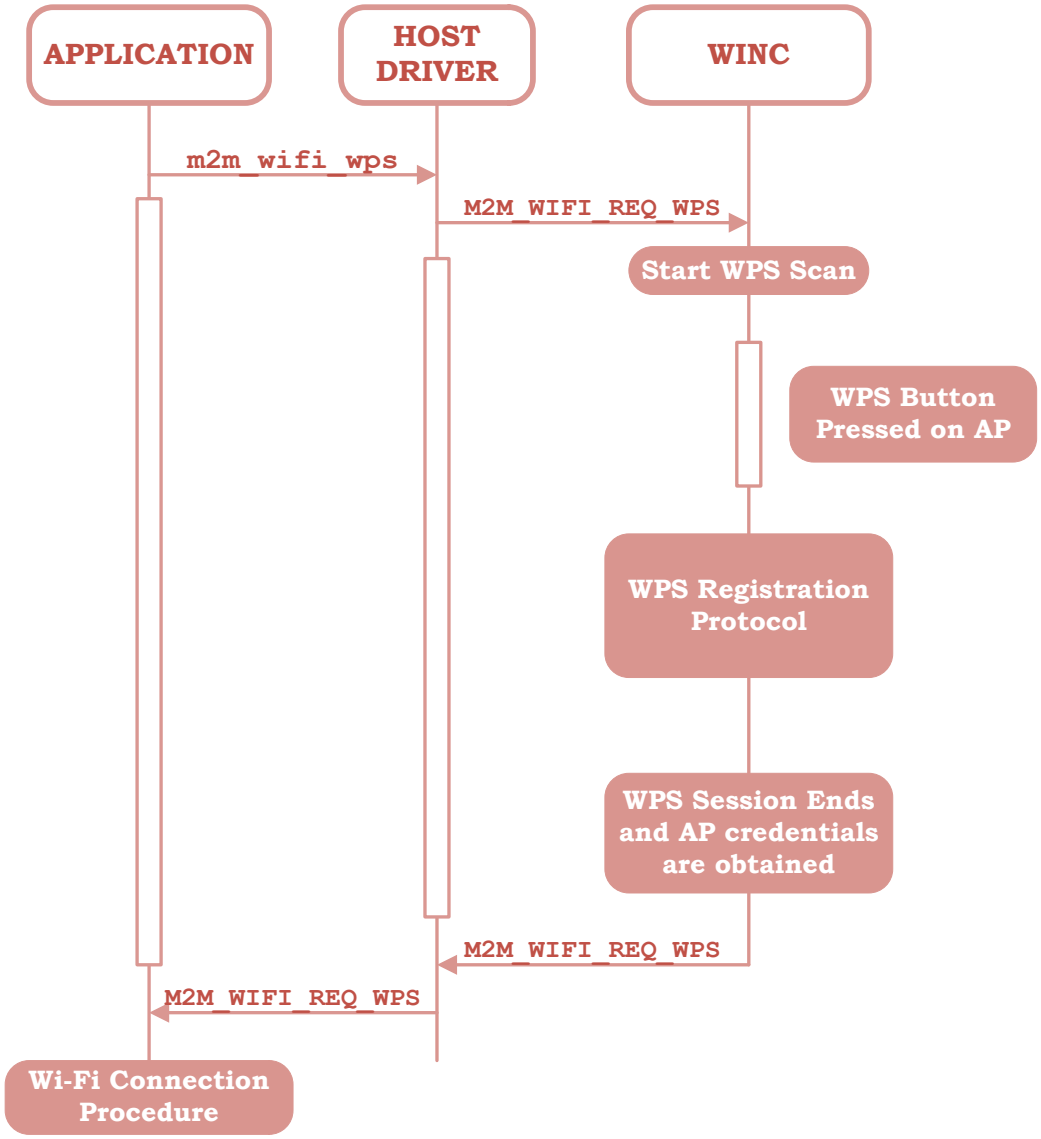
The flow of messages and actions for WPS operation is shown in Figure 12-4.

12.3.2 WPS Limitations

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

12.3.3 WPS Control Flow

Figure 12-4. WPS Operation for Push Button Trigger



12.3.4 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);
        }
    }
}
```

13 Multicast Sockets

13.1 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 is 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.

13.2 How to use Filters

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

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

13.3 Multicast Socket Code Example

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

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

- Server Initialization

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

   }
```

- Sockets Events Handler

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

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

- Server Socket Callback

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

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

- Parse mDNS Query

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

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

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

   }
```

- Send mDNS Response:

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

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

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

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

- Service Name

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

#define serviceName "_ATMELWIFI._tcp"
```

14 ATWINC Serial Flash Memory

14.1 Overview and Features

The ATWINC has internal serial (SPI) flash memory of either 4 or 8Mb capacity. The flash memory is used to store:

- User configuration
- Wi-Fi Firmware
- BLE Firmware
- Connection Profiles

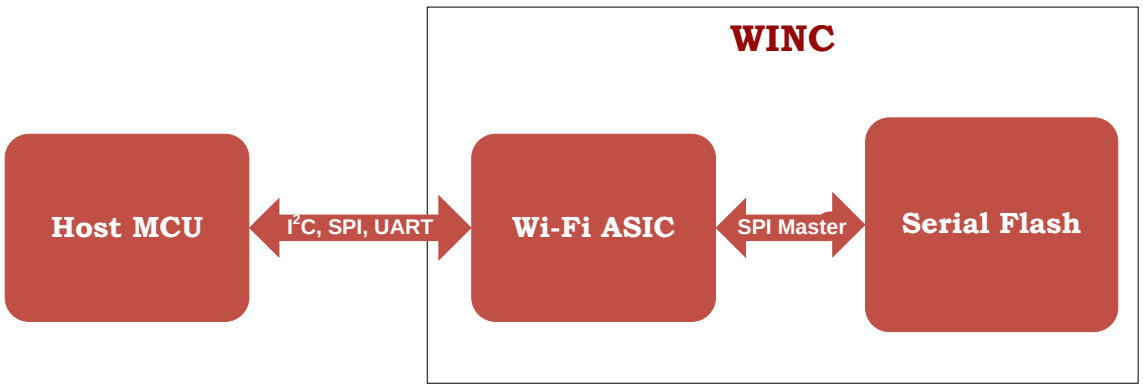
During startup and mode changes the firmware is loaded from the serial flash into program memory (IRAM) in which the firmware is executed. First the Wi-Fi firmware is loaded and started while holding the BLE processor in reset. Then the BLE firmware is loaded and placed into the BLE memory prior to releasing reset. The flash is accessed at other points during runtime to retrieve configuration and profile data.

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

14.2 Accessing to Serial Flash

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

Figure 14-1. System Block Diagram showing SPI Flash Connection



14.3 Read/Write/Erase Operations

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

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

```
sint32 m2m_wifi_download_mode();
```

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

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

- Query the size of the SPI Flash:

```
uint32 spi_flash_get_size();
```

This function returns with size of SPI Flash in Mb.

- Read data from the SPI Flash:

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

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

- Erase sectors in the SPI Flash:

```
sint8 spi_flash_erase(uint32 u32Offset, uint32 u32Sz)
```

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

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

- Write data to the SPI Flash:

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

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

14.4 Serial (SPI) Flash Map

The following map is valid for SPI Flash with size equals 8Mb (1MB)

Section	Offset [KB]	Size [KB]
Boot firmware	0	4
Control Section	4	8
Configuration	12	8
Certificate	20	4
Scratch Section	24	4
Firmware Image – Wi-Fi	28	196
HTTP Files	224	8
Connection Parameters	232	4
Firmware Image – BLE	236	128
[Unused]	[364]	[148]
OTA Image – Wi-Fi and BLE	512	324
[Unused]	[836]	[188]
Total used		**684**

14.5 Flash Read, Erase, Write Code Example

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

int main()
```

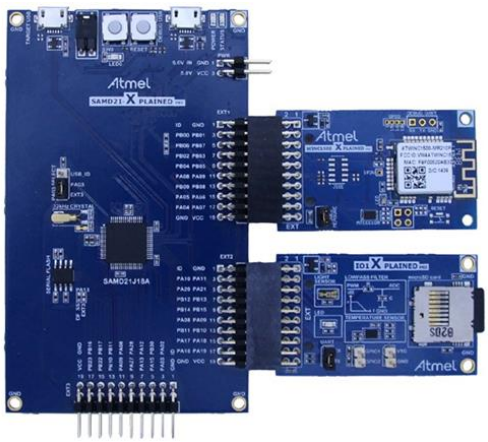
```
{
	uint8 au8FlashContent[FLASH_SECTOR_SZ] = {0};
	uint32 u32FlashTotalSize = 0, u32FlashOffset = 0;

	// 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;
}
```

15 Writing a Simple Networking Application

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

15.1 Prerequisites

- Hardware Prerequisites
  - Atmel SAMD21-XPRO Evaluation kit
  - Atmel I/O1 Xplained ATIO1-XPRO board
  - Atmel ATWINC3400 Xplained Pro Extension board
  - Micro-USB Cable (Micro-A / Micro-B)
  - Android Phone
- Software Prerequisites
  - Atmel Studio 6.2 (build 1153) or higher
  - Atmel Software Frameworks 3.15.0
  - Wi-Fi Network Controller demo for SAM D21
  - Android apk AtmelWirelessConnect application



15.2 Solution Overview

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

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

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

TO DO ATWINC3400 Wi-Fi extension and I/O1 extension should be plugged into SAM D21 Xplained Pro EXT1 and EXT2 respectively.

15.3 Project Creation

TO DO Open the empty Wi-Fi example Project.

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

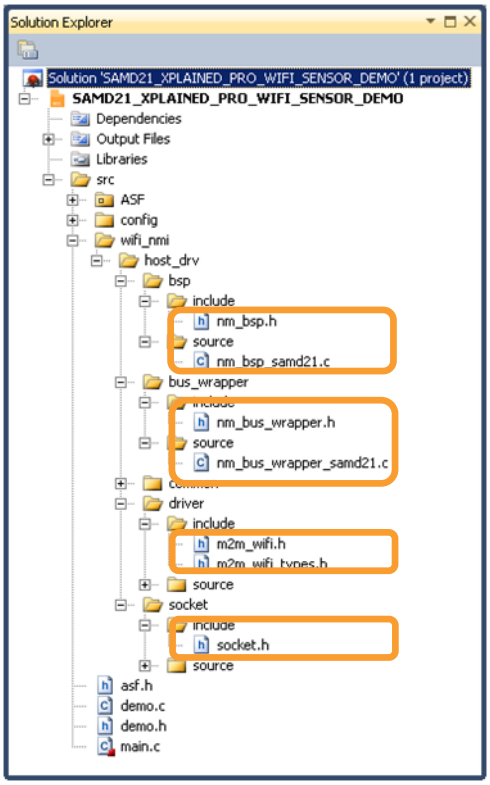
15.4 Wi-Fi Software API Files

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

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

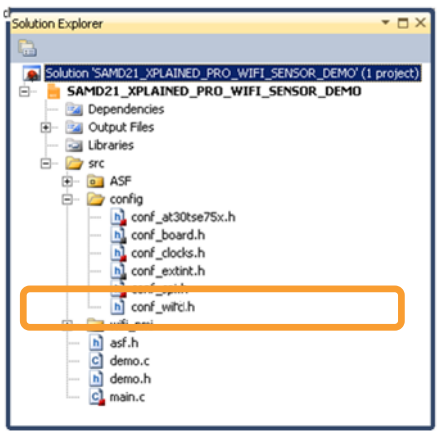
In order to add Wi-Fi connectivity into an existing user example project, the complete “wifi_nmi” folder should be added to the user project.

TO DO Locate these files in your project:



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

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

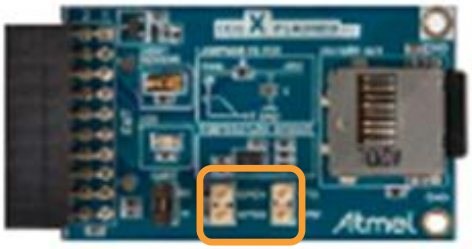


The file **`conf_winc.h`** provides configuration for the following:

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

15.5 Reading Temperature Sensor and Controlling LED Status

The empty Wi-Fi example Project comes with a specific driver for the AT30TSE temperature sensor, located on the I/O1 extension.



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

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

1. Include the driver header file.

```
#include “asf.h”
```

2. Initialize the temperature sensor driver.

```
at30tse_init();
```

3. Retrieve the current temperature value.

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

INFO

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

15.6 Step By Step Development

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

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

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

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

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

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

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

	return 0;
}
```

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

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

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

1. **Reset the ATWINC3400 Module.**

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

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

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

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

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

```
	tstrWifiInitParam param;
	sint8 ret;

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

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

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

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

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

4. **Implement an empty m2m_wifi_socket_handler.**

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

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

INFO All ATWINC3400 socket operations are non-blocking asynchronous operations.

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

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

5. **Initialize the temperature sensor.**

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

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

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

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

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

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

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

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

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

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

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

	/* Reset network controller */
	nm_bsp_init();

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

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

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

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

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

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

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

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

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

8. **Start association.**

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

   The **m2m_wifi_connect()** function request the ATWINC3400 Wi-Fi module to start association with the local SSID.

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

   The parameter **DEMO_WLAN_AUTH** specifies the local AP security type which can be either: OPEN, WEP, WPA/WPA2, or enterprise security. The last parameter **M2M_WIFI_CH_ALL** forces the ATWINC3400 to scan for the local AP on all channels.

TIPS The ATWINC3400 firmware contains a “**fast boot feature**” which optimizes the association time across power cycles. Firmware stores the channel on which the last successful association occurred. During next association attempt, firmware probes if the local AP is still on the same channel (which is most likely to happen since AP does not change channels frequently).
If local AP is not found, then firmware will trigger a new scan.

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

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

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

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

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

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

10. Obtaining DHCP configuration.

When DHCP client in ATWINC3400 Firmware obtains IP address from local AP, the demo app receives information about the obtained IP address. `m2m_wifi_state()` callback is invoked with **`M2M_WIFI_REQ_DHCP_CONF`** event type.

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

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

         break;
```

TIPS

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

RESULT

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

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

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

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

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

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

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

	/* Reset network controller */
	nm_bsp_init();

	/* Initialize Wi-Fi driver with data and Wi-Fi status callbacks. */
	ret = m2m_wifi_init(&param);
```

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

11. Open sockets and send transmit and receive data.

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

The demo Android app shall be connected to the same local AP and broadcasts a message on the local AP WLAN. All ATWINC3400 devices on the local WLAN receive the broadcast message. The message contains `s_msg_temp_report` structure. The message contains the name of the device which should receive the message. Each device matches the name member to a string DEMO_PRODUCT_NAME before changing the LED state as provided in led member in `s_msg_temp_report` message.

In return, the device broadcasts two messages every 1 sec. on local AP WLAN, namely: keep alive message `t_msg_temp_keepalive` and temperature report message `t_msg_temp_report`. Each device provides its name in transmitted messages in name field as defined in DEMO_PRODUCT_NAME. The `t_msg_temp_keepalive` message allows the demo Android app to discover the existence of ATWINC3400 devices on local WLAN and displays a list of discovered devices, to the user. When the user chooses to view temperature from a specific device, the Android app matches the received `t_msg_temp_report` message name field to the name of the user selected device before plotting the temperature value.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

12. Send a discovery frame and temperature information.

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

Add the following code listing to your demo application:

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

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

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

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

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

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

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

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

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

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

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

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

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

Listing continued below…

```
} t_msg_temp_report;

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

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

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

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

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

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

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

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

Listing continued below…

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

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

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

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

Listing continued below…

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

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

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

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

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

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

        /* Reset network controller */
        nm_bsp_init();

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

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

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

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

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

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

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

Listing continued below…

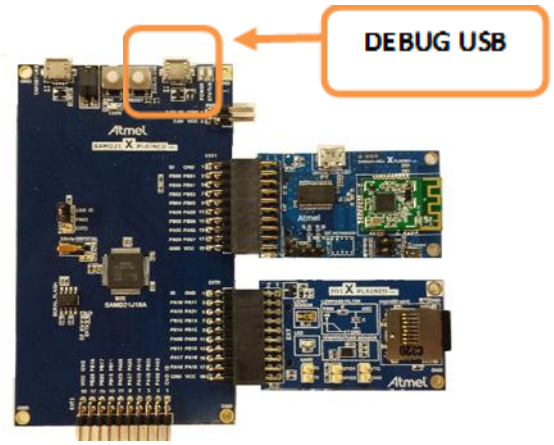
```
                                        puts("demo_start: failed to create TX UDP client socket error!");
                                        continue;
                                }
                        }

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

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

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

                }
        }
}
```

Listing page (5/5)

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

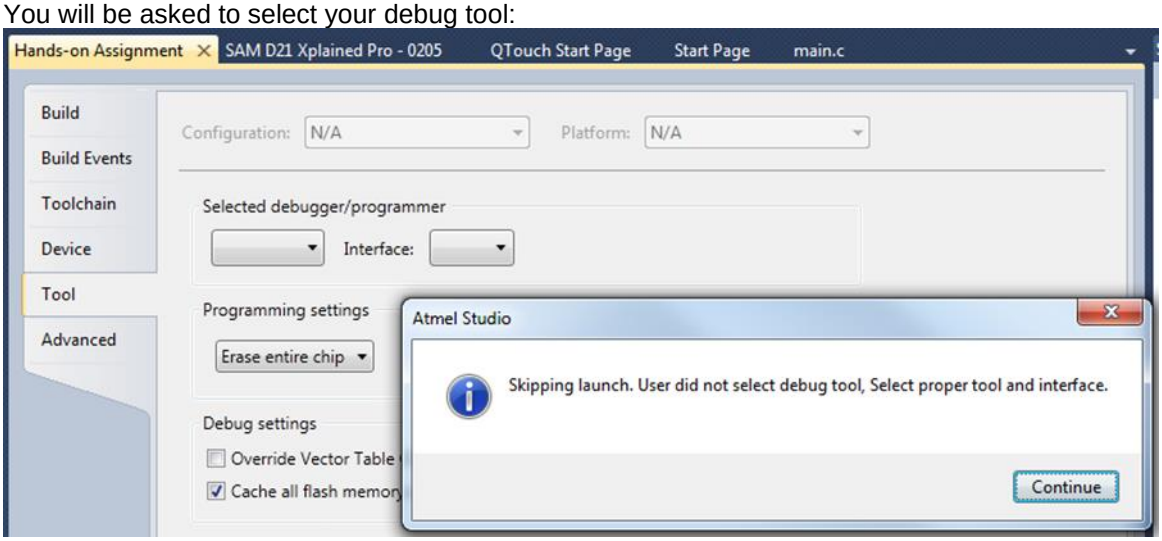
TO DO Program the SAM D21 Xplained Pro.

- Connect the ATWINC3400 Wi-Fi extension and the I/O1 extension to the SAM D21 Xplained Pro as displayed:

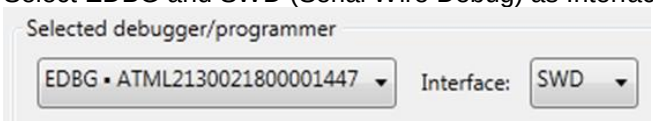


- Connect the SAM D21 Xplained Pro board to your PC using DEBUG USB connector
- Program the application by clicking on the Start Debugging and Break icon: 

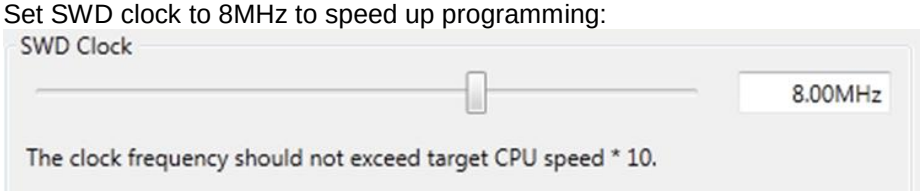
- You will be asked to select your debug tool:



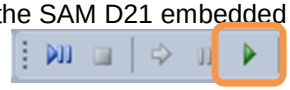
- Select EDBG and SWD (Serial Wire Debug) as Interface:



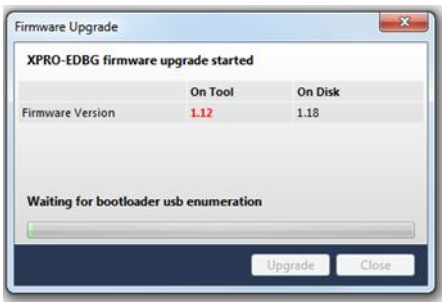
- Set SWD clock to 8MHz to speed up programming:



- Click again on the Start Debugging and Break icon: 

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

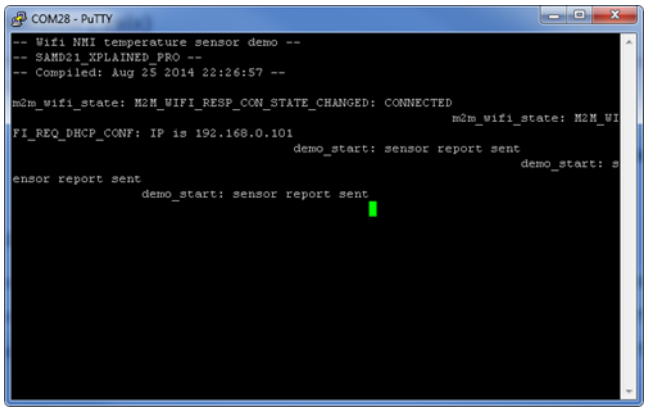
INFO You may be asked to upgrade your EDBG firmware. If so, click on Upgrade:



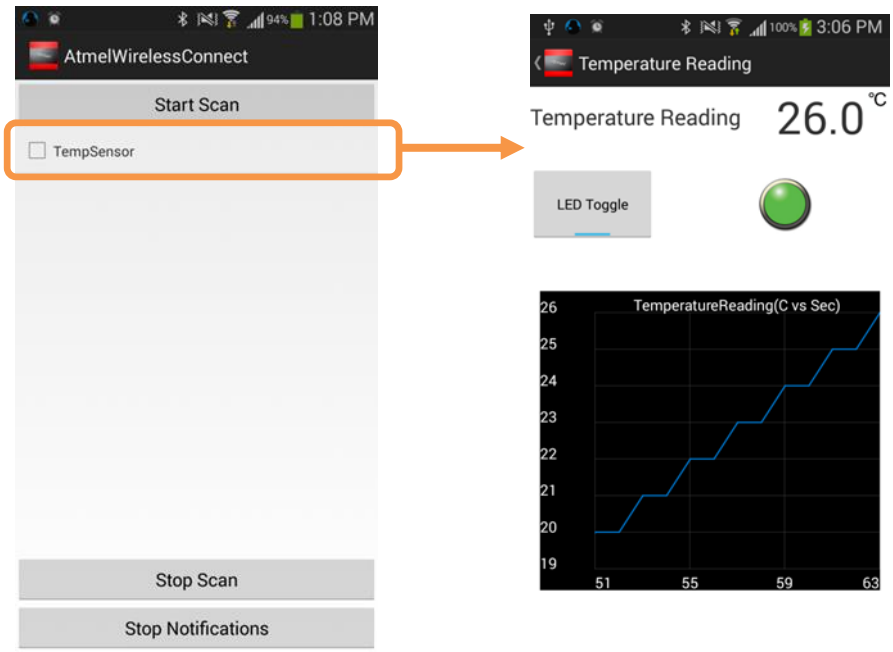
WARNING Upgrade operation may take a few minutes, **wait** for the operation to complete.

RESULT The IoT sensor application is now programmed and running.

