MICROCHIP

ATWINC15x0

ATWINC15x0 Wi-Fi®网络控制器软件设计指南

简介

Microchip 的 SmartConnect ATWINC15x0 是面向物联网（Internet of Things，IoT）应用的 IEEE® 802.11 b/g/n 网络控制器 SoC。它通过 SPI 转 Wi-Fi 接口实现 Wi-Fi 和网络功能，可与现有单片机（MCU）解决方案完美搭配。ATWINC15x0 仅需最少的资源便可连接到任何 Microchip AVR®或 Microchip SMART™ MCU。

特性

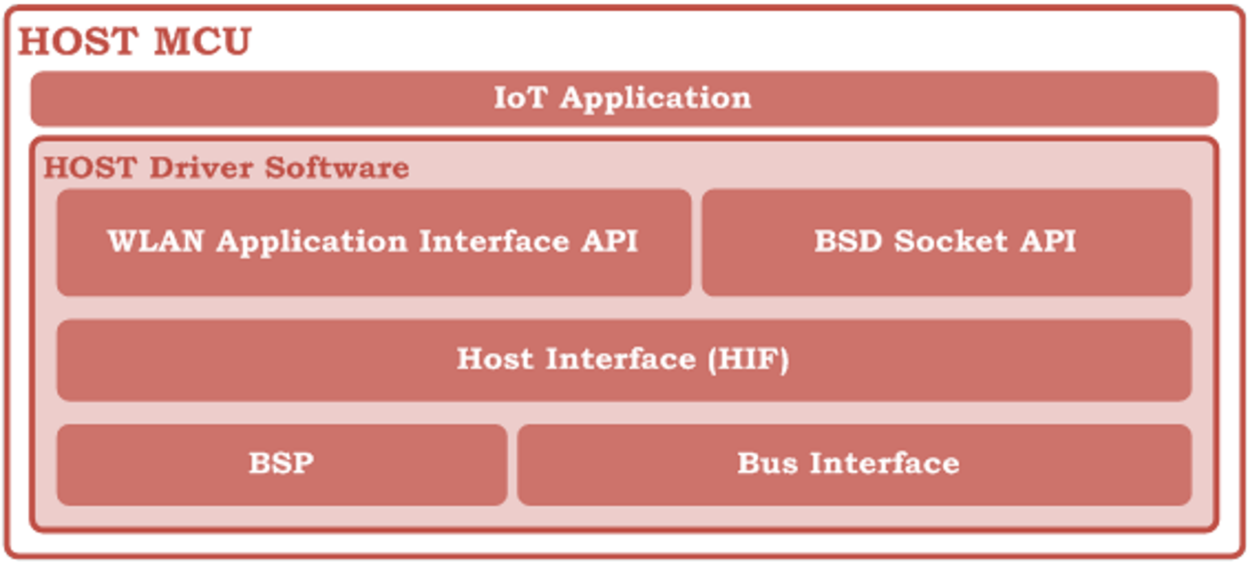
- Wi-Fi IEEE 802.11 b/g/n STA 和 AP 模式
- Wi-Fi 保护设置（Wi-Fi Protected Setup，WPS）
- 支持 WEP、WPA/WPA2 Personal 和 WPA/WPA2 Enterprise 安全功能
  - EAP-TLS
  - EAP-PEAPv0/TLS 和 EAP-PEAPv1/TLS
  - EAP-TTLSv0/MSCHAPv2
  - EAP-PEAPv0/MSCHAPv2 和 EAP-PEAPv1/MSCHAPv2
- 嵌入式网络协议栈协议，用于减轻 MCU 工作量（最大限度地减少对主机 CPU 的要求）。这使得 Wi-Fi 网络控制器（Wi-Fi Network Controller，WINC）可以与包括低端 MCU 在内的各种 MCU 搭配使用。
- 使用 BSD 样式套接字 API 的嵌入式 uIP TCP/IP 协议栈
- 嵌入式网络协议
  - DHCP 客户端/服务器
  - DNS 解析器客户端
  - 用于 UTC 时间同步的 SNTP 客户端
- 使用 BSD 样式套接字 API 抽象化的嵌入式 TLS 安全
- 用于在 AP 模式下配置服务的 HTTP 服务器
- 超低 C IEEE 802.11 b/g/n RF/PH/MAC SoC
- 从片上引导 ROM 快速启动
- 8 Mb（WINC1510）和 4 Mb（WINC1500）内部闪存，支持无线（Over-the-Air，OTA）固件升级
- WINC1510 支持主机文件下载功能，该功能可用于主机 MCU 的无线固件更新
- 不同节能模式实现低功耗
- 小尺寸主机驱动程序，具有以下功能：
  - 可以使用 SPI 接口在 8 位、16 位和 32 位 MCU 上运行
  - 支持小尾数法和大尾数法

目录

1. 主机驱动程序架构

下图显示了在主机 MCU 上运行的 WINC 主机驱动程序软件的架构。

图 1-1. 主机驱动程序软件架构



ATWINC15x0 主机驱动程序软件是一个 C 库，它为主机 MCU 应用程序提供必要的 API，以执行必要的 WLAN 和套接字操作。以下各小节将分别介绍主机驱动程序的各个组件。

1.1 WLAN API

该模块提供与应用程序的接口，以执行所有 Wi-Fi 操作和任何非 IP 相关操作。

其中包括以下服务：

- Wi-Fi STA 管理操作
  - Wi-Fi 扫描
  - Wi-Fi 连接管理（连接、断开连接和连接状态等）
  - WPS 激活/禁止
- Wi-Fi AP 使能/禁止
- Wi-Fi 节能控制 API

该接口在 **m2m_wifi.h** 文件中定义。

1.2 套接字 API

该模块提供的大部分套接字通信 API 都兼容常用的 BSD 套接字，因此可加快应用开发速度。为了符合 MCU 应用程序环境的性质，WINC 套接字与 BSD 套接字之间在 API 原型和某些 API 的使用方面存在差异。

该接口在 **socket.h** 文件中定义。

有关套接字操作的详细说明，请参见套接字编程。

1.3 主机接口（HIF）

主机接口（Host Interface，HIF）负责处理主机驱动程序与 WINC 固件之间的通信。其中包括中断处理、DMA 和 HIF 命令/响应管理。主机驱动程序以通过 HIF 层格式化的命令和响应形式与固件通信。

该接口在 **m2m_hif.h** 文件中定义。

有关 HIF 设计的详细说明，请参见主机接口协议。

1.4 板级支持包（BSP）

板级支持包（Board Support Package，BSP）负责抽象化特定主机 MCU 平台的功能，从而将驱动程序移植到各种硬件和主机上。抽象化包括：引脚分配、上电/掉电序列、复位序列和外设定义（按钮和 LED 等）。

BSP 功能的最低要求在 `nm_bsp.h` 文件中定义。

1.5 串行总线接口

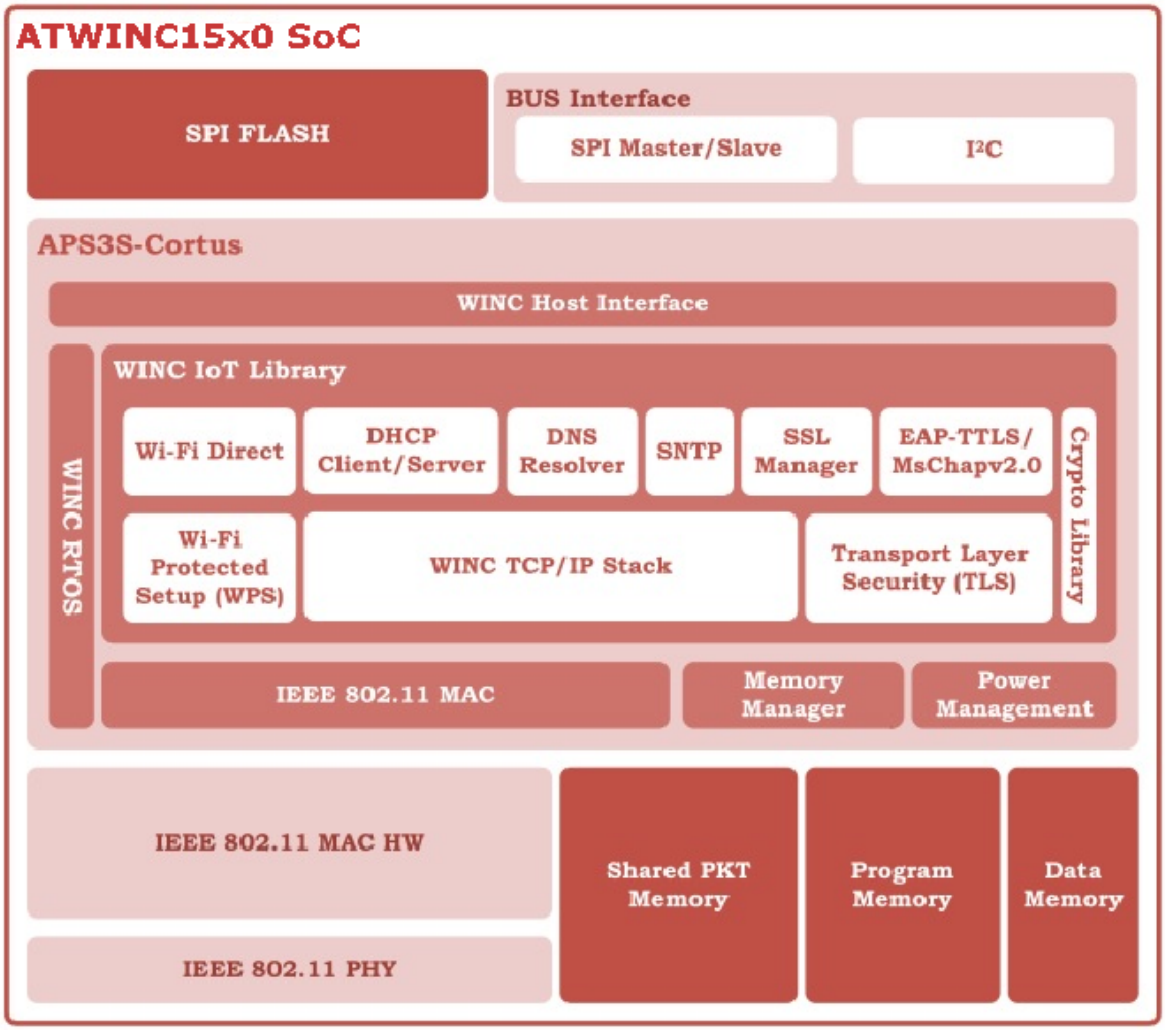
串行总线接口模块负责抽象化与在主机和 WINC 之间实现总线相关的硬件。串行总线接口可抽象化 I2C、SPI 或 UART 总线（目前，主机驱动程序仅支持 SPI 总线接口）。该模块中实现了基本的总线访问操作（读和写），以满足接口类型和特定硬件的要求。

总线接口 API 在 `nm_bus_wrapper.h` 文件中定义。

2. ATWINC15x0 系统架构

下图所示为 ATWINC15x0 系统架构。除了内置的 Wi-Fi IEEE-802.11 物理层和 RF 前端外，WINC ASIC 还包含一个嵌入式 APS3S-Cortus 32 位 CPU，用于运行 WINC 固件。固件包括 Wi-Fi IEEE-802.11 MAC 层和嵌入式协议栈，用于减轻主机 MCU 的负荷。以下各小节将分别介绍系统的各个组件。

图 2-1. ATWINC15x0 系统架构



2.1 总线接口

ATWINC15x0 通信支持的总线类型的硬件逻辑。

注： 主机驱动程序目前支持的总线接口为 SPI。

2.2 非易失性存储器

ATWINC1510 和 ATWINC1500 的 WINC 封装（SIP）内分别有一个 8 Mb 和 4 Mb 的集成串行闪存。其中存有 WINC 固件镜像，并且可存储第二个镜像以支持 OTA。此外，还存储了供 WINC 固件在运行时使用的信息。

有关串行闪存的详细说明，请参见 WINC 串行闪存。

2.3 CPU

该 SoC 包含一个以 40 MHz 时钟速度运行的 APS3S-Cortus 32 位 CPU，该 CPU 可执行嵌入式 WINC 固件。

2.4 IEEE 802.11 MAC 硬件

该 SoC 包含硬件加速器，可确保快速实现符合要求的 IEEE 802.11 MAC 层和相关时序。这样便无需通过固件执行 IEEE 802.11 MAC 功能，从而改善性能并提高 MAC 吞吐量。该加速器包括针对 Wi-Fi 流量的硬件加密/解密和流量过滤机制，可避免在软件中进行不必要的处理。

2.5 程序存储器

该系统提供 128 KB 指令 RAM，用于执行 ATWINC15x0 固件代码。

2.6 数据存储器

该系统提供 64 KB RAM，用于存储 ATWINC15x0 固件数据。

2.7 共用数据包存储器

该系统提供 128 KB 存储器，用于管理 TX/RX 数据包。该存储器由 MAC 硬件与 CPU 共用，并由存储器管理器软件组件管理。

2.8 IEEE 802.11 MAC 固件

该系统支持 IEEE 802.11 b/g/n Wi-Fi MAC，包括 WEP 和 WPA/WPA2 安全客户端。该系统在 MAC 硬件和固件之间实现并支持完整的 IEEE 802.11 功能，其中包括信标生成与接收、控制数据包生成与接收，以及数据包聚合与解聚合。

2.9 存储器管理器

存储器管理器负责在共用数据包存储器和数据存储器中分配和释放存储器块。

2.10 电源管理

电源管理模块负责处理 WINC 支持的不同节能模式，并使用 Wi-Fi 收发器协调这些模式。

2.11 WINC RTOS

该固件包含一段较小的实时调度程序，允许在 ATWINC15x0 CPU 上实现多任务并发。ATWINC15x0 RTOS 提供信号量和定时器功能。

2.12 WINC IoT 库

WINC IoT 库在 WINC 固件中提供一组网络协议。它从网络和传输层协议减轻主机 MCU 的工作量。以下各节将分别介绍 WINC IoT 库的各个组件。

2.12.1 WINC TCP/IP 协议栈

WINC TCP/IP 是一种基于 uIP（读作微 IP）TCP/IP 协议栈的 IPv4.0 协议栈。

uIP 是一种小型 TCP/IP 协议栈，能够在存储空间有限的单片机平台上运行。该协议栈最初由 Adam Dunkels 开发，并获得 BSD 样式许可证，而后由大批开发人员进一步不断完善开发而成。WINC TCP/IP 协议栈是原始 uIP 协议栈的定制版本，通过多项增强功能实现了 TCP 和 UDP 吞吐量的双重提升。

2.12.2 DHCP 客户端/服务器

DHCP 客户端嵌入在 WINC 固件中，可以在连接到 Wi-Fi 网络后自动获取 IP 配置。

WINC 固件提供 DHCP 服务器的实例，该实例在使能 WINC AP 模式时自动启动。当主机 MCU 应用程序激活 AP 模式时，可在 AP 配置参数中配置 DHCP 服务器 IP 地址池范围。

2.12.3 DNS 解析器

WINC 固件包含一个嵌入式 DNS 解析器实例。该模块可以通过解析随套接字 API 调用 gethostbyname 提供的主机域名来返回 IP 地址。

2.12.4 SNTP

简单网络时间协议（Simple Network Time Protocol，SNTP）模块实现用于将 WINC 内部时钟与 UTC 时钟进行同步的 SNTP 客户端。

2.12.5 Enterprise 安全模式

Enterprise 安全模块实现以下身份验证协议，以通过 WPA/WPA2-Enterprise 安全模式与 AP 建立 Wi-Fi 连接。

- EAP/TLS
- EAP-PEAPv0/MSCHAPV2 和 EAP-PEAPv1/MSCHAPV2
- EAP-TTLSv0/MSCHAPv2
- EAP-PEAPv0/MSCHAPv2 和 EAP-PEAPv1/MSCHAPv2

2.12.6 传输层安全

有关 TLS 实现的详细信息，请参见**第 7 章“传输层安全（TLS）”**。

2.12.7 WI-FI 保护设置

有关 WPS 协议实现的详细信息，请参见**第 10.3 节“Wi-Fi 保护设置（WPS）”**。

2.12.8 加密库

加密库包含一组公共安全协议使用的加密算法。该库实现了以下算法：

- MD4 哈希算法（仅用于 MsChapv2.0 摘要计算）
- MD5 哈希算法
- SHA-1 哈希算法
- SHA-256 哈希算法
- DES 加密（仅用于 MsChapv2.0 摘要计算）
- MS-CHAPv2.0（用作 EAP-TTLS 内部身份验证算法）
- MS-CHAPv2.0（用作 EAP-PEAP 和 EAP-TTLS 内部身份验证算法）
- AES-128 和 AES-256 加密（用于保护 WPS 和 TLS 流量）
- 用于大型整数算术的 BigInt 模块（用于公钥加密计算）
- RSA 公钥加密算法（包括 RSA 签名和 RSA 加密算法）

3. WINC 初始化和简单应用程序

WINC 器件上电后，必须执行一组同步初始化序列才能使 Wi-Fi 正常工作。本章旨在解释系统初始化阶段所需的不同步骤。初始化之后，主机 MCU 应用程序需要调用 WINC 驱动程序入口点来处理来自 WINC 固件的事件。

- BSP 初始化
- WINC 主机驱动程序初始化
- 套接字层初始化
- 调用 WINC 驱动程序入口点

注： 必须完成初始化序列才能成功运行 WINC 启动程序。

3.1 BSP 初始化

BSP 通过调用 `nm_bsp_init` API 来初始化。BSP 初始化程序执行以下步骤：

- 使用相应的主机 MCU 控制 GPIO 复位 WINC[1]
- 初始化连接到 WINC 中断线路的主机 MCU GPIO，将 GPIO 配置为主机 MCU 的中断源。在运行时，WINC 会中断主机以通知应用程序在 WINC 固件中有待处理的事件和数据。
- 初始化 `nm_bsp_sleep` 实现内使用的主机 MCU 延时函数。

3.2 WINC 主机驱动程序初始化

WINC 主机驱动程序通过调用 `m2m_wifi_init` API 来初始化。主机驱动程序初始化程序执行以下步骤：

- 初始化总线封装和 SPI 外设。必须在 `conf_winc.h` 中使能编译标志 `CONF_WINC_USE_SPI`（目前不支持总线接口 `CONF_WINC_USE_UART` 和 `CONF_WINC_USE_I2C`）。
- 注册应用程序定义的 Wi-Fi 事件处理程序。
- 初始化驱动程序并确保 WINC 固件版本和驱动程序版本之间的兼容性。
- 初始化主机接口和 Wi-Fi 层并注册 BSP 中断。

注： 为了确保所有 WINC 应用程序都能够正常工作，需要使用 Wi-Fi 事件处理程序。

3.3 套接字层初始化

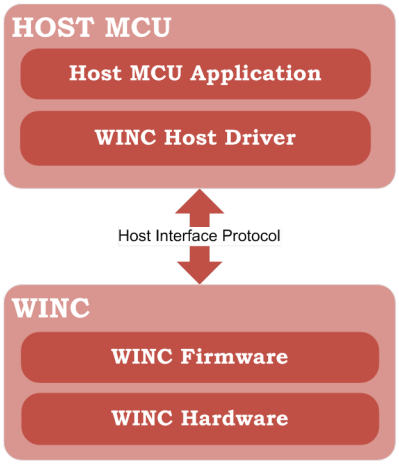
套接字层初始化通过调用 `socketInit` API 来执行，并且必须在任何套接字活动之前调用。有关套接字初始化和编程的更多信息，请参见 WINC 套接字 API。

3.4 WINC 事件处理

WINC 主机驱动程序 API 允许主机 MCU 应用程序与 WINC 固件交互。为了便于交互，WINC 驱动程序实现了主机接口（HIF）协议，如**第 15 章“主机接口（HIF）协议”**中所述。HIF 协议定义了如何通过串行总线接口 SPI（目前不支持 I2C 和 UART）对 API 请求和响应回调函数进行序列化和反序列化处理。

[1] 有关硬件上电/掉电序列的更多信息，请参见 ATWINC15x0-MR210xB Data Sheet（DS70005304）。

图 3-1. WINC 系统架构



WINC 主机驱动程序 API 为主机 MCU 应用程序提供服务，主要分为两大类：Wi-Fi 控制服务和套接字服务。Wi-Fi 控制服务支持通道扫描、网络标识、连接和断开连接等操作。套接字控制服务支持在建立 Wi-Fi 连接后传输应用程序数据。

3.4.1 异步事件

ATWINC15x0 主机驱动程序中的某些 API 为同步函数调用，通过返回函数使其结果就绪。不过，ATWINC15x0 主机驱动程序中的大多数 API 函数都是异步的。这意味着，当应用程序调用 API 来请求服务时，该调用是非阻塞的，会在请求的操作完成之前立即返回。完成后，会以 HIF 协议报文的形式从 WINC 固件向主机提供通知，然后通过回调函数将其传递给应用程序。[2] 当所请求的服务（例如 Wi-Fi 连接）可能需要很长时间才能完成时，异步操作必不可少。通常，ATWINC15x0 固件使用异步事件向主机驱动程序发送有关状态更改或待处理数据的信号。

HIF 使用推送架构，其以先到先服务（First-Come First-Served，FCFS）的方式将数据和事件从 ATWINC15x0 固件推送至主机 MCU。例如，主机 MCU 应用程序有两个打开的套接字：套接字 1 和套接字 2。如果 ATWINC15x0 接收到套接字 1 数据，然后接收到套接字 2 数据，则 HIF 按照接收顺序在两个 HIF 协议报文中传送套接字数据。HIF 不允许先读取套接字 2 数据，然后再读取套接字 1 数据。

3.4.2 中断处理

当 ATWINC15x0 固件中有一个或多个事件处于待处理状态时，HIF 会中断主机 MCU。主机 MCU 应用程序是一个大型状态机，可在 ATWINC15x0 驱动程序调用事件回调函数时，处理接收到的数据和事件。要接收事件回调函数，主机 MCU 应用程序需要调用 `m2m_wifi_handle_events` API 以允许主机驱动程序检索和处理来自 ATWINC15x0 固件的待处理事件。如果发生以下任意事件，建议调用此函数：

- 主机 MCU 应用程序在主循环或专用任务中轮询 API
- 主机 MCU 接收到来自 ATWINC15x0 固件的中断时

注： 通过 ATWINC15x0 驱动程序注册的所有应用程序定义的事件回调函数均在上下文 `m2m_wifi_handle_events` API 中运行。

上述 HIF 架构允许 ATWINC15x0 主机驱动程序灵活地在以下配置中运行：

- 无操作系统配置的主机 MCU——MCU 主循环负责处理来自中断处理程序的延迟工作

[2] 回调函数是 C 函数，其中包含应用程序定义的逻辑。回调函数使用 ATWINC15x0 主机驱动程序注册 API 进行注册，用于处理所请求服务的结果。

- 有操作系统配置的主机 MCU——需要专用任务或线程来调用 m2m_wifi_handle_events，以处理来自中断处理程序的延迟工作

注：

1. 主机驱动程序入口点 m2m_wifi_handle_events **不可重入**。在操作系统配置中，需要保护主机驱动程序不被同步对象重入。
2. 主机 MCU 轮询 m2m_wifi_handle_events 时，API 会检查来自 ATWINC15x0 的待处理中断。如果没有中断处于待处理状态，则立即返回。如果有中断处于待处理状态，m2m_wifi_handle_events 会按顺序读取所有待处理的 HIF 报文，并将 HIF 报文内容分配给相应的已注册回调函数。如果未注册回调函数来处理报文类型，则会丢弃 HIF 报文内容。

3.5 示例代码

以下示例代码说明了前面几节所述的初始化流程。

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

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

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

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

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

4. ATWINC15x0 配置

ATWINC15x0 固件提供了一组可配置的参数来控制其行为。通过为主机 MCU 应用程序提供的一组 API，可以配置这些参数。配置 API 按照其功能分为设备、网络和节能参数。

任何未经主机 MCU 应用程序设置的参数均使用其在 ATWINC15x0 固件初始化期间分配的默认值。主机 MCU 应用程序需要在退出冷启动时或需要更改特定配置时配置其参数。

4.1 器件参数

4.1.1 系统时间

为确保对 X509 证书到期日期进行适当的有效性检查，将 WINC 系统设置为 UTC 时间至关重要。由于 WINC 不包含内置实时时钟（Real-Time Clock，RTC），因此可通过两种方法获得 UTC 时间：

- 使用内部 SNTP 客户端——启动时默认在 WINC 固件中启用。SNTP 客户端将 WINC 系统时钟与时间服务器的 UTC 时间同步。可使用 API `m2m_wifi_configure_sntp` 配置 SNTP 客户端所用的 NTP 服务器。WINC 使用的默认 NTP 服务器为 `time-c.nist.gov`。SNTP 客户端使用一天一次的默认更新周期。
- 通过主机 MCU RTC——如果主机 MCU 具有 RTC，应用程序可在 WINC 初始化后通过调用 `m2m_wifi_enable_sntp(0)`（传递 0 作为参数）禁止 SNTP 客户端。应用程序通过调用 `m2m_wifi_set_system_time` API 提供 WINC 系统时间。

4.1.2 固件和驱动程序版本

初始化（`m2m_wifi_init`）期间，主机驱动程序会检查驱动程序和 WINC 固件之间的兼容性。相关参数如下：

- `M2M_HIF_MAJOR_VALUE`
- `M2M_HIF_MINOR_VALUE`

注： 这些参数在发行说明版本信息中声明为“主机接口级别：X.Y”。

如果驱动程序和 WINC 固件具有相同的 `M2M_HIF_MAJOR_VALUE` 值，则认为二者兼容，`m2m_wifi_init` 会返回 `M2M_SUCCESS`。

如果驱动程序和 WINC 固件具有不同的 `M2M_HIF_MAJOR_VALUE` 值，则认为二者不兼容，`m2m_wifi_init` 会返回 `M2M_ERR_FW_VER_MISMATCH`。在这种情况下，通信受限；惟一允许的通信是驱动程序请求 WINC 固件切换到 WINC 闪存非活动分区中的 WINC 固件镜像（通过 `m2m_wifi_check_ota_rb` 和 `m2m_ota_switch_firmware`）。

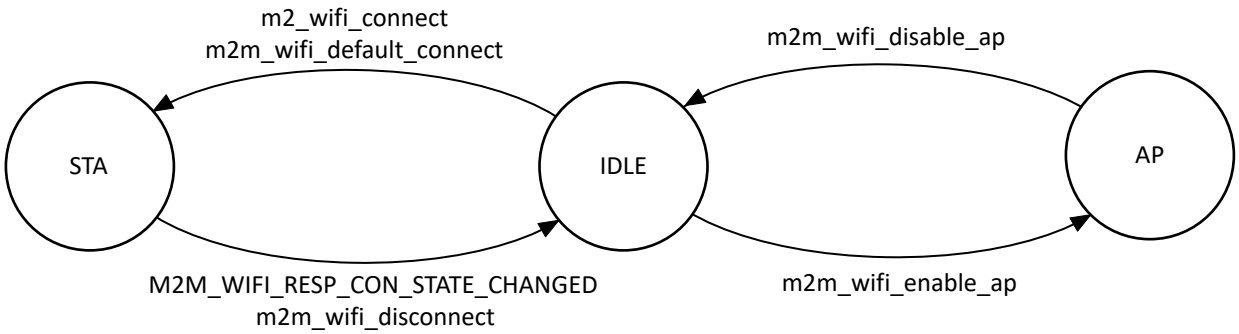
驱动程序文件 `m2m_ota.h` 中提供了处理此情况的示例代码。

4.2 WINC 工作模式

WINC 固件支持以下工作模式：

- 空闲模式
- Wi-Fi 站模式
- Wi-Fi 热点（AP）

图 4-1. WINC 工作模式



4.2.1 空闲模式

主机 MCU 应用程序调用 ATWINC15x0 驱动程序初始化 `m2m_wifi_init` API 后，ATWINC15x0 保持在空闲模式，等待命令更改模式或更新配置参数。在此模式下，ATWINC15x0 进入节能模式，这会禁止 IEEE 802.11 无线电和所有不需要的外设，并会暂停 ATWINC15x0 CPU。如果 ATWINC15x0 从主机 MCU 接收到任何配置命令，则会更新配置、将响应发送回主机 MCU，然后返回到节能模式。

4.2.2 Wi-Fi 站模式

主机 MCU 使用 `m2m_wifi_connect` 或 `m2m_wifi_default_connect` API 请求连接到 AP 时，ATWINC15x0 进入站（STA）模式。

注： 自 v19.6.1 起，弃用 `m2m_wifi_connect`。更多详细信息，请参见 5.3 Wi-Fi 安全。

ATWINC15x0 在接收到断开连接请求（通过事件回调函数 `M2M_WIFI_RESP_CON_STATE_CHANGED` 从 Wi-Fi AP 传递到主机 MCU 应用程序），或主机 MCU 应用程序决定通过 `m2m_wifi_disconnect` API 终止连接时，会退出 STA 模式。

注： 此模式下支持的 API 函数使用 HIF 命令类型：`tenuM2mConfigCmd` 和 `tenuM2mStaCmd`。请参见 `m2m_types.h` 头文件中的完整命令列表。

有关 STA 模式的更多信息，请参见 Wi-Fi 站模式。

4.2.3 Wi-Fi 热点（AP）模式

在 AP 模式下，WINC 允许 Wi-Fi 站连接并从 WINC DHCP 服务器获取 IP 地址。要进入 AP 模式，主机 MCU 应用程序会调用 `m2m_wifi_enable_ap` API。要退出 AP 模式，应用程序会调用 `m2m_wifi_disable_ap` API。

此模式下支持的 API 函数使用 HIF 命令类型：`tenuM2mApCmd` 和 `tenuM2mConfigCmd`。请参见 `m2m_types.h` 头文件中的完整命令列表。