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



- Connect the Android device to the same SSID network than the ATWINC3400 Wi-Fi module
- Open the Temperature sensor application on the Android device
- Hit the “Start Scan” button
- A “TempSensor” device (as defined in the `DEMO_PRODUCT_NAME` macro in demo.h file) appears



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

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

16 Host Interface Protocol

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

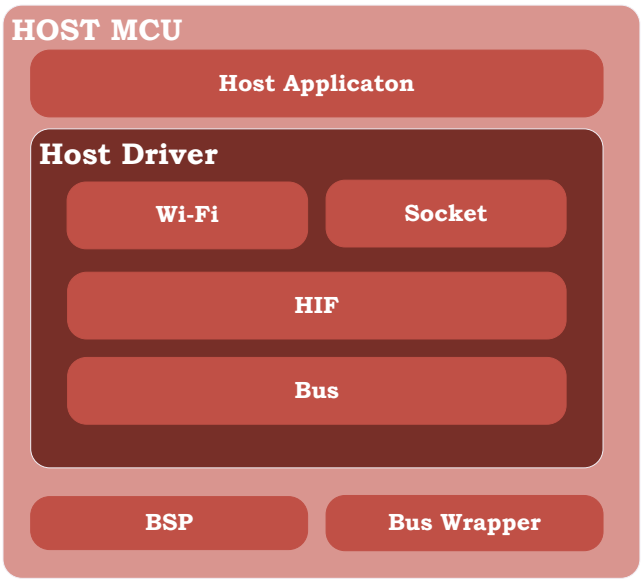
The host driver implements services asynchronously. This means that when the application calls an API to request a service action, the call is non-blocking and returns immediately, often before the action is completed. Where appropriate, notification that an action has completed is provided in a subsequent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

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

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

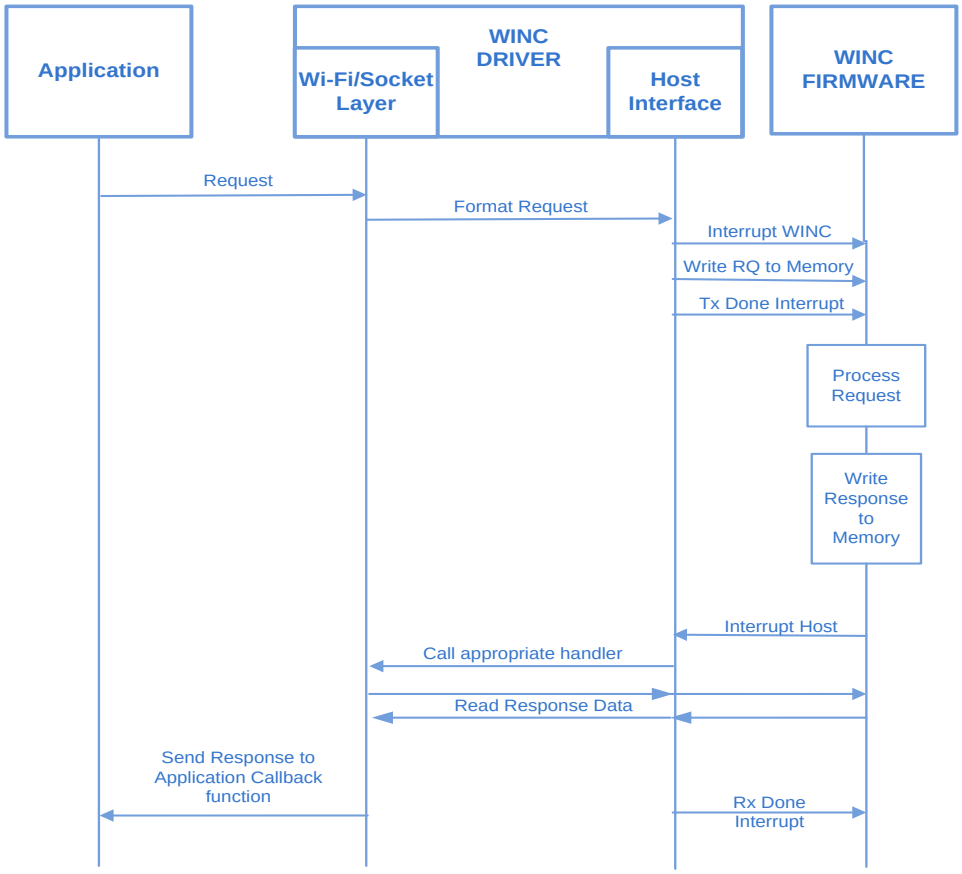
Figure 16-1. ATWINC Driver Layers



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

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

Figure 16-2. The Request/Response Sequence Diagram



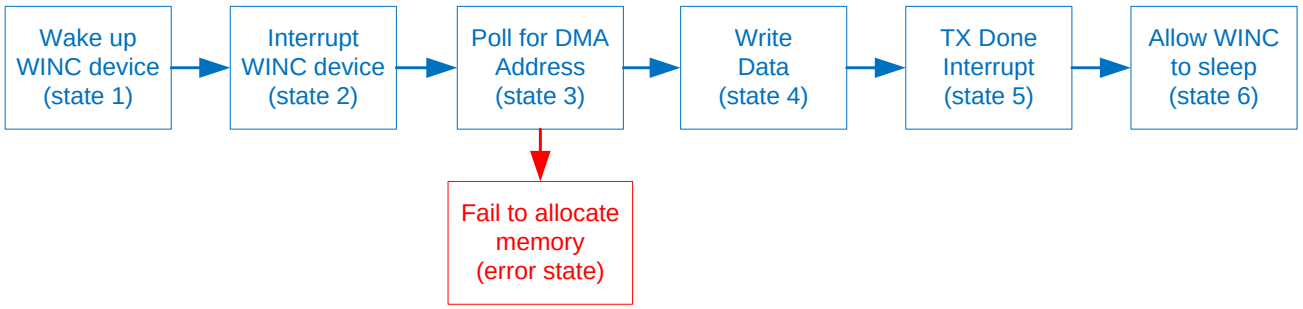
16.1 Transfer Sequence Between HIF Layer and ATWINC Firmware

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

16.1.1 Frame Transmit

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

Figure 16-3. HIF Frame Transmit to ATWINC

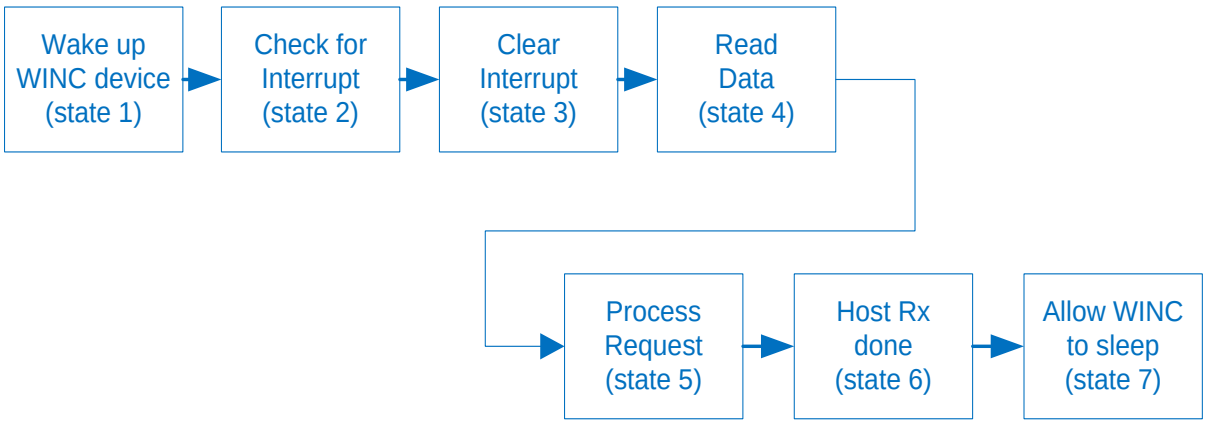


Step	Description
Step (1) Wake up the ATWINC device	Wakeup the device to be able to receive Host requests
Step (2) Interrupt the ATWINC device	Prepare and Set the HIF layer header to NMI_STATE_REG register (4-bytes header describing the sent packet). Set BIT [1] of WIFI_HOST_RCV_CTRL_2 register to raise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

16.1.2 Frame Receive

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

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

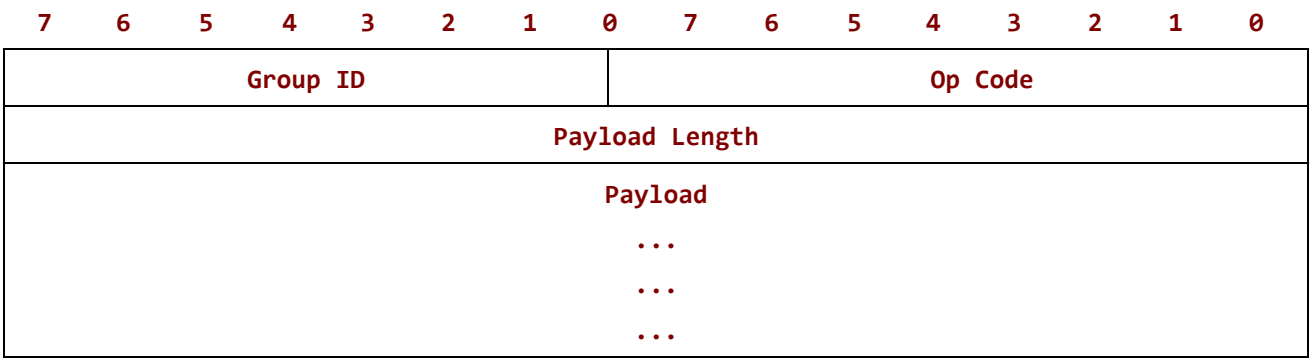


Step	Description
Step (1) Wake up the ATWINC device	Wakeup the device to be able to receive Host requests
Step (2) Check for Interrupt	Monitor BIT[0] of WIFI_HOST_RCV_CTRL_0 register. Disable the host from receiving interrupts (until this one has been processed).
Step (3) Clear interrupt	Write zero to BIT[0] of WIFI_HOST_RCV_CTRL_0 register
Step (4) Read Data	Get the address of the data block from WIFI_HOST_RCV_CTRL_1 register. Read Data block with size obtained from WIFI_HOST_RCV_CTRL_0 register BIT[13] <->BIT[2].

Step	Description
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 ATWINC device to sleep	Allow the ATWINC device to enter sleep mode again (if it wishes).

16.2 HIF Message Header Structure

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



- **The Group ID (8-bits)**: A group ID is the category of the message. Valid categories are `M2M_REQ_GRP_WIFI`, `M2M_REQ_GRP_IP`, `M2M_REQ_GRP_HIF`, `M2M_REQ_GRP_OTA` corresponding to Wi-Fi, Socket, HIF, and OTA respectively. A group ID can be assigned one of the values enumerated in `tenuM2mReqGrp`.
- **Op Code: (8-bit)**: A command number. Valid command number is a value enumerated in: `tenuM2mConfigCmd and tenuM2mStaCmd`, `tenuM2mApCmd`, and `tenuM2mP2pCmd` corresponding to configuration, STA mode, AP mode, and P2P mode commands. See the full list of commands in the header file `m2m_types.h`.
- **Payload Length (16-bits)**: The payload length in bytes (does not include header).

16.3 HIF Layer APIs

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

The following API functions are described:

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

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

sint8 hif_chip_wake(void):

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

sint8 hif_chip_sleep(void):

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

sint8 hif_register_cb(uint8 u8Grp,tpfHifCallBack fn):

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

sint8 hif_isr(void):

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

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

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

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

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

16.4 Scan Code Example

The following code example illustrates the Request/Response flow on a Wi-Fi Scan request. For more details on the code examples, refer to [R04].

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

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

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

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

        }
        else if(M2M_REQ_GRP_IP == strHif.u8Gid)
        {
            if(pfIpCb)
```

```
                                        pfIpCb(strHif.u8Opcode,strHif.u16Length - M2M_HIF_HDR_OFFSET,
                                                address + M2M_HIF_HDR_OFFSET);
                                }
                                else if(M2M_REQ_GRP_OTA == strHif.u8Gid)
                                {
                                        if(pfOtaCb)
                                                pfOtaCb(strHif.u8Opcode,strHif.u16Length - M2M_HIF_HDR_OFFSET,
                                                address + M2M_HIF_HDR_OFFSET);
                                }
                                else
                                {
                                        M2M_ERR("(hif) invalid group ID\n");
                                        ret = M2M_ERR_BUS_FAIL;
                                        goto ERR1;
                                }
                                /* RX STEP (6) */
                                if(!gu8HifSizeDone)
                                {
                                        M2M_ERR("(hif) host app didn't set RX Done\n");
                                        ret = hif_set_rx_done();
                                }
                        }
                        else
                        {
                                ret = M2M_ERR_RCV;
                                M2M_ERR("(hif) Wrong Size\n");
                                goto ERR1;
                        }
                }
                else
                {
#ifndef WIN32
                        M2M_ERR("(hif) False interrupt %lx",reg);
#endif
                }
        }
        else
        {
                M2M_ERR("(hif) Fail to Read interrupt reg\n");
                goto ERR1;
        }
}
else
{
        M2M_ERR("(hif) FAIL to wakeup the chip\n");
        goto ERR1;
}
/* RX STEP (7) */
ret = hif_chip_sleep();
ERR1:
        return ret;
}
```

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

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

```
                }
            }
        // …code eliminated…
}
```

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

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

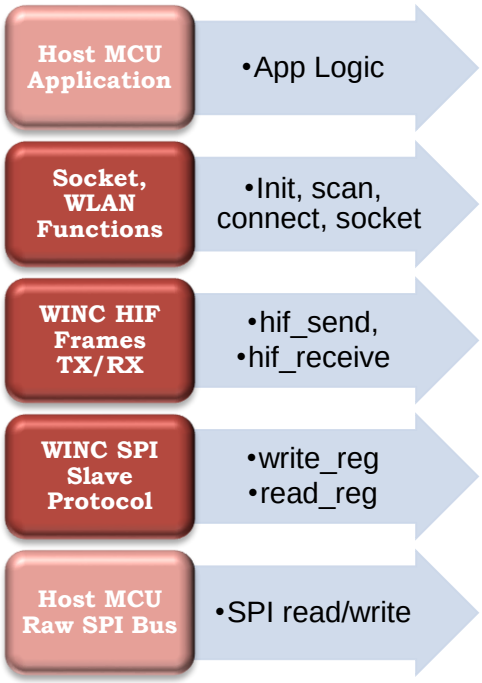
17 ATWINC SPI Protocol

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

The ATWINC SPI protocol consists of three layers:

- Layer 1: ATWINC SPI slave protocol, which allows the host MCU application to perform register/memory read and write operation in the ATWINC3400 device using raw SPI data exchange
- Layer 2: Host MCU application uses the register and memory read and write capabilities to exchange host interface frames with the ATWINC firmware. It also provides asynchronous callback from the ATWINC firmware to the host MCU through interrupts and host interface RX frames. This layer was discussed earlier in Chapter 16: Host Interface Protocol.
- Layer 3: Allows the host MCU application to exchange high level messages (e.g. Wi-Fi scan, socket connection, or TCP data received) with the ATWINC firmware to employ in the host MCU application logic

Figure 17-1. ATWINC SPI Protocol Layers



17.1 Introduction

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

- Command
- Response
- Data

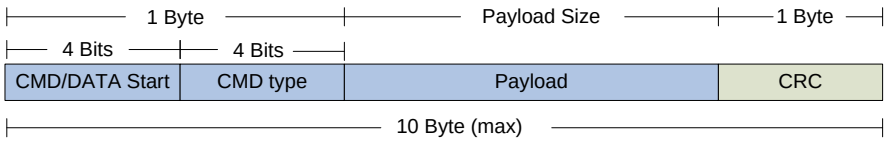
In the case of Command and Data messages, the last byte is used as data integrity check.

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

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

17.1.1 Command Format

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



The first byte contains two fields:

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

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

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

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

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

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

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

Type A commands include:

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

Type B commands include:

- DMA RD Transaction
- DMA WR Transaction

Type C commands include:

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

Type D commands include:

- DMA single-word WR

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

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

Payload	A: 3 B: 5 C: 6 D: 7	The Payload field may be of Type A, B, C, or D **Type A (length 3)** ***1- DMA single-word RD*** *Param: Read Address:* *Payload bytes:* B0: ADDRESS[23:16] B1: ADDRESS[15:8] B2: ADDRESS[7:0] ***2- internal register RD*** *Param: Offset address (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] B1: ADDRESS[15:8] B2: ADDRESS[7:0] B3: COUNT[15:8] B4: COUNT[7:0] ***2- DMA WR Transaction*** *Params:* DMA Start Address: 3 bytes DMA count: 2 bytes *Payload bytes*: B0: ADDRESS[23:16] B1: ADDRESS[15:8] B2: ADDRESS[7:0] B3: COUNT[15:8] B4: COUNT[7:0]

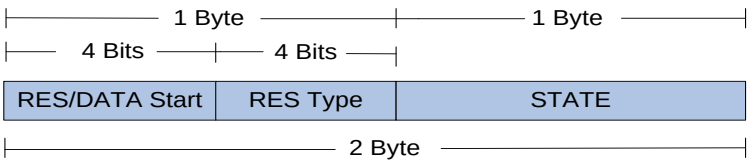
Field	Size	Description
		Type C (length 6) ***1- DMA Extended RD transaction*** *Params:* DMA Start Address: 3 bytes DMA extended count: 3 bytes *Payload bytes:* B0: ADDRESS[23:16] B1: ADDRESS[15:8] B2: ADDRESS[7:0] B3: COUNT[23:16] B4: COUNT[15:8] B5: COUNT[7:0] ***2- DMA Extended WR transaction*** *Params:* DMA Start Address: 3 bytes DMA extended count: 3 bytes *Payload bytes:* B0: ADDRESS[23:16] 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’ bits 1-7: 7 bit CRC value computed using polynomial G(x) = X^7 + X^3 + 1 with seed value: 0x7F

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

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

17.1.2 Response Format

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



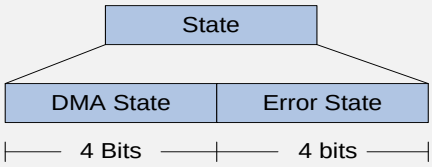
The first byte contains two 4-bit fields which identify the response message and the response type.

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

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

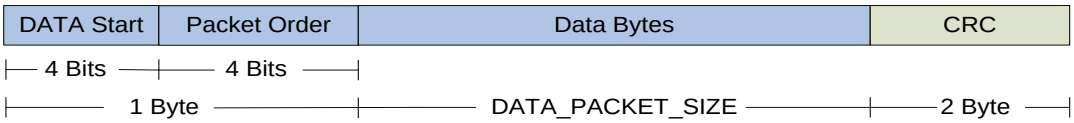
States that may be indicated are:

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

Field	Size	Description
Res Start	4 bits	Response Start : 4'b1100
Response Type	4 bits	If the response packet is for Command: • Contains of copy of the Command Type field in the Command. If the response packet is for received Data Packet: • 4'b0001: first data packet is received • 4'b0010: Receiving data packets • 4'b0011: last data packet is received • 4'b1111: Reserved value
State	1 byte	This field is divided into two subfields:  DMA State : • 4'b0000: DMA ready for any transaction • 4'b0001: DMA engine is busy Error State: • 4'b0000: No error • 4'b0001: Unsupported command • 4'b0010: Receiving unexpected data packet • 4'b0011: Command CRC7 error • 4'b0100: Data CRC16 error • 4'b0101: Internal general error

17.1.3 Data Packet Format

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

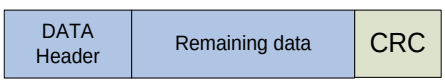


To support DMA hardware a large data transfer may be fragmented into multiple smaller Data Packets. This is controlled by the value of DATA_PACKET_SIZE which is agreed between the master and slave in software and is a fixed value such as 256B, 512B, 1KB (default), 2KB, 4KB, or 8KB. If a transfer has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

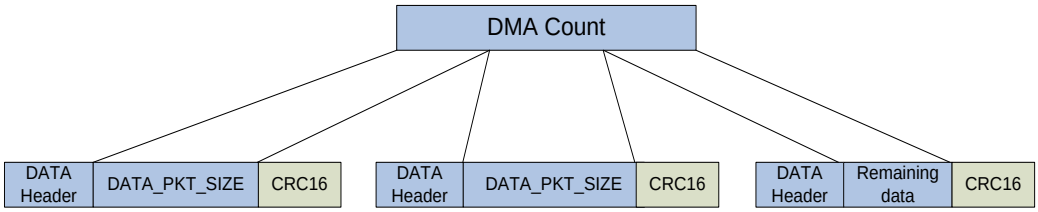
(m – (n-1)* DATA_PACKET_SIZE).This is shown diagrammatically below:

- **If DMA count <= DATA_PACKET_SIZE**

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



- **If DMA count > DATA_PACKET_SIZE**



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

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

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

Field	Size	Description
Data Start	4 bits	4'b1111 (Default) (Can be changed to any value by programming DATA_START_CTRL register)
Packet Order	4 bits	4'b0001: First packet in this transaction 4'b0010: Neither the first or the last packet in this 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 eight bits are transmitted first in the frame. The CRC16 value is computed on data bytes only based on the polynomial: G(x) = X^16 + X^12 + X^5 + 1, seed value: 0xFFFF

17.1.4 Error Recovery Mechanism

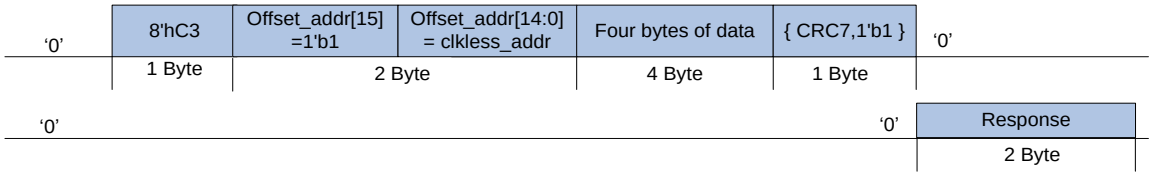
Error type	Recovery mechanism
Master:	
CRC error in command	1. Error response received from slave. 2. Retransmit the command.
CRC error in received data	1. Issue a repeat command for the data packet that has a CRC error. 2. Slave sends a response to the previous command. 3. Slave keeps the start DMA address of the previous data packet, so it can retransmit it. 4. Receive the data packet again.
No response is received from slave	• Synchronization is lost between master and slave • The worst case is when slave is in receiving data state • Solution: master should wait for maximum DATA_PACKET_SIZE period, then generate a soft reset command
Unexpected response	Retransmit the command

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

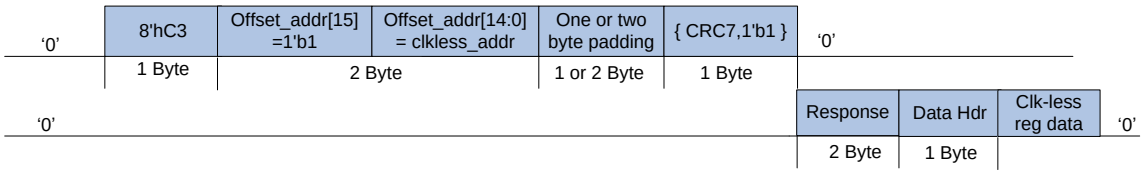
17.1.5 Clockless Registers Access

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

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



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

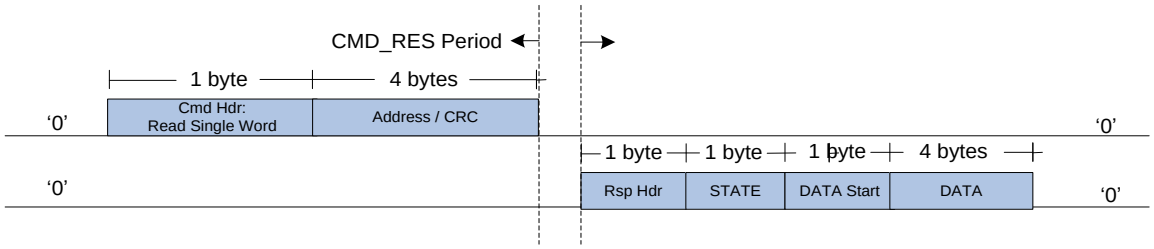


17.2 Message Flow for Basic Transactions

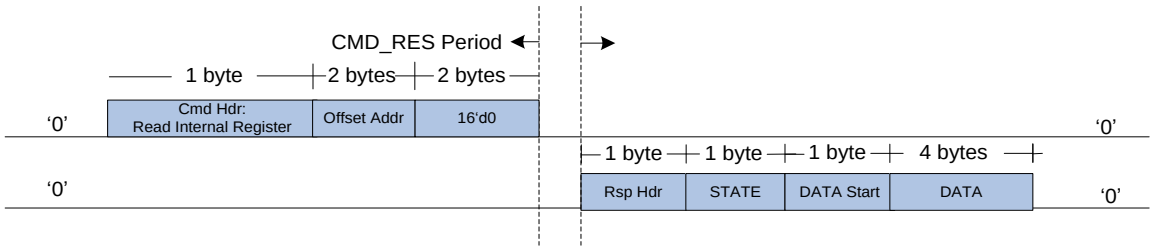
This section shows the essential message exchanges and timings associated with the following commands:

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

17.2.1 Read Single Word



17.2.2 Read Internal Register (for clockless registers)



17.2.3 Read Block

Normal transaction:

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

Slave: Sends a response after CMD_RES_PERIOD.

Master: Waits for a data packet start.

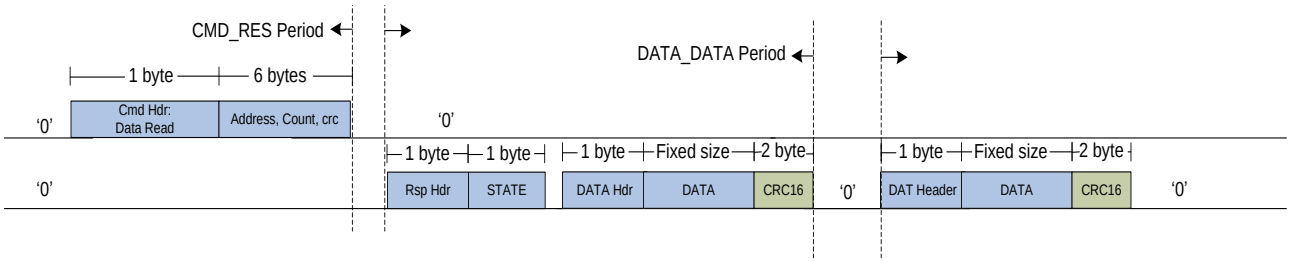
Slave: Sends the data packets, separated by DATA_DATA_PERIOD[5] where DATA_DATA_PERIOD is controlled by software and has one of these values:

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

Slave: Continues sending till the count ends.

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

The message sequence for this case is shown below:



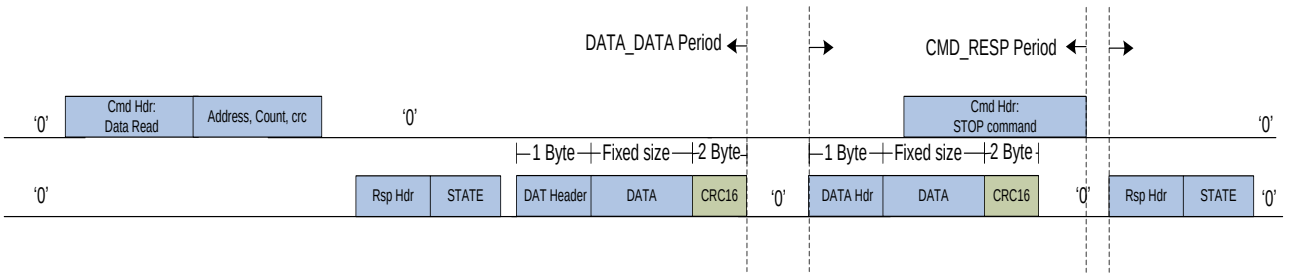
Termination command is issued:

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

Master: Should monitor for RES_START after CMD_RESP_PERIOD.

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

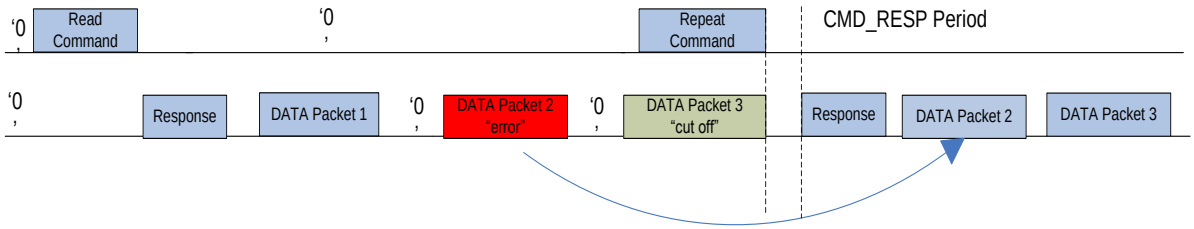
Slave: Should respond to the termination command after CMD_RESP_PERIOD from the end of the termination command packet.



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

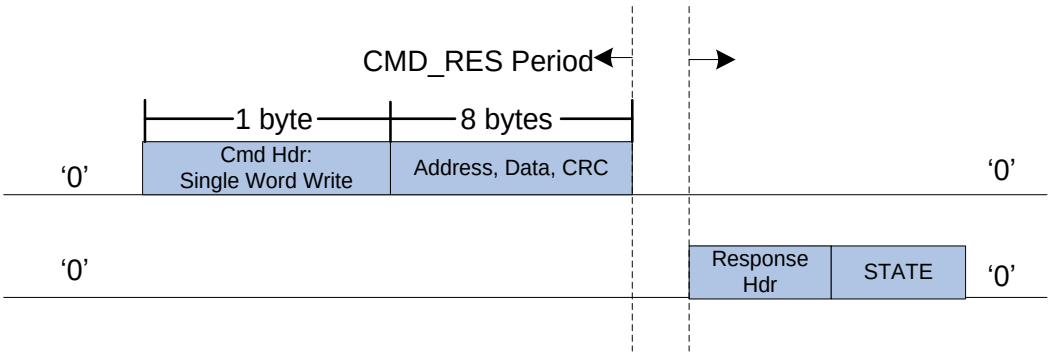
Repeat command is issued:

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



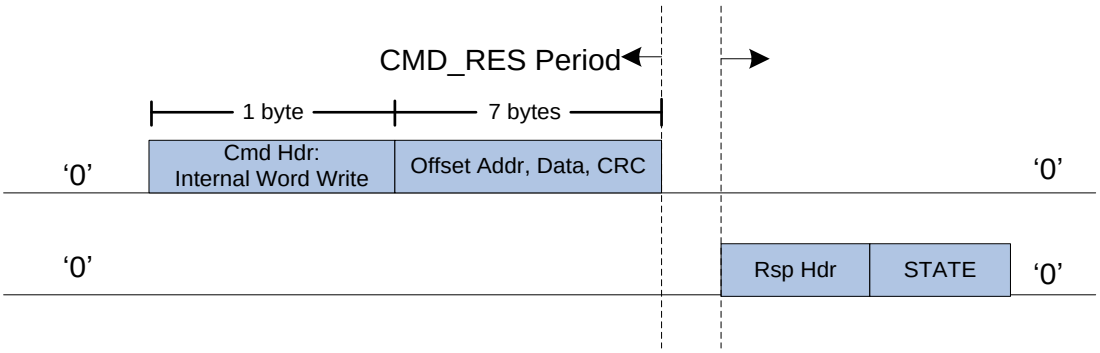
17.2.4 Write Single Word

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



17.2.5 Write Internal Register (for clockless registers)

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



17.2.6 Write Block

Case 1: Master waits for a command response:

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

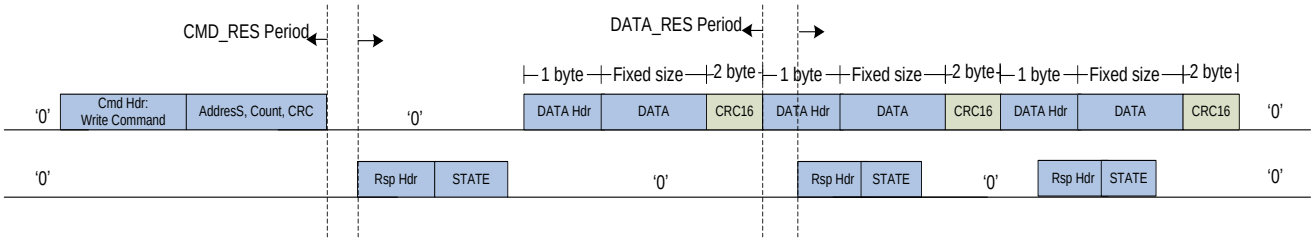
   CMD_RES_PERIOD is controlled by SW taking one of the values:

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

   The master should monitor for RES_START after CMD_RES_PERIOD

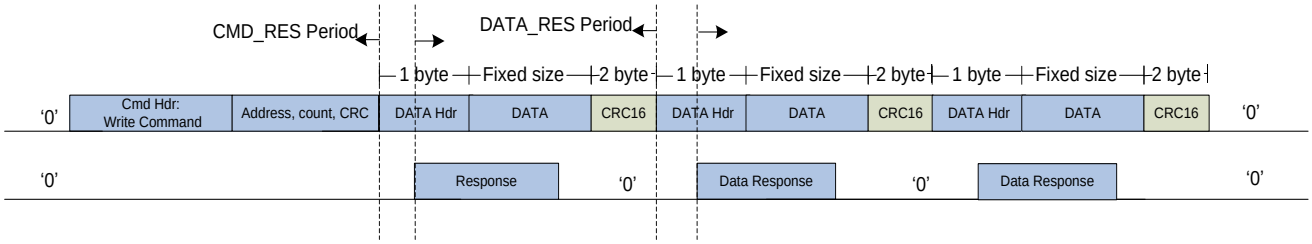
   DATA_RES_PERIOD is controlled by SW taking one of the values:

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



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

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



17.3 SPI Level Protocol Example

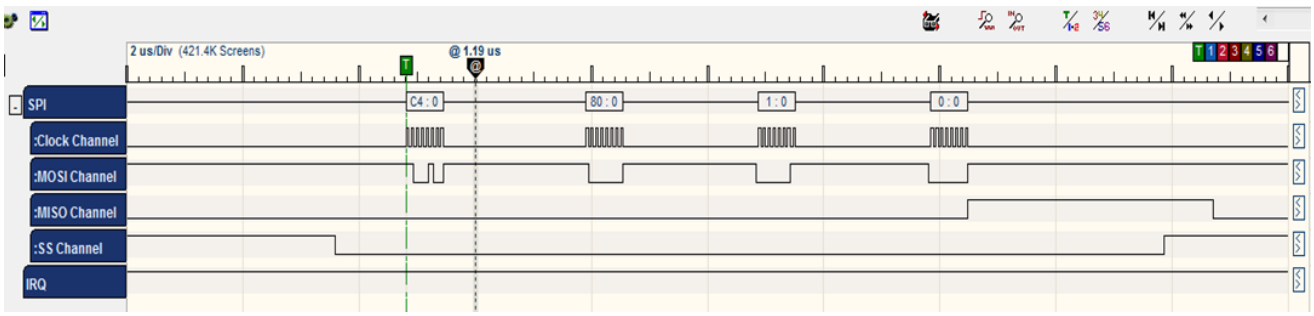
In order to illustrate how ATWINC SPI protocol works, SPI Bytes from the scan request example were dumped and the sequence is described below.

17.3.1 TX (Send Request)

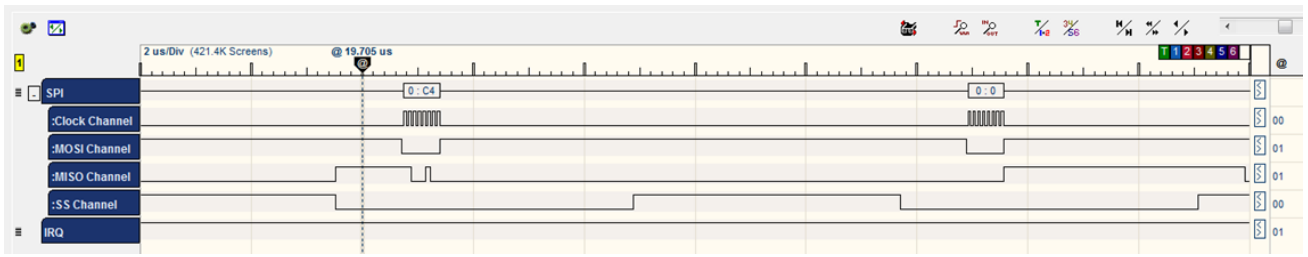
First step in hif_send() API is to wake up the chip:

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

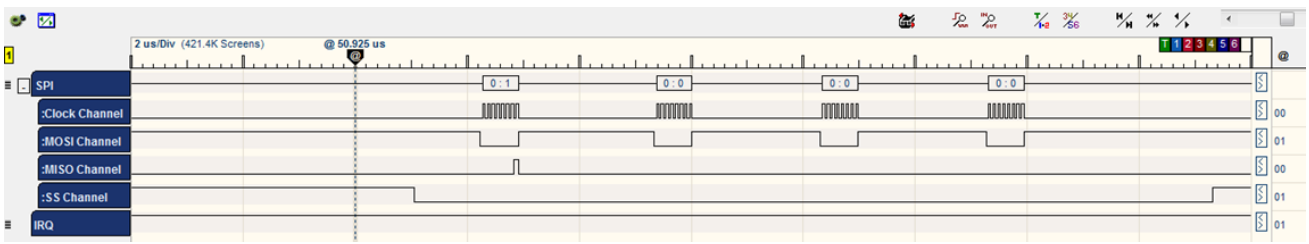
```
Command      CMD_INTERNAL_READ:  0xC4              /* internal register read */
      BYTE [0] = CMD_INTERNAL_READ
      BYTE [1] = address >> 8;                    /* address = 0x01 */
      BYTE [1] |= (1 << 7);                             /* clockless register */
      BYTE [2] = address;
   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



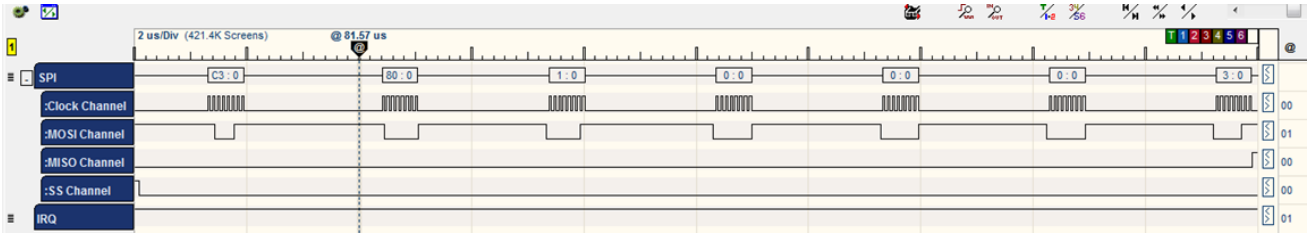
ATWINC acknowledges the command by sending three bytes [C4] [0] [F3].



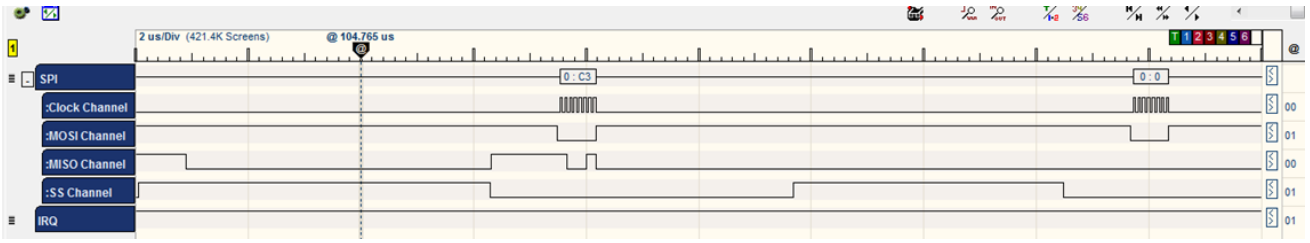
Then the ATWINC chip sends the value of the register 0x01 which equals 0x01.



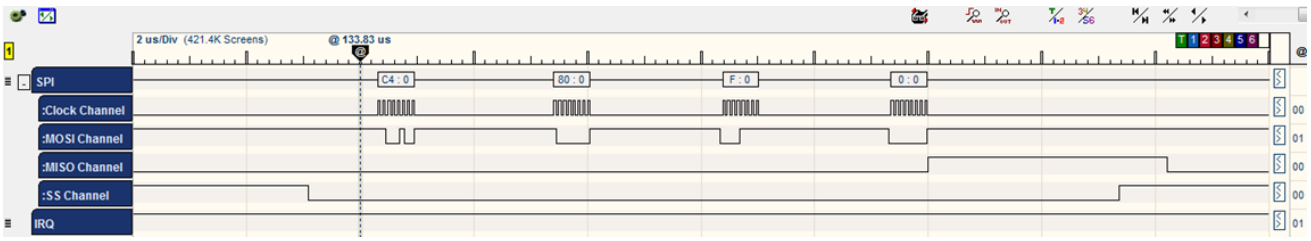
```
Command         CMD_INTERNAL_WRITE:   C3              /*    internal register write */
                BYTE [0] = CMD_INTERNAL_WRITE
                BYTE [1] = address >> 8;                    /*     address = 0x01              */
                BYTE [1] |= (1 << 7);               /*     clockless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



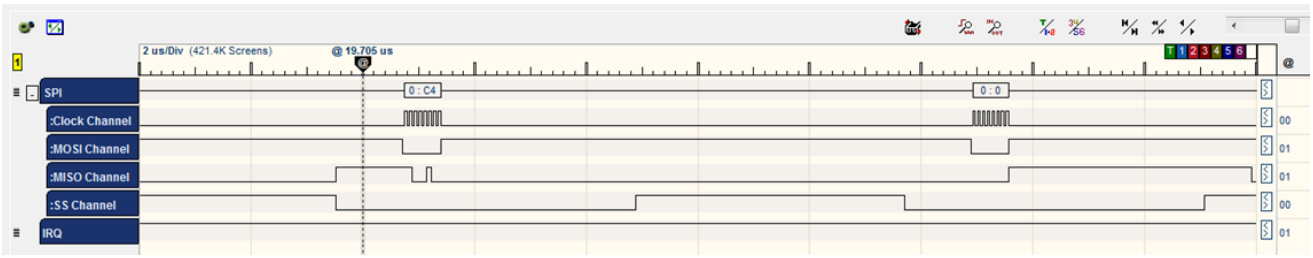
ATWINC acknowledges the command by sending two bytes [C3] [0].



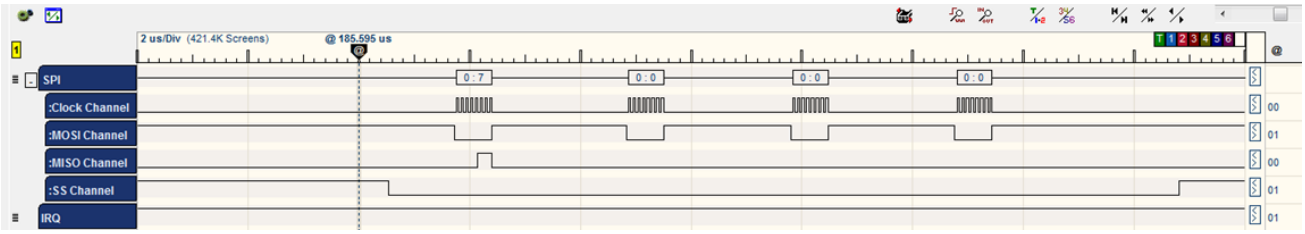

```
Command         CMD_INTERNAL_READ:    0xC4          /*    internal register read      */
                BYTE [0] = CMD_INTERNAL_READ
                BYTE [1] = address >> 8;                    /*     address = 0x0F              */
                BYTE [1] |= (1 << 7);                       /*  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



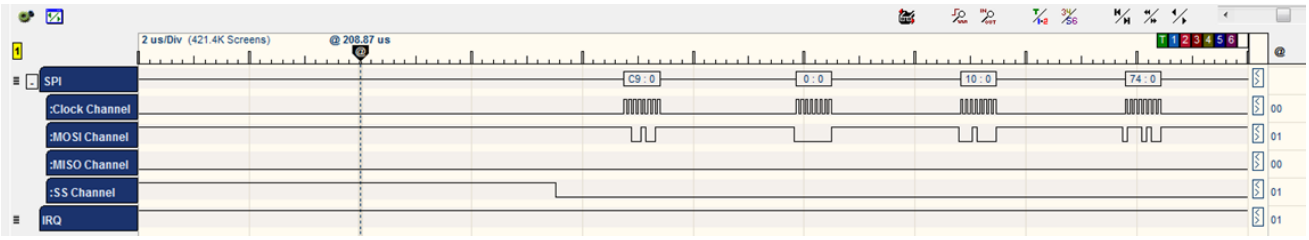
ATWINC acknowledges the command by sending three bytes [C4] [0] [F3].

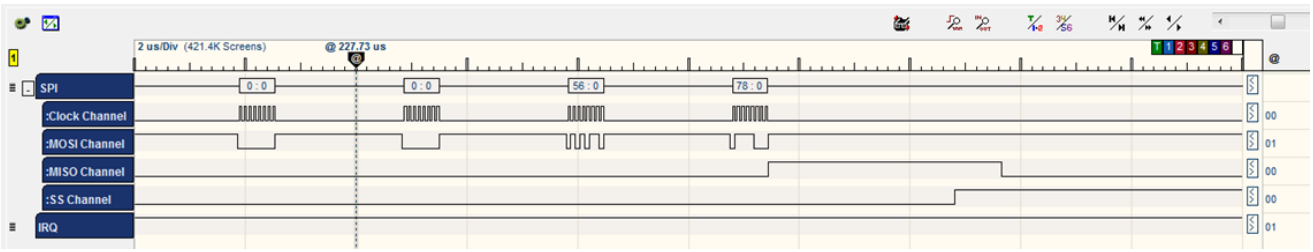


Then ATWINC chip sends the value of the register 0x01 which equals 0x07.

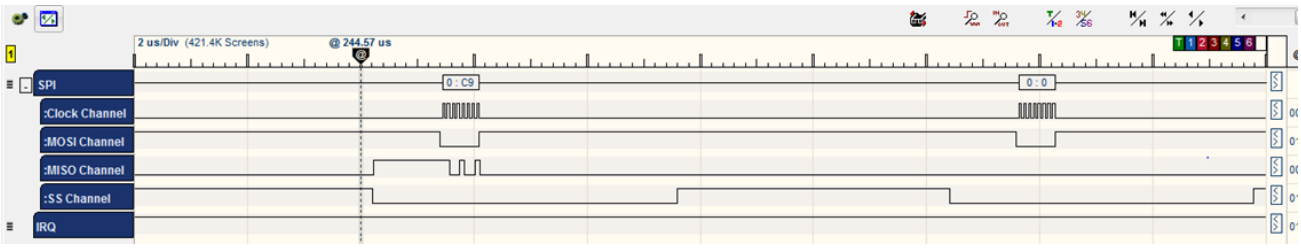



```
Command       CMD_SINGLE_WRITE:0XC9          /* single word write          */
                BYTE [0] = CMD_SINGLE_WRITE
                BYTE [1] = address >> 16;                   /* WAKE_REG address = 0x1074 */
                BYTE [2] = address >> 8;
                BYTE [3] = address;
     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





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



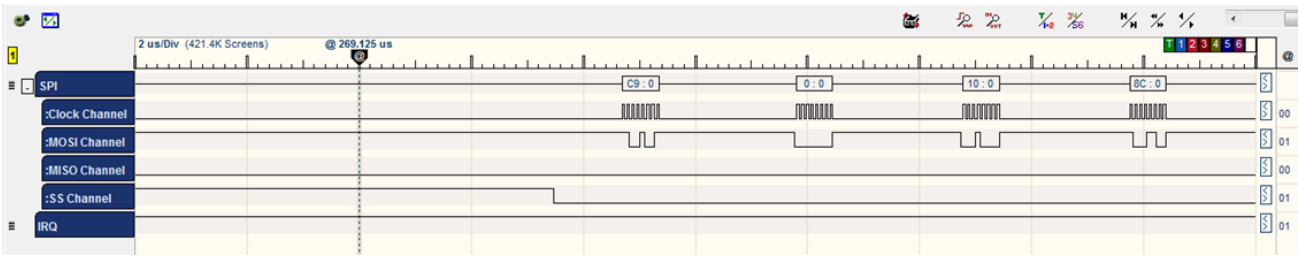
At this point, HIF finishes executing the clockless wakeup of the ATWINC chip.

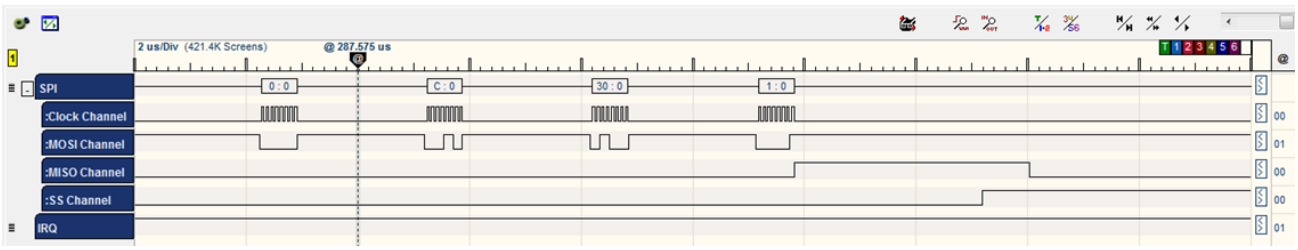
The HIF layer Prepares and Sets the HIF layer header to NMI_STATE_REG register (4 | 8 Byte header describing the packet to be sent).

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

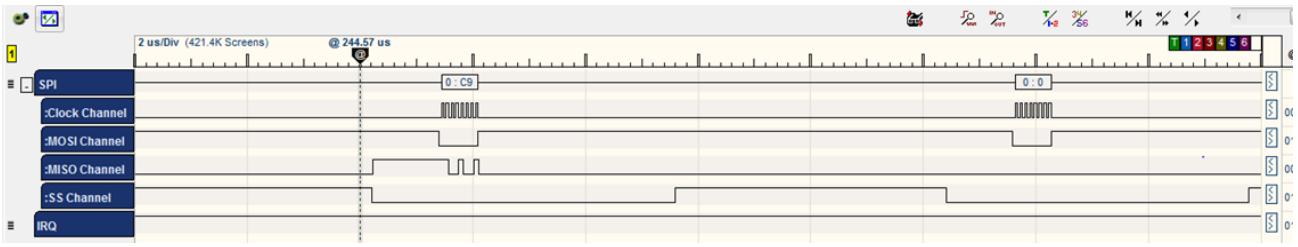
```
sint8 hif_send(uint8 u8Gid,uint8 u8Opcode,uint8 *pu8CtrlBuf,uint16 u16CtrlBufSize,
                                uint8 *pu8DataBuf,uint16 u16DataSize, uint16 u16DataOffset)
{
        volatile tstrHifHdr     strHif;
        volatile uint32 reg;
        strHif.u8Opcode      = u8Opcode&(~NBIT7);
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

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

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

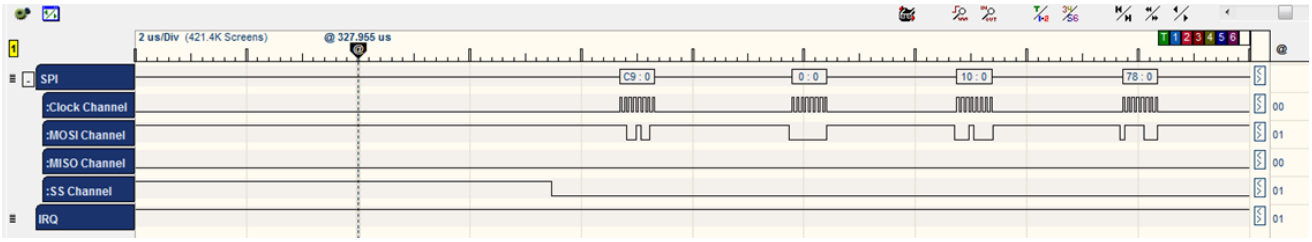


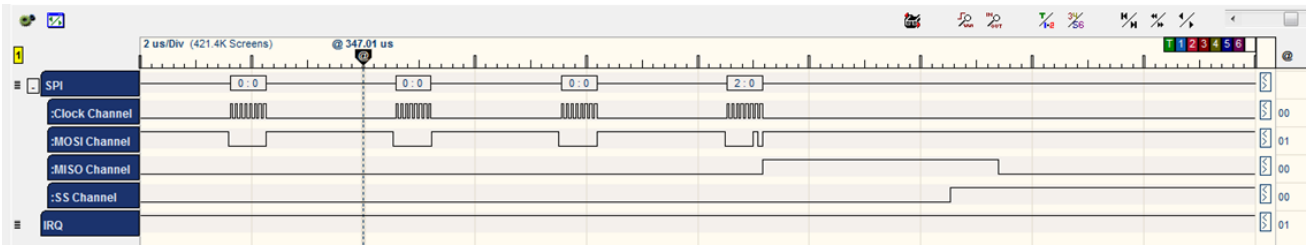


ATWINC acknowledges the command by sending two bytes [C9] [0].