有关此模式的更多信息，请参见 Wi-Fi AP 模式。

4.3 网络参数

4.3.1 Wi-Fi MAC 地址

WINC 固件提供两种方法来分配 WINC MAC 地址：

- 通过主机 MCU 分配——当主机 MCU 应用程序在使用 `m2m_wifi_init` API 进行完初始化后调用 `m2m_wifi_set_mac_address` API 时，会使用此方法。

- 通过 WINC 可一次性编程（One-Time-Programmable，OTP）存储器分配——WINC 支持通过内置 OTP 存储器实现内部 MAC 地址分配方法。如果在 WINC OTP 存储器中编程 MAC 地址，除非主机 MCU 应用程序在初始化后使用 API `m2m_wifi_set_mac_address` 以编程方式设置其他 MAC 地址，否则工作 WINC MAC 地址默认为 OTP MAC 地址。

注：

- OTP MAC 地址在生产期间编程到 WINC OTP 存储器中。
- 使用 `m2m_wifi_get_otp_mac_address` API 检查 WINC OTP 存储器中是否存在有效的已编程 MAC 地址。主机 MCU 应用程序也可使用相同的 API 来读取 OTP MAC 地址八位字节。不要将 `m2m_wifi_get_otp_mac_address` API 与 `m2m_wifi_get_mac_address` API 混淆，无论是通过主机 MCU 分配还是通过 WINC OTP 分配，后者始终读取 WINC 固件中的有效 WINC MAC 地址。
- 有关 API 的更多详细信息，请参见 Atmel Software Framework for ATWINC1500 (Wi-Fi)。

4.3.2 IP 地址

在成功连接 Wi-Fi 后，ATWINC15x0 固件使用嵌入式 DHCP 客户端自动获取 IP 配置。DHCP 是首选方法，因此将其用作默认方法。获得 IP 配置后，会通过异步事件 `M2M_WIFI_REQ_DHCP_CONF` 通知主机 MCU 应用程序。

此外，主机 MCU 应用程序可通过调用 `m2m_wifi_set_static_ip` API 来设置静态 IP 配置。在设置静态 IP 地址之前，建议使用 API `m2m_wifi_enable_dhcp(0)` 禁止 DHCP，然后再设置静态 IP，如下所示。

```
In Main(), disable dhcp after m2m_wifi_init as shown below
/* Initialize Wi-Fi driver with data and status callbacks.*/
param.pfAppWifiCb = wifi_cb;
ret = m2m_wifi_init(&param);
if (M2M_SUCCESS != ret)
{
    printf("main: m2m_wifi_init call error!(%d)\r\n", ret);
    while (1)
    {}
}
m2m_wifi_enable_dhcp(0);

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

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

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

4.4 节能模式

WINC 固件支持多种节能模式，可使主机 MCU 应用程序灵活调整系统功耗。主机 MCU 可使用 m2m_wifi_set_sleep_mode 和 m2m_wifi_set_lsn_int API 配置 WINC 节能策略。

WINC 支持以下节能模式：

- M2M_PS_MANUAL
- M2M_PS_DEEP_AUTOMATIC
- M2M_PS_AUTOMATIC（已弃用，在新实现方案中不使用）
- M2M_PS_H_AUTOMATIC（已弃用，在新实现方案中不使用）

注： 对于大多数应用程序，推荐使用 M2M_PS_DEEP_AUTOMATIC 模式。

4.4.1 M2M_PS_MANUAL

这是一种完全由主机驱动的节能模式。

- 主机使用 m2m_wifi_request_sleep API 时，WINC 休眠。在此期间，主机 MCU 也可以延长休眠的时间。
- 当主机 MCU 应用程序通过调用主机驱动程序 API 函数（例如，Wi-Fi 或套接字操作）从 WINC 请求服务时，WINC 将唤醒。

注： 在 M2M_PS_MANUAL 模式下，当 WINC 因 m2m_wifi_request_sleep API 而休眠时，WINC 不会唤醒来接收和监视 AP 信标。当主机 MCU 应用程序唤醒 WINC 时，信标监视将恢复。

对于活动的 Wi-Fi 连接，如果 WINC 因休眠时间较长而导致不可用，则 AP 可能退出连接。如果连接断开，WINC 将在下一个唤醒周期检测到断开连接，并通知主机再次重新连接到 AP。为长时间保持 Wi-Fi 连接有效，主机 MCU 应用程序必须定期唤醒 WINC，以向 AP 发送保持活动 Wi-Fi 帧。主机必须谨慎选择休眠周期，以在 Wi-Fi 连接不中断和最小化系统功耗之间保持平衡。

此模式对于因某个触发信号而导致很少发送通知的应用非常有用。它还适用于定期发送通知（通知之间间隔很长）的应用。使用此模式时需要谨慎进行功耗规划。如果主机 MCU 决定长时间休眠，则可以使用 M2M_PS_MANUAL 或关断 WINC。[3] 与关断 WINC 相比，此模式的优势在于，M2M_PS_MANUAL 节省了 WINC 固件引导所需的时间，因为固件始终在 WINC 存储器中。具体优缺点取决于应用的性质。在某些应用中，休眠时间非常长，以至于触发节能策略，因此会关断 WINC，然后再次将其开启并在主机 MCU 唤醒时重新连接到 AP。在其他情况下，对延时敏感的应用可能会选择使用 M2M_PS_MANUAL 避免 WINC 固件引导延时，但这会导致功耗略微增加。

在 WINC 休眠模式期间，与 M2M_PS_DEEP_AUTOMATIC 模式相比，WINC 在 M2M_PS_MANUAL 模式下更加节能。在 M2M_PS_MANUAL 模式下，WINC 会跳过信标监视，而在 M2M_PS_DEEP_AUTOMATIC 模式下，则会唤醒以接收信标。此外还比较了对 MCU 休眠持续时间的影响：如果主机 MCU 休眠时间较长，则 Wi-Fi 连接可能会频繁断开，由于 Wi-Fi 重新连接时也会增加功耗，因此 M2M_PS_MANUAL 不再具有功耗优势。相比之下，M2M_PS_DEEP_AUTOMATIC 模式可长时间保持 Wi-Fi 连接，但需要唤醒 WINC 以监视 AP 信标。

4.4.2 M2M_PS_AUTOMATIC

出于向后兼容性和开发的原因，不推荐使用此模式。不建议用于新实现方案。

[3] 有关硬件上电/掉电序列的更多信息，请参见 ATWINC15x0-MR210xB Data Sheet（DS70005304）。

4.4.3 M2M_PS_H_AUTOMATIC

出于向后兼容性和开发的原因，不推荐使用此模式。不建议用于新实现方案。

4.4.4 M2M_PS_DEEP_AUTOMATIC

该模式在 WINC 模块下实现了 Wi-Fi 标准节能方法。WINC 会休眠并定期唤醒以监视 AP 信标。当站进入节能模式和唤醒时，需要 AP 来缓冲和发送数据。AP 定期发送信标帧以针对每个信标周期与网络同步。处于节能模式的站会定期唤醒以接收信标。信标向站传送关于待处理单播数据的信息，这些数据在站处于休眠模式时在 AP 内部缓冲。信标还提供有关广播/多播数据的信息。

在此模式下，WINC 模块通过关闭 IEEE 802.11 无线电、MAC 和系统时钟进入休眠状态。在进入休眠模式之前，ATWINC15x0 会编程硬件定时器（在内部低功耗振荡器上运行），其休眠周期由 WINC 固件功耗管理模块决定。

以下任意事件都可以将 WINC 模块从休眠状态唤醒：

- 硬件休眠定时器到期。WINC 唤醒以接收即将来自 AP 的信标。
- WINC 在以下情况下唤醒：[4] 主机 MCU 应用程序通过调用主机驱动程序 API 函数（例如，Wi-Fi 或套接字操作）从 WINC 请求服务。

4.5 配置侦听间隔和 DTIM 监视

WINC 允许主机 MCU 应用程序通过配置信标监视参数来调整系统功耗。在每个 *DTIM 周期*（例如 100 ms），AP 会定期发送信标。信标包含一个 *TIM 元素*，此元素向站通知 AP 中缓冲的站的单播数据。站在*侦听间隔*内与 AP 协商。侦听间隔会通知 AP 站在唤醒以接收 AP 中缓冲的数据之前将休眠的信标周期数。如果站未检索到数据，则在侦听间隔后，某些 AP 可能会丢弃缓冲的数据。

WINC 驱动程序允许主机 MCU 应用程序配置信标监视参数，如下所述：

- 配置 DTIM 监视——即使用以下 API 启用或禁止广播/多播数据的接收：
  - `m2m_wifi_set_sleep_mode(desired_mode, 1)`（接收广播数据）
  - `m2m_wifi_set_sleep_mode(desired_mode, 0)`（忽略广播数据）
- 配置侦听间隔——使用 `m2m_wifi_set_lsn_int` API

注： 为 `m2m_wifi_set_lsn_int` API 提供的侦听间隔值以信标周期为单位表示。

[4] 唤醒序列由 `hif_chip_wake` API 在 WINC 主机驱动程序内部进行处理。有关更多信息，请参见**第 15 节“主机接口协议”**。

5. Wi-Fi 站模式

本章提供有关 WINC Wi-Fi 站（STA）模式的信息，如 Wi-Fi 站模式中所述。STA 模式涉及扫描操作；使用主机 MCU 提供的参数（SSID 和凭证）或使用存储在 WINC 非易失性存储器中的 AP 参数（默认连接）与 AP 关联。本章还提供了有关支持的安全模式信息以及代码示例。

5.1 扫描配置参数

5.1.1 扫描区域

支持的 RF 通道数因地域而异。例如，亚洲支持 13 个通道，而北美支持 11 个通道。默认情况下，WINC 初始区域配置支持 14 个通道，但可以通过使用 `m2m_wifi_set_scan_region` API 设置扫描区域来更改。可以从 `enum tenuM2mScanRegion` 选择扫描区域。

5.1.2 扫描选项

在 Wi-Fi 扫描操作期间，WINC 发送探测请求 Wi-Fi 帧，并等待扫描等待时间以接收当前 Wi-Fi 通道中的探测响应帧。扫描等待时间过后，WINC 切换到下一个通道。延长扫描等待时间会增加在扫描操作期间检测到更多接入点的可能性，但也会增加功耗和总扫描持续时间。WINC 固件默认扫描等待时间经过优化，可在功耗和扫描精度之间实现平衡。WINC 固件提供灵活的配置选项，允许主机 MCU 应用程序设置扫描时间。有关详细信息，请参见 `m2m_wifi_set_scan_options` API。

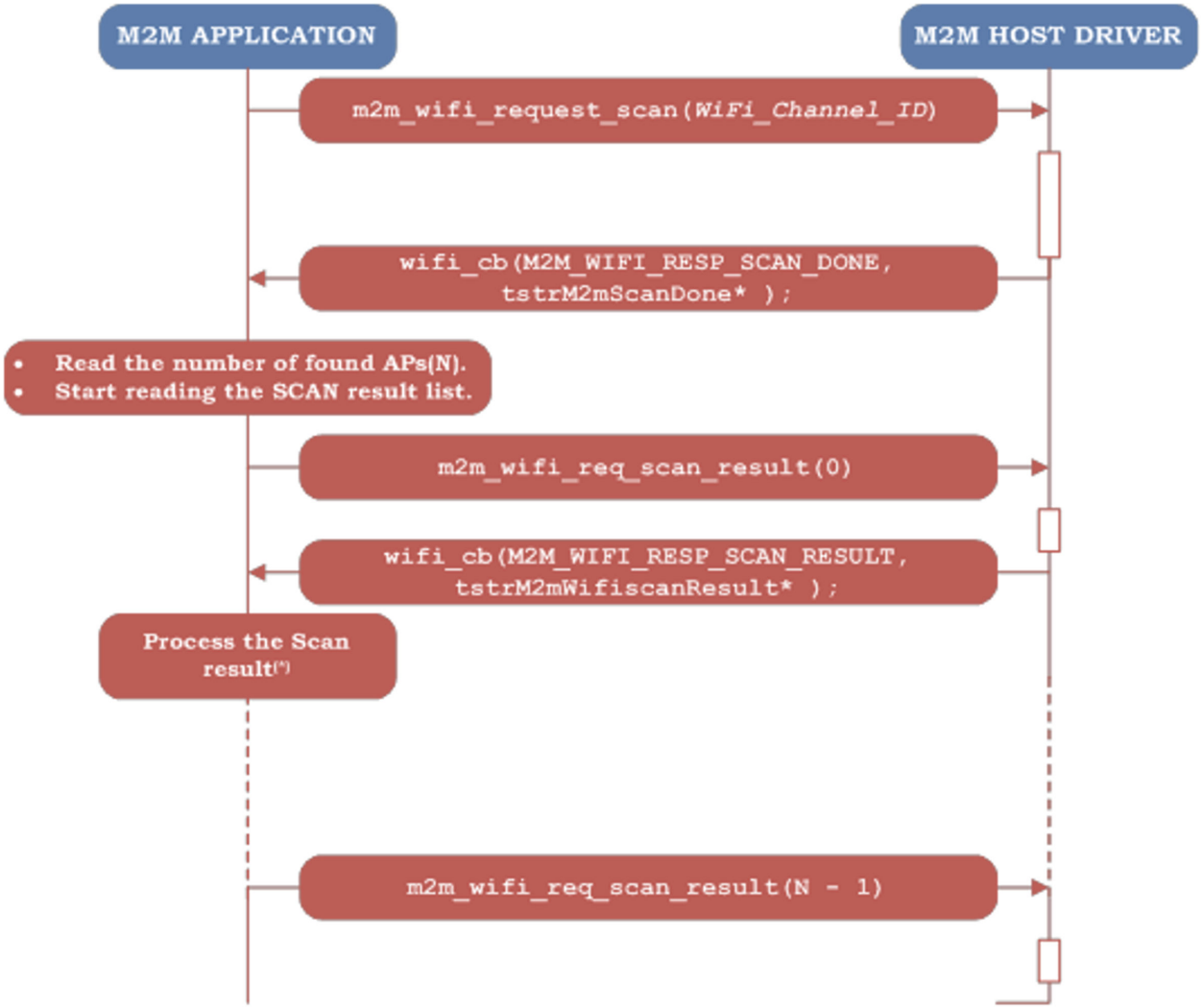
5.2 Wi-Fi 扫描

可通过调用 `m2m_wifi_request_scan` API 来启动 Wi-Fi 扫描操作。可在所有 2.4 GHz Wi-Fi 通道上或在特定请求的通道上执行扫描。

扫描响应时间取决于扫描选项，扫描选项可通过调用 `m2m_wifi_set_scan_options(tstrM2MScanOption* ptstrM2MScanOption)` 进行设置。例如，如果主机 MCU 应用程序请求扫描所有通道，则扫描时间等于 *NoOfChannels (13)* * *ptstrM2MScanOption->u8NumOfSlot* * *ptstrM2MScanOption->u8SlotTime*。

扫描操作如下图所示。

图 5-1. Wi-Fi 扫描操作



5.3 Wi-Fi 安全

WINC Wi-Fi STA 模式支持以下安全类型。

- OPEN
- WEP（有线等效协议）
- WPA/WPA2（Wi-Fi 保护接入——密码形式的个人安全模式）
- 802.1X（WPA/WPA2-Enterprise 安全模式）

对于 802.1X Enterprise 安全模式，ATWINC1500 固件版本 19.6.1 支持以下身份验证方法。

- EAP-TLS
- EAP-PEAPv0/TLS
- EAP-PEAPv1/TLS
- EAP-TTLSv0/MSCHAPv2
- EAP-PEAPv0/MSCHAPv2

- EAP-PEAPv1/MSCHAPv2

自固件版本 v19.6.1 起，m2m_wifi_connect 被弃用。传统 APIm2m_wifi_connect 和 m2m_wifi_connect_sc 可用作新 API 的包装器。从功能上来说，其行为与先前发布的驱动程序相同。

表 5-1 中汇总了适合 OPEN、WEP、WPA/WPA2 和 802.1X 等不同安全类型的推荐 API。

所有新连接 API 都通过指定其 BSSID 和 SSID 来与特定接入点连接。要将连接对象限制为特定接入点，除 SSID 之外，应用程序还可在参数 tstrNetworkId -> pu8Bssid 中指定 BSSID。

应用程序可指示 WINC 是否将凭证存储在闪存中，以及是否必须加密保存的凭证。通过配置 enum tenuCredStoreOption 来完成此操作。

对于 Enterprise 安全模式，应用程序可以将 WINC 配置为在阶段 1 身份验证期间发送实际身份或使用匿名身份。可通过将参数 tstrAuth1xTls 或 tstrAuth1xMschap2 中的 bUnencryptedUserName 置 1 或清零来完成此操作。

有关 API m2m_wifi_connect_1x_tls 使用的更多详细信息，请参见 ASF（v3.42 或更高版本）示例“WINC1500 连接 EAP-TLS：PEAPv0/TLS 和 PEAPv1/TLS 安全 AP 示例”。

有关 API m2m_wifi_connect_1x_mschap2 使用的更多详细信息，请参见 ASF（v3.42 或更高版本）示例“WINC1500 连接 EAP-TTLSv0/MSCHAPv2、EAP-PEAPv0/MSCHAPv2 和 EAP-PEAPv1/MSCHAPv2 安全 AP 示例”。

5.4 按需 Wi-Fi 连接

当主机应用程序知晓所有必需的连接参数（SSID 和安全凭证等）后，就可按需建立 Wi-Fi 连接。应用程序将根据安全类型调用以下 API，以便按需启动 Wi-Fi 连接。

表 5-1. 基于安全类型的 API 列表

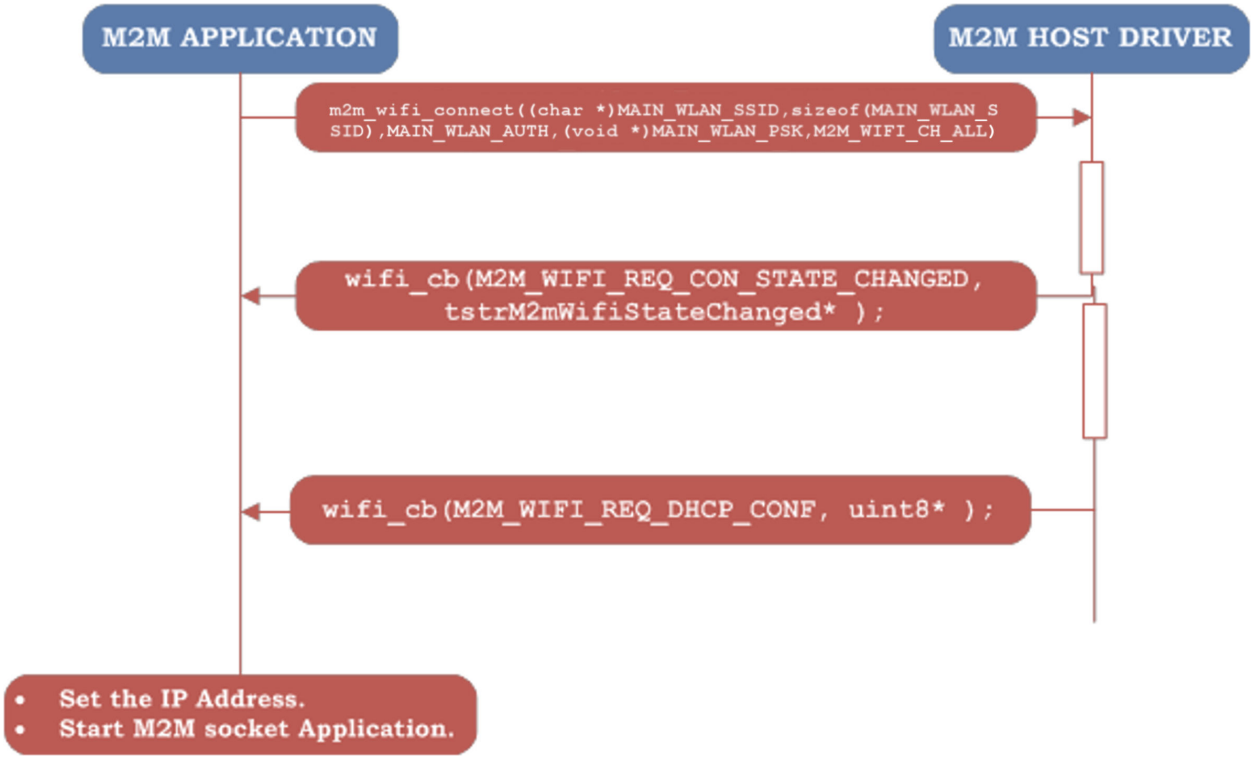
安全类型	API
Open	m2m_wifi_connect_open
WEP	m2m_wifi_connect_wep
WPA/WPA2	m2m_wifi_connect_psk
802.1x with MSCHAPv2	m2m_wifi_connect_1x_mschap2
802.1x with TLS	m2m_wifi_connect_1x_tls

此外，应用程序可调用 API m2m_wifi_connect 来连接支持 Open、WEP、WPA/WPA2 和 802.1x/MSCHAPv2 的接入点。m2m_wifi_connect 在 v19.6.1 中已弃用，但被保留以供旧版使用。

注： 使用表 5-1 中的 API 即表示主机 MCU 应用程序已事先知道连接参数。例如，可将连接参数存储在连接到主机 MCU 的非易失性存储器上。

按需 Wi-Fi 连接操作如下图所示。

图 5-2. 按需 Wi-Fi 连接



5.4.1 示例代码

5.4.1.1 使用 MSCHAPv2（作为阶段 2 的身份验证方法）时连接到企业网络（PEAP 和 TTLSv0）的代码示例

```
#define MAIN_WLAN_SSID                    "WINC1500_ENTERPRISE" /**< Destination SSID */
#define MAIN_WLAN_802_1X_USR_NAME        "DEMO_USER" /**< RADIUS user account name */
#define MAIN_WLAN_802_1X_PWD             "DemoPassword" /**< RADIUS user account password */

int main(void)
{
    int8_t ret;
    tstrWifiInitParam param;
    tstrNetworkId networkId;
    tstrAuth1xMschap2 mschapv2_credential;

    /* Initialize the board.*/
    system_init();

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

    /* Initialize the BSP.*/
    nm_bsp_init();

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

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

    networkId.pu8Bssid = NULL;
    networkId.pu8Ssid = (uint8 *)MAIN_WLAN_SSID;
```

```
    networkId.u8SsidLen = strlen(MAIN_WLAN_SSID);
    networkId.enuChannel = M2M_WIFI_CH_ALL;

    mschapv2_credential.pu8Domain = NULL;
    //mschapv2_credential.u16DomainLen = strlen(mschapv2_credential.pu8Domain);
    mschapv2_credential.pu8UserName = (uint8 *)MAIN_WLAN_802_1X_USR_NAME;
    mschapv2_credential.pu8Password = (uint8 *)MAIN_WLAN_802_1X_PWD;
    mschapv2_credential.u16UserNameLen = strlen(MAIN_WLAN_802_1X_USR_NAME);
    mschapv2_credential.u16PasswordLen = strlen(MAIN_WLAN_802_1X_PWD);
    mschapv2_credential.bUnencryptedUserName = false;
    mschapv2_credential.bPrependDomain = true;

    printf("Connecting to %s\r\n\tUsername:%s\r\n", MAIN_WLAN_SSID,
MAIN_WLAN_802_1X_USR_NAME);

    m2m_wifi_connect_1x_mschap2( WIFI_CRED_SAVE_ENCRYPTED, &networkId, &mschapv2_credential);

    /* Infinite loop to handle a event from the WINC1500.*/
    while (1) {
        while (m2m_wifi_handle_events(NULL) != M2M_SUCCESS) {
        }
    }

    return 0;
}
```

5.4.1.2 使用 TLS（作为阶段 2 的身份验证方法）和 EAP-TLS 时连接到 PEAP 企业网络的代码示例

```
/** security information for Wi-Fi connection */
#define MAIN_WLAN_SSID                   "WINC1500_ENTERPRISE" /**< Destination SSID */
#define MAIN_WLAN_802_1X_USR_NAME       "DEMO_USER" /**< RADIUS user account name */
const uint8_t modulus[] = { /** private key modulus extracted from key file */ };
const uint8_t exponent[] = { /** private key exponent coefficient extracted from  key file
*/ };
const uint8_t certificate[] = { /** certificate coefficient  corresponding to Private Key
*/ };

int main(void)
{
    int8_t ret;
    tstrWifiInitParam param;
    tstrNetworkId networkId;
    tstrAuth1xTls tls_credential;

    /* Initialize the board.*/
    system_init();

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

    /* Initialize the BSP.*/
    nm_bsp_init();

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

    /* Initialize Wi-Fi driver with data and status callbacks.*/
    param.pfAppWifiCb = wifi_cb;
    ret = m2m_wifi_init(&param);
    if (M2M_SUCCESS != ret) {
        printf("main: m2m_wifi_init call error!(%d)\r\n", ret);
        while (1) {
        }
    }
    printf("Username:%s\r\n",MAIN_WLAN_802_1X_USR_NAME);

    /* Connect to the enterprise network.*/
    networkId.pu8Bssid = NULL;
    networkId.pu8Ssid = (uint8 *)MAIN_WLAN_SSID;
    networkId.u8SsidLen = strlen(MAIN_WLAN_SSID);
    networkId.enuChannel = M2M_WIFI_CH_ALL;

    tls_credential.pu8Domain = NULL;
    tls_credential.pu8UserName = (uint8 *)MAIN_WLAN_802_1X_USR_NAME;
```

```
    tls_credential.pu8PrivateKey_Mod = (uint8 *)modulus;
     tls_credential.pu8PrivateKey_Exp = (uint8 *)exponent;
     tls_credential.pu8Certificate = (uint8 *)certificate;
     tls_credential.u16UserNameLen = strlen(MAIN_WLAN_802_1X_USR_NAME);
     tls_credential.u16PrivateKeyLen = sizeof(modulus);
     tls_credential.u16CertificateLen = sizeof(certificate);
      tls_credential.bUnencryptedUserName = true;
      tls_credential.bPrependDomain = true;

    printf("Connecting to %s...\r\n\t\tUsername:%s\r
\n",networkId.pu8Ssid,tls_credential.pu8UserName);

     m2m_wifi_connect_1x_tls(WIFI_CRED_SAVE_ENCRYPTED, &networkId, &tls_credential);

    /* Infinite loop to handle a event from the WINC1500.*/
    while (1) {
        while (m2m_wifi_handle_events(NULL) != M2M_SUCCESS) {
        }
    }

    return 0;
}
```

5.5 默认连接

主机 MCU 应用程序使用 m2m_wifi_default_connect API 基于存储在 WINC 串行闪存中的连接配置文件建立默认连接。此 API 不需要 AP 信息就可建立连接。

注： 当调用按需 Wi-Fi 连接 API 时，连接配置文件信息会自动存储在 WINC 闪存中（见表 5-1）。此连接配置文件的保存取决于 enum tenuCredStoreOption。

AP 或 Enterprise 证书密码等凭证以及其他参数（如 SSID、IP 地址和 BSSID）在写入串行闪存之前使用 AES128-CBC 进行加密。这样，即使攻击者可以物理访问器件，也很难检索敏感信息。如果没有缓存的配置文件，或者无法与任何缓存的配置文件建立连接，则会向主机驱动程序发送 M2M_WIFI_RESP_DEFAULT_CONNECT 类型的事件以指示失败。

成功进行默认连接后，主机应用程序可以通过调用 m2m_wifi_get_connection_info API 读取当前的 Wi-Fi 连接状态。m2m_wifi_get_connection_info 为异步 API。在 Wi-Fi 回调函数中，实际连接信息在异步事件 M2M_WIFI_RESP_CONN_INFO 中提供。tstrM2MConnInfo 类型的回调函数参数提供 AP SSID、RSSI（AP 接收的功率大小）、安全类型和 DHCP 获取的 IP 地址的相关信息。

注： 当且仅当与目标 AP 成功建立连接时，才会在串行闪存中缓存连接配置文件。

Wi-Fi 默认连接操作如下图所示。

图 5-3. Wi-Fi 默认连接

5.6 加密凭证存储

在 ATWINC15x0 固件 v19.6.1 及更高版本中，AP 或 Enterprise 证书密码等凭证以及其他参数（如 SSID、IP 地址和 BSSID）在写入串行闪存之前使用 AES128-CBC 进行加密。这样，即使攻击者可以物理访问器件，也很难检索敏感信息。不得将此功能提供的加密视为安全。加密仅用于防止通过伺机读取 ATWINC15x0 闪存以明文形式获得凭证。因此，必须尽可能采取其他安全措施，例如定期更改密码以及在不再需要时删除凭证。

请求连接到网络时，应用程序可指定连接凭证在 ATWINC15x0 闪存中的存储方式。选项如下：

- 不存储凭证
- 存储未加密凭证
- 存储加密凭证

凭证包括：

- SSID
- BSSID（如提供）
- WEP 密钥（用于 WEP 连接）
- 密码和 PSK（用于 WPA/WPA2 PSK 连接）
- 域、用户名和密码（用于 WPA/WPA2 1x MSCHAPv2 连接）
- 域、用户名、证书和私钥（用于 WPA/WPA2 1x TLS 连接）

凭证在连接成功时存储在 ATWINC15x0 闪存中，一次只存储一组凭证；如果需要存储新凭证，则会删除旧凭证（用 0 覆盖）。

如果凭证存储在 ATWINC15x0 闪存中，则应用程序可以使用 m2m_wifi_default_connect 请求后续连接，而无需再次提供凭证。

如果已启用漫游，则无论凭证是否存储在 ATWINC15x0 闪存中，都可进行漫游（连接期间凭证存储在数据存储器中）。应用程序可使用 m2m_wifi_delete_sc 从 ATWINC15x0 闪存中删除凭证。

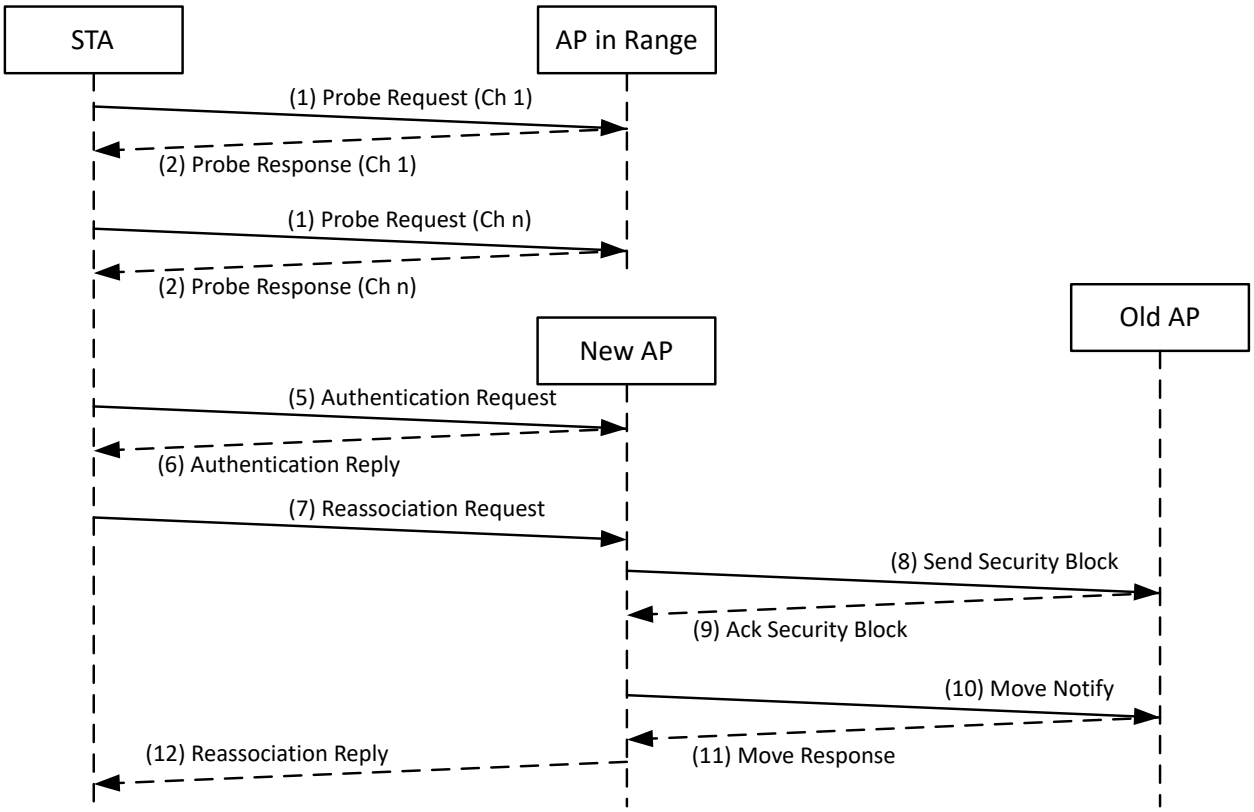
注： 版本 19.6.1 固件实现了新格式的 ATWINC15x0 闪存来存储连接参数。其作用如下：

- 固件升级到 v19.6.1 时，将重新格式化先前存储的凭证。第一次成功连接到接入点后，这些存储的凭证将被加密。
- 固件升级到 v19.6.1 时，先前存储的 IP 地址和 Wi-Fi 通道将被删除。
- 固件从 v19.6.1 降级到之前的固件后，v19.6.1 固件存储的凭证无法通过之前的固件读取。之前固件的操作不受影响。

5.7 简单漫游

简单漫游是 WINC 固件版本 19.6.1 及更高版本支持的自定义功能。使能简单漫游功能后，配置为站模式的 ATWINC1500 可在具有多个接入点的 ESS 区域中移动。WINC 自动切换到另一个具有相同 SSID、身份验证过程和凭证且信号强度更高的 AP。漫游可使站在保持连网的同时更改其 AP。下图说明了简单漫游功能。

图 5-4. 简单漫游



在 v19.6.1 中，AP 通过跟踪信标和发送 NULL 帧持续连接数据包来确定是否已断开与现有 AP 的连接，如果发现断开，就会开始 WINC 漫游。当 WINC 从一个 AP 漫游到另一个 AP 时（均位于同一个 IP 子网内），会发生 ISO/OSI 第 2 层漫游。当 WINC 从一个 AP 漫游到另一个处于不同子网的 AP 时（WINC 尝试通过 DHCP 获取新子网内的新 IP 地址），会发生第 3 层漫游。第 3 层漫游会导致任何现有网络连接中断，如果第 4 层及更高层需要连续连接，则上层协议会处理此 IP 地址更改。

漫游算法是 WINC 固件的内部算法。主机 MCU 可使用 API `m2m_wifi_enable_roaming` 和 `m2m_wifi_disable_roaming` 来启用或禁止漫游功能。必须在 WINC 初始化后调用漫游。

启用漫游后，如果 WINC 成功漫游到新 AP，则会向主机 MCU 发送状态为 `M2M_WIFI_ROAMED` 的 `M2M_WIFI_RESP_CON_STATE_CHANGED` 报文。如果 WINC 无法找到新 AP，则会向主机 MCU 发送状态为 `M2M_WIFI_DISCONNECTED` 的 `M2M_WIFI_RESP_CON_STATE_CHANGED` 报文。

API `call m2m_wifi_enable_roaming()` 将 ATWINC15x0 设置为检测连接是否断开，并在检测到与现有接入点断开连接时执行以下漫游步骤。

- 向旧 AP 发送预防性解除身份验证帧。
- 执行扫描以确定附近是否有与之前的 AP 处于同一 ESS 的其他 AP。
- 如果发现 AP，则与新 AP 交换身份验证并重新关联报文，然后在 WPA/WPA2 条件下进行常规 4 路安全握手，或在 802.1x Enterprise 安全条件下进行 EAPOL 交换。
- 将 DHCP 请求发送到新 AP 以尝试保留相同的 IP 地址。将通知事件发送到状态为 `M2M_WIFI_ROAMED` 的 `M2M_WIFI_RESP_CON_STATE_CHANGE` 类型的主机 MCU。此外，还会向主机 MCU 发送传送相同或新 IP 地址的 `M2M_WIFI_REQ_DHCP_CONF` 事件。
- 如果连接有问题或 DHCP 失败，则会向 AP 发送解除身份验证报文，并且 `M2M_WIFI_RESP_CON_STATE_CHANGED` 事件会被发送到主机 MCU（状态设为 `M2M_WIFI_DISCONNECTED`）。

`bEnableDhcp` 参数可以控制漫游到新 AP 后是否发送 DHCP 请求。API 调用 `m2m_wifi_disable_roaming` 用于禁止漫游。

5.8 多个增益表

无线设备的最大发射功率取决于所在地区监管机构的规定。Wi-Fi 设备的最大发射功率也受到所在地区的法规限制。可使用增益表配置 WINC 中的发送功率。用于不同通道和不同数据速率的数字增益（Digital Gain，DG）以名为“增益表”的表格形式存储在 ATWINC15x0 闪存中。在 ATWINC15x0 中，功率放大器（Power Amplifier，PA）和功率预放大器（Pre-power Amplifier，PPA）值直接在固件中配置。

下图给出了增益表的格式。

图 5-5. 增益表

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	Channels
Data Rates	1	-10	-9	-9	-9	-9	-9	-9	-9	-9	-9	-10	-9	-9	-9	Digital Gain
	2	-10	-9	-9	-9	-9	-9	-9	-9	-9	-9	-10	-9	-9	-9	
	5.5	-10	-9	-9	-9	-9	-9	-9	-9	-9	-9	-10	-9	-9	-9	
	11	-10	-9	-9	-9	-9	-9	-9	-9	-9	-9	-10	-9	-9	-9	
	6	-11	-7	-7	-7	-7	-7	-7	-7	-7	-7	-9	-7	-7	-7	
	9	-11	-7	-7	-7	-7	-7	-7	-7	-7	-7	-9	-7	-7	-7	
	12	-11	-7	-7	-7	-7	-7	-7	-7	-7	-7	-9	-7	-7	-7	
	18	-11	-7	-7	-7	-7	-7	-7	-7	-7	-7	-9	-7	-7	-7	
	24	-11	-7	-7	-7	-7	-7	-7	-7	-7	-7	-9	-7	-7	-7	
	36	-11	-7	-7	-7	-7	-7	-7	-7	-7	-7	-9	-7	-7	-7	
	48	-11	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	
	54	-11	-9	-9	-9	-8	-8	-8	-8	-8	-8	-9	-8	-8	-8	
	mcs0	-12	-7	-7	-7	-7	-7	-7	-7	-7	-7	-10	-7	-7	-7	
	mcs1	-12	-7	-7	-7	-7	-7	-7	-7	-7	-7	-10	-7	-7	-7	
	mcs2	-12	-7	-7	-7	-7	-7	-7	-7	-7	-7	-10	-7	-7	-7	
	mcs3	-12	-7	-7	-7	-7	-7	-7	-7	-7	-7	-10	-7	-7	-7	
	mcs4	-12	-7	-7	-7	-7	-7	-7	-7	-7	-7	-10	-7	-7	-7	
	mcs5	-12	-8	-8	-7	-7	-7	-7	-7	-7	-7	-10	-7	-7	-7	
	mcs6	-12	-9	-8	-8	-8	-8	-8	-8	-8	-8	-10	-8	-8	-8	
	mcs7	-12	-10	-9	-9	-9	-9	-9	-9	-9	-9	-10	-9	-9	-9	
	1e9c	0														Specific Configuration
	1edc	0														

增益表作为固件更新包的一部分提供，其格式为.csv 文件，位于 src/firmware/Tools/gain_builder/gain_sheets 文件夹中。增益值将作为完整下载过程的一部分进行下载。有关更多详细信息，请参见“WINC Devices – Integrated Serial Flash Memory Download Procedure”文档。

在 v19.6.1 之前，ATWINC15x0 仅支持一个增益表，致使 WINC 只能在不要求提供多种增益的监管地区使用。

ATWINC15x0 固件版本 19.6.1 或更高版本支持多个增益表，闪存最多可存储四个增益表。增益表可由主机 MCU 使用 API m2m_wifi_set_gain_table_idx 进行选择。如果 ATWINC15x0 必须在多个地区使用（采用相关地区允许的最大发射功率），可利用多个增益表功能来根据 ATWINC15x0 的所在地区选择增益表（由主机 MCU 完成）。

5.8.1 将增益表写入 ATWINC15x0

增益构建器应用程序使用多个.csv 文件（最多 4 个），并对增益表执行必要的数学运算以计算增益值并将其写入闪存：

```
gain_builder [-table <no_of_tables> <img_path1> <img_path2> <img_path3>
<img_path4>] [-index <gain_table_index>][-no_wait] [-port]
```

注： img_path*参数指定各个单独的表，索引参数指定在上电时使用的默认表。

5.8.2 选择特定的增益表

使用 API m2m_wifi_set_gain_table_idx 设置特定的增益表索引。
m2m_wifi_set_gain_table_idx 必须在初始化之后，任何连接请求之前调用。必须在闪存中提供相应的增益表。

注： ATWINC15x0 固件版本 v19.6.1 仅包含一个可在所有地区中使用的增益表。

5.9 主机文件下载

主机文件下载是 ATWINC15x0 固件版本 19.6.1 及更高版本支持的一项功能。只有 8 Mb 闪存的 ATWINC1510 器件支持此功能。ATWINC1500 只有 4 Mb 闪存，因此 ATWINC1500 不支持此功能。借助主机文件下载功能，主机 MCU 可指示 ATWINC1510 下载文件并将其保存在 ATWINC1510 闪存中。ATWINC1510 只能从 HTTP 或 HTTPS Web 服务器下载文件。存储在 ATWINC1510 中的文件不能超过 508 KB。此功能非常适合更新主机 MCU 的固件，但并非仅限于 MCU OTA。

执行 MCU OTA 更新时，没有规定文件格式，因此应用程序开发人员可选择一种策略对接收的文件执行完整性检查。WINC 不会对下载的文件执行任何完整性检查，因此建议应用程序执行此操作。

该功能支持单个文件，最大 508 KB。驱动程序通过文件处理程序识别每个文件，以避免对存储在 WINC 闪存中的文件进行无效访问。如果开始新的下载或文件被擦除，对分区的访问会被拒绝。此外，应用程序可请求显式擦除以从 ATWINC 闪存中删除文件，从而销毁任何可能的机密数据。

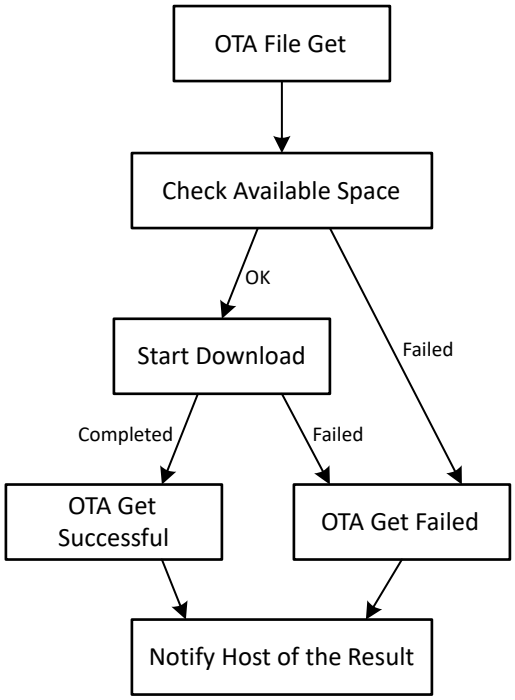
API m2m_ota_host_file_get 用于从远程位置下载文件并将其存储在 ATWINC1500 闪存中。使用该 API 一次只能下载一个文件。再次调用该 API 时，将擦除先前存储的文件并启动新文件下载。

要从 ATWINC1510 闪存检索下载的文件，主机 MCU 可使用 m2m_ota_host_file_read_spi 或 m2m_ota_host_file_read_hif API。文件下载完成后，会通过 m2m_ota_host_file_get API 中注册的回调函数来通知主机。用户可通过传递必要的参数来使用 m2m_ota_host_file_read_spi 或 m2m_ota_host_file_read_hif API，以便从 WINC 闪存读取文件。

5.9.1 概述

应用程序需要存储在远程位置的文件中的信息时，可以使用主机文件下载功能从远程位置检索文件并将其临时存储在 WINC 的闪存中。下载完成后，会生成文件处理程序并将其存储在 WINC 的 NVM 中，因此即使在 WINC 复位后仍有效。生成处理程序后，可通过提供的 API 访问该文件，并且可通过 HIF 和 SPI 两种机制读取文件。在任何一种情况下，读取操作要用到应用程序尝试访问的文件的处理程序，如果请求的处理程序和内部存储的处理程序匹配，则授予访问权限。此过程同样适用于文件擦除。使用文件处理程序可避免访问无效数据，例如尝试同时访问文件时。下图给出了执行主机文件下载时 WINC 遵循的步骤。

图 5-6. WINC 内的主机文件下载操作



仅当闪存中的可用空间足以存储请求下载的文件时，才会开始下载。如果在 ATWINC1500（4 Mb 闪存）中请求主机文件下载，由于闪存中没有主机文件分区，因此没有空间来存储该文件，下载将会失败。

“开始下载”步骤会导致之前可用的有效文件处理程序失效。接收到“OTA Get Successful”（OTA 获取成功）消息时，将生成一个新的文件处理程序并显示下载文件的状态和总大小，这些信息包含在发送给主机的下载完成通知中。

5.9.2 OTA 初始化

要使用主机文件下载功能，必须初始化 WINC 和 OTA 驱动程序。以下为 OTA 初始化过程：

1. `m2m_wifi_init` 或 `m2m_wifi_reinit`——需要此 API 来初始化 WINC 并设置用于 HIF 通信的回调函数。完成此步骤后，可以将 WINC 配置为连接到网络并下载文件。有关何时使用这两个选项的更多详细信息，请参见 API 文档。
2. `m2m_ota_init`——此 API 注册 OTA 回调函数，执行通过主机文件下载 API 配置的回调函数以及通知应用程序文件下载状态时需要此 API。

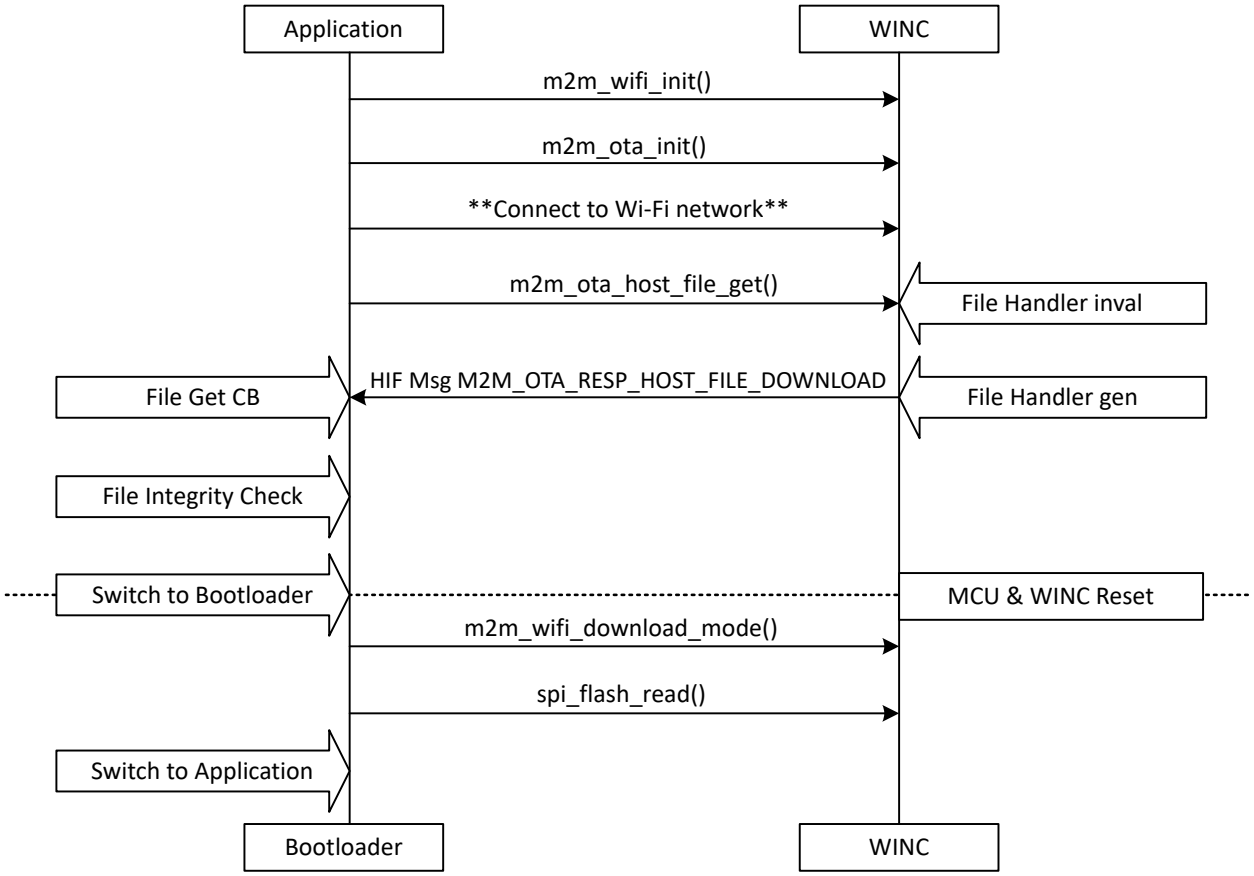
5.9.3 使用主机文件下载功能实现 MCU OTA

主机文件下载功能允许应用程序从远程位置下载文件。可通过安全连接访问文件的链接，下载文件后，会将其存储在 WINC 的闪存中，并通知应用程序。可下载任何类型的文件，并不限于 MCU 二进制文件，充

分体现出此功能灵活而又强大的特点。以文本文件下载为例，此时可以保存文件校验和，应用程序稍后可利用此校验和验证已下载二进制文件的完整性。主机 MCU OTA 需要以下步骤：

- 提供文件的 http/https 链接，通知 WINC 从特定的远程位置下载文件，可使用 API `m2m_ota_host_file_get` 完成此过程。
- 使用 `spi_flash_read` 从 WINC 读取镜像。由于目前存在限制，自举程序还需要执行 `m2m_wifi_init` 和 `m2m_ota_init`，之后才能执行 `m2m_ota_host_file_read_spi` 以从 WINC 读取镜像。MCU OTA 的 ASF 示例中未使用 `m2m_ota_host_file_read_hif` 和 `m2m_ota_host_file_read_spi`，以便在解决上述限制的同时保持较小的驱动程序占用空间。只有在应用程序需要复位时才会有此限制。如果在这种情况下切换到自举程序，WINC 驱动程序将无法再度跟踪文件处理程序，必须通过初始化过程重新加载。如果无需执行复位或关断，且在下载文件后无需加载不同的应用程序，则可使用这两个 API。

图 5-7. MCU OTA 的主机文件下载示例



应用程序开发人员必须同时考虑的其他步骤：

- 建议使用校验和计算验证镜像的完整性，并将其与先前已知的校验和进行匹配。用户可设计验证机制，因为 MCU OTA 没有要求预定义文件格式。
- 可选择从闪存中擦除文件。虽然不这样做也可以请求新的下载，但这样非常有利于安全，可确保敏感数据在使用后即失效。
  注： WINC 不会对通过主机文件下载获得的任何文件执行任何完整性检查，必须由应用程序执行此类检查。

5.9.4 API 说明

有关 API 的更多详细说明，请参见 WINC1500_SW_API.chm。

5.9.4.1 OTA 文件获取

```
NMI_API sint8 m2m_ota_host_file_get
(
unsigned char    *pcDownloadUrl,
tpfFileGetCb    pfHFDGetCb
);
```

此 API 用于获取远程存储文件的链接。随后，会将此链接传递给 WINC 来建立 TCP 连接以从该位置检索文件。此时也可以使用配置了 TLS 的服务器。

此外，还必须提供回调函数，以便在文件获取操作完成时执行。如果文件获取为成功状态，则会将此状态传递给此回调函数，WINC 生成的文件处理程序和下载文件的总大小会正确传递给回调函数。

5.9.4.2 文件获取回调函数

```
typedef void (*tpfFileGetCb)
(
uint8    u8Status,
uint8    u8Handler,
uint32    u32Size
);
```

文件获取回调函数会收到三个参数：文件获取请求的状态、文件处理程序 ID 和文件的总大小。如果状态为 OTA_STATUS_SUCCESS，则表示文件处理程序和大小可用，否则不会填充值。从应用程序的角度来看，不能将其视为有效。

文件处理程序在 WINC 中自动生成，用于标识文件。仅在下载成功完成时，才会生成相应的文件处理程序。从文件中读取或擦除文件时都要用到处理程序。类似地，如果下载被中止或中断，则不会生成处理程序，而是显示值 HFD_INVALID_HANDLER，随后会禁止通过 API 对闪存执行其他操作。

文件下载完成后，下载文件的总大小将传送给回调函数以通知应用程序。应用程序会借此跟踪下载数据的总大小和读取的数据量。

5.9.4.3 OTA 文件读取 HIF

```
NMI_API sint8 m2m_ota_host_file_read_hif
(
uint8            u8Handler,
uint32            u32Offset,
uint32            u32Size,
tpfFileReadCb        pfHFDReadCb
);
```

下载完成后，文件将存储在 WINC 闪存中。此 API 可用于通过 HIF 报文从 WINC 读取文件。必须具有一个有效的处理程序，如果没有，则可能意味着该文件已失效，因此无法用于任何操作。这样可防止读取无效或损坏的数据。

偏移标记闪存的读取位置（以字节计），从文件的开头开始计数。因此，偏移为零即表示从文件的开头读取。大小指定要读取的字节数，从定义的偏移量开始。 最后一个参数是读取完成时要执行的回调函数。

优点（与 SPI 读取相比）

- 使用 HIF 报文读取文件时，主机可继续运行，在数据读取完成后会由来自 WINC 的中断向其发出通知。
- 读取完成后无需复位 WINC。

缺点（与 SPI 读取相比）

- 通过 HIF 读取文件的速度略慢于通过 SPI 读取文件的速度。

5.9.4.4 文件读取 HIF 回调函数

```
typedef void (*tpfFileReadCb)
(
uint8    u8Status,
void    *pBuff,
uint32    u32Size
);
```

回调函数仅在通过 HIF 报文读取文件后执行，它会收到三个参数。

- 第一个参数是读取的状态，如果读取不成功，则其他参数将被忽略。
- 第二个参数是指向读取的数据缓冲区的指针。
- 第三个参数是大小，它表示要在缓冲区中（最大 128 字节）读取的数据量。

指定通过 HIF 读取大量数据时，可能会超出缓冲区最大大小（128 字节），因此建议使用 `u32Size` 从此回调函数中偏移第二次读取。这要求应用程序跟踪文件的总大小和读取的字节数，逐次请求读取每个部分，直至达到文件末尾。

5.9.4.5 OTA 文件读取 SPI

```
NMI_API sint8 m2m_ota_host_file_read_spi
(
uint8    u8Handler,
uint8    *pu8Buff,
uint32    u32Offset,
uint32    u32Size
);
```

通过 SPI 读取文件类似于通过 HIF 读取文件。要通过 SPI 访问 WINC 闪存，必须将 WINC 设置为允许安全读取/写入闪存的模式，因此不考虑使用回调。通常使用循环来读取 WINC 处于该状态时所需的所有数据，然后重新启动 WINC。

要使用此 API，应用程序必须调用 `m2m_wifi_download_mode` 以使 WINC 可安全用于读/写闪存访问，完成读取后，WINC 必须重新初始化（`m2m_wifi_reinit` 和 `m2m_ota_init`）才能再次连网（如果应用程序基于请求）。`pu8Buff` 指针指向应用程序提供的缓冲区以及读取数据的位置。

优点（与 HIF 读取相比）

- SPI 读取速度快于 HIF 读取速度。

缺点（与 HIF 读取相比）

- 需要 WINC 设置为特殊模式并稍后重启。
- 通常会受阻，因为读取在循环内完成以最大程度减少 WINC 复位。

5.9.4.6 OTA 文件擦除 API

```
NMI_API sint8 m2m_ota_host_file_erase
(
uint8            u8Handler,
tpfFileEraseCb    pfHFDEraseCb
);
```

文件擦除 API 需要以下两个参数：

- 第一个参数是要擦除文件的处理程序，用于确保有效执行闪存擦除。
- 另一个参数是擦除完成时执行的回调函数。

擦除完成时通过回调函数通知应用程序，从而有效地触发后续操作（例如下载第二个文件）。

注： 文件擦除 API 会擦除整个主机文件分区，无论 WINC 中擦除操作的最终结果如何，都会破坏文件处理程序。由于闪存中的数据被部分或完全破坏，因此当过程启动以确保安全时，处理程序无效。

5.9.4.7 文件擦除回调函数

```
typedef void (*tpfFileEraseCb)
(
uint8    u8Status
);
```

文件擦除回调会收到操作的擦除状态。OTA_STATUS_SUCCESS 状态确保数据已完全擦除，任何其他结果都不能保证数据仍有效或已完全擦除。

5.9.4.8 OTA 中止 API

```
NMI_API sint8 m2m_ota_abort
(
void
);
```

如果已启动主机文件下载但应用程序决定取消下载，则可调用此 API。这无需任何输入参数。

注： 此 API 与 WINC OTA 共享，如果在 WINC OTA 进程中发出，则会取消 WINC OTA。

5.9.5 限制

- 在 ATWINC1510 中的 512 KB 闪存中，WINC 使用第一个扇区（大小为 4 KB）来存储用于主机文件下载功能的文件信息。这表示，应用程序可使用共 508 KB 大小的闪存来存储主机文件。
- 仅 ATWINC1510 支持该功能，因为 ATWINC1500 仅具有 4 Mb 闪存，这意味它没有空间来存储文件。
- 没有文件系统，一次只存储一个文件。再次调用获取文件函数时，将擦除先前存储的文件并启动新文件下载。
- WINC OTA 固件下载和主机 OTA 文件下载不能同时运行。
- WINC 在应用程序尝试下载中断或死链接时会解释 404 Not Found 错误，并提供 OTA_STATUS_SERVER_ERROR 错误状态。WINC 不会解释任何其他无效链接的消息。WINC 将错误消息下载到 SPI 闪存中，并将主机指示为文件下载。由应用程序负责检查文件是否有效。

5.9.6 内置自动测试设备（Automated Test Equipment，ATE）机制

对于运行 v19.6.1 固件且在出厂时预装闪存的 ATWINC15x0 模块，在为 OTA 传输保留的闪存空间中有一个特殊的 ATE 固件（由第一次 OTA 更新覆盖）。

可在 WINC 初始化期间调用主机 API，以将器件引导至此特殊固件（m2m_ate_init）。\ASF\common\components\wifi\winc1500\driver\include\m2m_ate_mode.h 中详细介绍了用于控制此固件提供的 ATE 功能的 API。

以下为示例代码。

```
int main(void)
{
    /* Initialize the board.*/
    system_init();

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

    /* Initialize the BSP.*/
    nm_bsp_init();

    /*Check if initialization of ATE firmware is succeeded or not*/
```

```
    if(M2M_SUCCESS == m2m_ate_init())
    {
        /*Run TX test case if defined*/
        #if (M2M_ATE_RUN_TX_TEST_CASE == ENABLE)
        start_tx_test(M2M_ATE_TX_RATE_1_Mbps_INDEX);
        #endif
        /*Run RX test case if defined*/
        #if (M2M_ATE_RUN_RX_TEST_CASE == ENABLE)
        start_rx_test();
        #endif