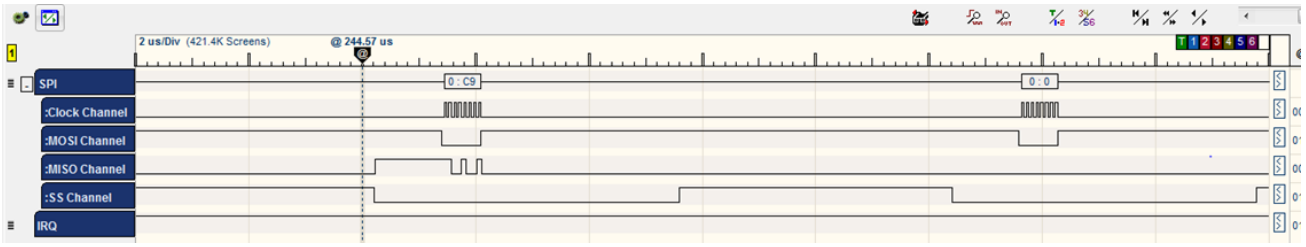


```
Command       CMD_SINGLE_WRITE:0XC9 /*     single word write */
               BYTE [0] = CMD_SINGLE_WRITE
               BYTE [1] = address >> 16;   /*     WIFI_HOST_RCV_CTRL_2address = 0x1087*/
               BYTE [2] = address >> 8;
               BYTE [3] = address;
               BYTE [4] =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





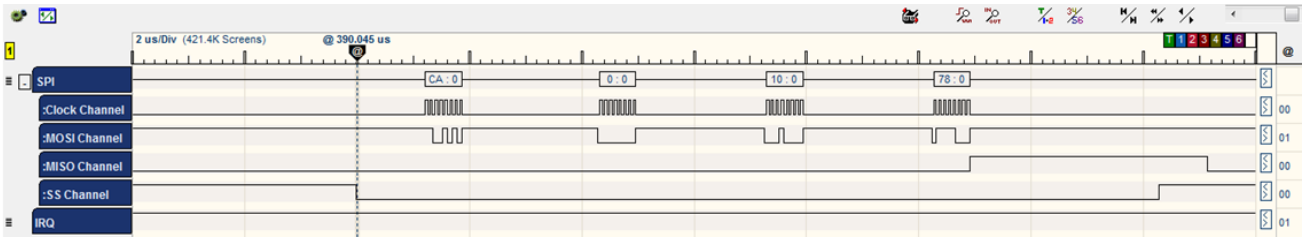
ATWINC acknowledges the command by sending two bytes [C9] [0].



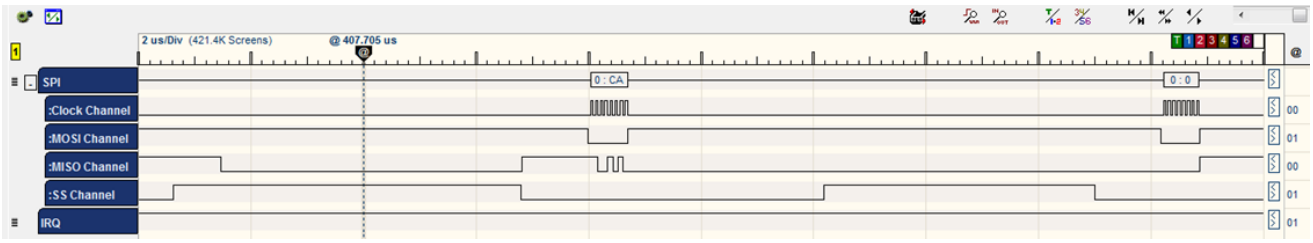
Then HIF polls for DMA address.

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

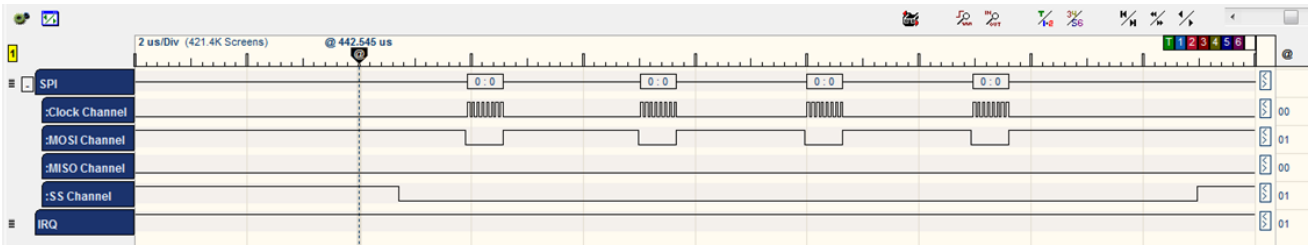
```
Command       CMD_SINGLE_READ:      0xCA          /* single word (4 bytes) read */
               BYTE [0] = CMD_SINGLE_READ
               BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_2 address = 0x1078 */
               BYTE [2] = address >> 8;
               BYTE [3] = address;
```



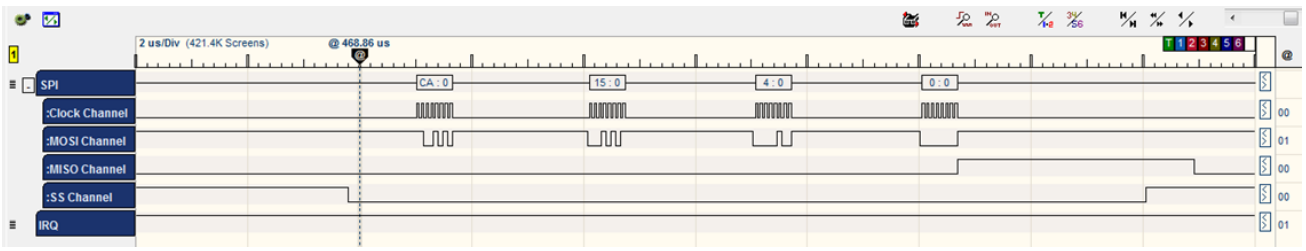
ATWINC acknowledges the command by sending three bytes [CA] [0] [F3].



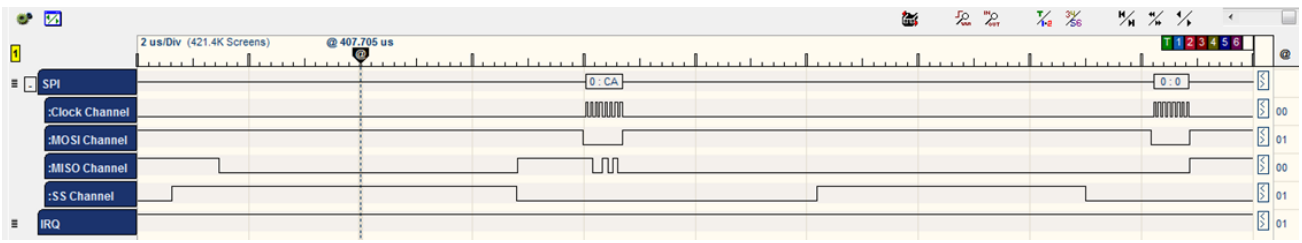
Then the ATWINC chip send the value of the register 0x1078 which equals 0x00.



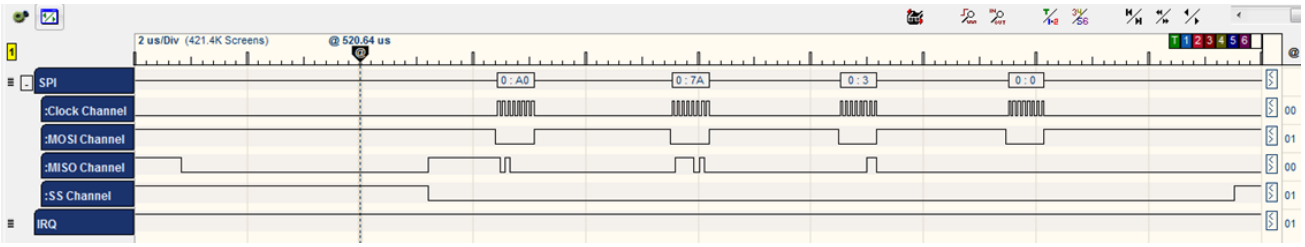

```
Command       CMD_SINGLE_READ:      0xCA          /* single word (4 bytes) read */
               BYTE [0] = CMD_SINGLE_READ
               BYTE [1] = address >> 16;          /* address = 0x1504 */
               BYTE [2] = address >> 8;
               BYTE [3] = address;
```



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



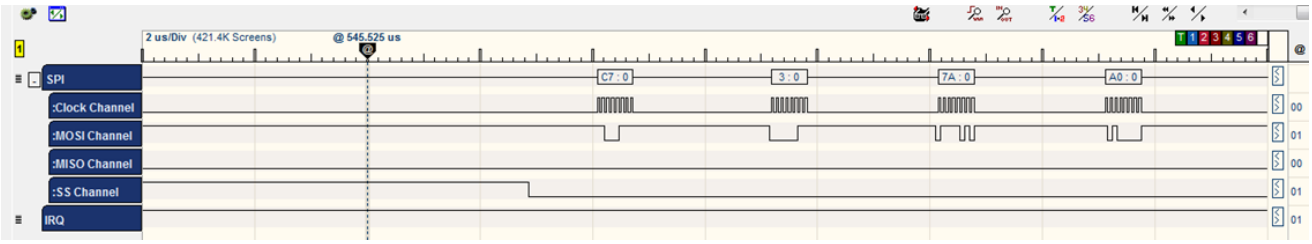
Then the ATWINC chip send the value of the register 0x1504 which equals 0x037AA0.

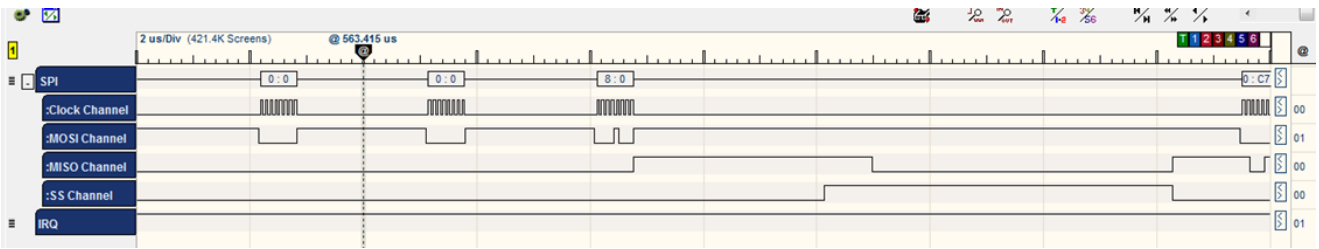


ATWINC writes the HIF header to the DMA memory address.

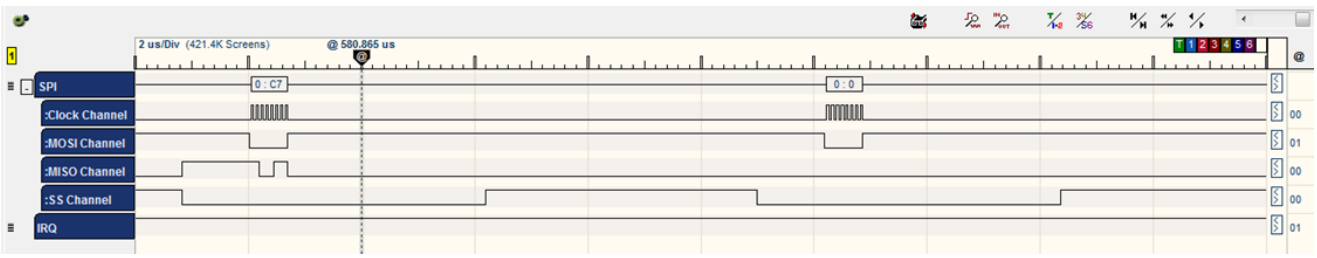
```
u32CurrAddr = dma_addr;
strHif.u16Length=NM_BSP_B_L_16(strHif.u16Length);
ret = nm_write_block(u32CurrAddr, (uint8*)&strHif, M2M_HIF_HDR_OFFSET);
```

```
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;
```

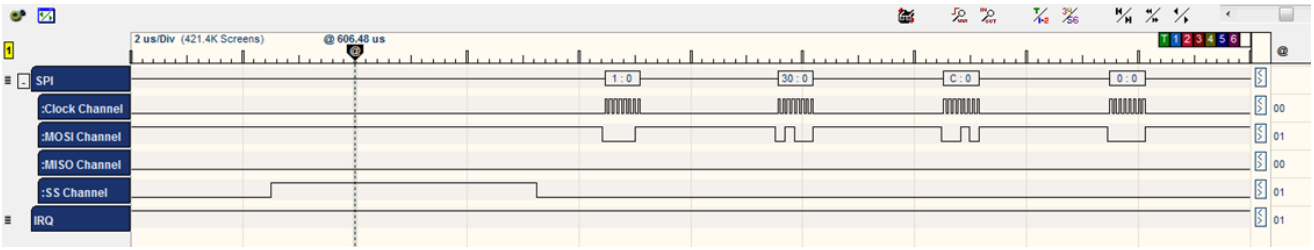


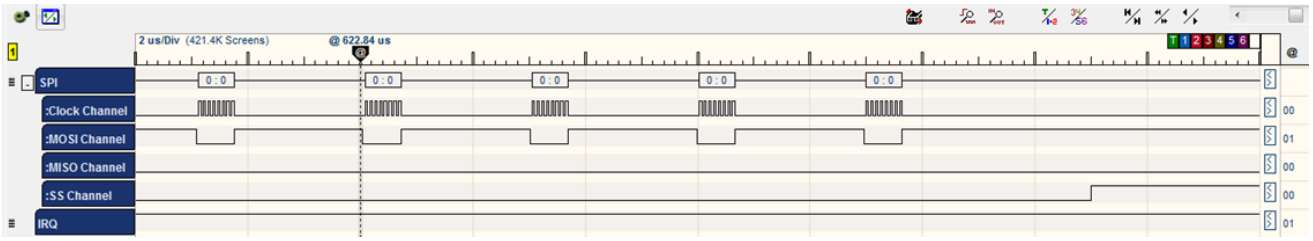


ATWINC acknowledges the command by sending three bytes [C7] [0] [F3].



The HIF layer writes the Data.

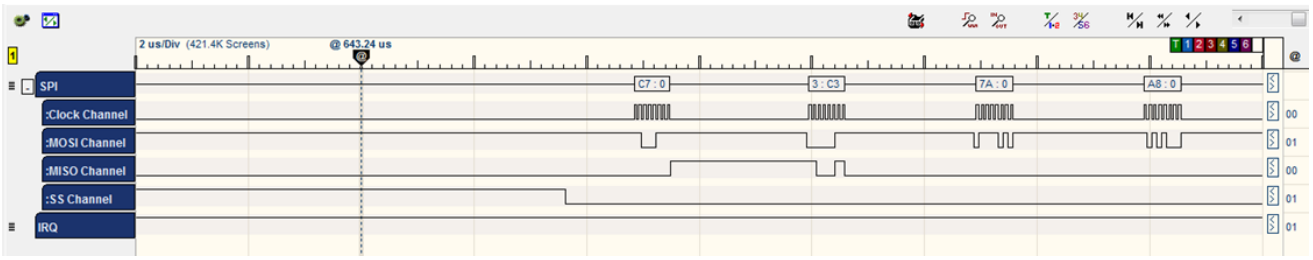


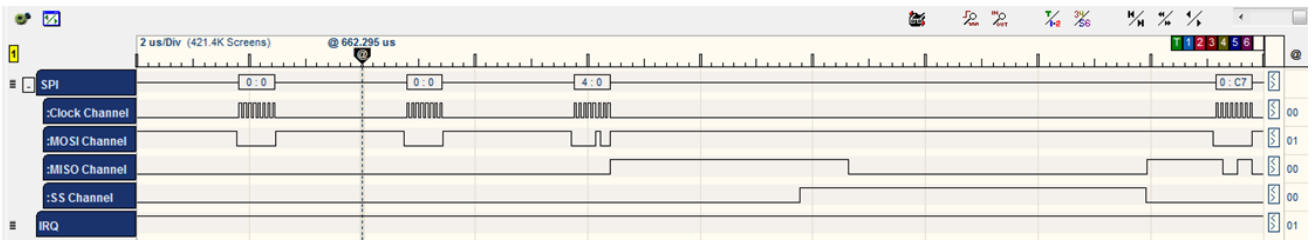


HIF writes the Control Buffer data (part of the framing of the request).

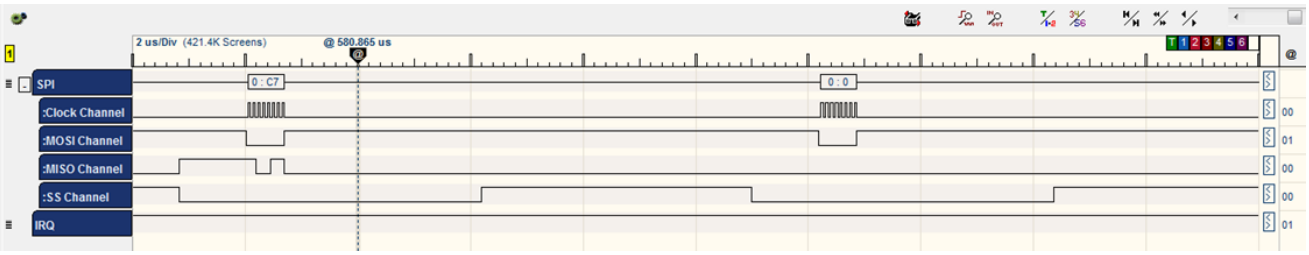
```
if (pu8CtrlBuf != NULL)
{
        ret = nm_write_block(u32CurrAddr, pu8CtrlBuf, u16CtrlBufSize);
        if(M2M_SUCCESS != ret) goto ERR1;
        u32CurrAddr += u16CtrlBufSize;
}
```

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

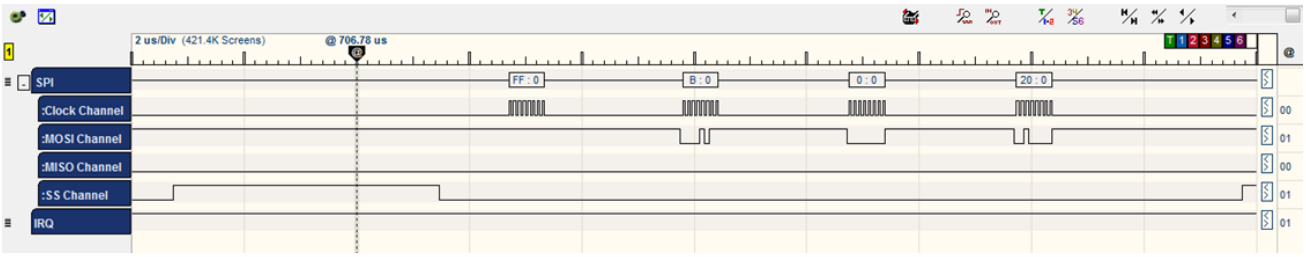




ATWINC acknowledges the command by sending three bytes [C7] [0] [F3].



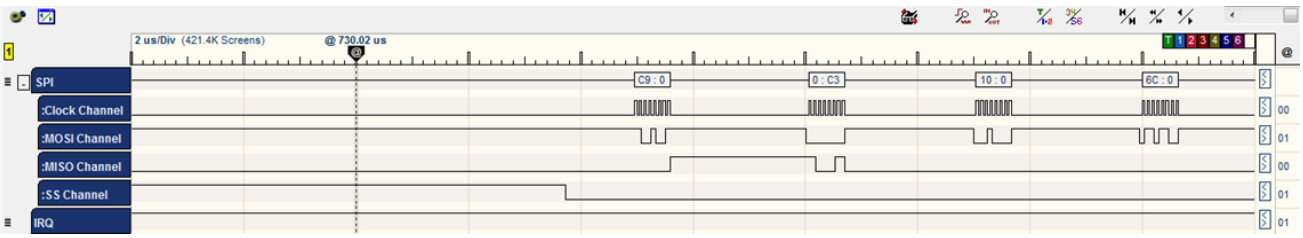
HIF layer writes the Data.

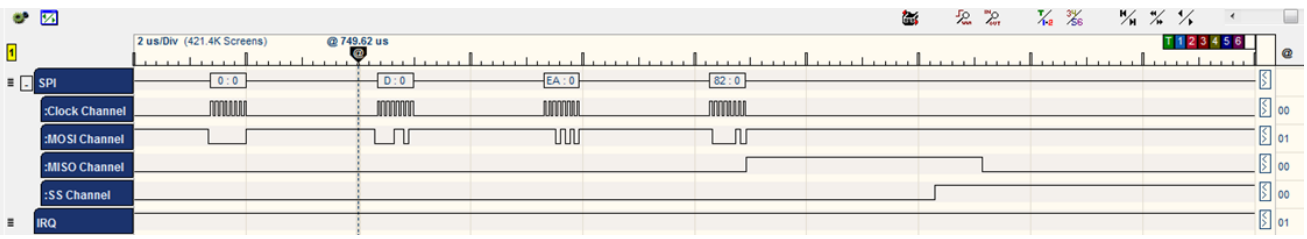


Finally, HIF finished writing the request data to memory and is going to interrupt the chip announcing that host TX is done.

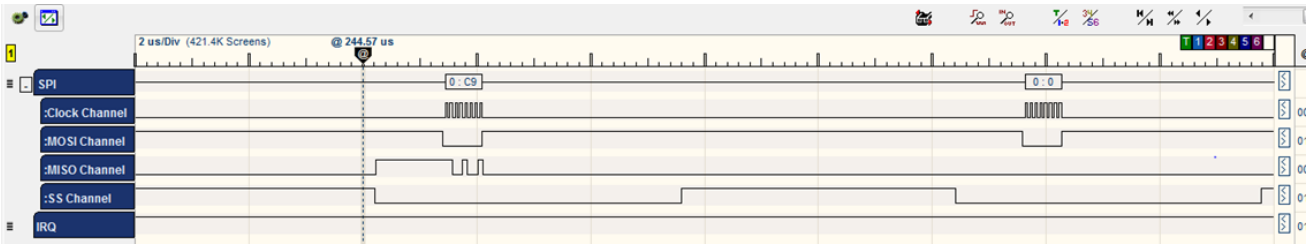
```
reg = dma_addr << 2;
reg |= (1 << 1);
ret = nm_write_reg(WIFI_HOST_RCV_CTRL_3, reg);
```

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





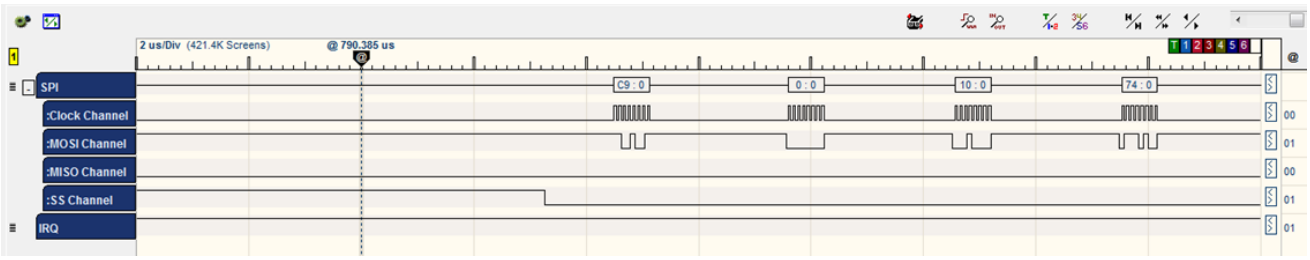
ATWINC acknowledges the command by sending two bytes [C9] [0].

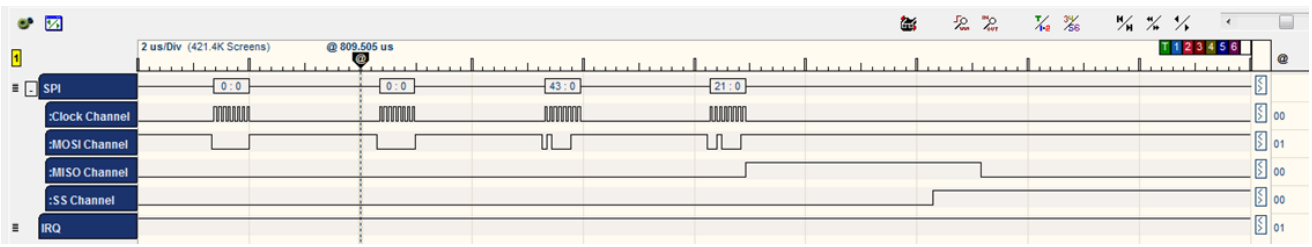


HIF layer allows the chip to enter sleep mode again.

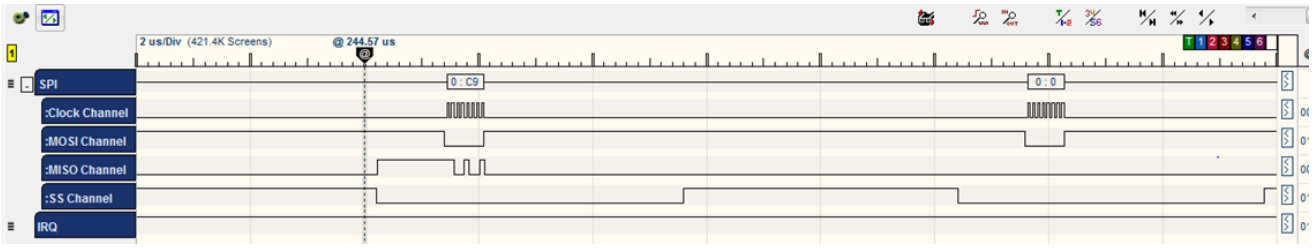
```
sint8 hif_chip_sleep(void)
{
		sint8 ret = M2M_SUCCESS;
		uint32 reg = 0;
		ret = nm_write_reg(WAKE_REG, SLEEP_VALUE);
		/* Clear bit 1 */
		ret = nm_read_reg_with_ret(0x1, &reg);
		if(reg&0x2)
		{
			reg &=~(1 << 1);
			ret = nm_write_reg(0x1, reg);
		}
}
```

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

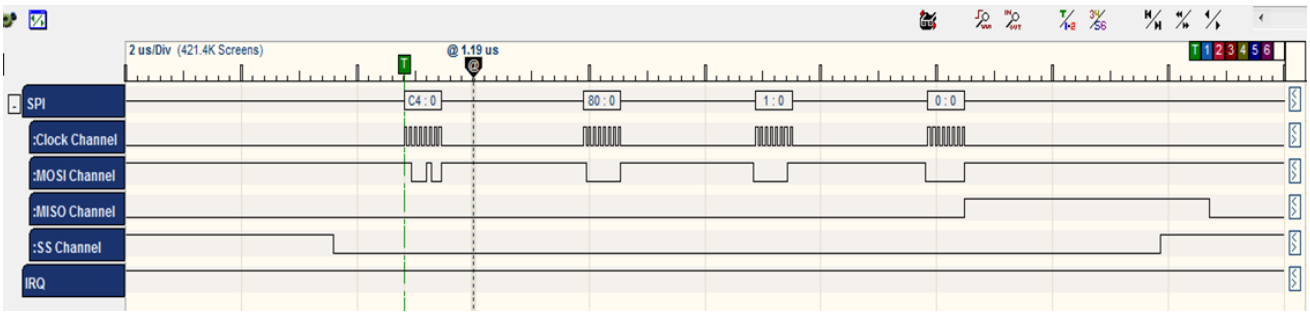




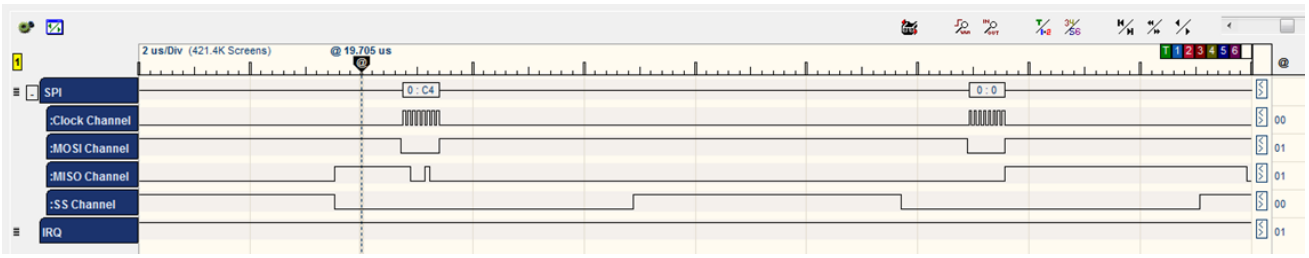
ATWINC acknowledges the command by sending two bytes [C9] [0].



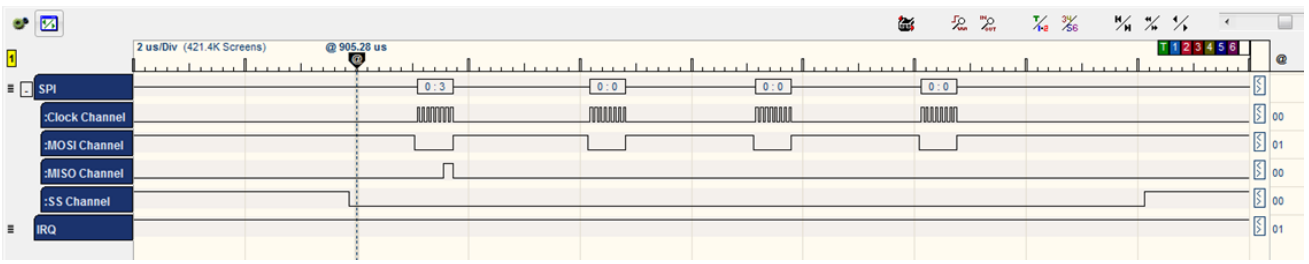