        /*De-Initialization of ATE firmware test mode*/
        m2m_ate_deinit();
    }
    else
    {
        M2M_ERR("Failed to initialize ATE firmware.\r\n");
        while(1);
    }

    #if ((M2M_ATE_RUN_RX_TEST_CASE == ENABLE) && (M2M_ATE_RUN_TX_TEST_CASE == ENABLE))
    M2M_INFO("Test cases have been finished.\r\n");
    #else
    M2M_INFO("Test case has been finished.\r\n");
    #endif

    while(1);
}


#if (M2M_ATE_RUN_TX_TEST_CASE == ENABLE)
static void start_tx_test(uint8_t tx_rate)
{
    tstrM2mAteTx tx_struct;

    /*Initialize parameter structure*/
    m2m_memset((uint8 *)&tx_struct, 0 , sizeof(tx_struct));

    /*Set TX Configuration parameters,
     *refer to tstrM2mAteTx for more information about parameters*/
    tx_struct.channel_num    = M2M_ATE_CHANNEL_11;
    tx_struct.data_rate        = m2m_ate_get_tx_rate(tx_rate);
    tx_struct.dpd_ctrl        = M2M_ATE_TX_DPD_DYNAMIC;
    tx_struct.duty_cycle    = M2M_ATE_TX_DUTY_1;
    tx_struct.frame_len        = 1024;
    tx_struct.num_frames    = 0;
    tx_struct.phy_burst_tx    = M2M_ATE_TX_SRC_MAC;
    tx_struct.tx_gain_sel    = M2M_ATE_TX_GAIN_DYNAMIC;
    tx_struct.use_pmu        = M2M_ATE_PMU_DISBLE;
    tx_struct.cw_tx            = M2M_ATE_TX_MODE_CW;
    tx_struct.xo_offset_x1000 =  0;

    /*Start TX Case*/
    if(M2M_ATE_SUCCESS == m2m_ate_start_tx(&tx_struct))
    {
        uint32 u32TxTimeout = M2M_ATE_TEST_DURATION_IN_SEC;

        M2M_INFO(">>Running TX Test case on CH<%02u>.\r\n", tx_struct.channel_num);
        do
        {
            nm_bsp_sleep(1000);
            printf("%02u\r", (unsigned int)u32TxTimeout);
        }while(--u32TxTimeout);

        if(M2M_ATE_SUCCESS == m2m_ate_stop_tx())
        {
            M2M_INFO("Completed TX Test successfully.\r\n");
        }
    }
    else
    {
        M2M_INFO("Failed to start TX Test case.\r\n");
    }
}
#endif
```

```
#if (M2M_ATE_RUN_RX_TEST_CASE == ENABLE)
static void start_rx_test(void)
{
    tstrM2mAteRx rx_struct;

    /*Initialize parameter structure*/
    m2m_memset((uint8 *)&rx_struct, 0, sizeof(rx_struct));

    /*Set RX Configuration parameters*/
    rx_struct.channel_num    = M2M_ATE_CHANNEL_6;
    rx_struct.use_pmu        = M2M_ATE_PMU_DISBLE;
    rx_struct.xo_offset_x1000 =  0;

    /*Start RX Case*/
    if(M2M_ATE_SUCCESS == m2m_ate_start_rx(&rx_struct))
    {
        tstrM2mAteRxStatus rx_data;
        uint32 u32RxTimeout = M2M_ATE_TEST_DURATION_IN_SEC;

        M2M_INFO(">>Running RX Test case on CH<%02u>.\r\n", rx_struct.channel_num);
        do
        {
            m2m_ate_read_rx_status(&rx_data);
            M2M_INFO("Num Rx PKTs: %d, Num ERR PKTs: %d, PER: %1.3f",
(int)rx_data.num_rx_pkts, (int)rx_data.num_err_pkts,
                (rx_data.num_rx_pkts>0)?((double)rx_data.num_err_pkts/
(double)rx_data.num_rx_pkts):(0));
            nm_bsp_sleep(1000);
        }while(--u32RxTimeout);
        printf("\r\n");
        if(M2M_ATE_SUCCESS == m2m_ate_stop_rx())
        {
            M2M_INFO("Compeleted RX Test successfully.\r\n");
        }
    }
    else
    {
        M2M_INFO("Failed to start RX Test case.\r\n");
    }
}
#endif
```

6. 套接字编程

6.1 概述

ATWINC15x0 套接字应用程序编程接口（Application Programming Interface，API）允许主机 MCU 应用程序与 Intranet 和远程 Internet 主机进行交互。ATWINC15x0 套接字 API 基于 BSD（Berkeley）套接字。本章介绍了 ATWINC15x0 套接字编程及其与常规 BSD 套接字的区别。

注： 在阅读本章之前，读者必须对以下主题有基本了解：

- BSD 套接字
- TCP
- UDP
- Internet 协议

6.1.1 套接字类型

ATWINC15x0 套接字 API 提供两种类型的套接字：

- 数据报套接字（无连接套接字）——使用 UDP 协议
- 流套接字（面向连接的套接字）——使用 TCP 协议

6.1.2 套接字属性

每个 ATWINC15x0 套接字通过以下惟一组合来标识：

- 套接字 ID——每个套接字的惟一标识符。这是套接字 API 的返回值。
- 本地套接字地址——ATWINC15x0 IP 地址和 ATWINC15x0 固件为套接字分配的端口号的组合。
- 协议——传输层协议，TCP 或 UDP。
- 远程套接字地址——仅适用于 TCP 流套接字。这是必需项，因为 TCP 是面向连接的。针对特定 IP 地址和端口号建立的每个连接都需要一个单独的套接字。可以在后续部分介绍的套接字事件回调函数中获取远程套接字地址。

注： TCP 端口 53 和 UDP 端口 53 代表两个不同的套接字。

6.1.3 限制

- ATWINC15x0 套接字 API 最多支持 7 个 TCP 套接字和 4 个 UDP 套接字。
- ATWINC15x0 套接字 API 仅支持 IPv4。不支持 IPv6。

6.2 套接字 API

6.2.1 API 先决条件

- C 头文件 `socket.h`——包括所有必要的套接字 API 函数声明。使用如下所述的 ATWINC15x0 套接字时，主机 MCU 应用程序必须包括 `socket.h` 头文件。
- 初始化——ATWINC15x0 套接字 API 在调用任何套接字 API 函数之前都会初始化一次。使用套接字 API 函数中介绍的 `socketInit` API 完成此操作。

6.2.2 非阻塞异步套接字 API

大多数 ATWINC15x0 套接字 API 都不会阻塞主机 MCU 应用程序的异步函数调用。异步事件中介绍了 ATWINC15x0 异步 API 的行为。

例如，主机 MCU 应用程序可使用 ATWINC15x0 套接字 API `registerSocketCallback` 注册应用程序定义的套接字事件回调函数。主机 MCU 应用程序调用套接字 API `connect` 时，API 返回零值（`SUCCESS`），立即表明已接受请求。随后，主机 MCU 应用程序必须等待 ATWINC15x0 套接字 API，以在建立连接或发生连接超时时调用已注册的套接字回调函数。套接字回调函数提供必要的信息来确定连接状态。

6.2.3 套接字 API 函数

ATWINC15x0 套接字 API 提供以下函数。

6.2.3.1 socketInit

主机 MCU 应用程序在初始化期间必须调用一次 API `socketInit`。此 API 是一个异步 API。

6.2.3.2 registerSocketCallback

`registerSocketCallback` 函数允许主机 MCU 应用程序为 ATWINC15x0 套接字提供应用程序定义的事件回调函数，用于进行套接字操作。此 API 是一个异步 API。API 注册以下回调函数：

- 套接字事件回调函数
- DNS 解析回调函数

套接字事件回调函数是应用程序定义的函数，发生套接字事件时由 ATWINC15x0 套接字 API 调用。在此处理程序中，主机 MCU 应用程序必须提供应用程序定义的逻辑，以处理相关事件。

DNS 解析事件处理程序是应用程序的定义函数，ATWINC15x0 套接字 API 会调用此函数以返回 `gethostbyname` 的结果。这表示仅在主机 MCU 应用程序调用 `gethostbyname` 函数之后才会发生这种情况。如果成功，则回调函数将提供所需域名的 IP 地址。

6.2.3.3 socket

`socket` 函数创建指定类型的新套接字并返回相应的套接字 ID。此 API 是一个异步 API。

大部分其他套接字函数都需要套接字 ID，同时也可将套接字 ID 作为参数传递给套接字事件回调函数，以确定生成事件的套接字。

6.2.3.4 connect

`connect` 函数与 TCP 套接字一起用于建立与 TCP 服务器的新连接。

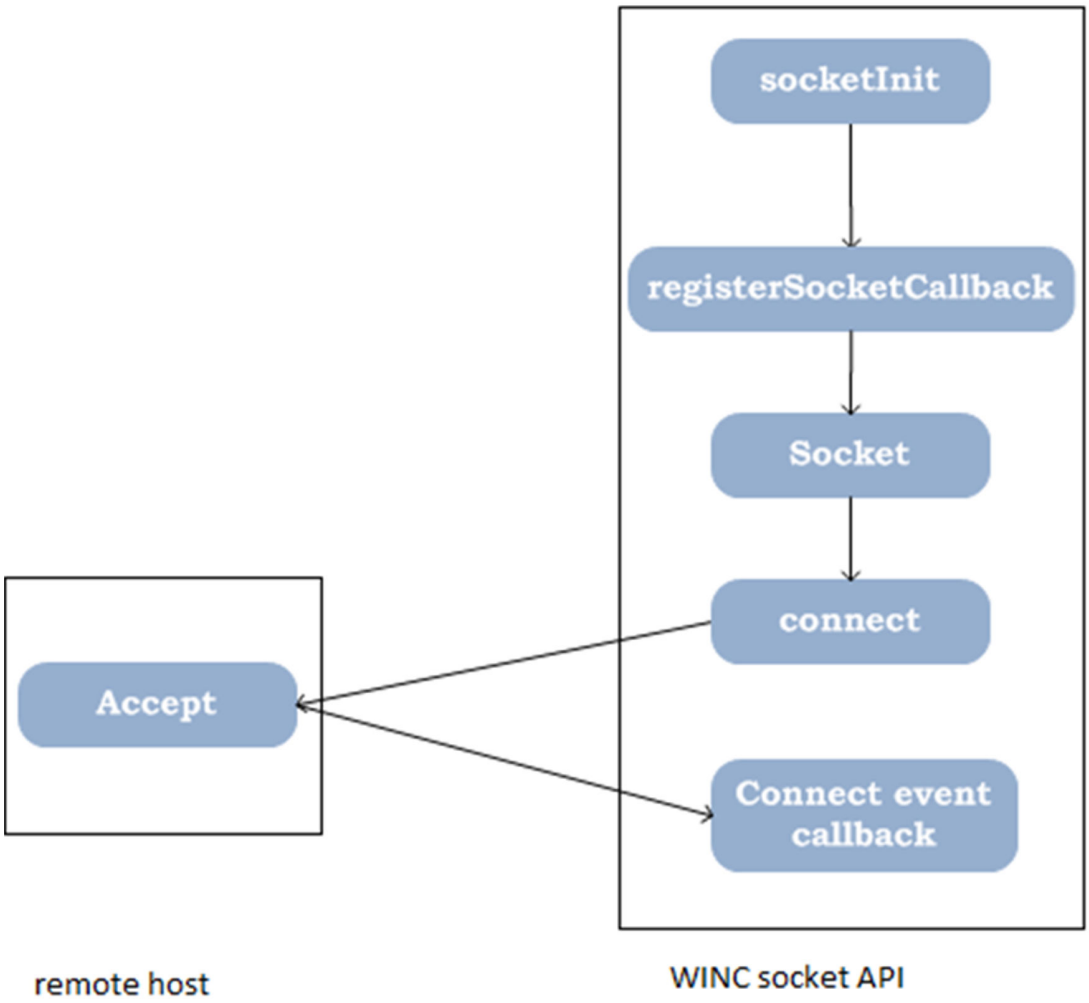
`connect` 函数会在完成后将 `SOCKET_MSG_CONNECT` 发送至套接字事件处理程序回调函数。当 TCP 服务器接受连接或者未收到远程主机响应时，则在大约 30 秒的超时间隔后发送 connect 事件。

注： `SOCKET_MSG_CONNECT` 事件回调函数提供包含错误代码的 `tstrSocketConnectMsg`。错误代码值表示：

- 零值，指示连接成功，或
- 负值，指示因超时条件而导致出错或将 `connect` 用于 UDP 套接字。

下图给出了 ATWINC15x0 套接字 API 与远程服务器主机的连接。

图 6-1. TCP 客户端 API 调用序列



6.2.3.5 bind

bind 函数可用于面向 UDP 和 TCP 套接字的服务器操作。它用于将套接字与地址结构（端口号和 IP 地址）相关联。

bind 函数调用会使 SOCKET_MSG_BIND 事件发送到具有绑定状态的套接字回调函数处理程序。在接收到 bind 回调函数之前，不得发出对 listen、send、sendto、recv 和 recvfrom 函数的调用。

6.2.3.6 listen

listen 函数用于面向 TCP 流套接字的服务器操作。调用 listen API 后，套接字接受来自远程主机的连接请求。listen 函数的作用是，当套接字端口准备好指示侦听操作成功或失败后，将 SOCKET_MSG_LISTEN 事件通知发送到主机。

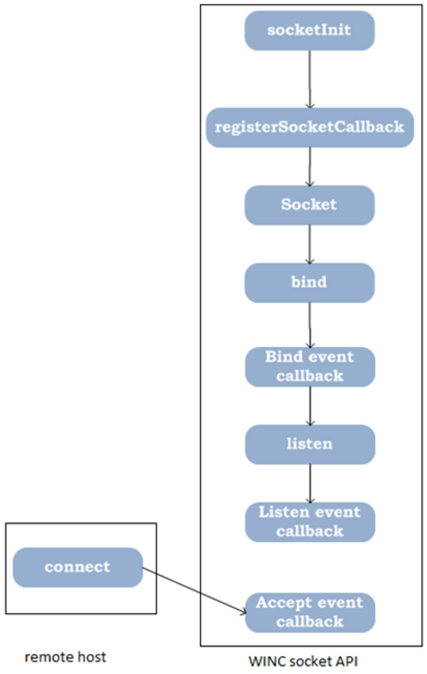
远程对等设备建立连接时，SOCKET_MSG_ACCEPT 事件通知会发送到应用程序。

6.2.3.7 accept

accept 函数已弃用，调用此 API 无效。仅出于向后兼容原因保留此函数。

注： listen API 隐式接受 TCP 远程对等设备连接请求。

图 6-2. TCP 服务器 API 调用序列



虽然已弃用 accept 函数，但在远程主机连接至 ATWINC15x0 TCP 服务器时，仍会发生 SOCKET_MSG_ACCEPT 事件。事件报文包含已连接远程主机的 IP 地址和端口号。

6.2.3.8 send

send 函数由应用程序用于将数据发送到远程主机。send 函数可用于发送 UDP 或 TCP 数据，具体取决于套接字的类型。

- 对于 TCP 套接字，必须先建立连接。
- 对于 UDP 套接字，建议使用定义了目标地址的 sendto API。不过，可使用 send API 代替 sendto API。为此，在连续调用 send 函数前，必须至少成功调用一次 sendto API。这样可确保目标地址保存在 ATWINC15x0 固件中。

将数据传送到远程主机后，send 函数会生成 SOCKET_MSG_SEND 事件回调函数。对于 TCP 套接字，此事件可确保将数据传送到远程主机 TCP/IP 堆栈（远程应用程序必须使用 recv 函数读取数据）。对于 UDP 套接字，这意味着数据已传送，但无法保证数据按 UDP 协议传送到远程主机。对于 UDP 套接字，由应用程序负责保证数据传送。

SOCKET_MSG_SEND 事件回调函数在成功时返回传送的数据大小，在出错时返回零或负值。

6.2.3.9 sendto

sendto 函数由应用程序用于将 UDP 数据发送到远程主机。它只能与 UDP 套接字搭配使用。目标远程主机的 IP 地址和端口作为一个参数包含在 sendto 函数中。

SOCKET_MSG_SENDTO 事件回调函数在成功时返回传送的数据大小，在出错时返回零或负值。

6.2.3.10 recv/recvfrom

recv 和 recvfrom 函数用于分别从 TCP 和 UDP 套接字读取数据，二者的工作原理相同。

主机 MCU 应用程序使用预分配缓冲区调用 recv 或 recvfrom 函数。SOCKET_MSG_RECV 或 SOCKET_MSG_RECVFROM 事件回调函数到达后，此缓冲区必须具有已接收的数据。

接收到的数据大小表示状态，具体如下：

- 正数——接收到数据
- 零——套接字连接终止
- 负数——指示错误

对于 TCP 套接字，建议在每次成功进行套接字连接（客户端或服务器）后都调用 recv 函数。否则，在使用 close 函数调用显式关闭套接字之前，接收的数据会始终在 ATWINC15x0 固件中缓冲，这将浪费系统资源。

6.2.3.11 close

close 函数用于释放分配给套接字的资源，对于 TCP 流套接字，还会终止打开的连接。

每次对 socket 函数的调用都必须与对 close 函数的调用匹配。此外，必须使用此函数关闭服务器套接字端口上接受的套接字。

6.2.3.12 setsockopt

setsockopt 函数可用于设置套接字选项，从而控制套接字行为。

支持的选项如下：

- SO_SET_UDP_SEND_CALLBACK——启用或禁止 send/sendto 事件回调函数。用户可能想要为 UDP 套接字禁止 sendto 事件回调函数，以便提高套接字连接吞吐量。
- IP_ADD_MEMBERSHIP——启用 IP 多播地址订阅。
- IP_DROP_MEMBERSHIP——启用 IP 多播地址取消订阅。
- SOL_SSL_SOCKET——设置 SSL 套接字。以下是 SSL 套接字支持的选项：
  - SO_SSL_BYPASS_X509_VERIF 命令允许打开 SSL 套接字以绕过 X509 认证身份验证过程。
    示例：

```
struct sockaddr_in addr_in;
                            int    optVal =1;
                            addr_in.sin_family = AF_INET;
                            addr_in.sin_port = _htons(MAIN_HOST_PORT);
                            addr_in.sin_addr.s_addr = gu32HostIp;

                            /* Create secure socket */
                            if (tcp_client_socket < 0) {
                            tcp_client_socket = socket(AF_INET, SOCK_STREAM,
SOCKET_FLAGS_SSL);
                            }

                            /* Check if socket was created successfully */
                            if (tcp_client_socket == -1) {
                            printf("socket error.\r\n");
                            close(tcp_client_socket);
                            return -1;
                            }

                            /* Enable X509 bypass verification */
                            setsockopt(tcp_client_socket,

SOL_SSL_SOCKET,SO_SSL_BYPASS_X509_VERIF,&optVal,sizeof(optVal));

                            /* If success, connect to socket */
                            if (connect(tcp_client_socket, (struct sockaddr
*)&addr_in,
                            sizeof(struct sockaddr_in)) !=
                            SOCK_ERR_NO_ERROR) {
                            printf("connect error.\r\n");
                            return SOCK_ERR_INVALID;
                            }
```

- SO_SSL_SNI 命令用于设置服务器名称指示符（Server Name Indicator，SNI）。在 TLS 握手过程中，客户端可通过在（扩展）客户端问候中设置服务器名称来指示其尝试连接的主机名。SNI 允许服务器提供基于同一 IP 地址和 TCP 端口号的多个证书，因此可通过同一 IP 地址处理多个安全网站，而不需要所有网站使用同一证书。
- SO_SSL_ENABLE_SNI_VALIDATION 用于在已设置 SNI 的情况下使能 SNI 验证功能。默认情况下，服务器名称验证处于禁止状态。要启用服务器名称验证，必须由应用程序通过 setsockopt() 设置 SO_SSL_SNI 和 SO_SSL_ENABLE_SNI_VALIDATION，如示例代码片段所示。启用 SNI 验证后，SNI 将与所接收服务器证书中的公用名称（Common Name，CN）进行比较。如果提供的 SNI 与 CN 不匹配，则 ATWINC15x0 固件将强制关闭 SSL 连接。
  示例：

```
    #define MAIN_HOST_NAME     "www.google.com"
                            struct sockaddr_in addr_in;
                            int    optVal =1;
                            addr_in.sin_family = AF_INET;
                            addr_in.sin_port = _htons(MAIN_HOST_PORT);
                            addr_in.sin_addr.s_addr = gu32HostIp;

                            /* Create secure socket */
                            if (tcp_client_socket < 0) {
                            tcp_client_socket = socket(AF_INET, SOCK_STREAM,
SOCKET_FLAGS_SSL);
                            }

                            /* Check if socket was created successfully */
                            if (tcp_client_socket == -1) {
                            printf("socket error.\r\n");
                            close(tcp_client_socket);
                            return -1;
                            }

                            /* set SNI on SSL Socket */
                            setsockopt(tcp_client_socket, SOL_SSL_SOCKET,SO_SSL_SNI,
                            MAIN_HOST_NAME,sizeof(MAIN_HOST_NAME));
                            /* Enable SSL SNI validation */
                            setsockopt(tcp_client_socket, SOL_SSL_SOCKET,
                            SO_SSL_ENABLE_SNI_VALIDATION,&optVal,sizeof(optVal));

                            /* If success, connect to socket */
                            if (connect(tcp_client_socket, (struct sockaddr
*)&addr_in, sizeof(
                            struct sockaddr_in)) != SOCK_ERR_NO_ERROR) {
                            printf("connect error.\r\n");
                            return SOCK_ERR_INVALID;
                            }
```

- SO_SSL_ENABLE_SESSION_CACHING 命令允许 TLS 缓存会话信息以加快后续 TLS 会话建立的速度。
  示例：

```
    struct sockaddr_in addr_in;
                                int    optVal =1;
                                addr_in.sin_family = AF_INET;
                                addr_in.sin_port = _htons(MAIN_HOST_PORT);
                                addr_in.sin_addr.s_addr = gu32HostIp;

                                /* Create secure socket */
                                if (tcp_client_socket < 0) {
                                tcp_client_socket = socket(AF_INET, SOCK_STREAM,
SOCKET_FLAGS_SSL);
                                }

                                /* Check if socket was created successfully */
                                if (tcp_client_socket == -1) {
                                printf("socket error.\r\n");
                                close(tcp_client_socket);
                                return -1;
                                }
```

```
                                /* Enable SSL Session cache */
                                setsockopt(tcp_client_socket,
SOL_SSL_SOCKET,SO_SSL_ENABLE_SESSION_CACHING,&optVal,sizeof(optVal));

                                /* If success, connect to socket */
                                if (connect(tcp_client_socket, (struct sockaddr
*)&addr_in, sizeof(struct
                                sockaddr_in)) != SOCK_ERR_NO_ERROR) {
                                printf("connect error.\r\n");
                                return SOCK_ERR_INVALID;
                                }
```

⚠WARNING SO_SSL_BYPASS_X509_VERIF 仅用于调试和测试目的。建议不要在生产软件应用程序中使用此套接字选项。

6.2.3.13 gethostbyname

gethostbyname 函数用于通过域名系统（Domain Name System，DNS）将主机名（例如 URL）解析为主机 IP 地址（仅限于 IPv4 地址）。该操作取决于 DNS 服务器 IP 地址的配置以及通过 Internet 访问 DNS 层级的权限。

调用 gethostbyname 后，会对 DNS 解析器处理程序进行回调。如果确定了 IP 地址，则返回正值。如果无法确定 IP 地址或者 DNS 服务器不可访问（30 秒超时），则指示 IP 地址值 0。

注： IP 返回值为 0 时表示出现错误（例如，Internet 连接中断或 DNS 不可用），主机 MCU 应用程序可以稍后再次尝试调用函数 gethostbyname。

6.2.4 汇总

下表汇总了 ATWINC15x0 套接字 API 并给出了其与 BSD 套接字 API 的兼容性。

表 6-1. ATWINC15x0 套接字 API 汇总

BSD API	ATWINC15x0 API	ATWINC15x0 API 类型	服务器/客户端	TCP/UDP	简介
socket	socket	同步	两者	两者	创建新套接字。
connect	connect	异步	客户端	TCP	初始化对远程服务器的 TCP 连接请求。
bind	bind	异步	服务器	两者	将套接字绑定到地址（地址/端口）。
listen	listen	异步	服务器	TCP	允许绑定的套接字侦听其本地端口的远程连接。
accept	accept		弃用，隐式接受侦听。		
send	send	异步	两者	两者	发送数据包。
sendto	sendto	异步	两者	UDP	通过 UDP 套接字发送数据包。
write	-	不支持			

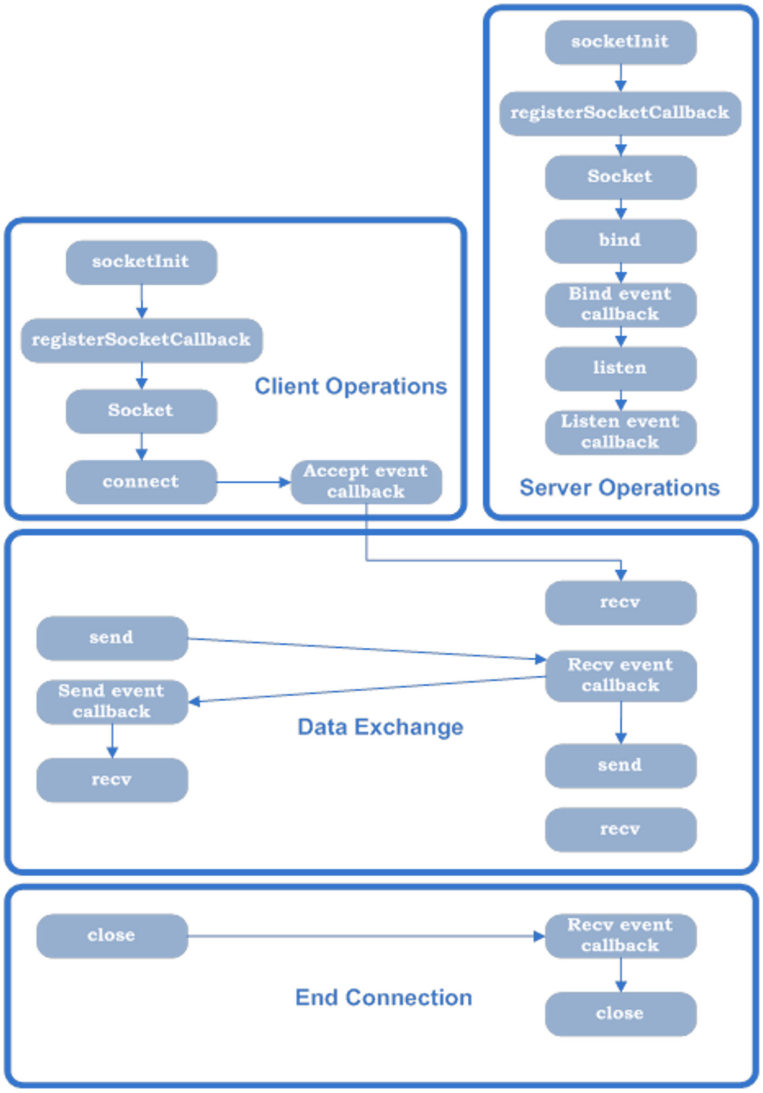
.....（续）

BSD API	ATWINC15x0 API	ATWINC15x0 API 类型	服务器/客户端	TCP/UDP	简介
recv	recv	异步	两者	两者	接收数据包。
recvfrom	recvfrom	异步	两者	两者	接收数据包。
read	-	不支持			
close	close	同步	两者	两者	终止 TCP 连接并释放系统资源。
gethostbyname	gethostbyname	异步	两者	两者	获取某个主机名的 IP 地址
gethostbyaddr	-	不支持			
select	-	不支持			
poll	-	不支持			
setsockopt	setsockopt	同步	两者	两者	设置套接字选项。
getsockopt		不支持			
htons/ntohs	_htons/_ntohs	同步	两者	两者	将 2 字节整数在主机表示与网络字节顺序表示之间相互转换。
htonl/ntohl21	_htonl/_ntohl	同步	两者	两者	将 4 字节整数在主机表示与网络字节顺序表示之间相互转换。

6.3 套接字连接流程

以下小节将详细介绍 TCP 和 UDP（客户端和服务器）操作。

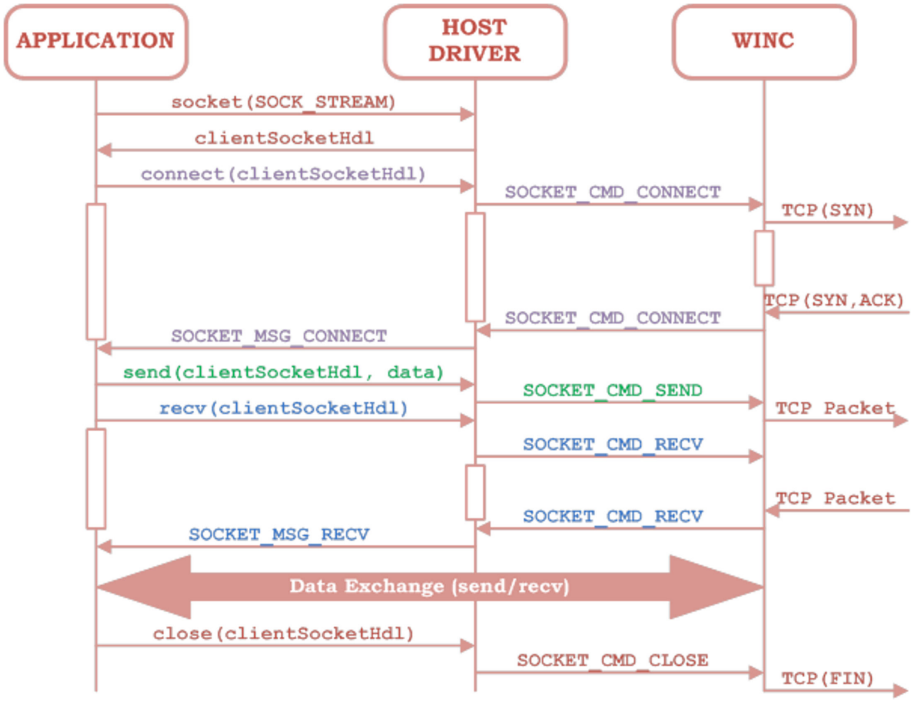
图 6-3. 典型套接字连接流程



6.3.1 TCP 客户端操作

下图所示为使用 TCP 客户端传输数据的流程。

图 6-4. TCP 客户端序列图

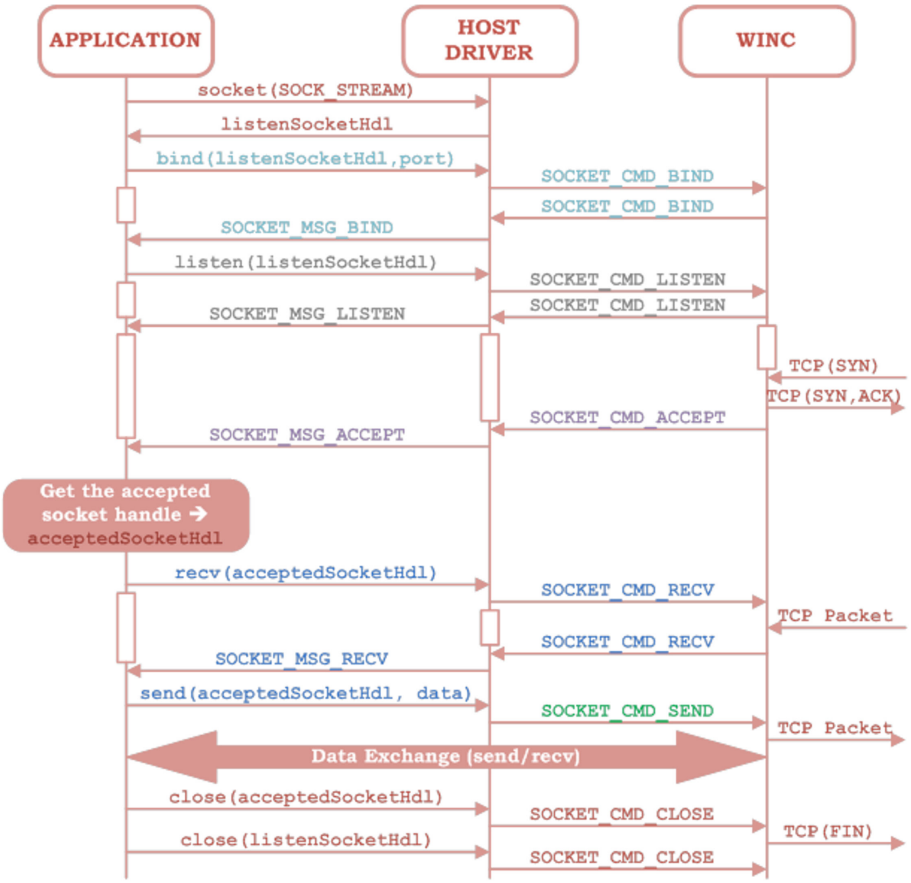


注：

1. 主机应用程序必须注册套接字通知回调函数。该函数必须为 `tpfAppSocketCb` 类型，并且必须适当地处理套接字事件通知。
2. 如果客户端知道服务器的 IP，则可以直接调用 `connect`，如上图所示。如果只知道服务器 URL，则应用程序必须先调用 `gethostbyname` API 来解析服务器 URL。

6.3.2 TCP 服务器操作

图 6-5. TCP 服务器序列图

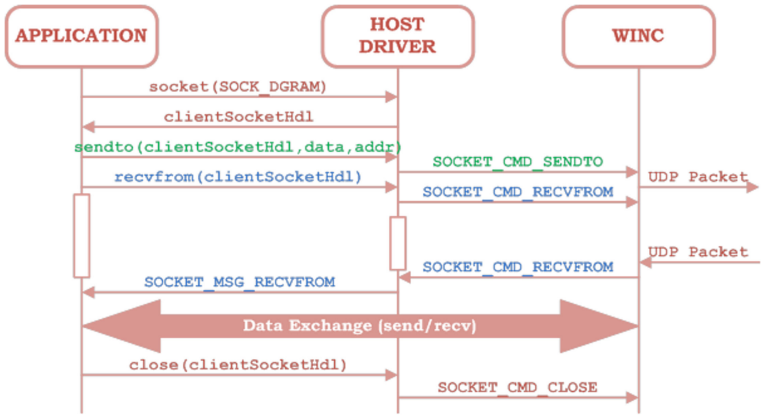


注： 主机应用程序必须注册套接字通知回调函数。该函数必须为 `tpfAppSocketCb` 类型，并且必须适当地处理套接字事件通知。

6.3.3 UDP 客户端操作

下图所示为使用 UDP 客户端传输数据的流程。

图 6-6. UDP 客户端序列图



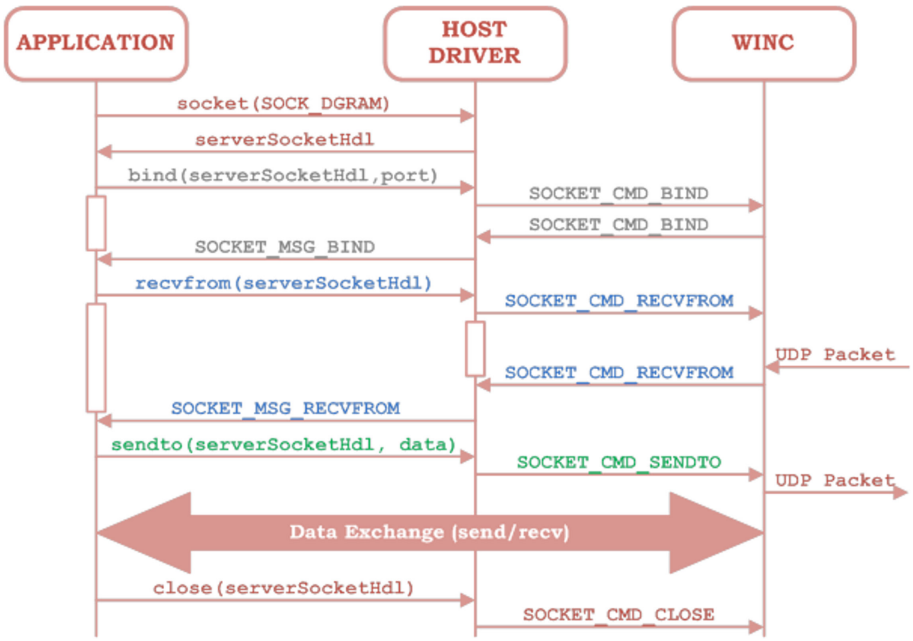
注：

1. 必须使用具有指定目标地址的 sendto API 发送第一条报文。
2. 如果后续报文将发送到同一地址，也可使用 send API。更多详细信息，请参见 send。
3. 可使用 recv 代替 recvfrom。

6.3.4 UDP 服务器操作

下图所示为建立 UDP 服务器后传输数据的流程。

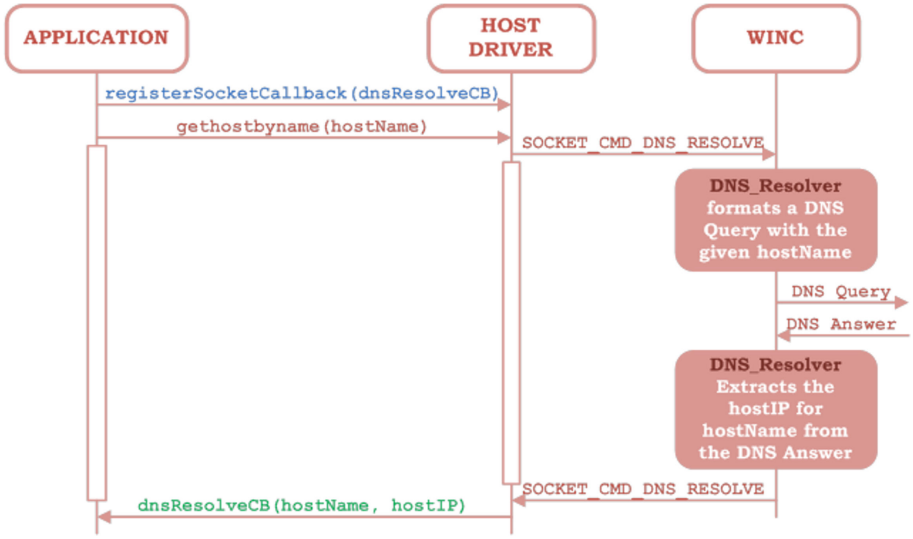
图 6-7. UDP 服务器序列图



6.3.5 DNS 主机名解析

下图所示为 DNS 主机名解析流程。

图 6-8. DNS 解析序列



注：

1. 主机应用程序通过调用函数 gethostbyname 请求解析主机名（例如 http://www.foobar.com）。
2. 在调用 gethostbyname 之前，应用程序必须使用函数 registerSocketCallback 注册 DNS 响应回调函数。
3. 在 ATWINC15x0 DNS_Resolver 模块获得给定主机名对应的 IP 地址（hostIP）后，调用 dnsResolveCB 将得到 hostIP。
4. 如果发生错误或 DNS 请求发生超时，则调用 dnsResolveCB 将得到 IP 地址值 0，表示无法解析域名。

6.4 示例代码

本节给出了不同套接字应用程序的代码示例。有关更多套接字代码示例，请参见 Wi-Fi Network Controller Software Programming Guide。

6.4.1 TCP 客户端示例代码

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


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

                // Fill in the acSendBuffer with some data here

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

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

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

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

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

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

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


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

6.4.2 TCP 服务器示例代码

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

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

            recv(acceptedSocketHdl, rxBuffer, sizeof(rxBuffer), 0);
        }
        else
        {
```

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

            uint8    acSendBuffer[256];
            uint16    u16MsgSize;

            // Fill in the acSendBuffer with some data here

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

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

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

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

    // Initialize the socket layer.
    socketInit();

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

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

6.4.3 UDP 客户端示例代码

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

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

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

```
            close(clientSocketHdl);
        }
    }
}

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

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

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

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

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

6.4.4 UDP 服务器示例代码

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

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

            // Fill in the acSendBuffer with some data

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

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

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

```
}

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

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

7. 传输层安全（TLS）

传输层安全（Transport Layer Security，TLS）层位于 TCP 之上，用于提供安全服务，包括隐私、真实性和报文完整性。WINC 固件中提供多种基于 TLS 的安全方法。

7.1 TLS 概述

ATWINC15x0 在 WINC 固件中绑定了内存占用较低的嵌入式 TLS 协议栈。

该协议栈具有以下功能：

- 支持 TLS 版本 TLS1.0、TLS1.1 和 TLS1.2。
- 支持 TLS 客户端身份验证。
- 支持 TLS 服务器模式。
- TLS 协议栈的简单应用程序接口。TLS 功能通过 ATWINC15x0 套接字接口抽象化，向应用程序开发人员隐藏了实现复杂性，并最大限度地减少了将现有纯 TCP 代码移植到 TLS 的工作量。

7.2 建立 TLS 连接

从应用程序的角度来看，TLS 功能被抽象化为套接字 API。这向应用程序隐藏了 TLS 的复杂性，应用程序可以按照与 TCP（非 TLS）客户端和服务器相同的方式使用 TLS。TLS 套接字和常规 TCP 套接字之间的主要区别在于，应用程序在创建 TLS 客户端和服务器侦听套接字时设置了 `SOCKET_FLAGS_SSL`。下图显示了建立 TLS 连接的详细序列。

注：

- 要正确进行 TLS 客户端操作，请确保在 TLS 客户端应用程序中设置 `SOCKET_FLAGS_SSL` 标志和正确的端口号。例如，HTTP 客户端应用程序在调用 `socket` API 和 `connect` 函数（至端口 80）时不使用标志。如果使用标志 `SOCKET_FLAGS_SSL` 并将 `connect` API 中的端口号更改为端口 433，则相同的应用程序源代码将变为 HTTPS 客户端应用程序。
- 要正确进行 TLS 服务器操作，请确保在 TLS 服务器应用程序中设置 `SOCKET_FLAGS_SSL` 标志和正确的端口号。例如，HTTP 服务器应用程序在调用 `socket` API 和 `bind` 函数（至端口 80）时不使用标志。如果使用标志 `SOCKET_FLAGS_SSL` 并将 `bind` API 中的端口号更改为端口 443，则相同的应用程序源代码将变为 HTTPS 服务器应用程序。

图 7-1. 建立 TLS 客户端应用程序连接

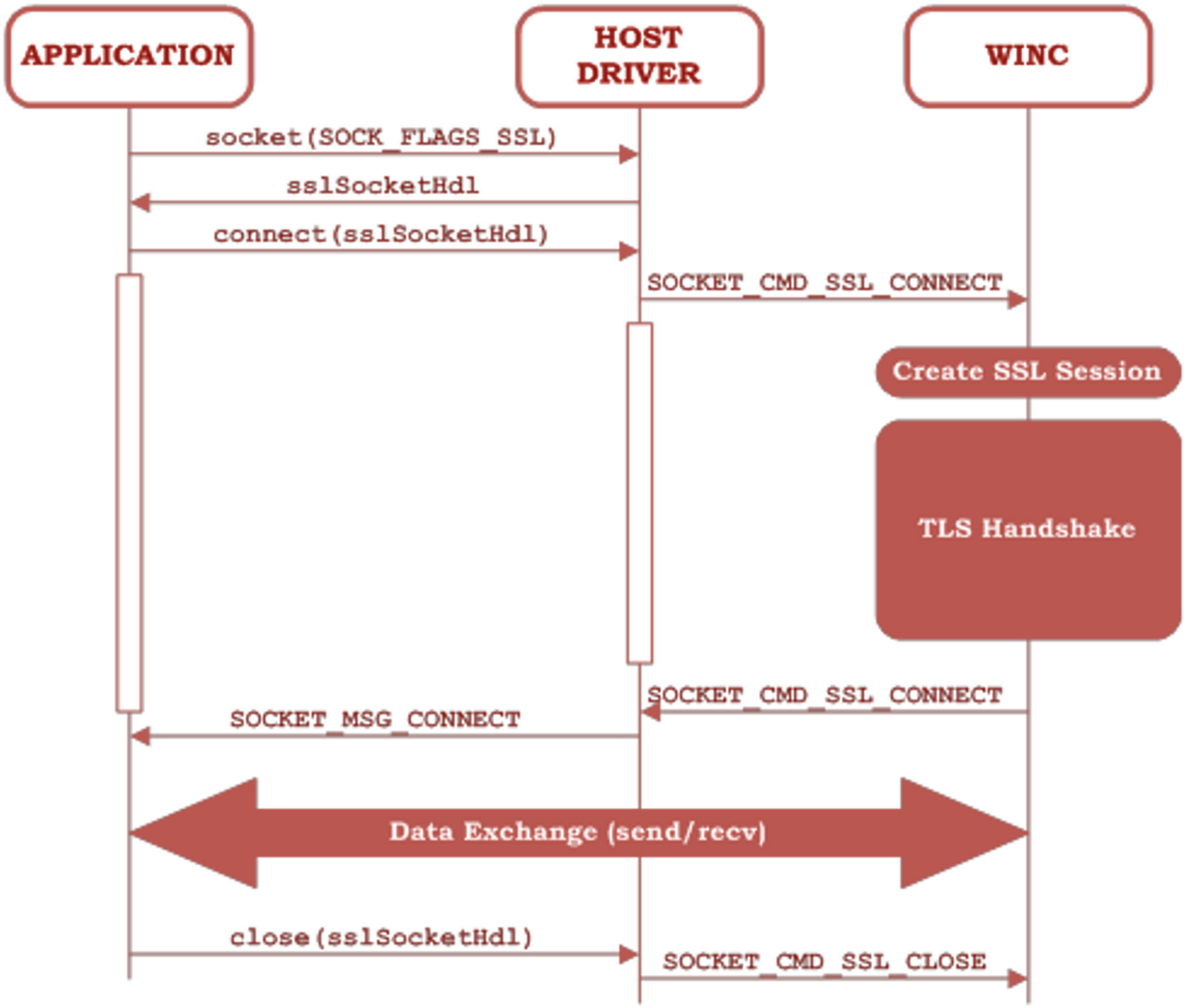
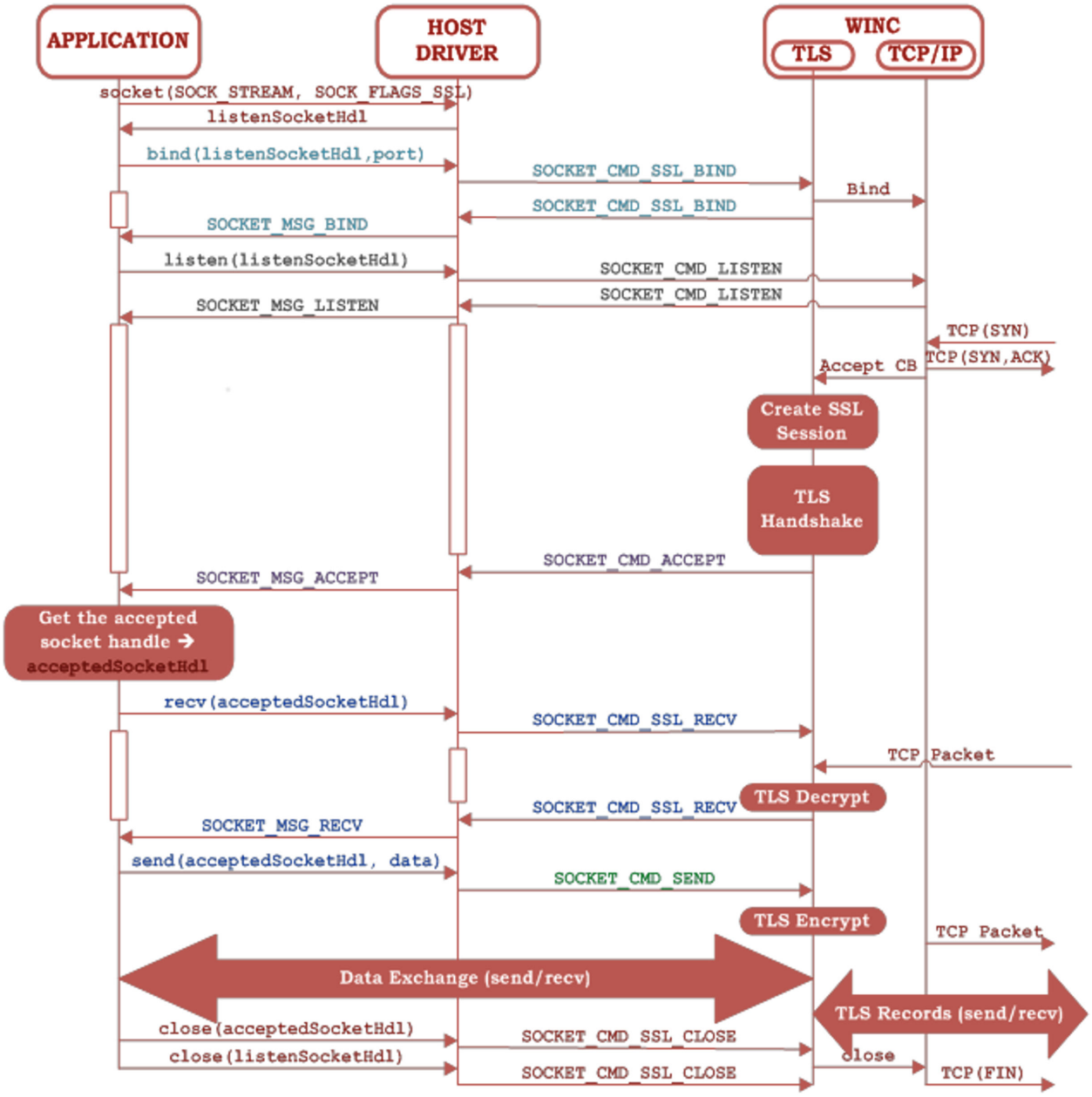


图 7-2. 建立 TLS 服务器应用程序连接



7.3 安装服务器证书

7.3.1 技术背景

7.3.1.1 公钥基础架构

TLS 安全基于公钥基础架构 PKI，其中：

- 服务器将其公钥存储在采用 X.509 标准格式的数字证书中。
- 服务器的 X.509 证书必须由证书颁发机构（Certificate Authority，CA）颁发，而该 CA 可由其他 CA 认证。

- 这种结构形成了一种称作信任链的 X.509 证书链。
- 众所周知，链中最顶级的 CA 是链中的*信任根证书颁发机构*。

7.3.1.2 TLS 服务器身份验证

- 当 TLS 客户端发起与服务器的连接时，服务器会将其 X.509 证书链（可能包括也可能不包括根证书）发送到客户端。
- 客户端必须在开始数据交换之前对服务器进行身份验证（验证服务器身份）。
- 客户端必须验证整个证书链，并验证证书链的根证书颁发机构是否在客户端的信任根证书存储区中。

7.3.2 将证书添加到 WINC 信任根证书存储区

- 在连接到 TLS 服务器之前，必须在 ATWINC15x0 上安装服务器的根证书。否则，WINC 将在本地中止与服务器的 TLS 连接。
- 根证书必须采用 **DER** 格式，否则必须在安装前进行转换。有关证书格式和转换方法的信息，请参见**第 17 章“如何生成证书”**。
- 要安装证书，请执行 **root_certificate_downloader.exe**，语法如下：

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

7.4 WINC TLS 限制

7.4.1 并发连接

仅允许 2 个 TLS 并发连接。

7.4.2 TLS 支持的密码

ATWINC15x0 支持以下密码套件（对客户端和服务器模式均适用）。

- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA256

ATWINC15x0 还可选择支持以下 ECC 密码套件。

- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ ECDSA _WITH_AES_128_CBC_SHA256

7.4.3 支持的哈希算法

当前实现（WINC 固件版本 19.5.2 及更高版本）支持以下哈希算法：

- MD5
- SHA-1
- SHA256
- SHA384
- SHA512
- RSA 4096

7.4.4 TLS 证书约束

对于 TLS 服务器和 TLS 客户端身份验证，ATWINC15x0 可接受以下证书类型：

- 密钥大小不超过 2048 位的 RSA 证书
- 仅适用于 NIST P256 EC 曲线（secp256r1）的 ECDSA 证书；有条件的支持

7.4.5 ECC 密码套件

ATWINC15x0 TLS 库支持 ECC 密码套件。ATWINC15x0 器件不包含用于 ECC 数学运算的内置硬件加速器，但 WINC TLS 库可利用主机 MCU 提供的 ECC 数学运算。为了执行 ECC 密码所需的 ECC 计算，主机 MCU 上必须提供 ECC 硬件加速器（或软件库）。

WINC TLS 使用默认禁止的 ECC 密码套件进行初始化。主机 MCU 应用程序可通过 API `sslSetActiveCipherSuites` 使能密码。

7.5 SSL 客户端代码示例

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

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

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

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

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

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

    if(u32ServerIP != 0)
    {
```

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

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

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

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

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

8. Wi-Fi AP 模式

8.1 概述

本章概述了 WINC 接入点（Access Point，AP）模式，并介绍了如何设置此模式和配置其参数。

在 ATWINC1500 v19.6.1 固件及更高版本中，进入 AP 和配置模式时，可自定义 DHCP 默认网关、DNS 服务器和子网掩码。在旧版本中，默认网关和 DNS 服务器与 WINC 的主机 IP 相同，子网掩码为 255.255.255.0。通过配置这些值，可使用 0.0.0.0 作为默认网关和 DNS 服务器，以便移动设备能够连接到 WINC AP 而无需断开与移动网络的连接。可使用除 0.0.0.0 外的 IP，但无任何作用，因为任何时候都只有一个设备可连接到 WINC AP。

8.2 设置 WINC AP 模式

使用 `tstrM2MAPConfig` 结构设置 WINC AP 模式配置参数。

可通过以下两种函数来启用/禁止 WINC AP 模式：

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

有关 API 的更多详细信息，请参见 Atmel Software Framework for ATWINC1500 (Wi-Fi)。

在 ATWINC1500 v19.6.1 固件及更高版本中，为了保持与旧版驱动程序的向后兼容性，引入了新的结构和 API。

要在进入 AP 或配置模式时自定义这些字段，必须填充 `tstrM2MAPModeConfig` 结构并将其传送给新的 `m2m_wifi_enable_ap_ext()` 或 `m2m_wifi_start_provision_mode_ext()` API。`tstrM2MAPModeConfig` 结构包含用于存储 AP SSID 和密码等信息的原始 `tstrM2MAPConfig` 结构以及另一个用于配置默认路由器、DNS 服务器和子网掩码的 `tstrM2MAPConfigExt` 结构。

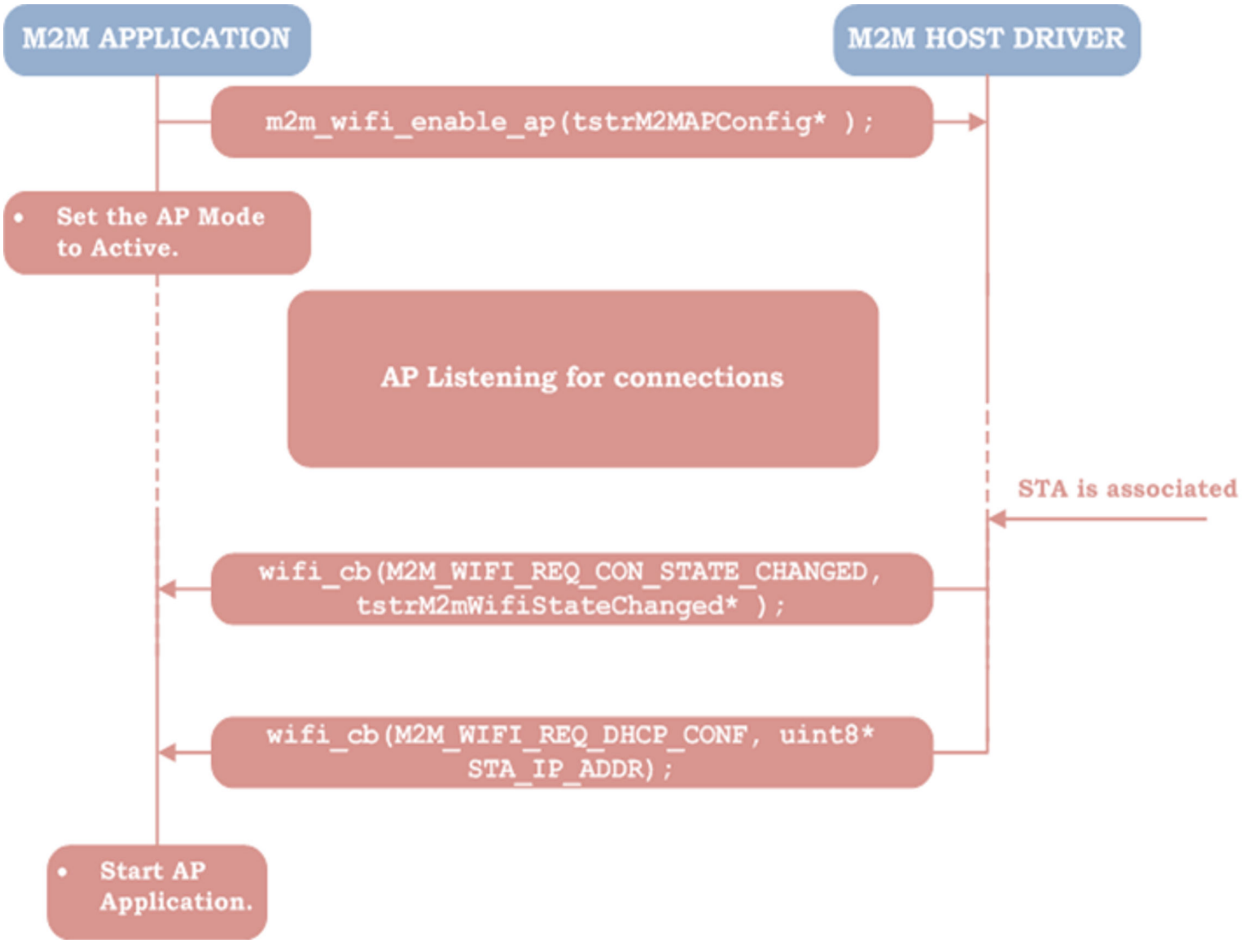
8.3 限制

- AP 只能支持单个关联站。它会拒绝进一步的连接尝试。
- ATWINC15x0 支持 WPA2 安全功能（自固件版本 19.5.x 起）。
- 不支持并发（同时使用 STA 和 AP 模式）。在激活 AP 模式之前，主机 MCU 应用程序必须禁止当前正在运行的模式。

8.4 序列图

建立 AP 模式后，在站关联到 AP 之前，数据接口不存在；因此，应用程序需要等待，直到通过事件回调函数接收到通知。该过程如下图所示。

图 8-1. ATWINC15x0 AP 模式建立



8.5 AP 模式代码示例

以下示例说明了如何在 ATWINC15x0 AP 模式下将 WINC_SSID 配置为支持开放式通道 1 上的广播 SSID 以及将 IP 地址配置为 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);
        }
    }
}
```