```
Command         CMD_INTERNAL_READ:   0xC4          /* internal register read   */
                BYTE [0] = CMD_INTERNAL_READ
                BYTE [1] = address >> 8;                        /* address = 0x01    */
                BYTE [1] |= (1 << 7);                           /* clockless register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



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

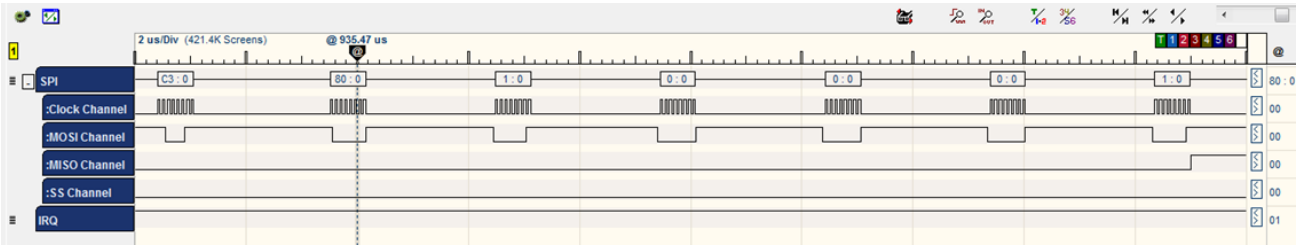


Then the ATWINC chip sends the value of the register 0x01 which equals 0x03.

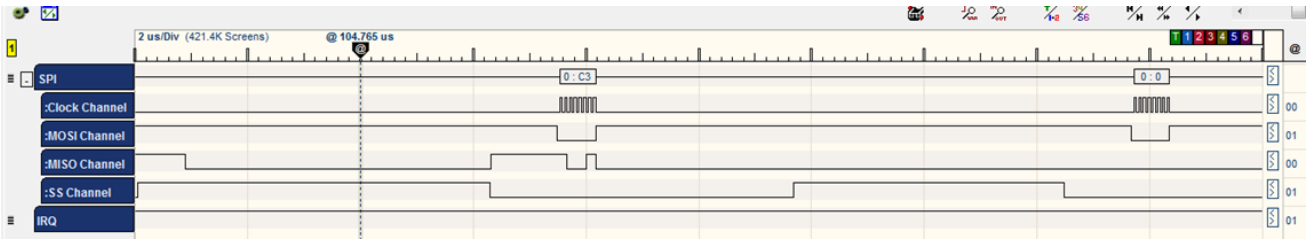



```
Command         CMD_INTERNAL_WRITE:  C3             /* internal register write   */
                BYTE [0] = CMD_INTERNAL_WRITE
                BYTE [1] = address >> 8;                        /* address = 0x01     */
                BYTE [1] |= (1 << 7);                           /* clockless register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

```
BYTE [3] = u32data >> 24;                /* Data = 0x01              */
BYTE [4] = u32data >> 16;
BYTE [5] = u32data >> 8;
BYTE [6] = u32data;
```



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



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

17.3.2 RX (Receive Response)

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

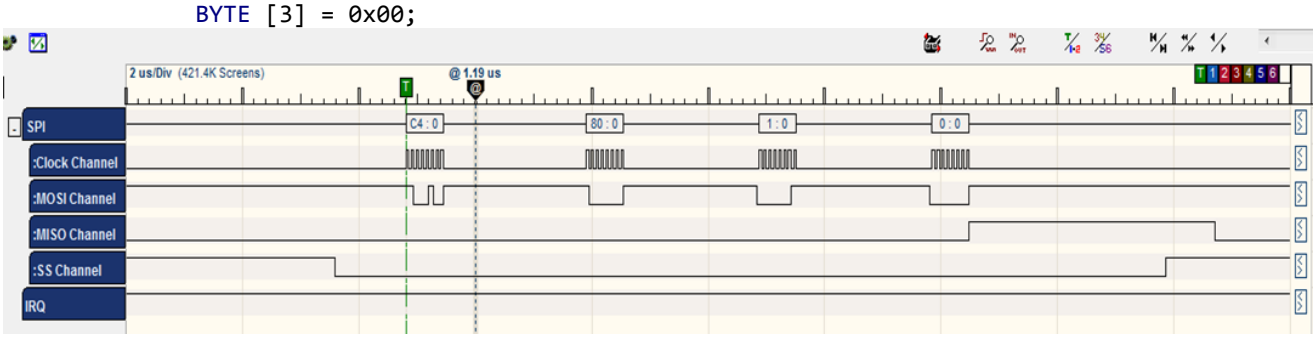
Host will handle this interrupt to receive the response.

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

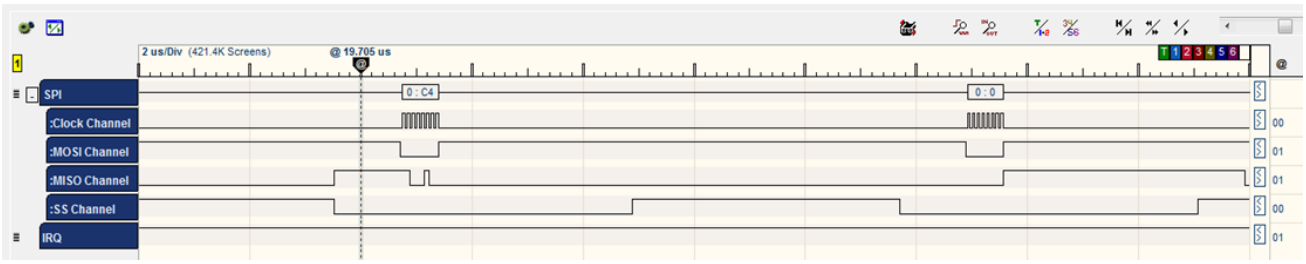
```
sint8 nm_clkless_wake(void)
{
        ret = nm_read_reg_with_ret(0x1, &reg);
        /* Set bit 1 */
        ret = nm_write_reg(0x1, reg | (1 << 1));
        // Check the clock status
        ret = nm_read_reg_with_ret(clk_status_reg_adr, &clk_status_reg);
        // Tell Firmware that Host waked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

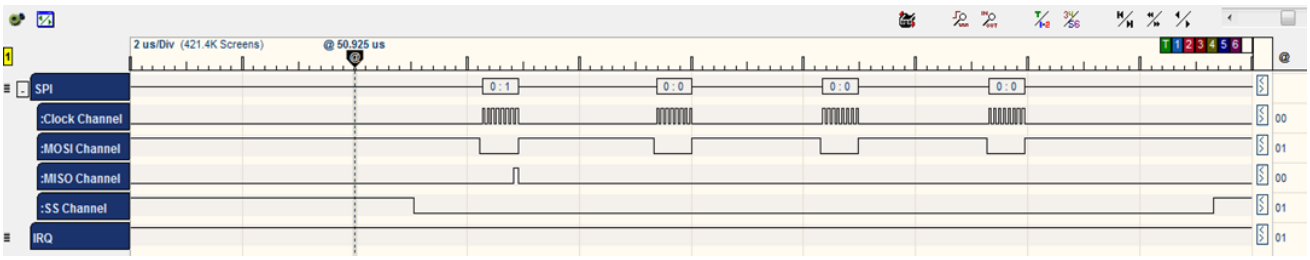
```
BYTE [3] = 0x00;
```



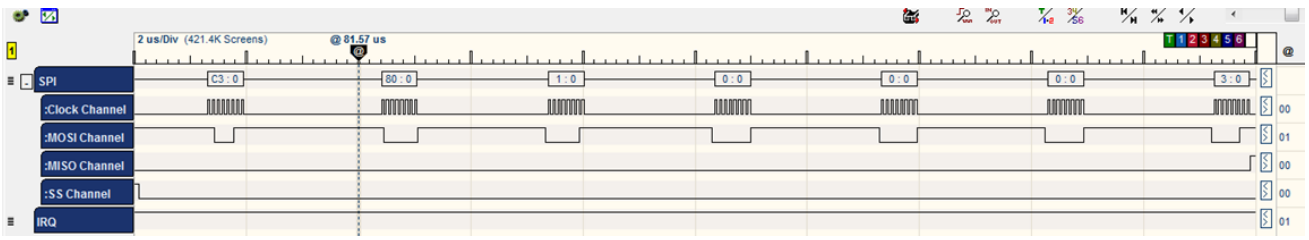
ATWINC acknowledges the command by sending three bytes [C4] [0] [F3].



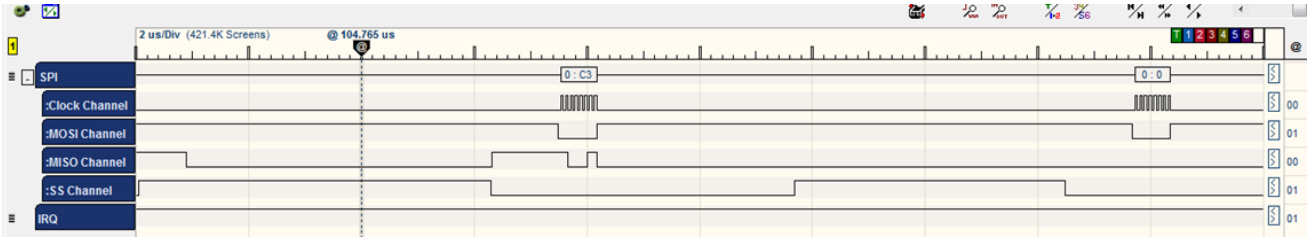
Then the ATWINC chip sends the value of the register 0x01 which equals 0x01.



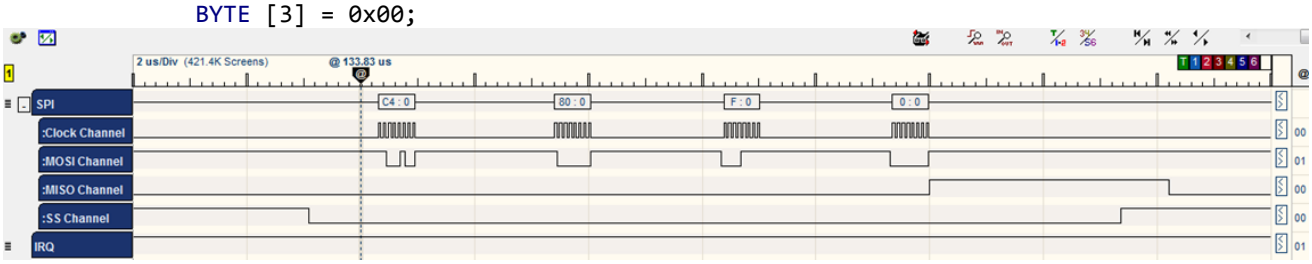

```
Command         CMD_INTERNAL_WRITE:  C3              /*    internal register write */
                BYTE [0] = CMD_INTERNAL_WRITE
                BYTE [1] = address >> 8;                        /*     address = 0x01
        */
                BYTE [1] |= (1 << 7);                           /* 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



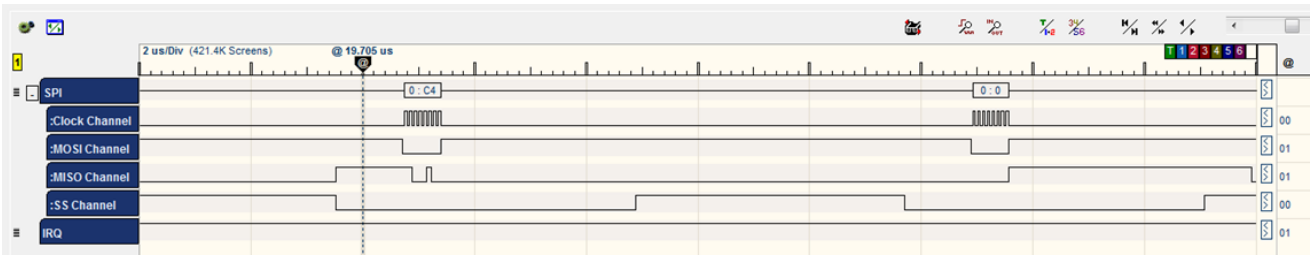
ATWINC acknowledges the command by sending two bytes [C3] [0].



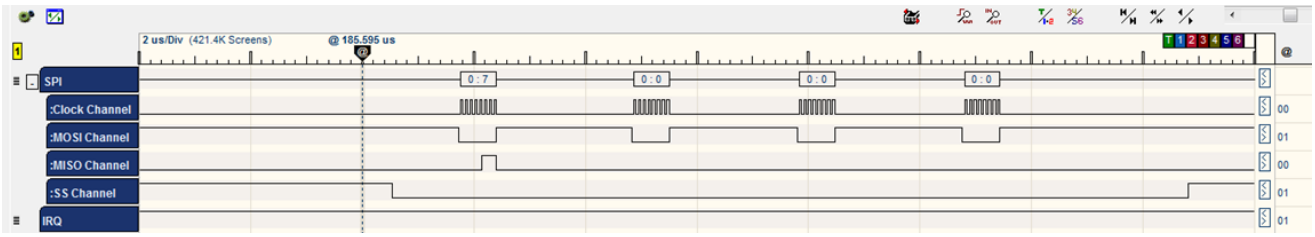

```
Command         CMD_INTERNAL_READ:   0xC4          /*     internal register read      */
                BYTE [0] = CMD_INTERNAL_READ
                BYTE [1] = address >> 8;                        /*     address = 0x0F
        */
                BYTE [1] |= (1 << 7);                           /*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



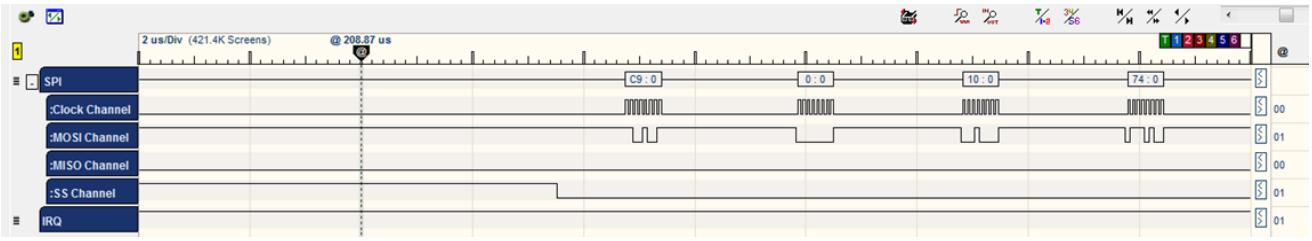
ATWINC acknowledges the command by sending three bytes [C4] [0] [F3].

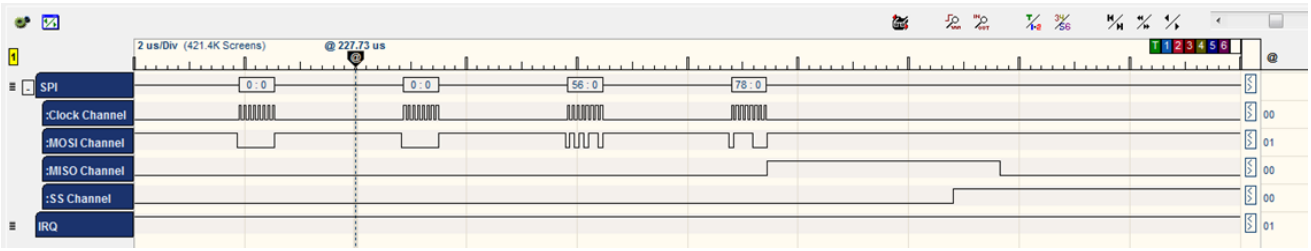


Then the ATWINC chip sends the value of the register 0x01 which equals 0x07.