注： 在 ATWINC15x0 AP 模式下，不支持节能模式。

9. 配置

为了确保 ATWINC15x0 器件正常工作，需要为其加载某些参数。特别是在站模式下工作时，该器件必须知道其需要连接的接入点的身份（SSID）和凭证。通过以下配置步骤可轻松输入这类信息。

当前 ATWINC15x0 软件支持以下配置方法：

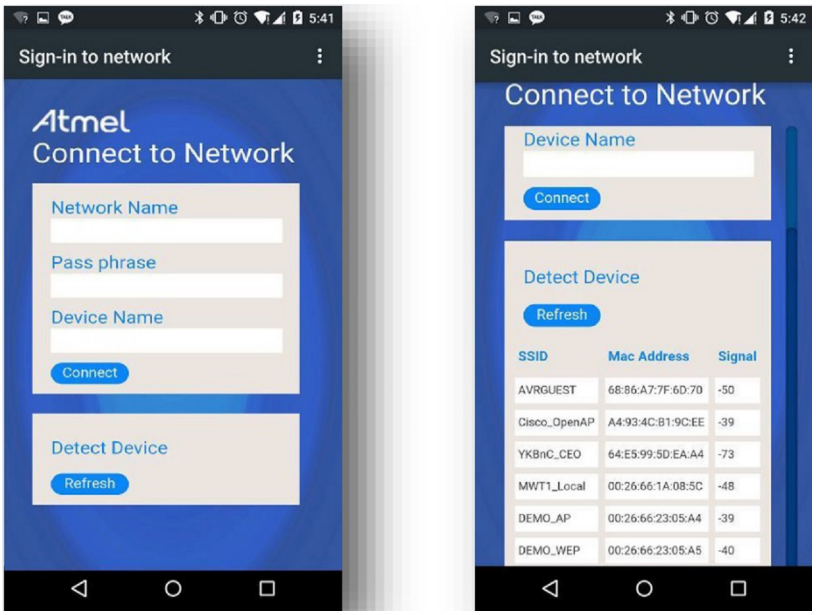
- 基于 HTTP（浏览器）的配置，WINC 处于 AP 模式
- Wi-Fi 保护设置（WPS）

9.1 HTTP 配置

在该方法中，ATWINC15x0 处于 AP 模式，根据说明将其他支持浏览器功能的设备（手机、平板电脑和 PC 等）连接到 ATWINC15x0 HTTP 服务器。建立连接后，可以输入所需的配置。

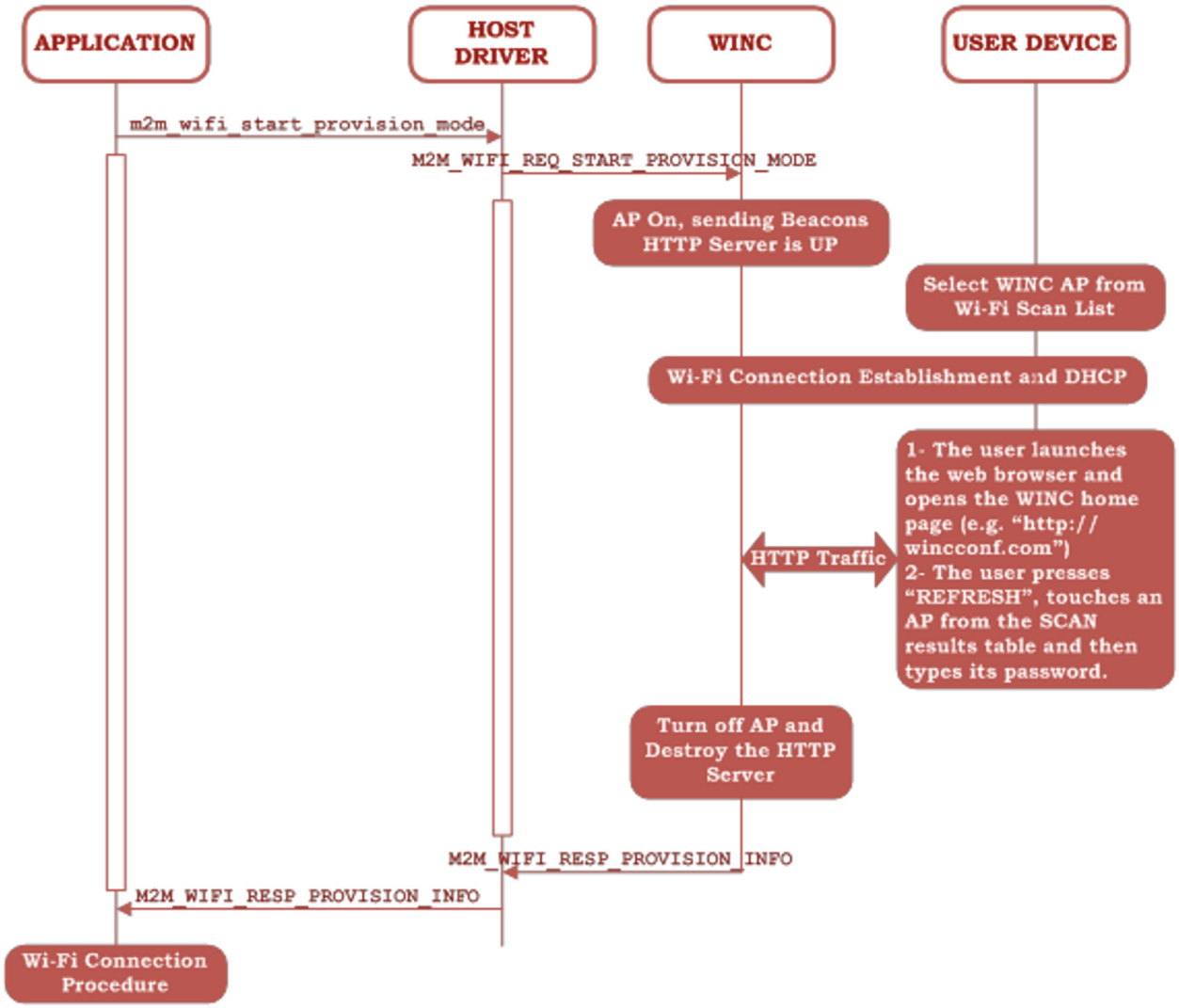
HTTP 配置主页如下图所示。

图 9-1. ATWINC15x0 HTTP 配置页面



9.1.1 配置控制流程

图 9-2. HTTP 配置序列图



上图所示为 WINC 器件的配置操作。详细步骤的说明如下：

1. WINC 器件启动 HTTP 配置模式。
2. 智能手机用户在 Wi-Fi 搜索列表中找到 WINC AP SSID。
3. 用户连接到 WINC AP。
4. 用户启动 Web 浏览器并将 WINC 主页写入地址栏。
5. 如果 `m2m_wifi_start_provision_mode` API 中的 HTTP 重定向位（`bEnableHttpRedirect`）置 1，则关联设备（手机和 PC 等）的所有 http 流量（http://URL）都将重定向到 WINC HTTP 配置主页。有些手机会显示一条“登录 Wi-Fi 网络？”的通知消息，接受后会自动加载 WINC 主页。浏览器上将显示 WINC 主页，如图 10.1 所示。
6. 要发现相应区域中的 Wi-Fi AP 列表，用户可以按“Refresh”（刷新）。
7. 然后从搜索列表中选择所需的 AP（通过单击或触摸），其名称将自动显示在“Network Name”（网络名称）文本框中。

8. 然后，用户必须在“Pass Phrase”（密码）文本框中输入正确的 AP 密码（WPA/WPA2 Personal 安全密码）。如果所需的 AP 为开放式（M2M_WIFI_SEC_OPEN），则 Pass Phrase 字段保留为空。
9. 如果需要，用户可以在“Device Name”（器件名称）文本框中选择性地配置 WINC 器件名称。
10. 然后用户应按 **Connect**（连接）。

WINC 关闭 AP 模式并开始连接到配置的 AP。

9.1.2 HTTP 重定向功能

ATWINC15x0 HTTP 配置服务器支持 HTTP 重定向功能，该功能会将来自关联用户设备的所有 HTTP 流量强制重定向到 ATWINC15x0 配置主页。

这样便无需键入 HTTP 配置服务器的确切 Web 地址，简化了加载配置页面的机制。

要启用该功能，请在调用 API `m2m_wifi_start_provision_mode` 时将重定向标志置 1。更多详细信息，请参见以下代码示例。

9.1.3 配置代码示例

```
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);
```

```
            }
        }
}
```

9.2 限制

目前的 HTTP 配置实现具有以下限制：

- ATWINC15x0 AP 限制适用于配置模式。 有关 AP 模式限制列表，请参见限制。
- 配置模式使用开放式 AP 模式，不使用 Wi-Fi 安全设置或应用程序级安全设置（例如，TLS）；因此，用户输入的 AP 凭证将显式发送，窃听者可以看到。
- WINC 配置主页是一个静态 HTML 页面。WINC HTTP 服务器不允许服务器端脚本。
- 只能配置具有 WPA-personal 安全（基于密码）和无安全（开放网络）的 AP。无法配置 WEP 和 WPA-Enterprise AP。
- 配置负责向应用程序提供连接参数，而应用程序负责确保连接过程和连接参数的有效性。

9.3 Wi-Fi 保护设置（WPS）

大多数现代接入点都支持 Wi-Fi 保护设置方法，并且通常使用按钮来实现。从用户的角度来看，WPS 是一种简单的机制，可以使设备安全地连接到 AP，而且无需记住密码。WPS 使用非对称加密形成临时安全链路，随后使用该链路将密码（和其他信息）从 AP 传输到新站。传输完成后，即可针对普通静态 PSK 配置建立安全连接。

9.3.1 WPS 配置方法

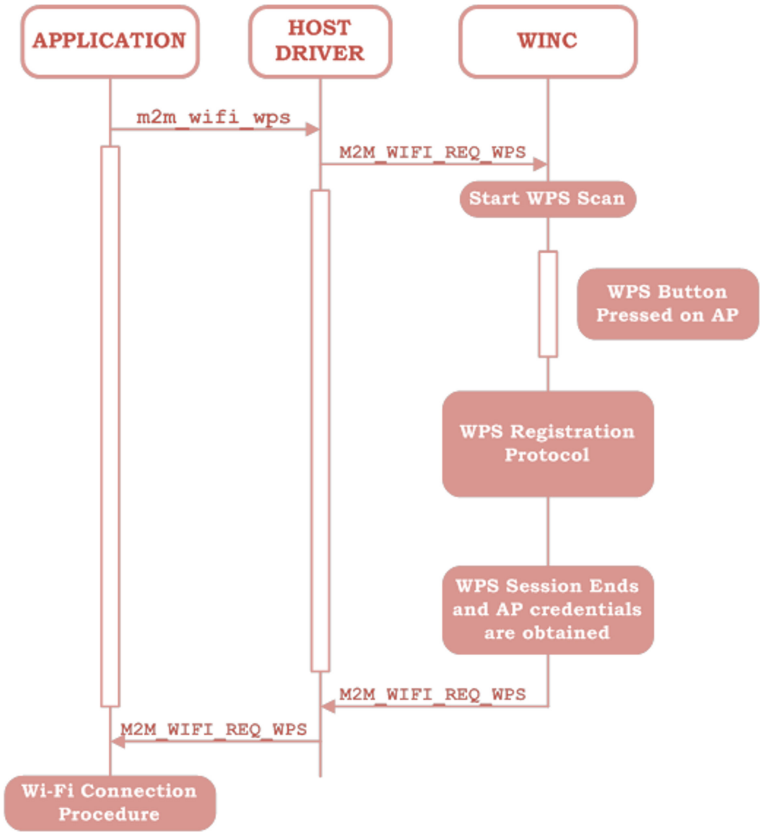
有两种身份验证方法可与 WPS 配合使用：

1. PBC（按钮）方法——按下 AP 上的物理按钮，使 AP 进入 WPS 模式并持续一段有限的时间。在 ATWINC15x0 上通过输入参数 `WPS_PBC_TRIGGER` 调用 `m2m_wifi_wps`，继而启动 WPS。
2. PIN 方法——任何情况下都可以使用 AP 启动 WPS，但用户需要知道 8 位 PIN，PIN 通常印在 AP 主体上。由于 WINC 通常用于无头设备（无用户界面），因此必须反向执行此过程并强制 AP 使用随 WINC 器件提供的 PIN 号。某些 AP 允许通过配置更改 PIN。通过使用输入参数 `WPS_PIN_TRIGGER` 调用 `m2m_wifi_wps` 在 ATWINC15x0 上启动 WPS。鉴于这种方法比较有难度，大多数应用程序均不建议使用该方法。

WPS 操作的报文和操作流程如下图所示。

9.3.2 WPS 控制流程

图 9-3. 按钮触发的 WPS 操作



9.3.3 WPS 限制

- WPS 仅用于传输 WPA/WPA2 密钥；不支持其他安全类型。
- 如果按下邻近的多个 AP 上的 WPS 按钮，则 WPS 标准将拒绝会话（WPS 响应失败），而应用程序可以在几分钟后再次尝试。
- 如果没有按下 AP 上的 WPS 按钮，则 WPS 扫描将在第一次 WPS 触发起的两分钟后超时。
- WPS 负责向应用程序提供连接参数，而应用程序负责确保连接过程和连接参数的有效性。

9.3.4 WPS 代码示例

```
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);
        }
    }
}
```

10. 无线升级

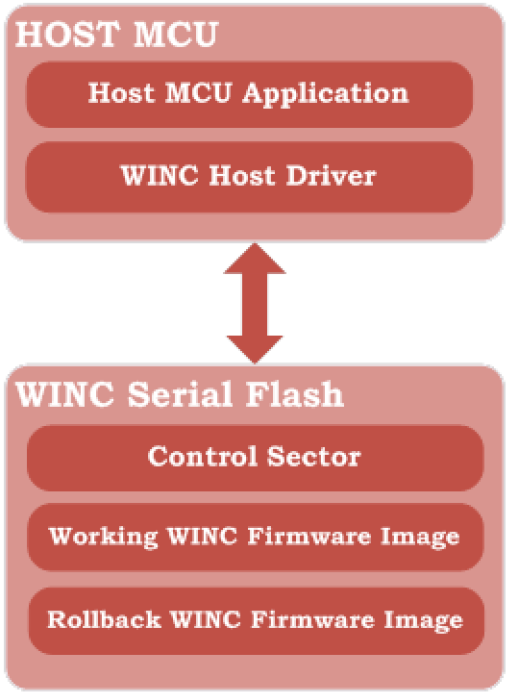
10.1 概述

ATWINC15x0 支持在内部串行闪存上对固件进行 OTA 升级。无需主机闪存资源即可存储固件。ATWINC15x0 使用内部 HTTP 客户端从远程服务器检索固件。

10.2 OTA 镜像架构

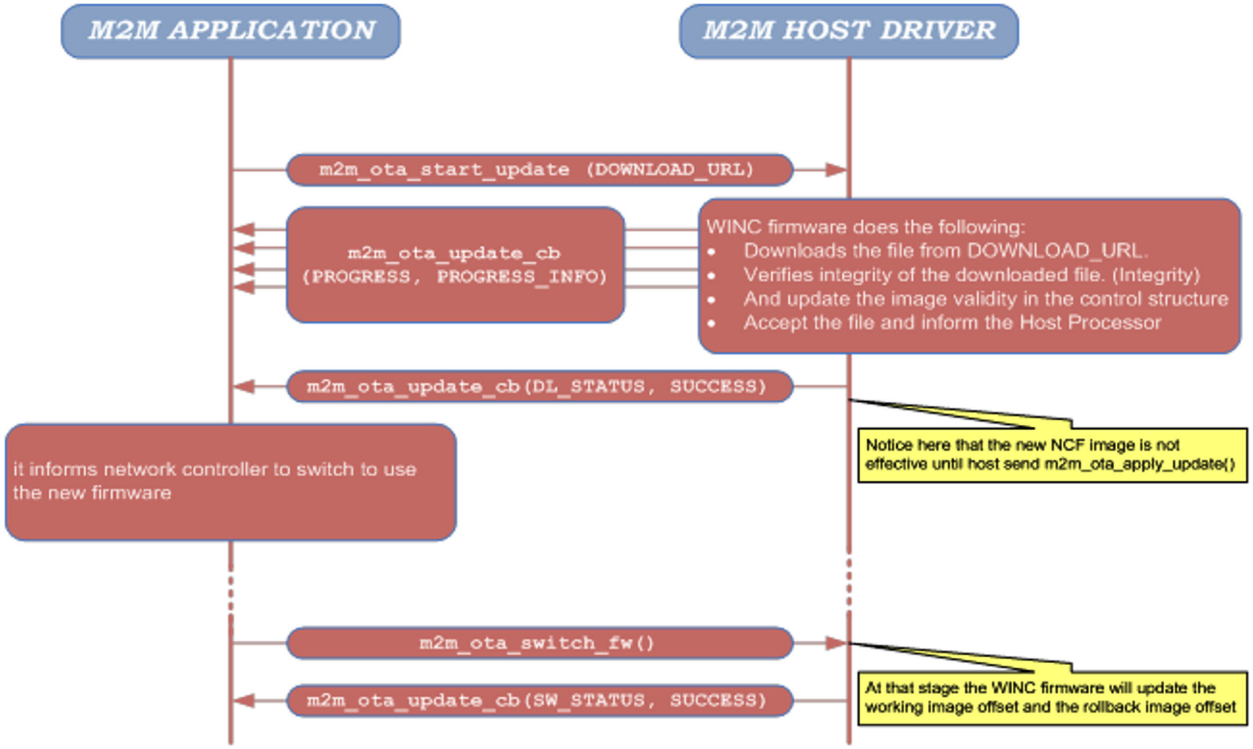
WINC 串行闪存可以存储固件镜像的两个副本：工作镜像和回滚镜像。首次引导时，工作镜像是出厂镜像，回滚镜像在 WINC 闪存中不可用。而 ATE 固件将在回滚镜像固件部分中可用。执行 OTA 固件升级后，ATE 固件将被擦除，新接收的固件将写入回滚镜像部分。在 OTA 升级期间，WINC 的内部存储空间不足，无法将整个镜像保存在 RAM 中；因此，接收到的每个已下载数据块都会写入闪存。如果 OTA 失败，现有（工作）镜像将被保留，而回滚镜像将变为无效。如果传输成功，则会更新闪存控制结构以反映新的工作镜像，现有镜像将标记为有效的回滚镜像。

图 10-1. OTA 镜像构成



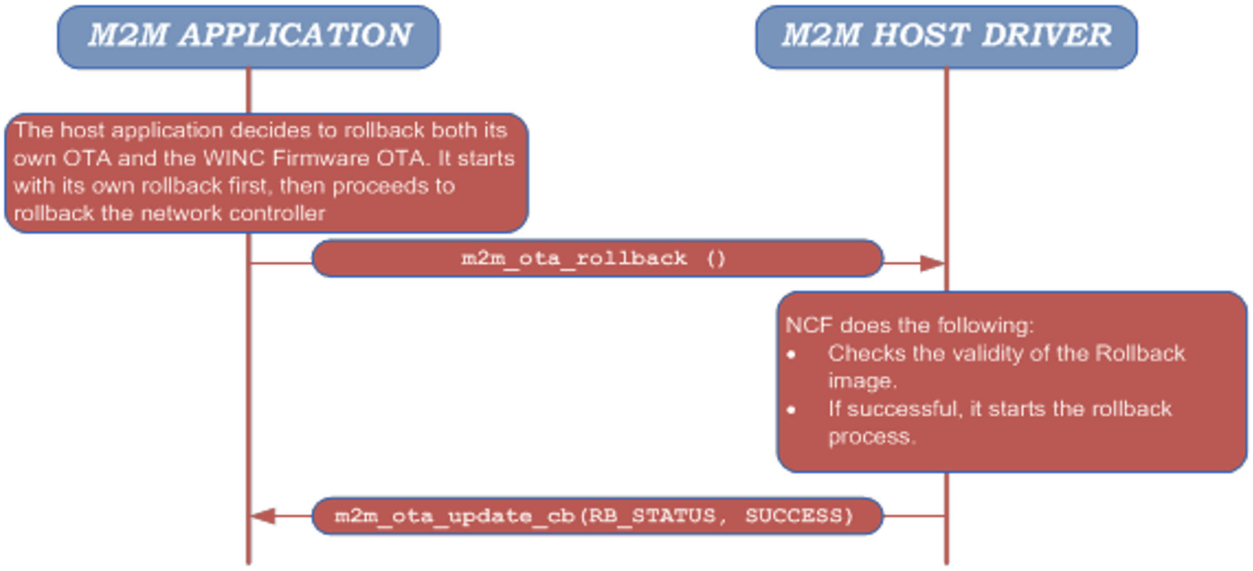
10.3 OTA 下载序列图

图 10-2. OTA 镜像下载和安装



10.4 OTA 固件回滚

图 10-3. OTA 镜像回滚序列



10.5 OTA 限制

- 必须至少成功下载 OTA 一次才能回滚。
- 无论新的 OTA 尝试成功或失败，都会覆盖回滚镜像。

10.6 OTA 代码示例

```
/*!<OTA update callback typedef> */
static void OtaUpdateCb(uint8 u8OtaUpdateStatusType ,uint8 u8OtaUpdateStatus)
{
    if(u8OtaUpdateStatusType == DL_STATUS)
    {
        if(u8OtaUpdateStatus == OTA_STATUS_SUCSESS)
        {
            //switch to the upgraded firmware
            m2m_ota_switch_firmware();
        }
    }
    else if(u8OtaUpdateStatusType == SW_STATUS)
    {
        if(u8OtaUpdateStatus == OTA_STATUS_SUCSESS)
        {
            M2M_INFO("Now OTA suceesfully done");
            //start the host SW upgrade then system reset is required (Reintilize the driver)
        }
    }
}

void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    case M2M_WIFI_REQ_DHCP_CONF:
    {
        //after suceesfull connection, start the over air upgrade
        m2m_ota_start_update(OTA_URL);
    }
    break;
    default:
    break;
}

int main (void)
{
    tstrWifiInitParam param;
    tstr1xAuthCredentials gstrCred1x    = AUTH_CREDENTIALS;
    nm_bsp_init();
    m2m_memset((uint8*)&param, 0, sizeof(param));
    param.pfAppWifiCb = wifi_event_cb;

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

注： 有关示例代码的更多详细信息，请参见 Wi-Fi Network Controller Software Programming Guide。

11. 多播套接字

11.1 概述

多播过滤器用于提供向多播地址发送报文或从中接收报文的功能。此功能适用于通过网络进行一对多通信，适合在单次传送中将 Internet 协议（Internet Protocol，IP）数据报发送给一组相关接收器、参与零配置联网或侦听多播流或者任何其他应用。

11.2 如何使用过滤器

每当应用程序希望使用多播 IP 地址时，无论是发送还是接收，都需要使用过滤器。应用程序可通过为所需套接字设置 IP_ADD_MEMBERSHIP 选项来实现此目的，该套接字附带应用程序要使用的多播地址。如果随后主机想要停止接收多播流，则可为附带多播地址的所需套接字设置 IP_DROP_MEMBERSHIP 选项。

添加或删除多播地址过滤器时，WINC 芯片固件会添加/删除 MAC 层过滤器和 IP 层过滤器，以便同意或拒绝报文到达主机。

11.3 多播套接字代码示例

下面通过一个简单示例来说明这项功能：主机应用程序响应计算机/移动应用程序发送的 mDNS（多播域名系统）查询。计算机/移动应用程序搜索支持由 mDNS 响应指示的零配置服务的器件。WINC 将作出响应，表示自己存在并能够发送和接收多播报文。

示例中包含一个 UDP 服务器，该服务器绑定在端口 5353（mDNS 端口）上并等待报文，随后会解析报文并使用先前保存的响应报文进行回复。

- 服务器初始化：

```
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);
}
```

- 套接字事件处理程序：

```
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");
    }
}
```

- 服务器套接字回调函数：

```
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");
    }
}
```

- 解析 mDNS 查询：

```
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;
}
```

- 发送 mDNS 响应：

```
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);
}
```

- 服务名称：

```
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"
```

12. WINC 串行闪存

12.1 概述和特性

ATWINC1500 和 ATWINC1510 这两款 WINC 的内部串行（SPI）闪存容量分别为 4 Mb 和 8 Mb。闪存用于存储：

- 用户配置
- 固件
- 连接配置文件

在启动和模式切换期间，固件从串行闪存装入执行固件的程序存储器（IRAM）中。运行期间的其他时刻将访问闪存以检索配置和配置文件数据。

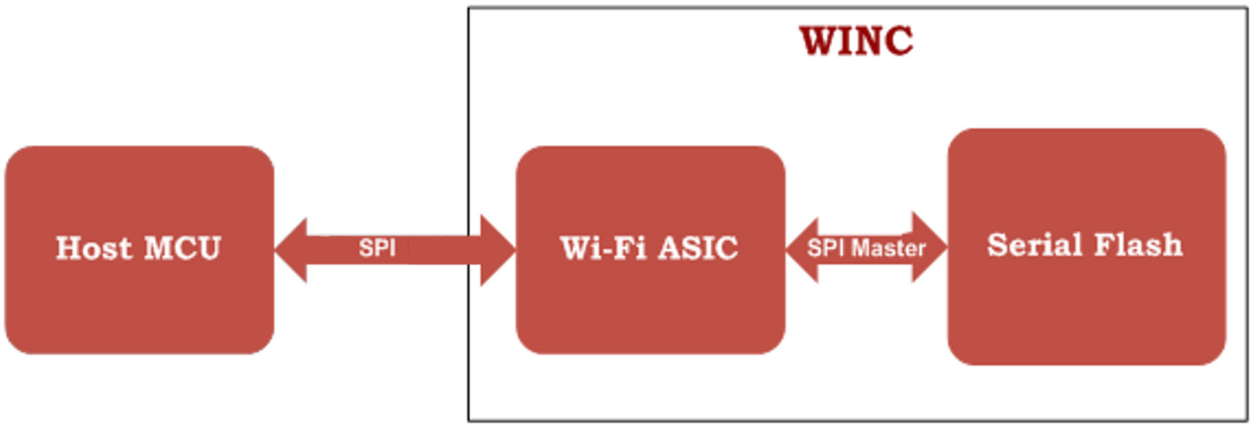
OTA 功能至少需要 4 Mb 闪存来存储工作镜像和回滚镜像。

可以直接通过主机读取、写入和擦除闪存，无需使用 WINC 固件。但是，如果已装入工作固件，则必须先停止任何正在运行的 WINC 固件，然后再访问串行闪存，这样可避免主机与 WINC 处理器之间发生访问冲突。

12.2 访问串行闪存

- 主机可通过 WINC SPI 主器件透明地访问串行（SPI）闪存。
- 主机可对串行（SPI）闪存进行编程，无需使用 WINC 中的工作固件。必须先调用函数 `m2m_wifi_download_mode`。

图 12-1. 系统框图（显示 SPI 闪存连接）



12.3 读/写/擦除操作

可以访问 SPI 闪存进行读取、写入和擦除操作。

在尝试访问 SPI 闪存之前，需要先通过调用以下函数将 WINC 的模式切换为下载模式：

```
sint32 m2m_wifi_download_mode();
```

所有 SPI 闪存函数都是阻塞函数。返回 `M2M_SUCCESS` 表示所请求的操作成功完成。

下面列出了可使用的闪存函数：

- 查询 SPI 闪存的大小：