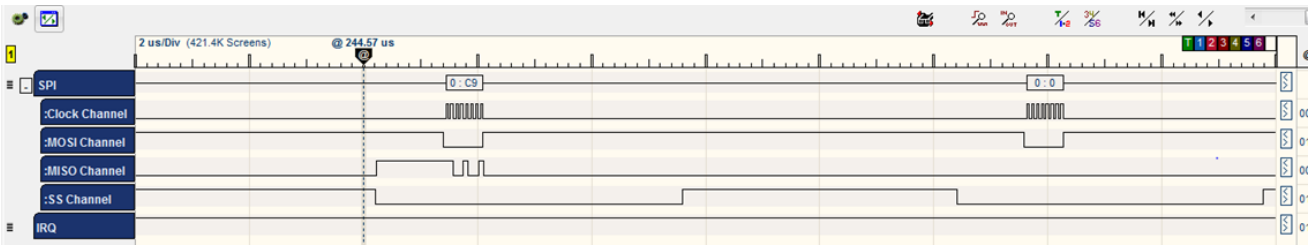



```
Command         CMD_SINGLE_WRITE:0XC9        /* single word write */
                BYTE [0] = CMD_SINGLE_WRITE
                BYTE [1] = address >> 16;           /* WAKE_REG address = 0x1074 */
                BYTE [2] = address >> 8;
                BYTE [3] = address;
                BYTE [4]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





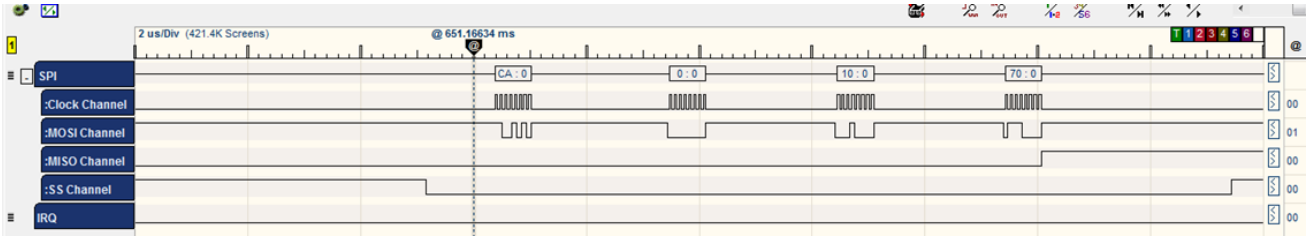
The chip acknowledges the command by sending two bytes [C9] [0].



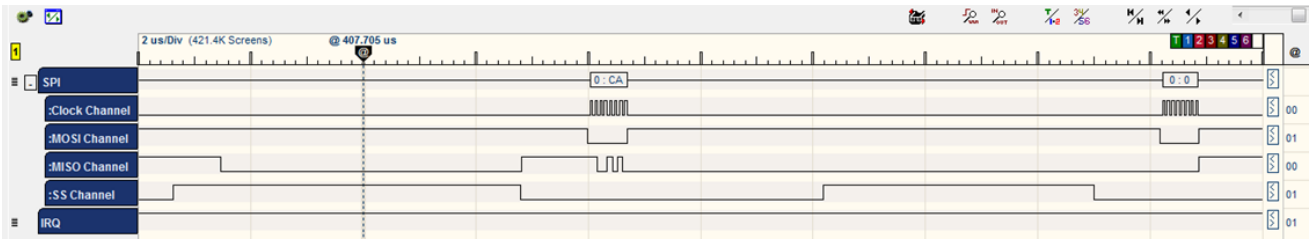
Read register WIFI_HOST_RCV_CTRL_0 to check if there is new interrupt, and if so, clear it (as it will be handled now).

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

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

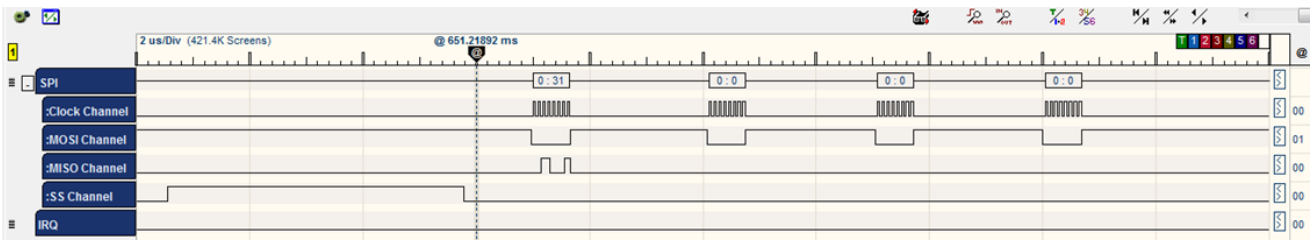
```
Command      CMD_SINGLE_READ:      0xCA          /* single word (4 bytes) read              */
              BYTE [0] = CMD_SINGLE_READ
              BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
              BYTE [2] = address >> 8;
              BYTE [3] = address;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



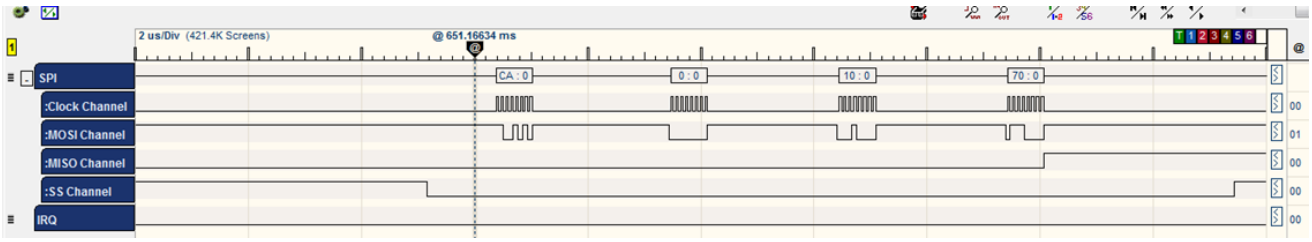
ATWINC acknowledges the command by sending three bytes [CA] [0] [F3].



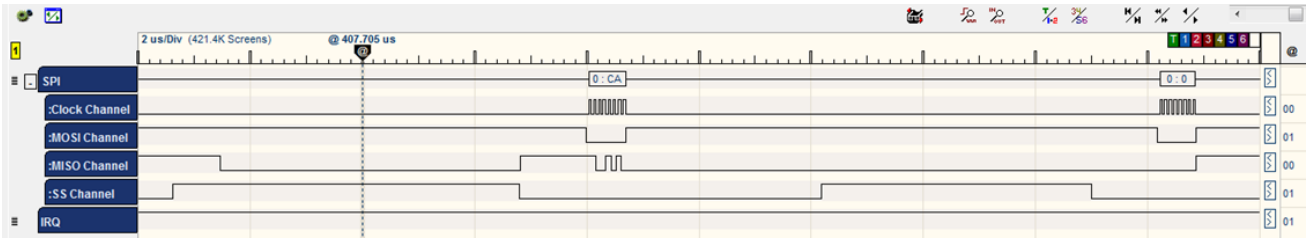
Then the ATWINC chip sends the value of the register 0x1070 which equals 0x31.



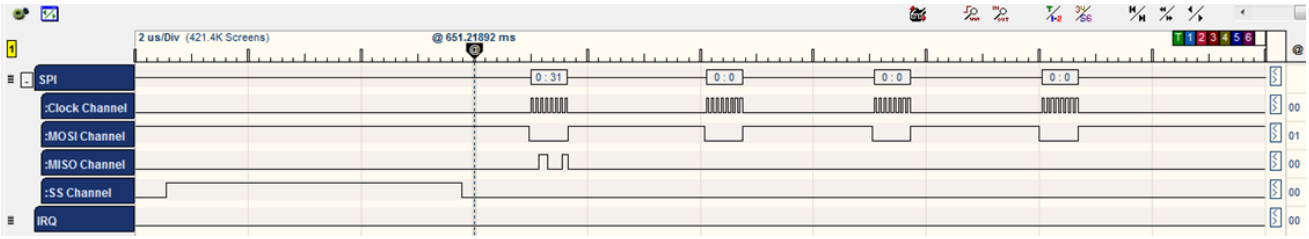

```
Command       CMD_SINGLE_READ:      0xCA          /* single word (4 bytes) read              */
               BYTE [0] = CMD_SINGLE_READ
               BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
               BYTE [2] = address >> 8;
               BYTE [3] =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



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

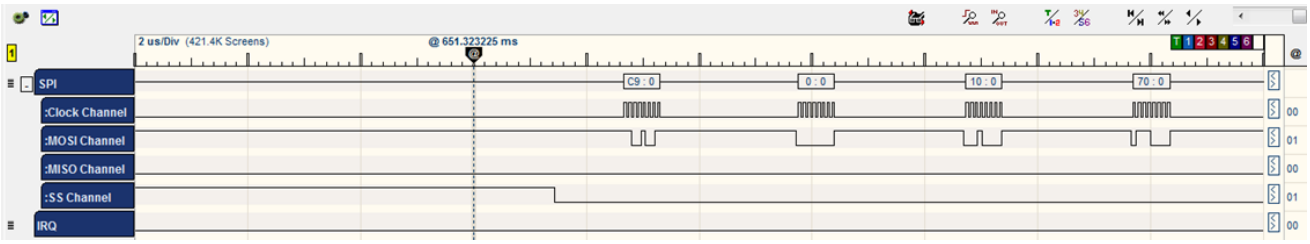


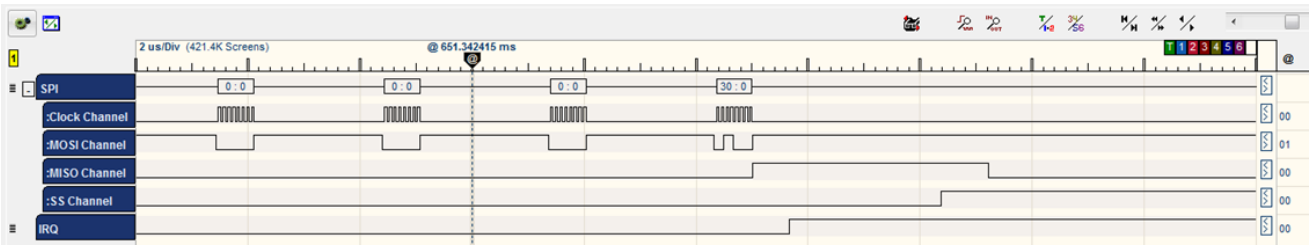
Then the ATWINC chip sends the value of the register 0x1070 which equals 0x31.



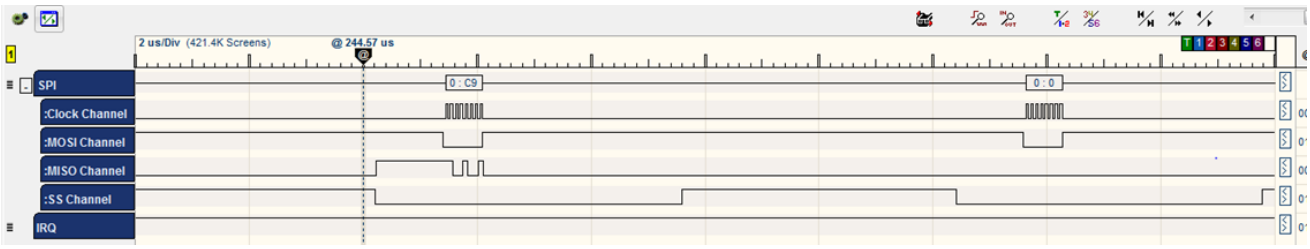
Clear the ATWINC Interrupt.

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





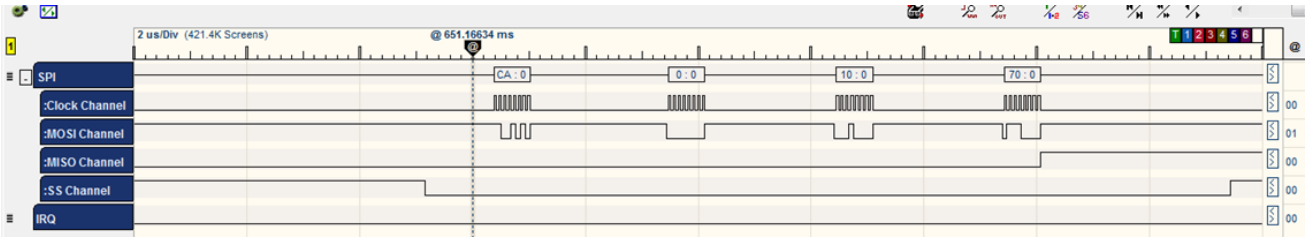
The chip acknowledges the command by sending two bytes [C9] [0].



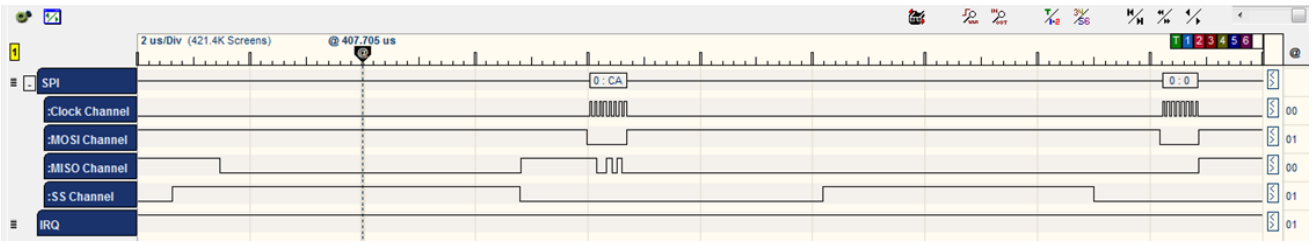
Then HIF reads the data size.

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

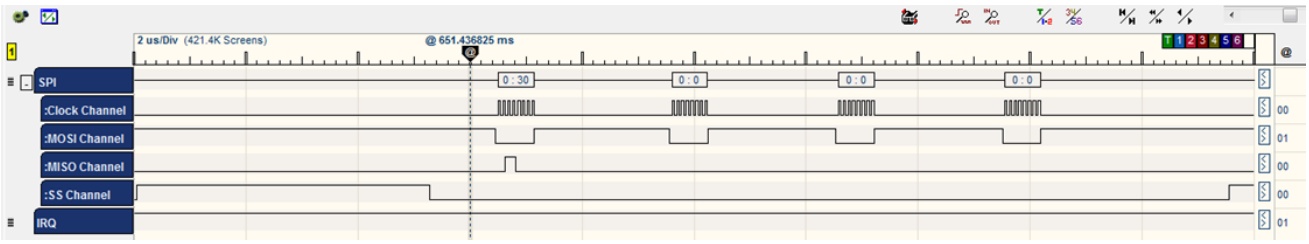
```
Command        CMD_SINGLE_READ:      0xCA          /* single word (4 bytes) read              */
                BYTE [0] = CMD_SINGLE_READ
                BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
                BYTE [2] = address >> 8;
                BYTE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



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



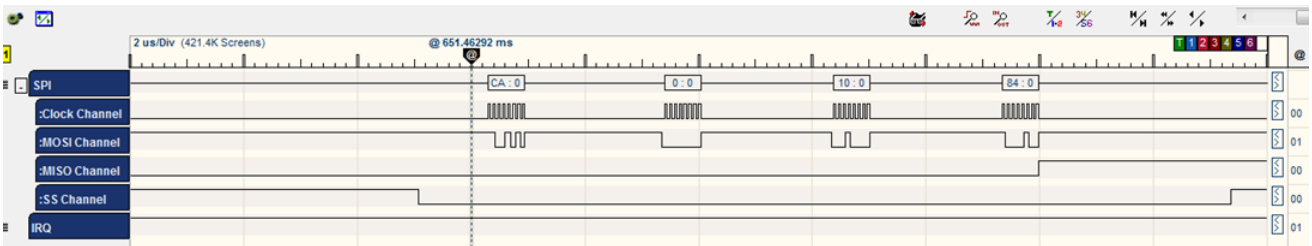
Then the ATWINC chip sends the value of the register 0x1070 which equals 0x30.



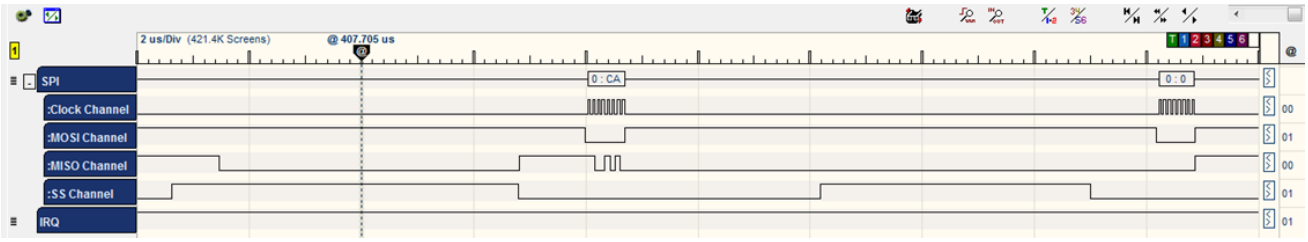
HIF reads hif header address.

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

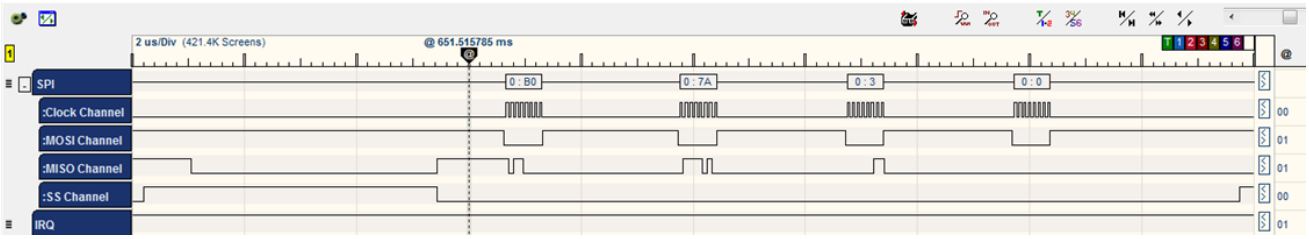
```
Command        CMD_SINGLE_READ:      0xCA          /* single word (4 bytes) read */
                BYTE [0] = CMD_SINGLE_READ
                BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_1 address = 0x1084 */
                BYTE [2] = address >> 8;
                BYTE [3] = address;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



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



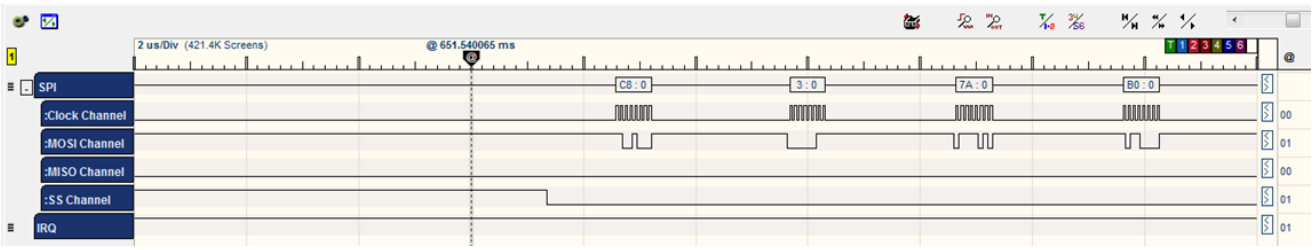
Then the ATWINC chip sends the value of the register 0x1078 which equals 0x037AB0.

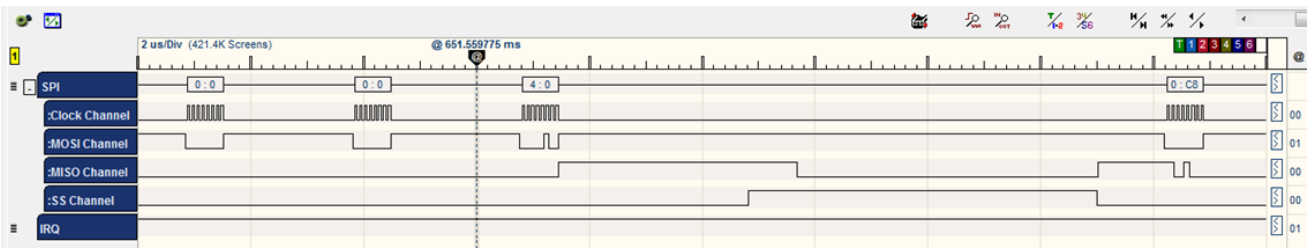


HIF reads the hif header data (as a block).

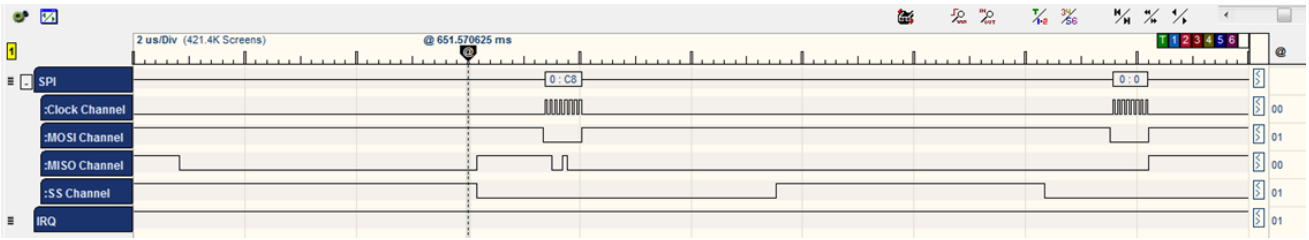
```
ret = nm_read_block(address, (uint8*)&strHif, sizeof(tstrHifHdr));
```

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

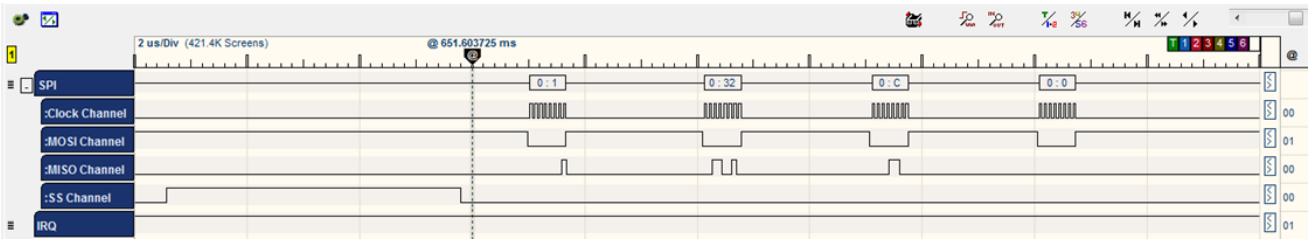




ATWINC acknowledges the command by sending three bytes [C8] [0] [F3].



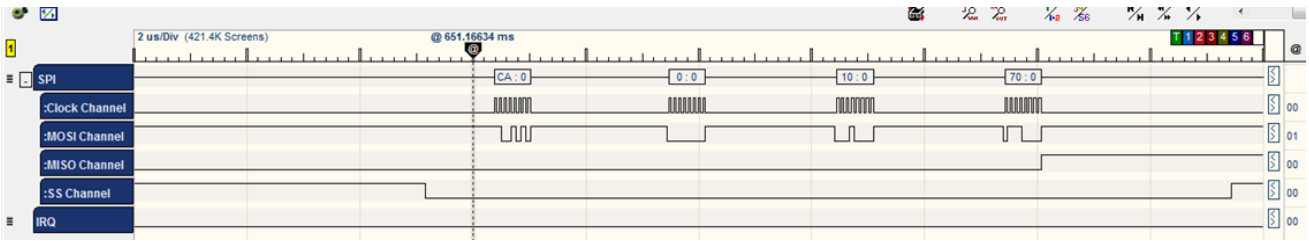
ATWINC sends the data block (four bytes).



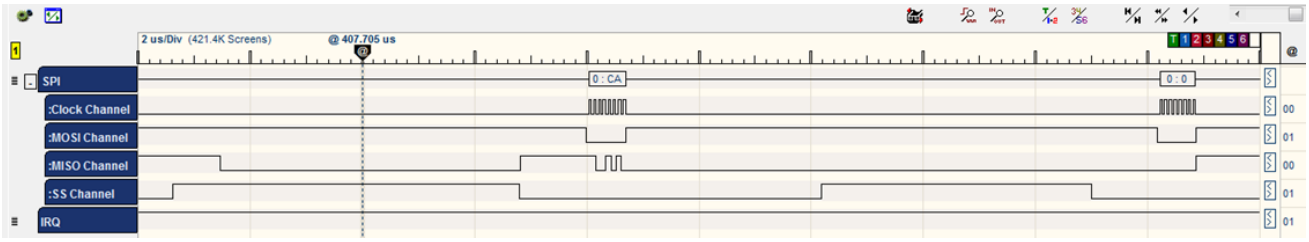
HIF then calls the appropriate handler according to the hif header received which tries to receive the Response data payload. (Note that hif_receive ( ) obtains some data again for checks.)

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

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

}
```

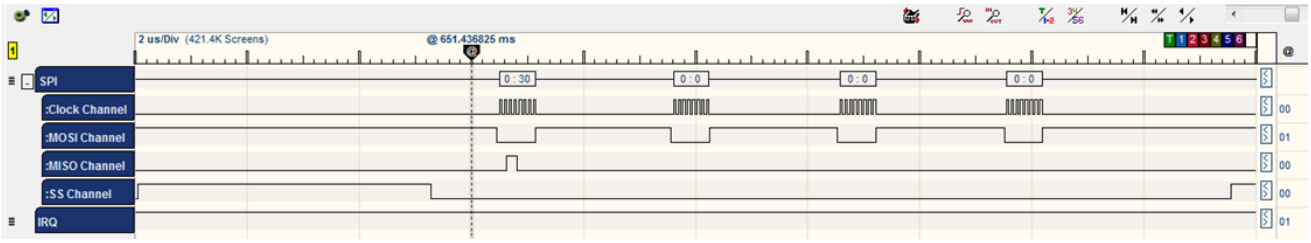
```
Command		CMD_SINGLE_READ:	0xCA		/* single word (4 bytes) read		*/
		 BYTE [0] = CMD_SINGLE_READ
		 BYTE [1] = address >> 16;		/* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
		 BYTE [2] = address >> 8;
		 BYTE [3] = address;
```



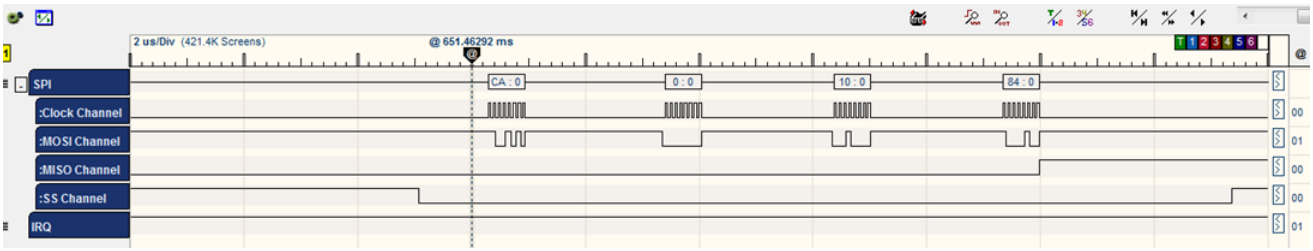
ATWINC acknowledges the command by sending three bytes [CA] [0] [F3].