```
uint32 spi_flash_get_size();
```

该函数将返回 SPI 闪存的大小（单位为 Mb）。

- 从 SPI 闪存读取数据：

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

其中，数据大小受 SPI 闪存大小的限制。

- 擦除 SPI 闪存中的扇区：

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

注： 大小受 SPI 闪存大小的限制。

在写入任何扇区之前，需先擦除该扇区。如果需要在扇区内更改某些数据，建议先读取扇区，修改数据，然后再擦除并写入整个扇区。

- 向 SPI 闪存写入数据：

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

如果应用程序想要在任何扇区内写入任意数量的字节，则必须先擦除整个扇区。可能有必要读取整个扇区，擦除扇区，然后回写修改。另外，建议在函数返回成功信息后验证是否已写入数据，方法是再次读取数据并将其与原始值进行比较。

12.3.1 闪存读取、擦除、写入代码示例

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

int main()
{
    uint8    au8FlashContent[FLASH_SECTOR_SZ] = {0};
    uint32u32FlashTotalSize = 0, u32FlashOffset = 0;

    // Platform specific initializations.

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

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

        ret = spi_flash_erase(u32FlashOffset, FLASH_SECTOR_SZ);
        if(M2M_SUCCESS != ret)
        {
```

```
            printf("Unable to erase SPI sector\r\n");
            break;
        }

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

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

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

13. 主机接口（HIF）协议

用户应用程序和 WINC 器件之间的通信由驱动程序软件实现。该驱动程序实现主机接口（HIF）协议，并向应用程序公开 API 以提供各种服务。这些服务大致分为两类：Wi-Fi 设备控制和 IP 套接字。Wi-Fi 设备控制服务允许通道扫描、网络标识、连接和断开连接等操作。一旦建立连接，套接字服务就会允许数据传输，类似于 BSD 套接字定义。

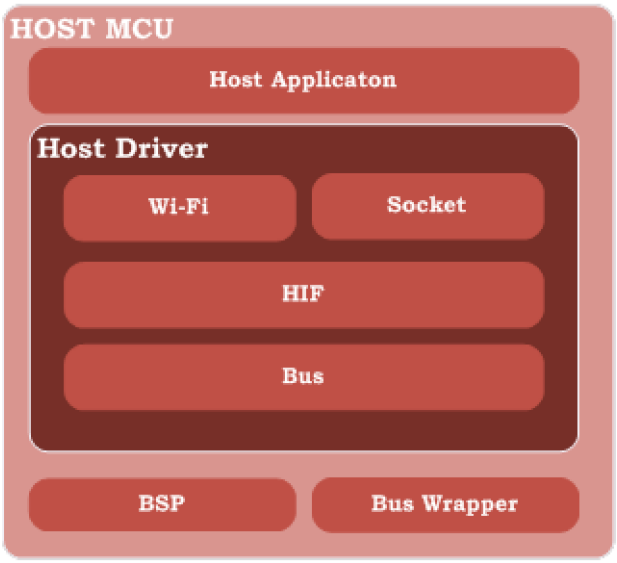
主机驱动程序异步执行服务。这意味着，当应用程序调用 API 来请求服务操作时，该调用是非阻塞的，通常在操作完成之前立即返回。在适当的情况下，会在从 WINC 器件到主机的后续报文中提供关于操作已完成的通知，然后通过回调函数将其传递给应用程序。通常，WINC 固件使用异步事件向主机驱动程序发送有关某些状态更改的信号。当功能（例如 Wi-Fi 连接）可能需要很长时间才能完成时，异步操作必不可少。

调用 API 时，将激活一系列层以格式化请求并安排通过串行协议将其传输到 WINC 器件。

注： 处理主机 MCU 应用程序中的 HIF 报文是一个深入的话题。对于大多数应用，建议使用 Wi-Fi 层和套接字层。这两层都隐藏了 HIF API 的复杂性。

应用程序发送请求后，主机驱动程序（Wi-Fi/套接字层）将格式化请求并将其发送到 HIF 层，然后中断 WINC 器件以发出新请求已发布的通知。收到请求后，WINC 固件会对其进行解析并启动所需操作。

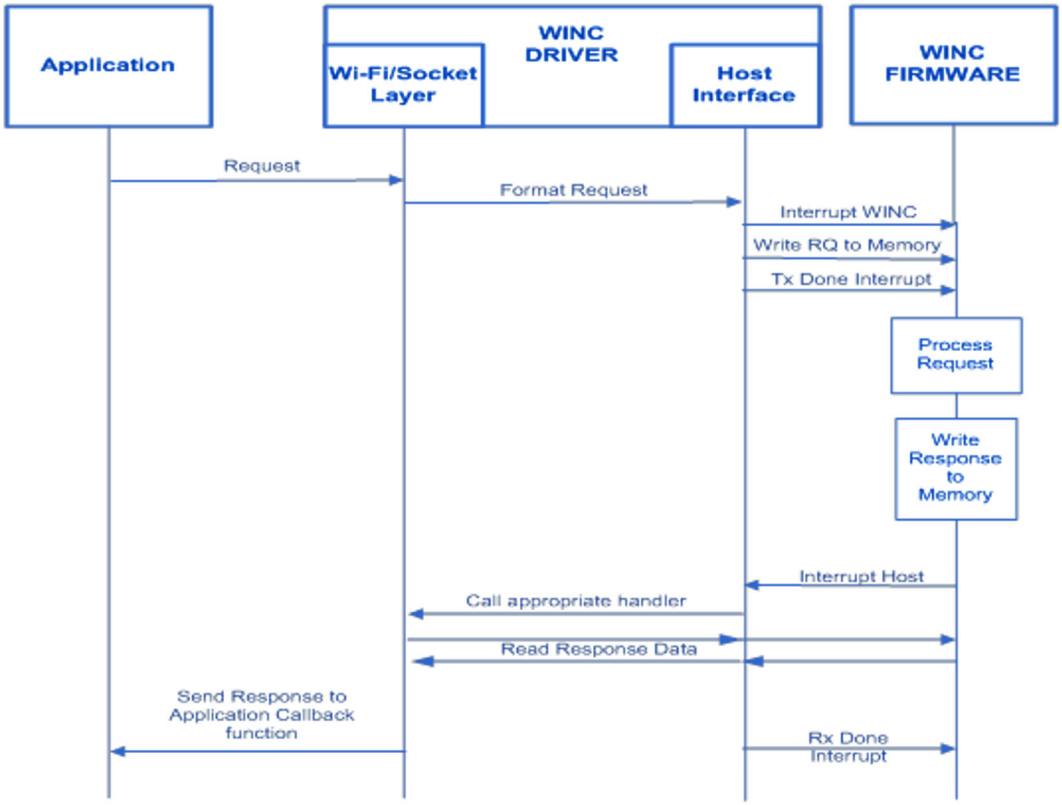
图 13-1. WINC 驱动程序层



主机接口层负责处理主机 MCU 与 WINC 器件之间的通信。其中包括针对主机固件驱动程序与 WINC 固件之间的通信逻辑的中断处理、DMA 控制和管理。

主机和 WINC 芯片之间的请求/响应序列如下图所示。

图 13-2. 请求/响应序列图



13.1 HIF 层和 WINC 固件之间的传输序列

以下部分介绍在 HIF 帧发送（HIF 报文发送到 WINC）和 HIF 帧接收（从 WINC 接收 HIF 报文）期间的各个步骤。

13.1.1 帧发送

下图所示为从主机向 WINC 器件发送报文时涉及的步骤和状态。

图 13-3. HIF 帧发送到 WINC

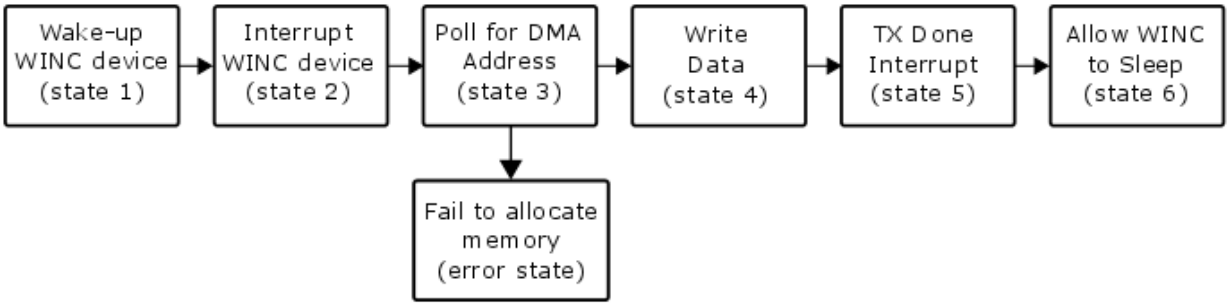


表 13-1. HIF 帧发送到 WINC 的步骤

步骤	说明
步骤(1)唤醒 WINC 器件	唤醒器件以便能够接收主机请求。
步骤(2)中断 WINC 器件	准备并设置针对 NMI_STATE_REG 寄存器的 HIF 层报头（4 字节报头，用于说明发送的数据包）。 通过将 WIFI_HOST_RCV_CTRL_2 寄存器的 BIT [1]置 1 向 WINC 芯片发送中断。
步骤(3)轮询 DMA 地址	等待 WINC 芯片将 WIFI_HOST_RCV_CTRL_2 寄存器的 BIT [1]清零。 从寄存器 0x150400 获取 DMA 地址（用于分配的存储器）。
步骤(4)写入数据	依次写入数据块、HIF 报头、控制缓冲区（如果存在）和数据缓冲区（如果存在）。
步骤(5)TX 完成中断	通过将 WIFI_HOST_RCV_CTRL_3 寄存器的 BIT [1]置 1 发送关于完成数据写入操作的通知。
步骤(6)允许 WINC 器件休眠	允许 WINC 器件再次进入休眠模式（如果需要）。

13.1.2 帧接收

下图所示为从 WINC 器件向主机发送报文时涉及的步骤和状态。

图 13-4. 主机从 WINC 接收 HIF 帧

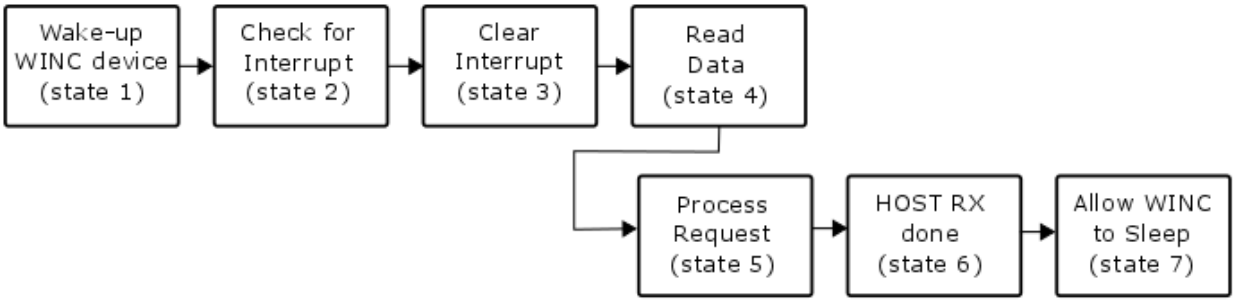


表 13-2. 主机从 WINC 接收 HIF 帧的步骤

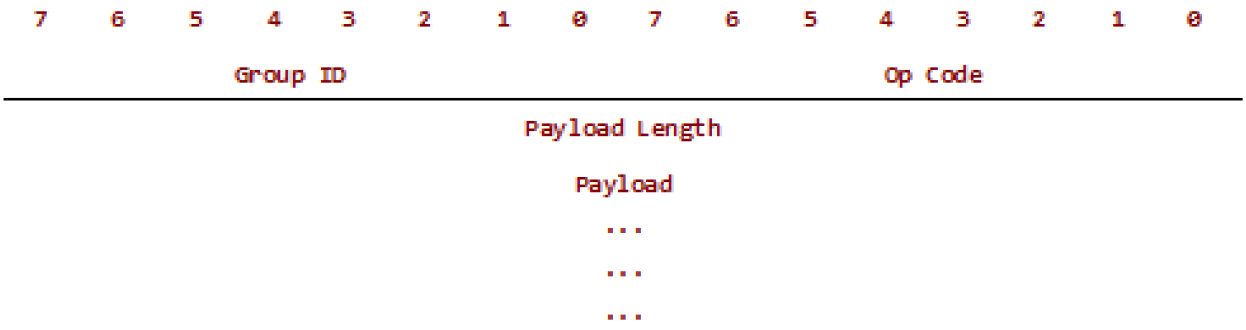
步骤	说明
步骤(1)唤醒 WINC 器件	唤醒器件以便能够接收主机请求。
步骤(2)检查中断	监视 WIFI_HOST_RCV_CTRL_0 寄存器的 BIT [0]。 禁止主机接收中断（直到已处理该中断）。
步骤(3)清除中断	向 WIFI_HOST_RCV_CTRL_0 寄存器的 BIT [0]写入 0。

.....（续）

步骤	说明
步骤(4)读取数据	从 WIFI_HOST_RCV_CTRL_1 寄存器获取数据块的地址。 读取数据块（大小从 WIFI_HOST_RCV_CTRL_0 寄存器 BIT [13] <-> BIT [2]获取）。
步骤(5)处理请求	解析数据起始处的 HIF 报头并将数据转发到适当的已注册回调函数。
步骤(6) 主机 RX 完成	通过将 WIFI_HOST_RCV_CTRL_0 寄存器的 BIT [1]位置 1 向芯片发送中断，以释放保存数据的存储空间。 再次使能主机中断接收。
步骤(7)允许 WINC 器件休眠	允许 WINC 器件再次进入休眠模式（如果需要）。

13.2 HIF 报文报头结构

HIF 报文是在主机接口与 WINC 固件之间来回交换的数据结构。HIF 报文报头结构由三个字段组成：



- 组 ID（8 位）——组 ID 是报文的类别。有效类别在 tenuM2mReqGroup 中枚举。
- 操作码（8 位）——是命令编号。有效命令编号是按照以下形式枚举的值：tenuM2mConfigCmd、tenuM2mStaCmd、tenuM2mApCmd 和 tenuM2mP2pCmd，分别对应于配置、STA 模式、AP 模式和 P2P 模式命令。
  注：
  - 有关完整的命令列表，请参见 m2m_types.h。
  - 自版本 v19.5.3 起，不再支持 P2P 模式。
- 有效负载长度（16 位）——有效负载长度以字节显示（不包括报头）。

13.3 HIF 层 API

应用程序与驱动程序之间的接口在上层 API 接口（Wi-Fi/套接字）上实现。如前所述，驱动程序上层使用下层 API 来访问主机接口协议的服务。本节介绍上层使用的主机接口 API：

将介绍以下 API 函数：

- hif_chip_wake

- hif_chip_sleep
- hif_register_cb
- hif_isr
- hif_receive
- hif_send
- hif_set_sleep_mode
- hif_get_sleep_mode

对于所有函数，成功时的返回值为 M2M_SUCCESS（0）,失败时的返回值为负值。

- sint8 hif_chip_wake (void)——该函数将通过无时钟寄存器访问将 WINC 芯片从休眠模式唤醒。它会将寄存器 0x01 的 bit 1 置 1，并将 WAKE_REG 寄存器的值设置为 WAKE_VALUE。
- sint8 hif_chip_sleep (void)——该函数通过将 WAKE_REG 寄存器的值设置为 SLEEP_VALUE 并将寄存器 0x01 的 bit 1 清零来使能 WINC 芯片的休眠模式。
- sint8 hif_register_cb (uint8 u8Grp, tpfHifCallBack fn)——该函数为不同组件（例如 M2M_WIFI、M2M_HIF 和 M2M_OTA 等）设置回调函数。上层注册回调函数以接收特定报文组的特定事件。
- sint8 hif_isr (void)——这是主机接口中断服务程序。它处理 WINC 芯片生成的中断并解析 HIF 报头以回调相应的处理程序。
- sint8 hif_receive (uint32 u32Addr, uint8 *pu8Buf, uint16 u16Sz, uint8 is Done)——该函数使主机驱动程序从 WINC 芯片读取数据。必须事先知道并指定数据的位置和长度。这类信息通常在事务的早期阶段提取。
- sint8 hif_send (uint8 u8Gid, uint8 u8Opcode, uint8 *pu8CtrlBuf, uint16 u16CtrlBufSize, uint8 *pu8DataBuf, uint16 u16DataSize, uint16 16DataOffset)——该函数使主机驱动程序向 WINC 芯片发送数据。WINC 芯片必须根据上一节中描述的流程做好接收准备。
- void hif_set_sleep_mode (uint8 u8Pstype)——该函数用于设置 HIF 层的休眠模式。
- uint8 hif_get_sleep_mode (void)——该函数返回 HIF 层的休眠模式。

13.4 扫描代码示例

以下代码示例说明了关于 Wi-Fi 扫描请求的请求/响应流程。

注： 有关示例代码的更多详细信息，请参见 Wi-Fi Network Controller Software Programming Guide。

- 应用程序请求 Wi-Fi 扫描。

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

- 主机驱动程序 Wi-Fi 层格式化请求并将其转发到 HIF（主机接口）层。

```
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;
}
```

- HIF 层将请求发送到 WINC 芯片。

```
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;
}
```

- WINC 芯片处理请求并在完成操作后中断主机。
- 然后 HIF 层接收响应。

```
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;
}
```

- Wi-Fi 层中的相应处理程序（从 HIF 层调用）。

```
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…
}
```

- Wi-Fi 层通过其回调函数将响应发送给应用程序。