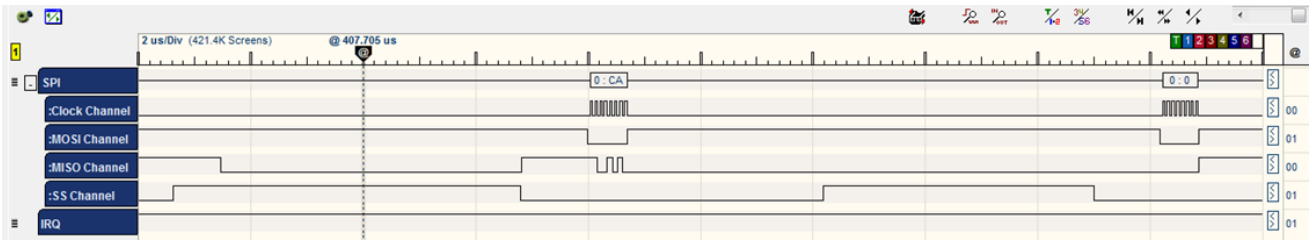
Then the ATWINC chip sends the value of the register 0x1070 which equals 0x30.



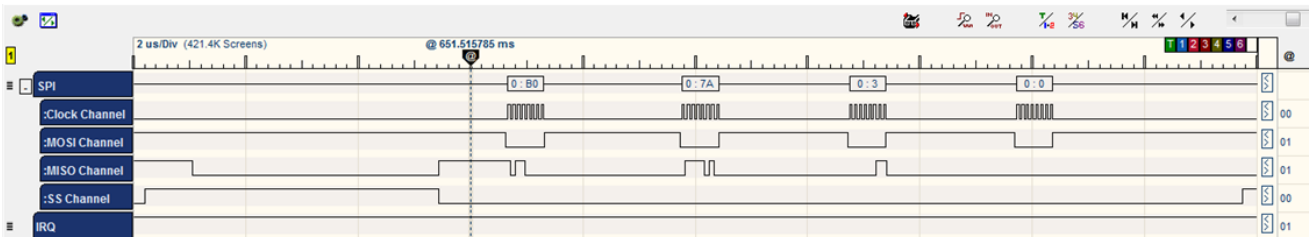

```
Command       CMD_SINGLE_READ:      0xCA          /* single word (4 bytes) read              */
               BYTE [0] = CMD_SINGLE_READ
               BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_1 address = 0x1084 */
               BYTE [2] = address >> 8;
               BYTE [3] =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



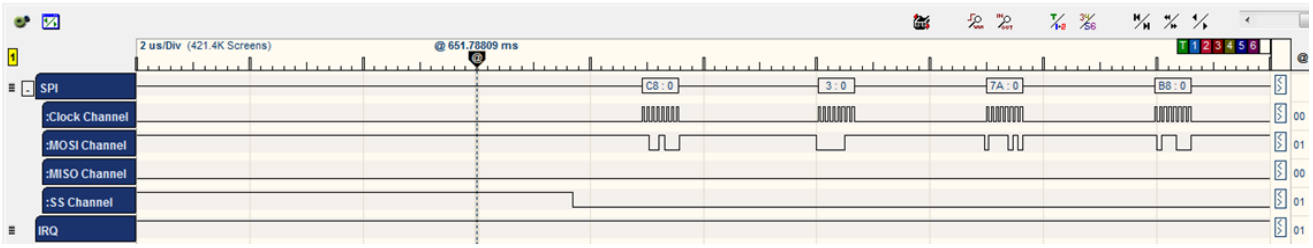
ATWINC acknowledges the command by sending three bytes [CA] [0] [F3].

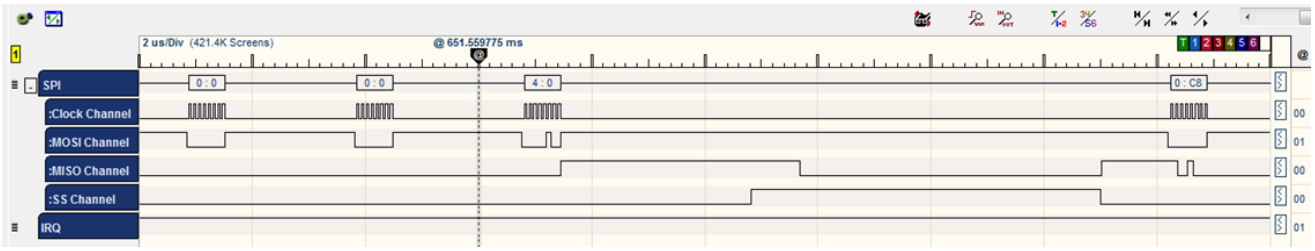


Then the ATWINC chip sends the value of the register 0x1078 which equals 0x037AB0.

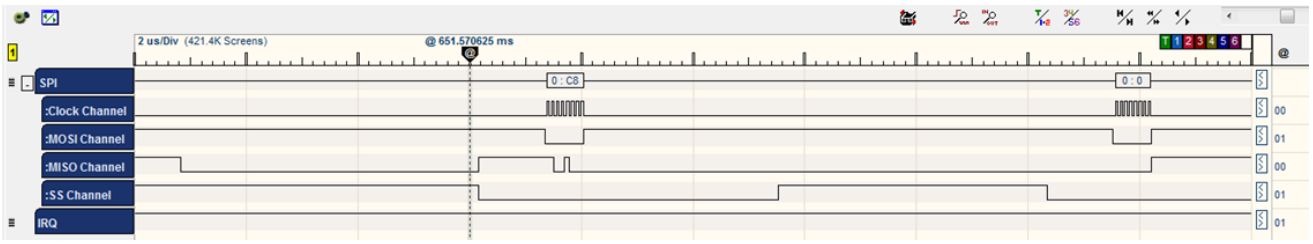


```
Command         CMD_DMA_EXT_READ:     C8              /* dma extended read */
                 BYTE [0] = CMD_DMA_EXT_READ
                 BYTE [1] = address >> 16;           /* address = 0x037AB8*/
                 BYTE [2] = address >> 8;
                 BYTE [3] = address;
                 BYTE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

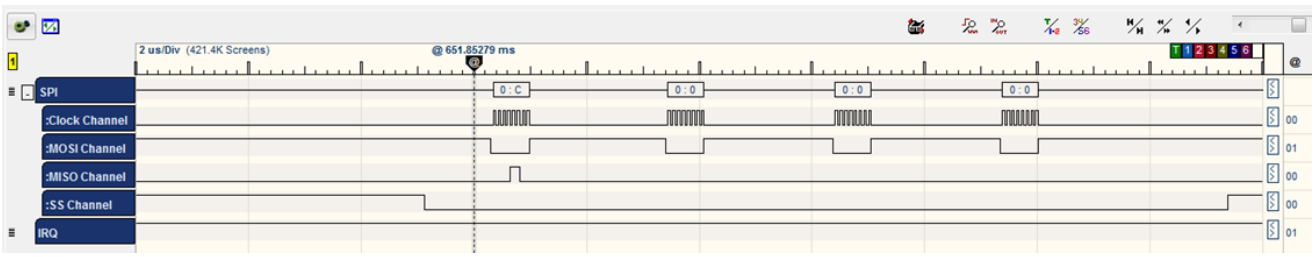




ATWINC acknowledges the command by sending three bytes [C8] [0] [F3].



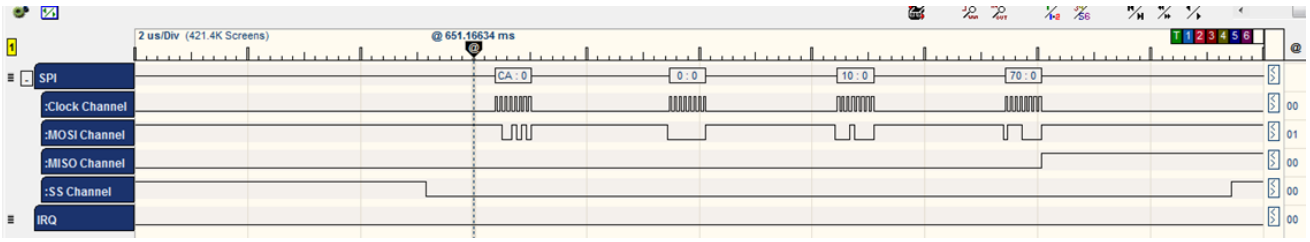
ATWINC sends the data block (four bytes).



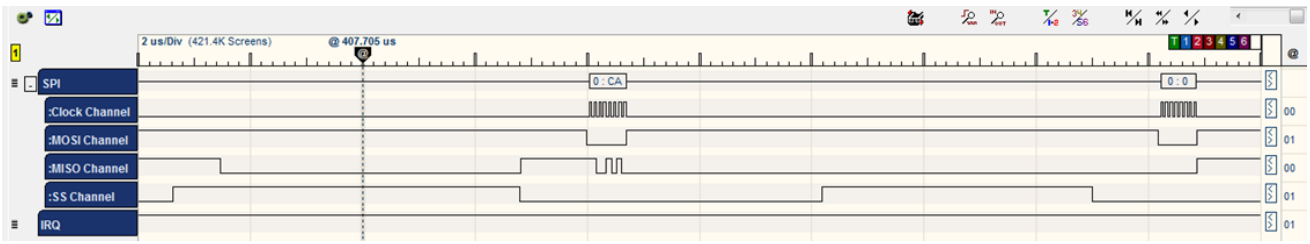
Now, after HIF layer received the response, it interrupts the chip to announce host RX is done.

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

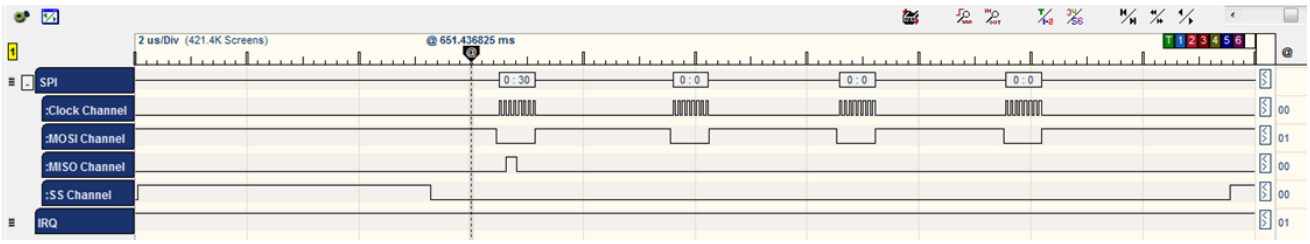
```
Command		CMD_SINGLE_READ:	0xCA		/* single word (4 bytes) read		*/
		 BYTE [0] = CMD_SINGLE_READ
		 BYTE [1] = address >> 16;		/* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
		 BYTE [2] = address >> 8;
		 BYTE [3] = address;
```



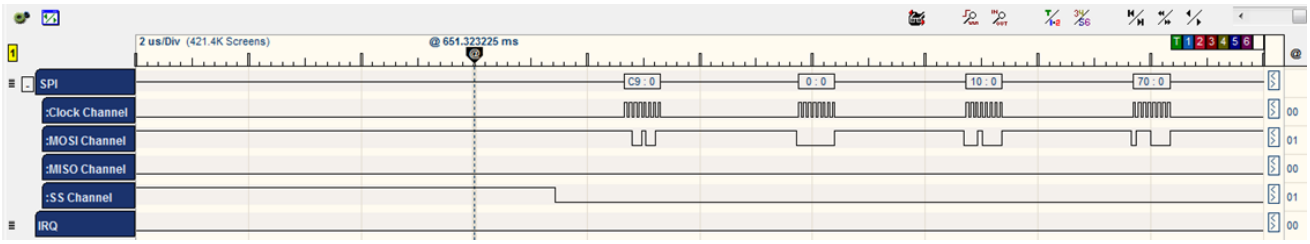
ATWINC acknowledges the command by sending three bytes [CA] [0] [F3].

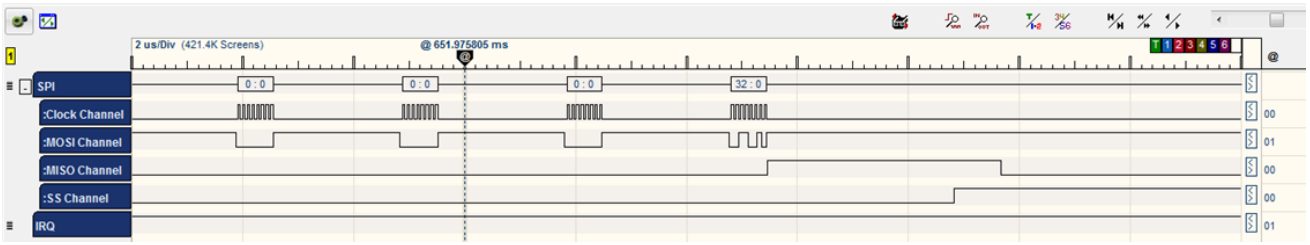


Then the ATWINC chip sends the value of the register 0x1070 which equals 0x30.

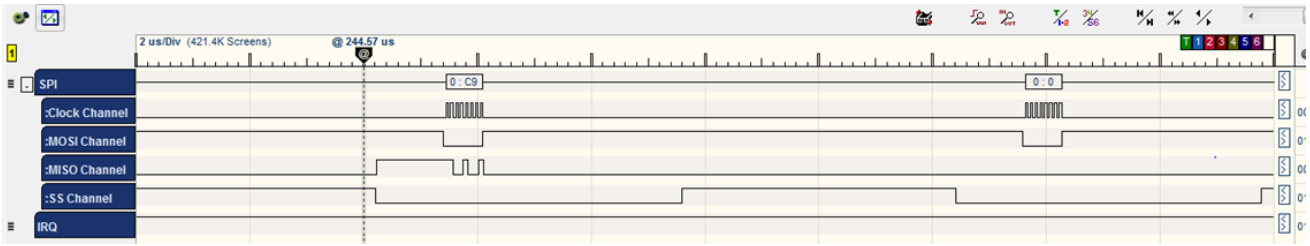



```
Command		CMD_SINGLE_WRITE:0XC9		/* single word write */
		 BYTE [0] = CMD_SINGLE_WRITE
		 BYTE [1] = address >> 16;		/* WIFI_HOST_RCV_CTRL_0 address = 0x1070  */
		 BYTE [2] = address >> 8;
		 BYTE [3] = address;
		 BYTE [4] = u32data >> 24;		/* Data = 0x32*/
		 BYTE [5] = u32data >> 16;
		 BYTE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





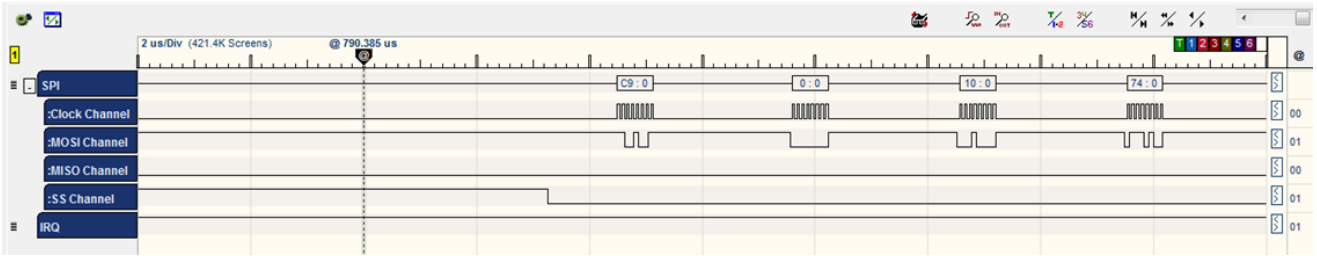
The chip acknowledges the command by sending two bytes [C9] [0].

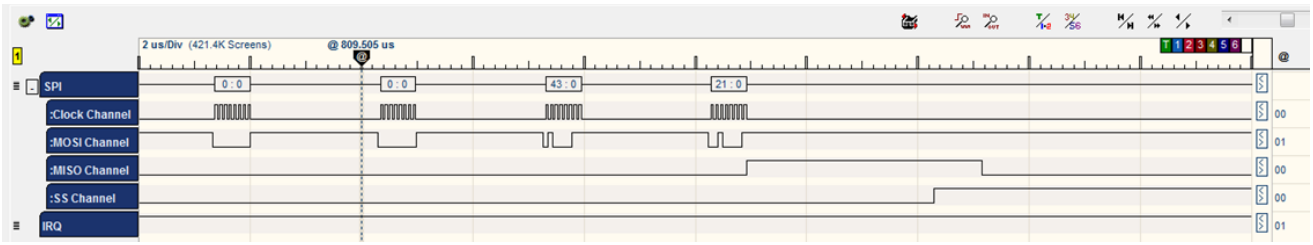


The HIF layer allows the chip to enter sleep mode again.

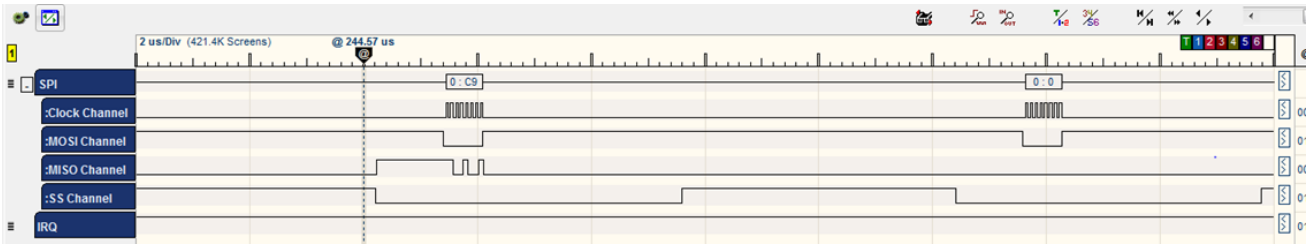
```
sint8 hif_chip_sleep(void)
{
	sint8 ret = M2M_SUCCESS;
	uint32 reg = 0;
	ret = nm_write_reg(WAKE_REG, SLEEP_VALUE);
	/* Clear bit 1 */
	ret = nm_read_reg_with_ret(0x1, &reg);
	if(reg&0x2)
	{
		reg &=~(1 << 1);
		ret = nm_write_reg(0x1, reg);
	}
}
```

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



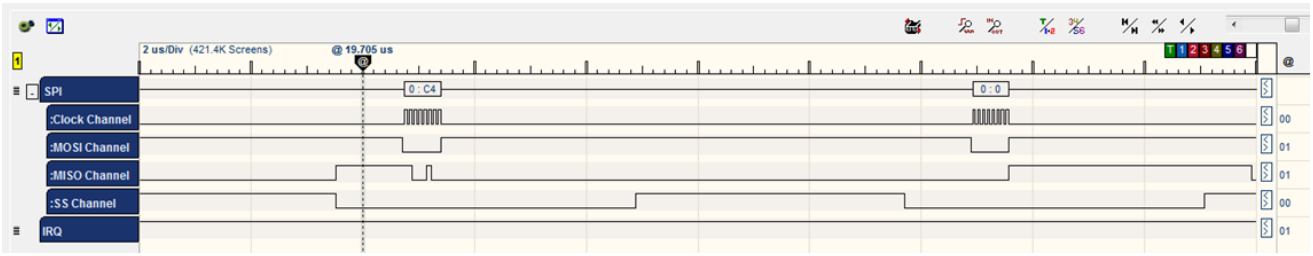


ATWINC acknowledges the command by sending two bytes [C9] [0].

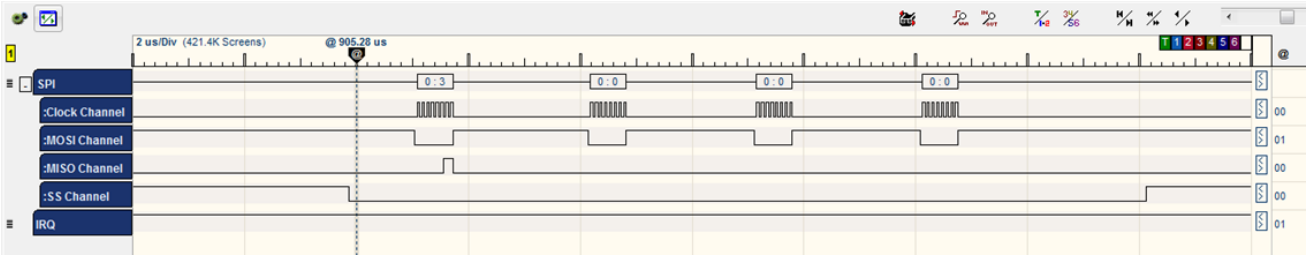



```
Command         CMD_INTERNAL_READ:   0xC4          /* internal register read   */
                BYTE [0] = CMD_INTERNAL_READ
                BYTE [1] = address >> 8;           /* address = 0x01           */
                BYTE [1] |= (1 << 7);              /* clockless register       */
     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

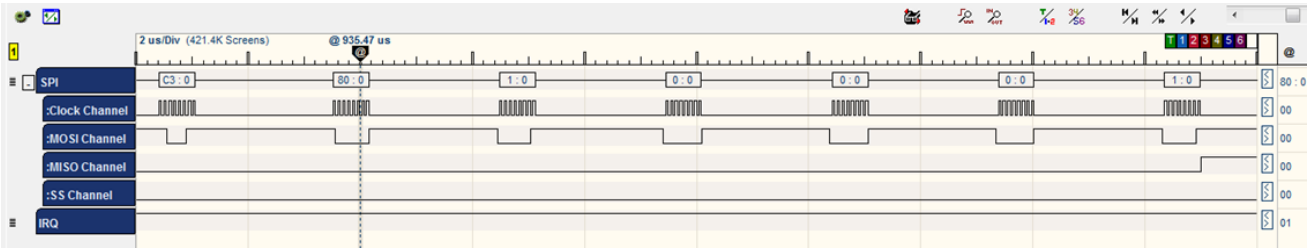
ATWINC acknowledges the command by sending three bytes [C4] [0] [F3].



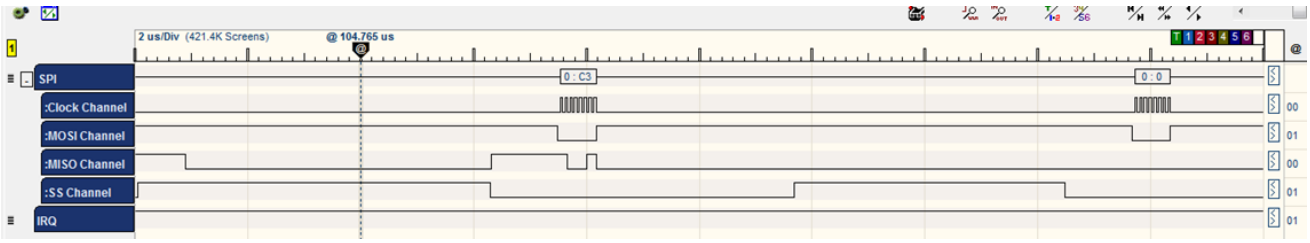
Then the ATWINC chip sends the value of the register 0x01 which equals 0x03.




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



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



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

Appendix A. How to Generate Certificates

A.1 Introduction

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

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

OpenSSL can be found here: https://www.openssl.org

A.2 Steps

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

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

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

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

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

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

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

- Using the key generated above, you should generate a certificate request file (csr):

  CMD: **openssl req -new -key Custom.key -out CertReq.csr**

- Finally: process the request for the certificate and get it signed by the root CA.

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

Appendix B. X.509 Certificate Format and Conversion

B.1 Introduction

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

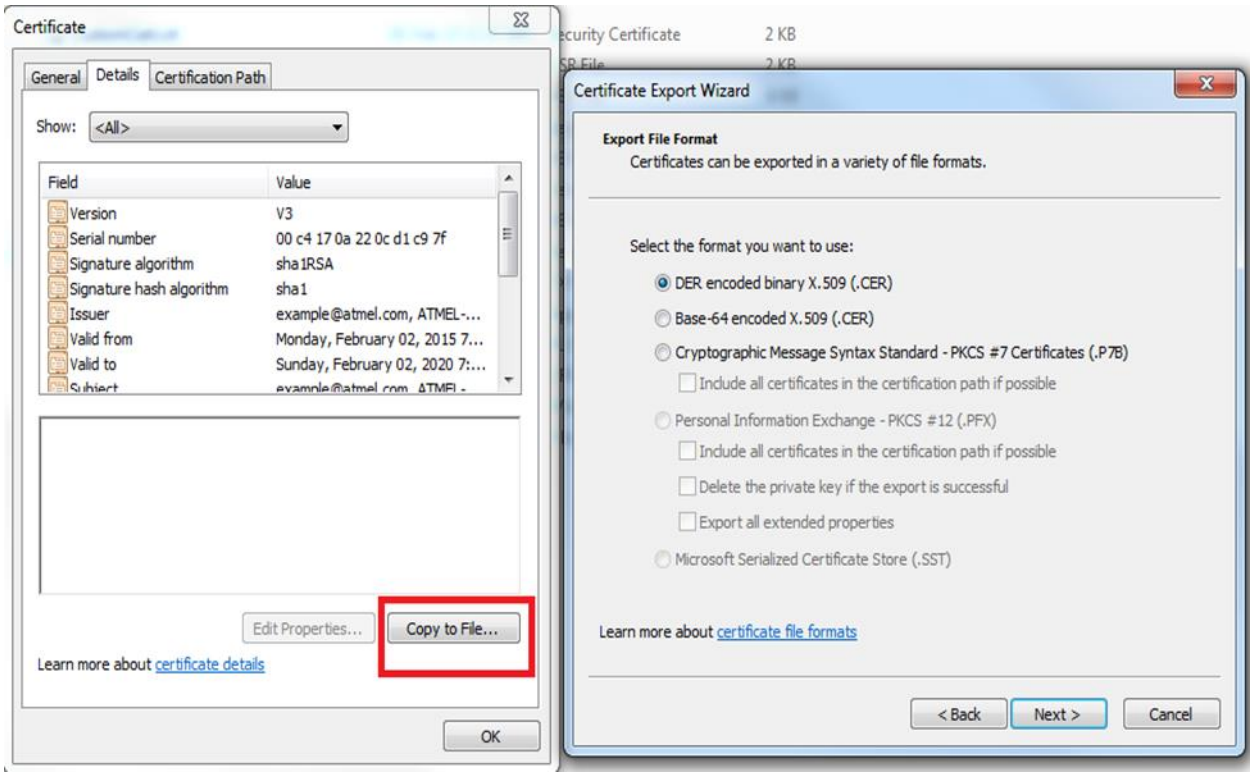
The PEM format is base64 encoding of the DER enclosed with between "-----BEGIN CERTIFICATE-----" and "-----END CERTIFICATE-----".

B.2 Conversion Between Different Formats

The current implementation of the ATWINC root_certificate_downloader supports only DER format. So, if the certificate is not in DER it must be converted to DER. This conversion can be done by several methods as described in the following sub-sections.

B.2.1 Using Windows

From Windows®, double click on the .pem certificate file and then go to Details Tab and press “Copy to File”. Follow the wizard until finish.



B.2.2 Using OpenSSL

The famous OpenSSL could be used for certificate conversion by the following command.

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

B.2.3 Online Conversion

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

Appendix C. How to Download the Certificate into the ATWINC

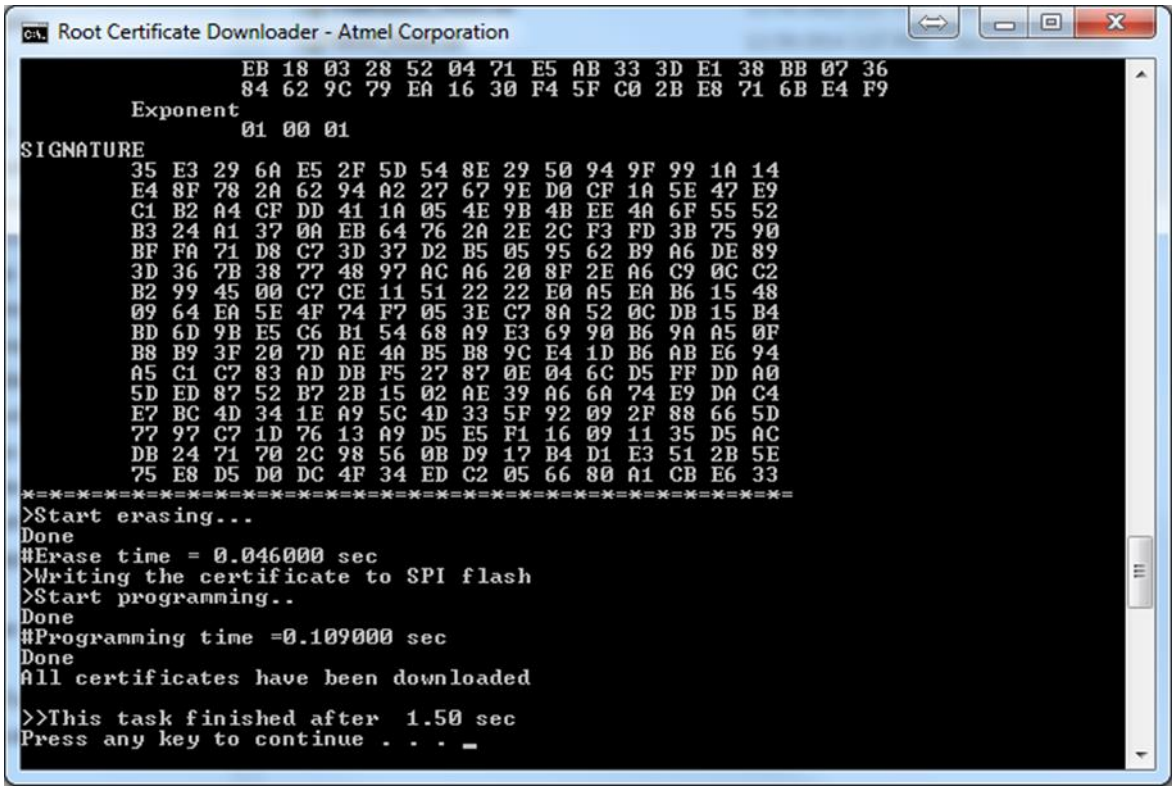
C.1 Overview

The ATWINC save the certificate inside the SPI flash in 4K sector (so the maximum size of all certificates in flash should be less than 4K).

C.2 Certificate Downloading

To download the root certificate Execute the batch file ***RootCertDownload.bat*** inside the release package:

- I²C Downloader
  /src/Tools/root_certificate_downloader/debug_I2C/RootCertDownload.bat
- UART Downloader
  /src/Tools/root_certificate_downloader/debug UART/RootCertDownload.bat



C.3 Adding New Certificate

- Open the file RootCertDownload.bat. There you find the following command:

```
root_certificate_downloader -n 2 NMA_Root.cer PROWL_Root.cer
```

- Update the batch for example to add NMI_root.cer (by update the n number of certificated and add the new certificate to the argument)

```
root_certificate_downloader -n 3 NMA_Root.cer PROWL_Root.cer NMI_root.cer
```

Appendix D. Firmware Image Downloader

The Firmware Downloader script use the EDBG SAMD21/W25 UART as the main interface.

1. Downloads the serial bridge application on the SAMD21/W25 using the Atmel atprogram.exe.
2. After downloading, the application will initialize the ATWINC in download mode and start listen on EDBG UART for UART commands.
3. The application will convert the UART commands to SPI commands.
4. The script will wait a couple of seconds until the application initialization finishes, then executes the firmware downloader, and the gain builder given the UART argument and the firmware/gain sheets path argument.

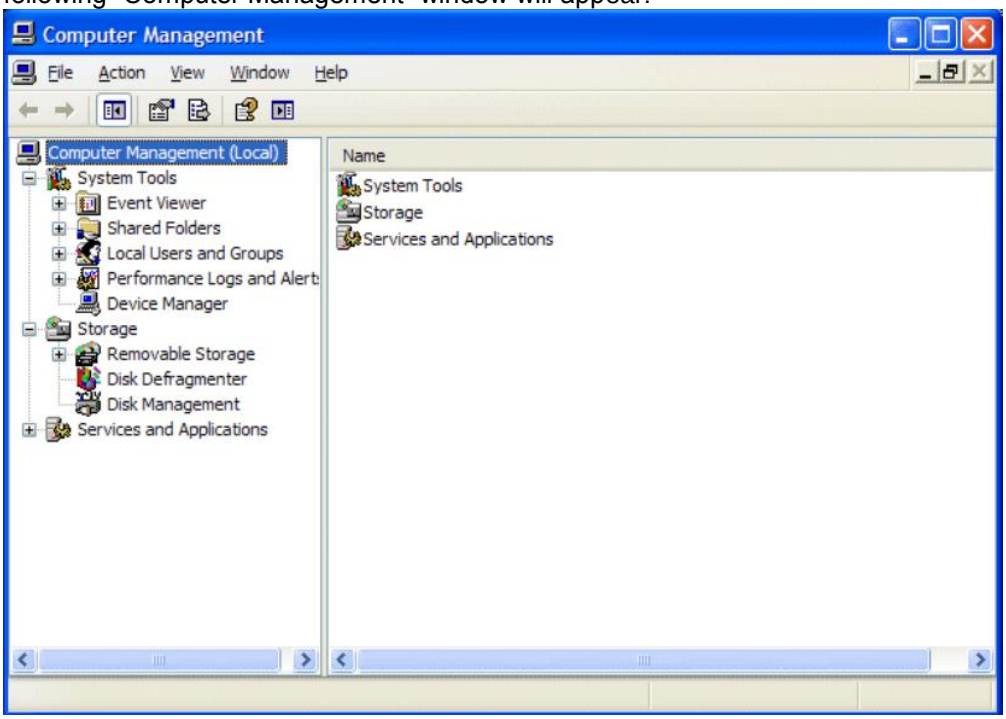
D.1 Preparing Environment

After connecting SAM D21 Debug USB and ATWINC Virtual COM Port USB port to computer, make sure that their drivers are already installed and correctly detected by Windows.

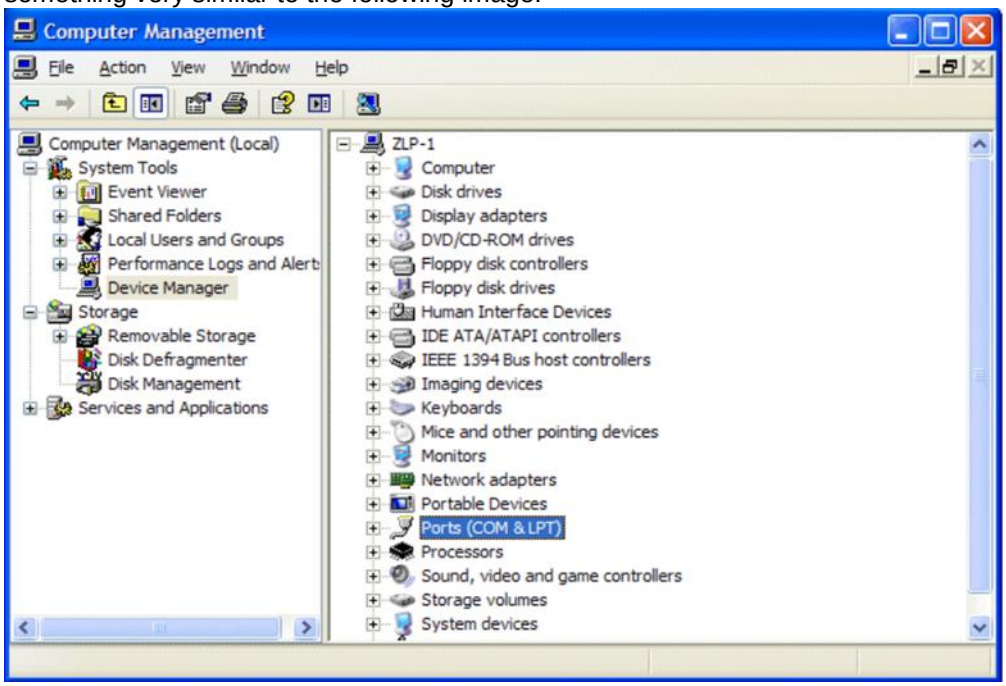
The ATWINC COM Port is configured (460800N-8-1) for Debug traces.

To check this, here are the steps for Windows XP and 7:

1. Right click on the icon “My Computer” and a menu will appear. Scroll down and select “Manage”. The following “Computer Management” window will appear:



2. From the list on the left hand side, select (click on) “Device Manager”. The “View” will change to something very similar to the following image:



3. In the Right Hand Panel, select (double click) Ports (COM and LPT). The “View” will change to something like the following (actual info shown depends on your PC).

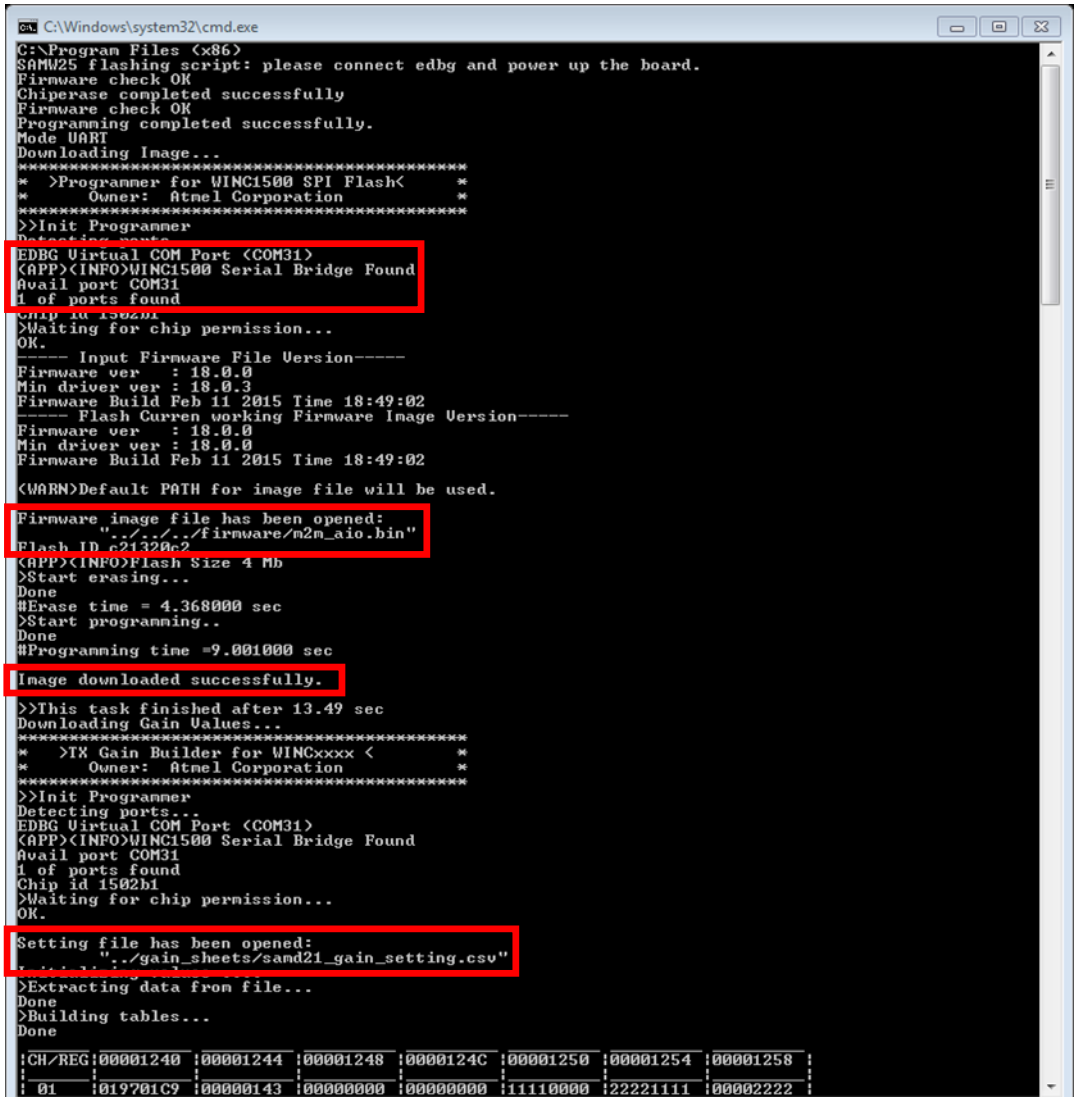
   In the below image, the boards are connected and using COM15, and COM26:

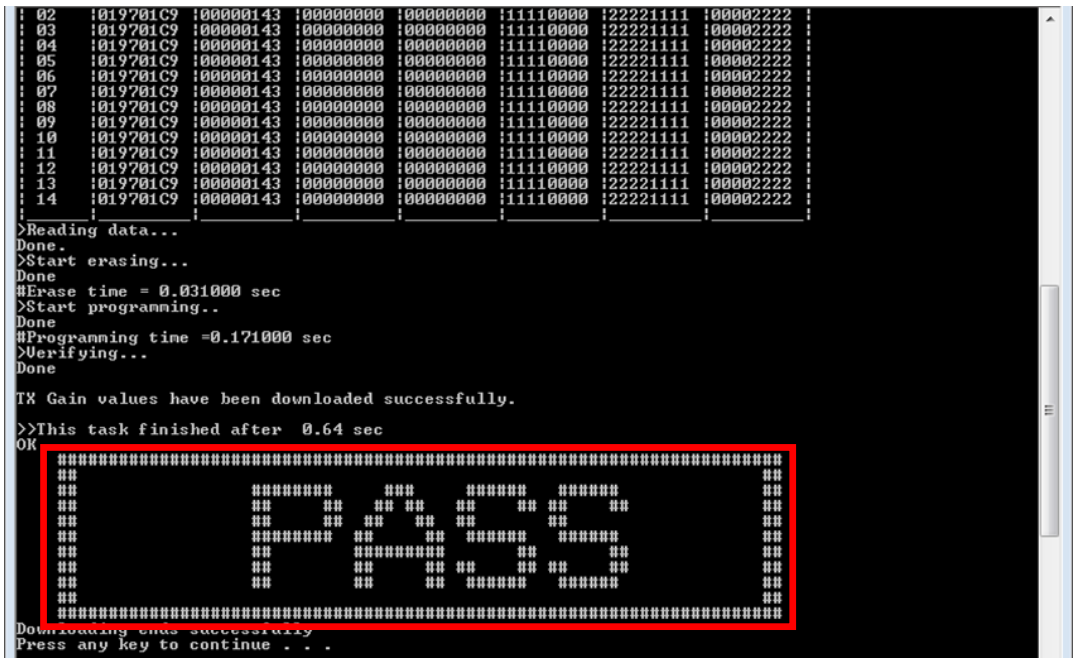
WARNING Make sure the EDBG port is not in use at Atmel Studio or with any other serial monitor before downloading. Also make sure that the firmware bin file is located at “./firmware/m2m_aio.bin” and gain files are located at “./Tools/gain_builder/gain_sheets/”.

TIPS The same thing is valid for SAMW25, by running “download_all_sb_samw25_xplained_pro.bat”.

D.2 Download Firmware

Run the "download_all_sb_samd21_xplained_pro.bat" script that is associated with the release to download firmware and gain settings for SAM D21.


C:\Windows\system32\cmd.exe
C:\Program Files (x86)
SAMW25 flashing script: please connect edbg and power up the board.
Firmware check OK
Chiperase completed successfully
Firmware check OK
Programming completed successfully.
Mode UART
Downloading Image...
>Programmer for WINC1500 SPI Flash<
Owner: Atmel Corporation
>>Init Programmer
EDBG Virtual COM Port (COM31)
(APP)(INFO)WINC1500 Serial Bridge Found
Avail port COM31
1 of ports found
>Waiting for chip permission...
OK.
----- Input Firmware File Version-----
Firmware ver : 18.0.0
Min driver ver : 18.0.3
Firmware Build Feb 11 2015 Time 18:49:02
----- Flash Curren working Firmware Image Version-----
Firmware ver : 18.0.0
Min driver ver : 18.0.0
Firmware Build Feb 11 2015 Time 18:49:02
(WARN)Default PATH for image file will be used.
Firmware image file has been opened:
"../../../firmware/m2m_aio.bin"
(APP)(INFO)Flash Size 4 Mb
>Start erasing...
Done
#Erase time = 4.368000 sec
>Start programming..
Done
#Programming time =9.001000 sec
Image downloaded successfully.
>>This task finished after 13.49 sec
Downloading Gain Values...
>TX Gain Builder for WINCxxxx <
Owner: Atmel Corporation
>>Init Programmer
Detecting ports...
EDBG Virtual COM Port (COM31)
(APP)(INFO)WINC1500 Serial Bridge Found
Avail port COM31
1 of ports found
Chip id 1502b1
>Waiting for chip permission...
OK.
Setting file has been opened:
"../gain_sheets/samd21_gain_setting.csv"
>Extracting data from file...
Done
>Building tables...
Done
CH/REG 00001240 00001244 00001248 0000124C 00001250 00001254 00001258
01 019701C9 00000143 00000000 00000000 11110000 22221111 00002222


02 019701C9 00000143 00000000 00000000 11110000 22221111 00002222
03 019701C9 00000143 00000000 00000000 11110000 22221111 00002222
04 019701C9 00000143 00000000 00000000 11110000 22221111 00002222
05 019701C9 00000143 00000000 00000000 11110000 22221111 00002222
06 019701C9 00000143 00000000 00000000 11110000 22221111 00002222
07 019701C9 00000143 00000000 00000000 11110000 22221111 00002222
08 019701C9 00000143 00000000 00000000 11110000 22221111 00002222
09 019701C9 00000143 00000000 00000000 11110000 22221111 00002222
10 019701C9 00000143 00000000 00000000 11110000 22221111 00002222
11 019701C9 00000143 00000000 00000000 11110000 22221111 00002222
12 019701C9 00000143 00000000 00000000 11110000 22221111 00002222
13 019701C9 00000143 00000000 00000000 11110000 22221111 00002222
14 019701C9 00000143 00000000 00000000 11110000 22221111 00002222
>Reading data...
Done.
>Start erasing...
Done
#Erase time = 0.031000 sec
>Start programming..
Done
#Programming time =0.171000 sec
>Verifying...
Done
TX Gain values have been downloaded successfully.
>>This task finished after 0.64 sec
OK
Press any key to continue . . .

Appendix E. Gain Settings Builder

E.1 Introduction

Gain setting values, are those values used by RF with different rates to configure transmission power.

This appendix helps to calculate these values and store them in Flash to use them otherwise default values will be used.

E.2 Preparing Environment

Make sure the environment is ready for building and downloading gain settings as in Appendix D: Firmware Image Downloader.

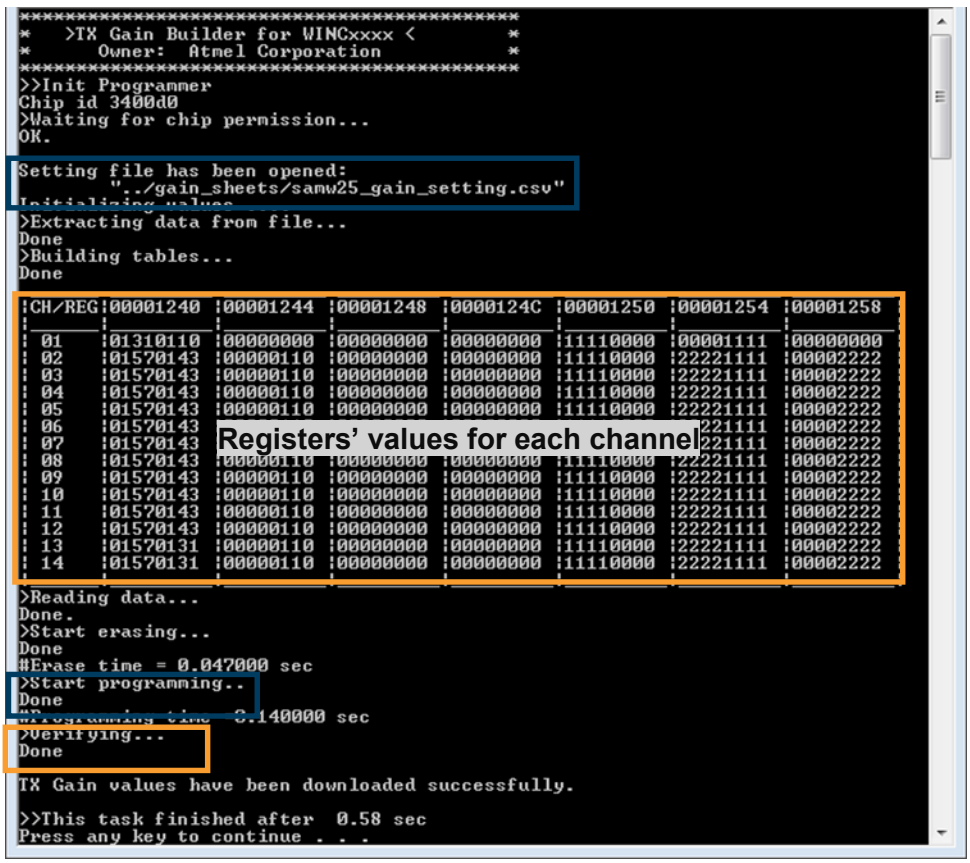
E.3 How to use

E.3.1 Method 1

- Replace the data of samd21_gain_setting.csv file in the project's folder with the new data where the file's location is the default (./Tools/gain_builder/gain_sheets/)
- Then run your application. It will calculate and store you data in Flash.

E.3.2 Method 2

- If you have a different file with data in a different path, then open "gain_build_and_download.bat" patch file and update it with the new path and file like:
  -fw_path ../gain_sheets/samd21_gain_setting.csv → -fw_path c:/gain_values.csv
- Then run your application by double click on modified patch file



WARNING

- The .csv file must be sorted based on gain rates like the templates
- There are two different buses to run your app I^2C or UART
- Your file must have 15 columns and 21 Rows for all channels such as the following template:

ch	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1														
2														
5.5														
11														
6														
9														
12														
18														
24														
36														
48														
54														
mcs0														
mcs1														
mcs2														
mcs3														
mcs4														
mcs5														
mcs6														
mcs7														

Insert your values here

Appendix F. Revision History

Doc Rev.	Date	Comments
42566A	04/2016	Initial document release.

Atmel | Enabling Unlimited Possibilities®

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