```
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);
        }
    }
}
```

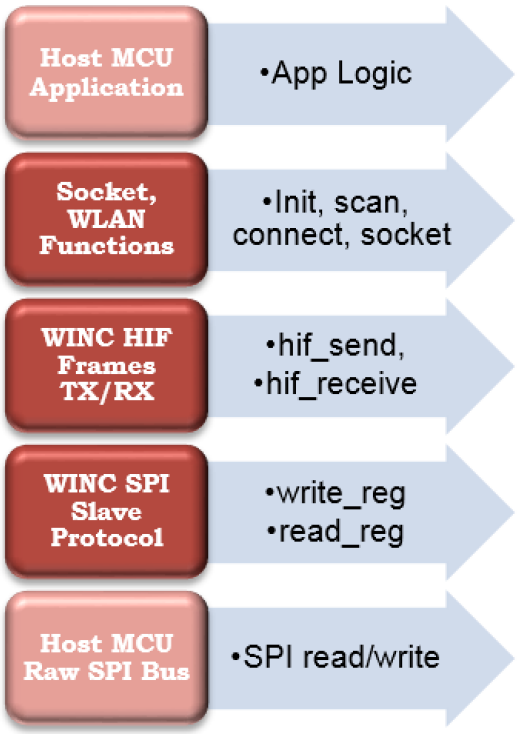
14. WINC SPI 协议

WINC 主接口是 SPI。WINC 器件采用的协议允许在 WINC 固件和主机 MCU 应用程序之间交换格式化的二进制报文。WINC 协议使用在 SPI 总线上交换的原始字节来形成高级结构，如请求和回调函数。

WINC SPI 协议由三层组成：

- 第 1 层——WINC SPI 从器件协议，允许主机 MCU 应用程序通过原始 SPI 数据交换在 ATWINC15x0 器件中执行寄存器/存储器读写操作。
- 第 2 层——主机 MCU 应用程序使用寄存器和存储器读写功能与 WINC 固件交换主机接口帧。它还通过中断和主机接口 RX 帧提供从 WINC 固件到主机 MCU 的异步回调函数。有关该层的更多信息，请参见**第 15 章“主机接口（HIF）协议”**。
- 第 3 层——允许主机 MCU 应用程序与 WINC 固件交换高级报文（例如，Wi-Fi 扫描、套接字连接或接收的 TCP 数据），以便在主机 MCU 应用程序逻辑中使用。

图 14-1. WINC SPI 协议层



14.1 简介

WINC SPI 协议以命令-响应事务的形式实现，并假定其中一方是主器件，另一方是从器件。这两个角色分别对应于 SPI 总线上的主器件和从器件。每条报文的第一个字节中都有一个标识符，用于指示报文的类型：

- 命令
- 响应
- 数据

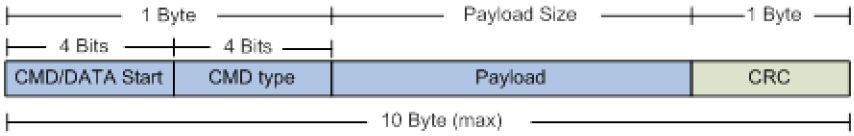
对于命令和数据报文，最后一个字节用于数据完整性检查。

以下各节将分别介绍命令帧、响应帧和数据帧的格式。以下几点适用：

- 每条命令都有响应。
- 发送/接收的数据被分成多个大小固定的数据包。
- 对于 WR 事务（*从器件正在接收数据包*），从器件会为每个数据包发送响应。
- 对于 RD 事务（*主器件正在接收数据包*），主器件不会发送响应。如果存在错误，主器件会请求重发丢失的数据包。
- 可以选择通过 CRC 保护命令和数据包。

14.1.1 命令格式

以下帧格式用于支持三字节 DMA 地址的主机命令。



第一个字节包含两个字段：

- CMD/数据开始字段指示这是一个命令帧。
- CMD 类型字段指定要执行的命令。

CMD 类型可能为以下 15 种命令之一：

- DMA 写操作
- DMA 读操作
- 内部寄存器写操作
- 内部寄存器读操作
- 事务终止
- 重复数据包
- DMA 扩展写操作
- DMA 扩展读操作
- DMA 单字写操作
- DMA 单字读操作
- 软复位

有效负载字段包含命令特定数据，其长度取决于 CMD 类型。

CRC 字段是可选的，通常用软件计算。

有效负载字段可以是以下四种类型之一，每种类型具有不同的长度：

- A：三字节
- B：五字节
- C：六字节
- D：七字节

类型 A 的命令包括：

- DMA 单字 RD
- 内部寄存器 RD

- 事务终止命令
- 重复数据 PKT 命令
- 软复位命令

类型 B 的命令包括：

- DMA RD 事务
- DMA WR 事务

类型 C 的命令包括：

- DMA 扩展 RD 事务
- DMA 扩展 WR 事务
- 内部寄存器 WR

类型 D 的命令包括：

- DMA 单字 WR

下表提供了帧格式字段的完整详细信息：

表 14-1. 帧格式字段

字段	大小	说明
CMD 开始	4 位	命令开始：4'b1100
CMD 类型	4 位	命令类型： 4'b0001：DMA 写事务 4'b0010：DMA 读事务 4'b0011：内部寄存器写操作 4'b0100：内部寄存器读操作 4'b0101：事务终止 4'b0110：重复数据包命令 4'0111：DMA 扩展写事务 4'b1000：DMA 扩展读事务 4'b1001：DMA 单字写操作 4'b1010：DMA 单字读操作 4'b1111：软复位命令

.....（续）

字段	大小	说明
有效负载	A：3	有效负载字段可以是类型 A、B、C 或 D **类型 A（长度 3）** ***1——DMA 单字 RD*** *参数：读地址：* *有效负载字节：* B0：ADDRESS[23:16] B1：ADDRESS[15:8] B2：ADDRESS[7:0] ***2——内部寄存器 RD*** *参数：偏移地址（两字节）：* *有效负载字节：* B0：OFFSET-ADDR[15:8] B1：OFFSET-ADDR[7:0] B2：0 ***3——事务终止命令*** *参数：无* *有效负载字节：* B0：0 B1：0 B2：0 ***4——重复数据 PKT 命令*** *参数：无* *有效负载字节：* B0：0 B1：0 B2：0 ***5——软复位命令*** *参数：无* *有效负载字节：* B0：0xFF B1：0xFF B2：0xFF

.....（续）

字段	大小	说明
有效负载	B：5	**类型 B（长度 5）** ***1——DMA RD 事务*** *参数：* DMA 起始地址：3 字节 DMA 计数：2 字节 *有效负载字节：* B0：ADDRESS[23:16] B1：ADDRESS[15:8] B2：ADDRESS[7:0] B3：COUNT[15:8] B4：COUNT[7:0] ***2——DMA WR 事务*** *参数：* DMA 起始地址：3 字节 DMA 计数：2 字节 *有效负载字节*： B0：ADDRESS[23:16] B1：ADDRESS[15:8] B2：ADDRESS[7:0] B3：COUNT[15:8] B4：COUNT[7:0]

.....（续）

字段	大小	说明
有效负载	C：6	**类型 C（长度 6）** ***1——DMA 扩展 RD 事务*** *参数：* DMA 起始地址：3 字节 DMA 扩展计数：3 字节 *有效负载字节：* 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 扩展 WR 事务*** *参数：* DMA 起始地址：3 字节 DMA 扩展计数：3 字节 *有效负载字节：* B0：ADDRESS[23:16] B1：ADDRESS[15:8] B2：ADDRESS[7:0] B3：COUNT[23:16] B4：COUNT[15:8] B5：COUNT[7:0]

.....（续）

字段	大小	说明
有效负载	C：6	***3——内部寄存器 WR**** *参数：* 偏移地址：3 字节 写数据：3 字节 *“有时钟或无时钟寄存器” *有效负载字节：* 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]
有效负载	D：7	**类型 D（长度 7）** ***1——DMA 单字 WR*** *参数：* 地址：3 字节 DMA 数据：4 字节 *有效负载字节：* 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 字节	可选的数据完整性字段，包括两个子字段： bit 0：固定值 1 bit 1-7：使用多项式 G(x) = X^7 + X^3 + 1 计算的 7 位 CRC 值，种子值为：0x7F

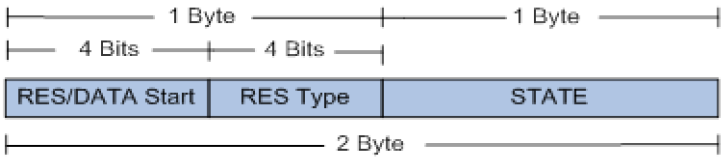
下表按照有效负载类型（DMA 地址 = 3 字节）汇总了不同命令：

表 14-2. 有效负载中的命令

有效负载类型	有效负载大小	带 CRC 的命令包大小	命令
类型 A	3 字节	5 字节	1——DMA 单字读操作 2——内部寄存器读操作 3——事务终止 4——重复数据包 5——软复位
类型 B	5 字节	7 字节	1——DMA 读操作 2——DMA 写操作
类型 C	6 字节	8 字节	1——DMA 扩展读操作 2——DMA 扩展写操作 3——内部寄存器写操作
类型 D	7 字节	9 字节	1——DMA 单字写操作

14.1.2 响应格式

以下帧格式用于 WINC 器件在收到命令或特定数据帧后发送的响应。响应报文为两字节固定长度。



第一个字节包含两个四位字段，分别用于标识响应报文和响应类型。

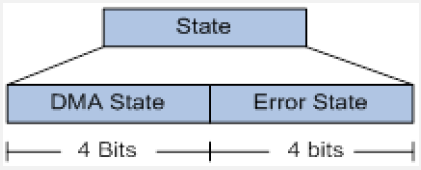
第二个字节指示 WINC 在收到并可能执行命令/数据后的状态。该字节包含两个子字段：

- B0-B3：错误状态
- B4-B7：DMA 状态

可指示的状态如下：

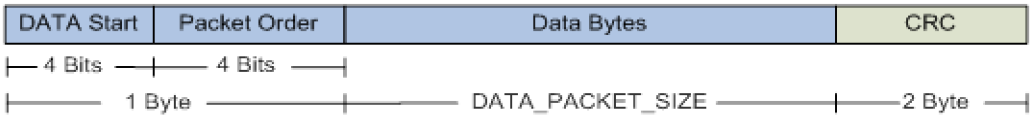
- DMA 状态：
  - DMA 已准备好执行任何事务
  - DMA 引擎忙
- 错误状态：
  - 无错误
  - 不支持的命令
  - 收到意外的数据包
  - 命令 CRC7 错误

表 14-3. 响应格式

字段	大小	说明
响应开始	4 位	响应开始：4'b1100
响应类型	4 位	如果响应数据包是针对命令的： • 包含命令中“命令类型”字段的副本。 如果响应数据包是针对所接收数据包的： • 4'b0001：已收到第一个数据包 • 4'b0010：正在接收数据包 • 4'b0011：已收到最后一个数据包 • 4'b1111：保留值
状态	1 字节	该字段分为两个子字段： DMA 状态：  • 4'b0000：DMA 已准备好执行任何事务 • 4'b0001：DMA 引擎忙 错误状态： • 4'b0000：无错误 • 4'b0001：不支持的命令 • 4'b0010：收到意外的数据包 • 4'b0011：命令 CRC7 错误 • 4'b0100：数据 CRC16 错误 • 4'b0101：内部一般错误

14.1.3 数据包格式

数据包格式用于在主器件与从器件之间双向传输不透明数据。命令帧可用于通知从器件即将发送数据包或请求从器件向主器件发送数据包。在从主器件向从器件传输的情况下，从器件会在收到命令和每个后续数据帧之后发送响应。数据包的格式如下所示。



为了支持 DMA 硬件，可以将大型数据传输拆分为多个较小的数据包。具体大小由 DATA_PACKET_SIZE 的值控制，该值在主器件和从器件之间通过软件达成一致并且是固定值，例如 256B、512B、1 KB（默认）、2 KB、4 KB 或 8 KB。如果数据传输长度 **m** 超过 DATA_PACKET_SIZE，则发送方必须将其拆分为多个 DATA_PACKET_SIZE，如公式 1 所示：

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

其中，

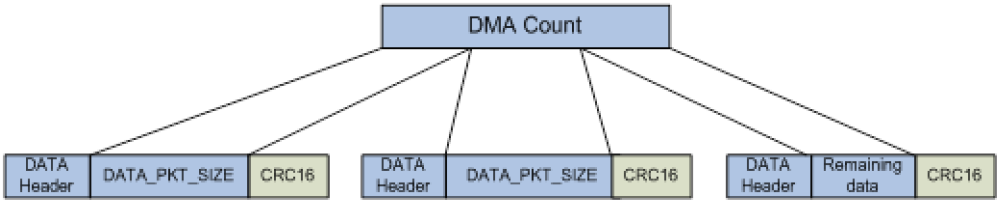
1.. n-1 = DATA_PACKET_SIZE 的长度

n = 帧长

下面对此进行了说明。

- 如果 DMA 计数 <= DATA_PACKET_SIZE：
  数据包为“*DATA_Header + DMA 计数 + 可选 CRC16*”，无填充。

- 如果 DMA 计数 > DATA_PACKET_SIZE：



- 如果剩余数据 < DATA_PACKET_SIZE，则最后一个数据包为：
  “DATA_Header + 剩余数据 + 可选 CRC16”，无填充。

下表详细说明了帧字段：

表 14-4. 帧字段

字段	大小	说明
数据开始	4 位	4'b1111（默认值） （可以通过编程 DATA_START_CTRL 寄存器将其更改为任何值）
数据包顺序	4 位	4'b0001：该事务中的第一个数据包 4'b0010：既不是该事务中的第一个数据包也不是最后一个数据包 4'b0011：该事务中的最后一个数据包 4'b1111：保留
数据字节	DATA_PACKET_SIZE	用户数据
CRC16	2 字节	可选的数据完整性字段，包括以两个字节编码的 16 位 CRC 值。先发送帧中的高 8 位。 仅基于以下多项式计算数据字节的 CRC16 值： *G(x) = X^16 + X^12 + X^5 + 1*，种子值：0xFFFF

14.1.4 错误恢复机制

表 14-5. 错误恢复机制

错误类型	恢复机制
主器件	

.....（续）

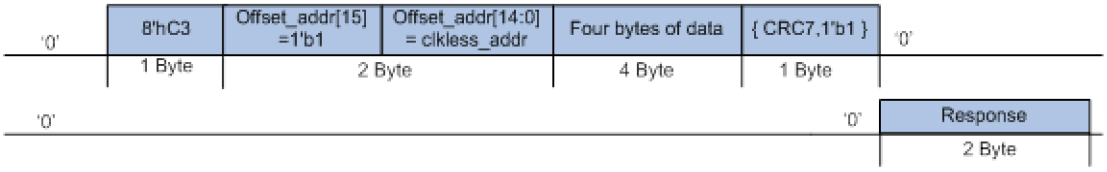
错误类型	恢复机制
命令中的 CRC 错误	1. 收到从器件发送的错误响应。 2. 重新发送命令。
接收的数据中存在 CRC 错误	1. 对具有 CRC 错误的数据包发出重复命令。 2. 从器件发送前一个命令的响应。 3. 从器件保留前一个数据包的起始 DMA 地址，以便重新发送。 4. 再次接收数据包。
未收到从器件发送的响应	• 主器件和从器件之间失去同步。 • 最糟糕的情况发生在从器件处于接收数据状态时。 • 解决方案：主器件必须等待最大 DATA_PACKET_SIZE 周期，然后再生成软复位命令。
意外的响应	重新发送命令。
TX/RX 数据计数错误	重新发送命令。
软复位命令无响应	• 发送全 1，直到主器件接收到来自从器件的全 1 响应。 • 然后禁止输出数据线。
从器件	
不支持的命令	• 发送含错误的响应。 • 返回命令监视状态。
接收的命令中存在 CRC 错误	• 发送含错误的响应。 • 等待命令重新发送。
接收的数据中存在 CRC 错误	• 发送含错误的响应。 • 等待数据包重新发送。
内部一般错误	• 主器件必须对从器件执行软复位。
TX/RX 数据计数错误	• 只有主器件可以检测到此错误。 • 从器件使用接收到的数据计数进行操作，直到计数完成或主器件终止事务。 • 在这两种情况下，主器件都可以从头开始重试命令。
软复位命令无响应	1. 首先接收到 4'b1001，它决定数据开始位置。 2. 然后接收到数据包顺序 4'b1111，即保留值。 3. 然后监视 7 字节全 1，以决定执行软复位操作。 4. 从器件必须激活输出数据线。 5. 等待接收线禁止。 6. 然后，从器件将禁止输出数据线并返回 CMD/DATA 开始监视状态。

.....（续）	
错误类型	**恢复机制**
一般说明	• 从器件必须随时监视用于接收命令的接收线路。 • 当检测到 CMD 开始时，从器件将接收到 8 个字节，然后再次返回到命令接收状态。 • 当从器件正在发送数据时，还必须监视是否收到命令。 • 当从器件正在接收数据时，会监视是否在数据包之间收到命令。 • 在所有情况下都会检测是否发出软复位命令。

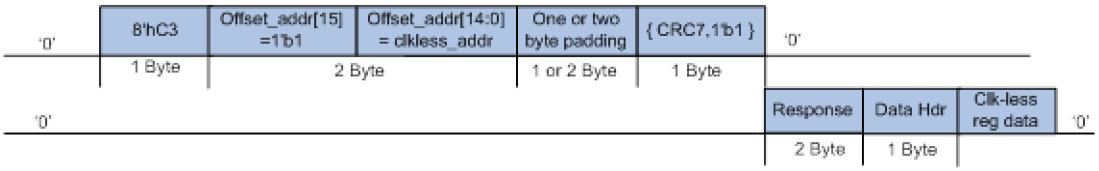
14.1.5 无时钟寄存器访问

无时钟寄存器访问允许主机器件在 WINC 器件处于复位状态时访问 WINC 器件上的寄存器。只能使用“内部寄存器读操作”和“内部寄存器写操作”命令来执行此类访问。对于无时钟访问，命令中 `Offset_addr` 的 bit 15 必须是 `1`，以便区分无时钟和有时钟访问模式。

对于无时钟寄存器**写操作**：协议主器件必须等待如下所示的响应：



对于无时钟寄存器**读操作**：根据接口的不同，协议从器件可能不会发送 CRC16。填充单字节还是双字节取决于 DMA 地址为三字节还是四字节。

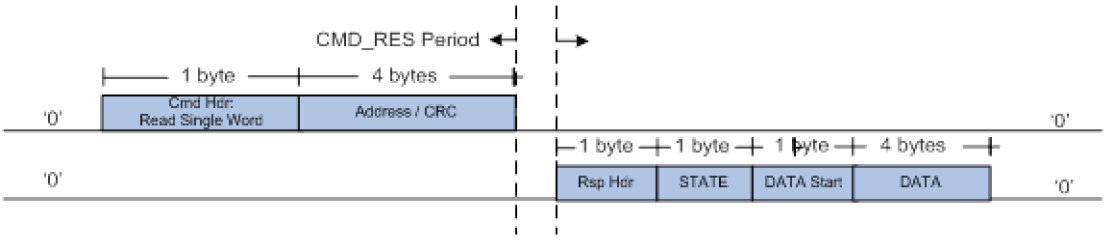


14.2 基本事务的报文流

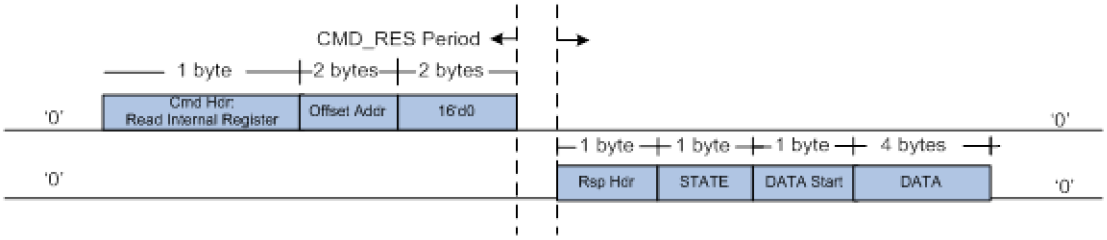
本节介绍与以下命令关联的基本报文交换和时序：

- 读取单字
- 读取内部寄存器（无时钟）
- 读取块
- 写入单字
- 写入内部寄存器（无时钟）
- 写入块

14.2.1 读取单字



14.2.2 读取内部寄存器（对于无时钟寄存器）



14.2.3 读取块

正常事务：

主器件——发出 DMA 读取事务并等待响应。

从器件——在 CMD_RES_PERIOD 之后发送响应。

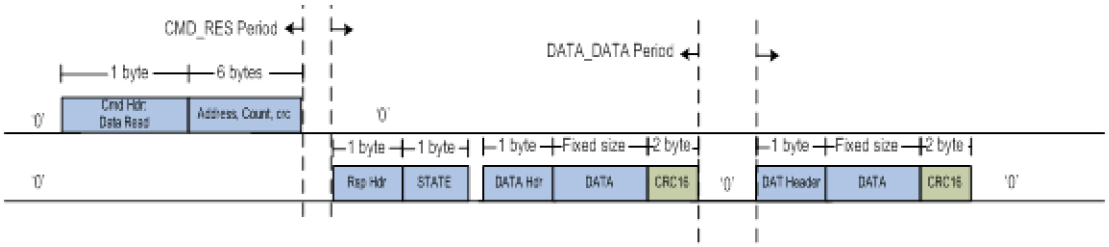
主器件——等待数据包开始。

从器件——发送数据包，各数据包之间由 DATA_DATA_PERIOD[1]隔开，其中 DATA_DATA_PERIOD 由软件控制，并具有以下值之一：NO_DELAY（默认值）、4_BYTE_PERIOD、8_BYTE_PERIOD 和 16_BYTE_PERIOD。

从器件——继续发送，直到计数结束。

主器件——接收数据包。不为数据包发送响应，但如果存在错误，可以发送终止/重新发送命令。

本例的报文序列如下所示：



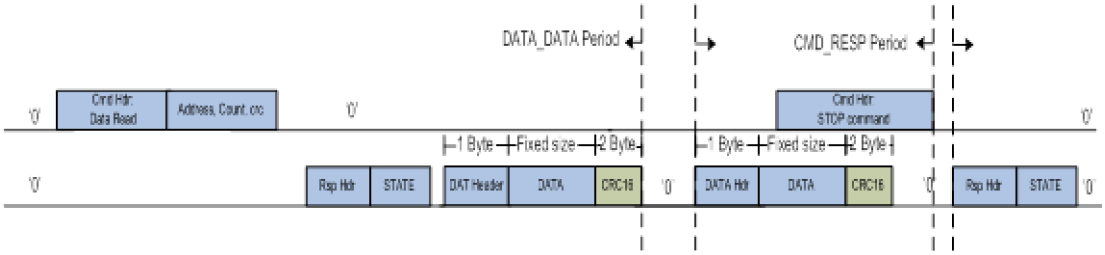
发出终止命令：

主器件——可以在执行事务期间随时发出终止命令。

主器件——在 CMD_RESP_PERIOD 之后监视 RES_START。

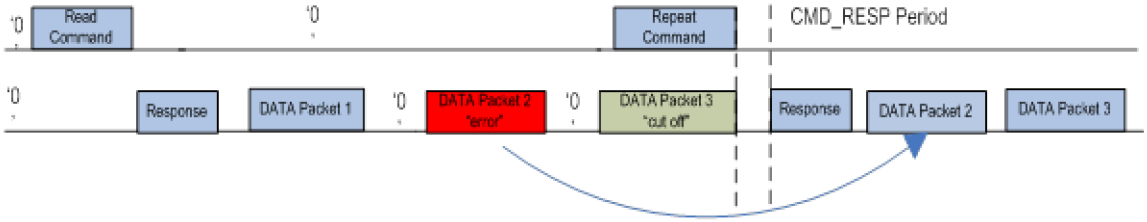
从器件——中断当前运行的数据包（如果存在）。

从器件——在终止命令包末尾的 CMD_RESP_PERIOD 之后响应终止命令。



发出重复命令：

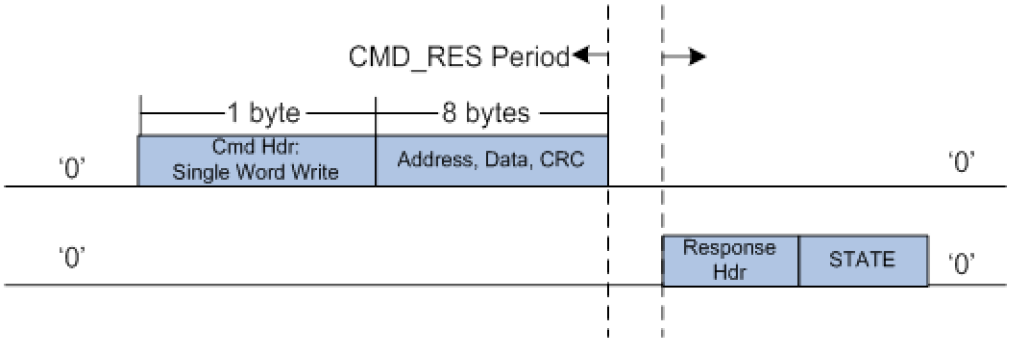
1. 主器件——可以在执行事务期间随时发出重复命令。
2. 主器件——在 CMD_RESP_PERIOD 之后监视 RES_START。
3. 从器件——中断当前运行的数据包（如果存在）。
4. 从器件——在重复命令包末尾的 CMD_RESP_PERIOD 之后响应重复命令。
5. 从器件——再次发送存在错误的数据包，然后继续正常执行事务。



[1]数据包的间隔时间为“DATA_DATA_PERIOD + DMA 访问时间”。主器件在 DATA_DATA_PERIOD 之后立即监视 DATA_START。

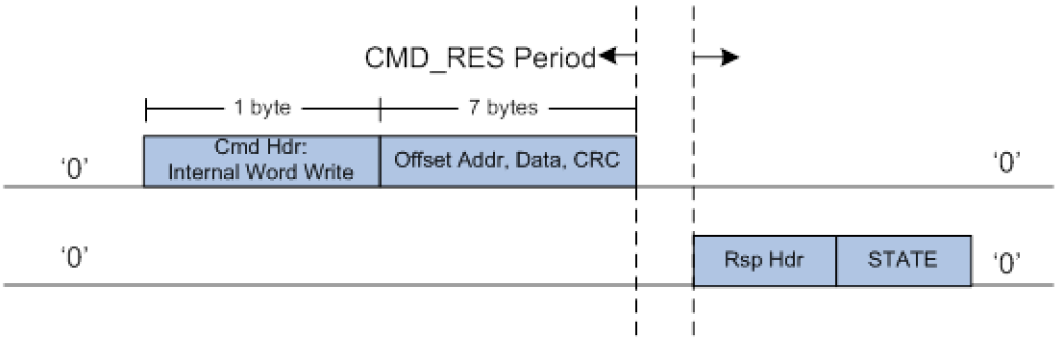
14.2.4 写入单字

1. 主器件——发出 DMA 单字写入命令，包括数据。
2. 从器件——获取数据并发送命令响应。



14.2.5 写入内部寄存器（对于无时钟寄存器）

1. 主器件——发出内部寄存器写入命令，包括数据。
2. 从器件——获取数据并发送命令响应。



14.2.6 写入块

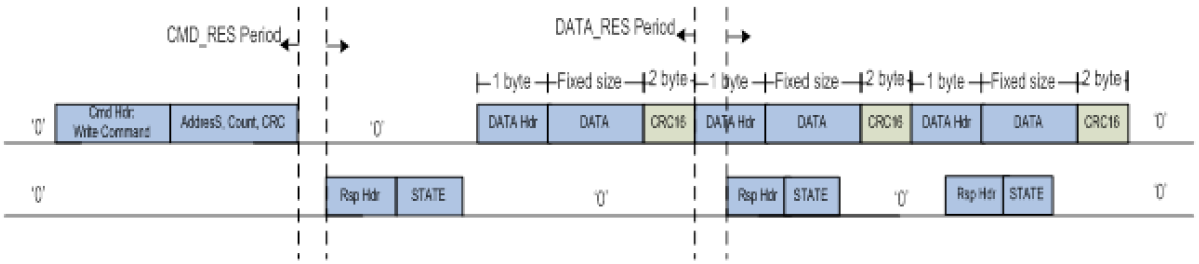
- **情形 1：主器件等待命令响应：**
  1. 主器件——发出 DMA 写入命令并等待响应。
  2. 从器件——在 CMD_RES_PERIOD 之后发送响应。
  3. 主器件——接收到响应后发送数据包。
  4. 从器件——为 DATA_RES_PERIOD 之后接收的每个数据包发送响应数据包。
  5. 主器件——在发送以下数据包注释之前不等待数据响应：
     CMD_RES_PERIOD 由软件控制，取以下值之一：

     NO_DELAY（默认值）、1_BYTE_PERIOD、2_BYTE_PERIOD 和 3_BYTE_PERIOD

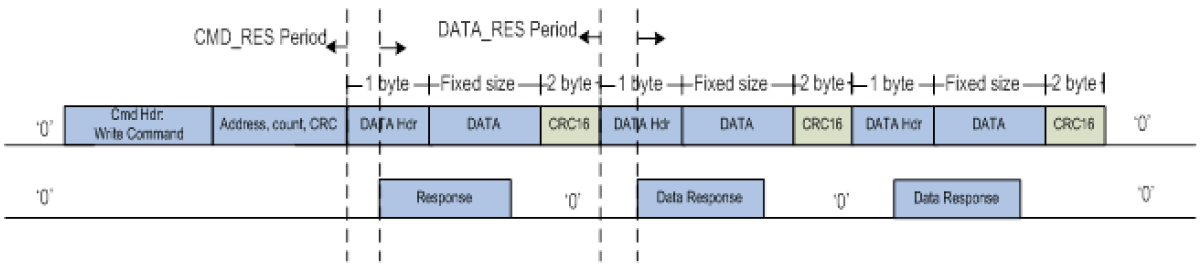
     主器件必须在 CMD_RES_PERIOD 之后监视 RES_START

     DATA_RES_PERIOD 由软件控制，取以下值之一：

     NO_DELAY（默认值）、1_BYTE_PERIOD、2_BYTE_PERIOD 和 3_BYTE_PERIOD



- **情形 2：主器件不等待命令响应：**
  1. 主器件——在命令之后立即发送数据包，但仍在 CMD_RESP_PERIOD 之后监视命令响应。
  2. 主器件——如果命令中存在错误，则重新发送数据包。



14.3 SPI 层协议示例

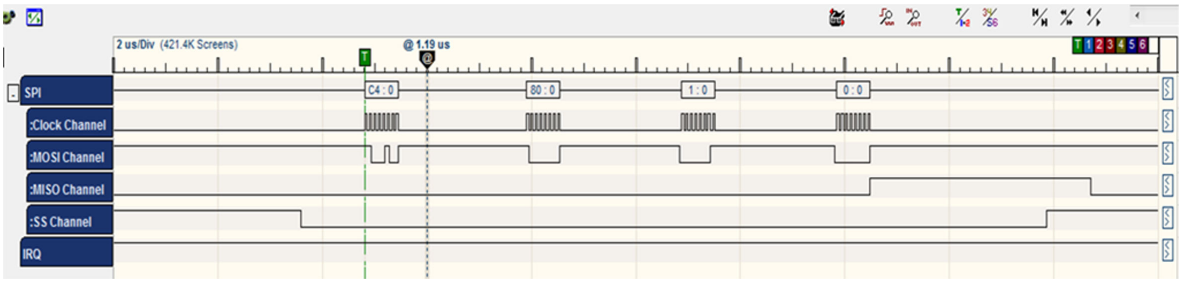
为了说明 WINC SPI 协议的工作原理，下面将以扫描请求为例对 SPI 字节进行转储，序列如下所述。

14.3.1 TX（发送请求）

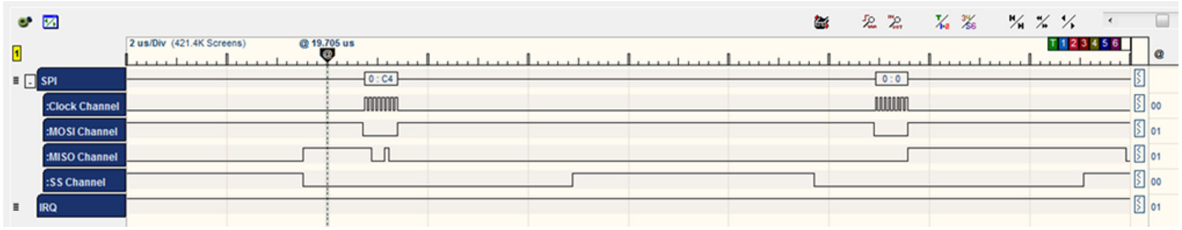
1. hif_send() API 中的第一步是唤醒芯片。

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

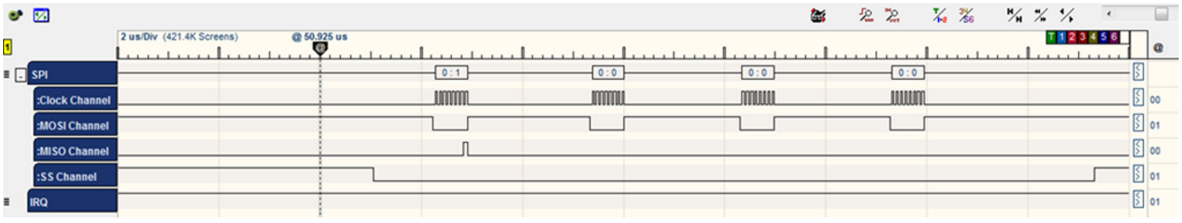
```
Command         CMD_INTERNAL_READ:      0xC4    /* internal register read */
            BYTE [0] = CMD_INTERNAL_READ
               BYTE [1] = address >> 8;             /* address = 0x01 */
            BYTE [1] |= (1 << 7);                 /* clockless register */
            BYTE [2] = address;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



2. WINC 通过发送三个字节[C4] [0] [F3]来应答命令。

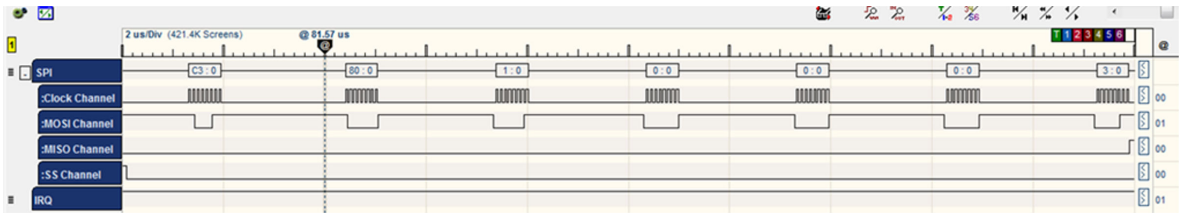


3. WINC 芯片发送寄存器 0x01 的值（等于 0x01）。

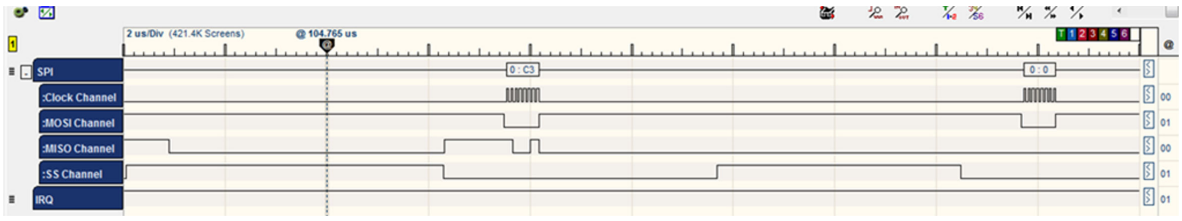



```
Command    CMD_INTERNAL_WRITE:      C3          /*     internal register write */
        BYTE [0] = CMD_INTERNAL_WRITE
        BYTE [1] = address >> 8;              /*     address = 0x01          */
        BYTE [1] |= (1 << 7);              /*     clockless register     */
        BYTE [2] = address;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

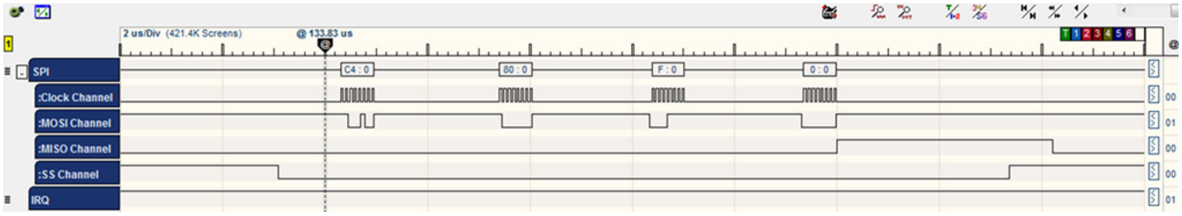
```
        BYTE [5] = u32data >> 8;
        BYTE [6] = u32data;
```



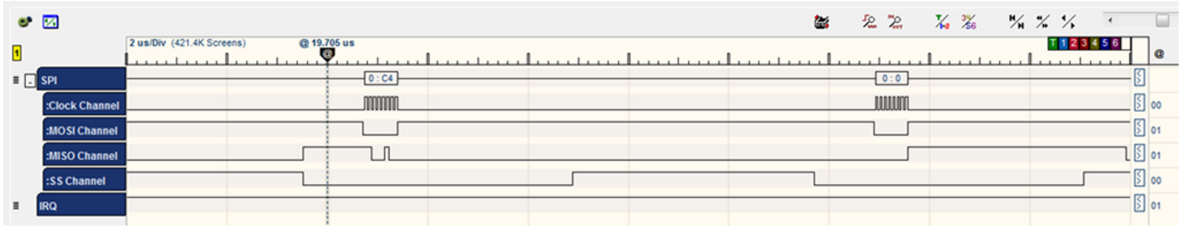
4. WINC 通过发送两个字节[C3] [0]来应答命令。



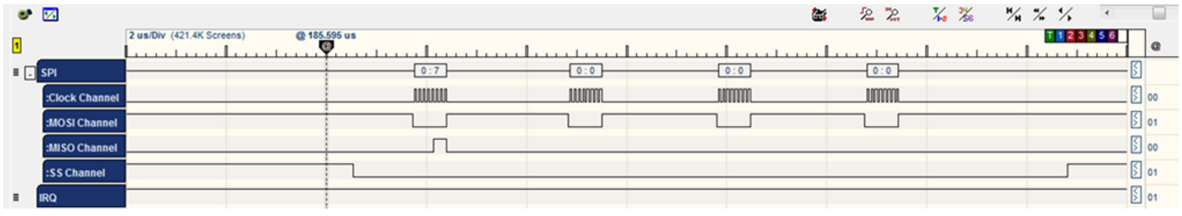

```
Command    CMD_INTERNAL_READ:     0xC4    /*     internal register read    */
        BYTE [0] = CMD_INTERNAL_READ
        BYTE [1] = address >> 8;            /*     address = 0x0F          */
        BYTE [1] |= (1 << 7);         /*     clockless register     */
        BYTE [2] = address;
    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



5. WINC 通过发送三个字节[C4] [0] [F3]来应答命令。

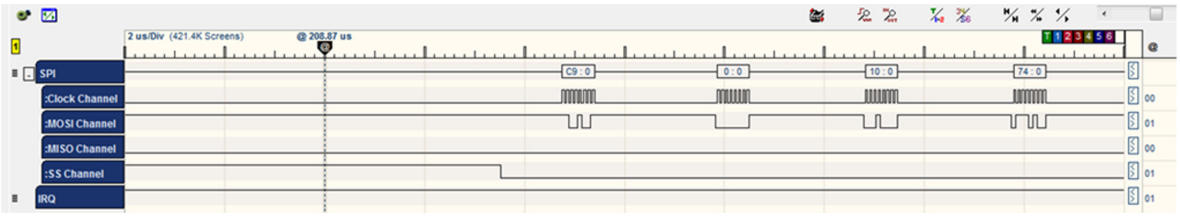


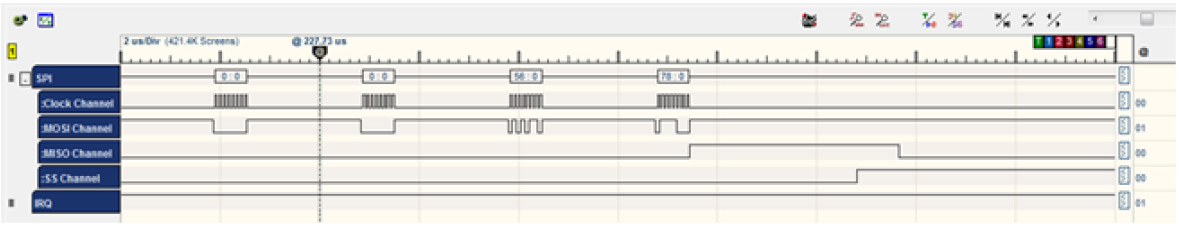
6. WINC 芯片发送寄存器 0x01 的值（等于 0x07）。



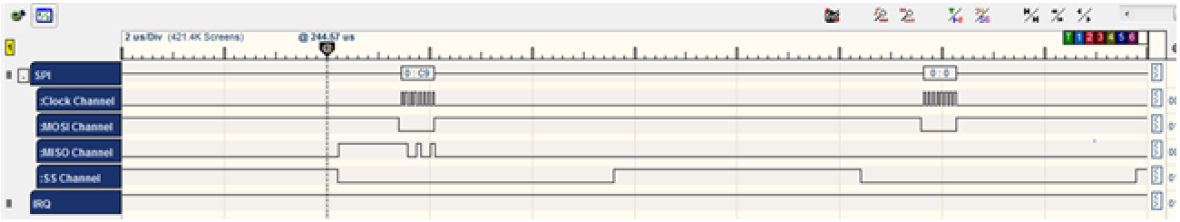

```
Command    CMD_SINGLE_WRITE:0XC9        /* single word write         */
        BYTE [0] = CMD_SINGLE_WRITE
        BYTE [1] = address >> 16;       /* WAKE_REG address = 0x1074 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = u32data >> 24;        /* WAKE_VALUE Data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

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





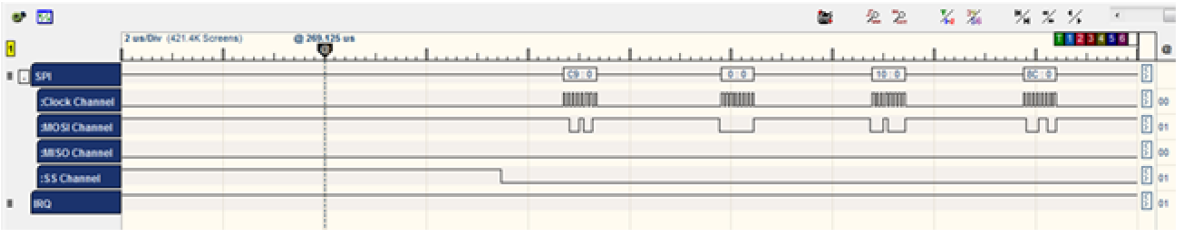
7. 芯片通过发送两个字节[C9] [0]来应答命令。

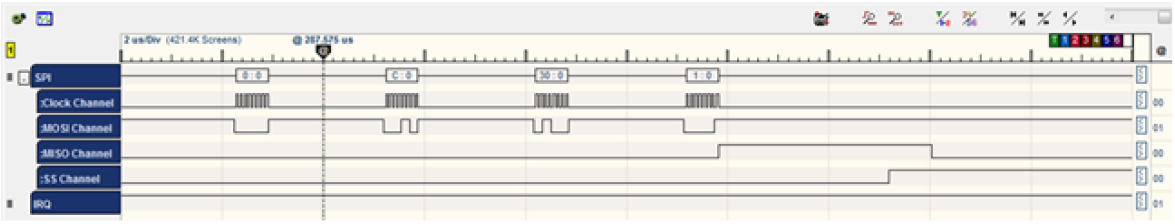


8. 至此，HIF 完成执行 WINC 芯片的无时钟唤醒。
9. HIF 层准备并设置针对 NMI_STATE_REG 寄存器的 HIF 层报头（4 字节或 8 字节报头，用于说明要发送的数据包）。
10. 通过将 WIFI_HOST_RCV_CTRL_2 寄存器的 bit 1 置 1 向 WINC 芯片发送中断。

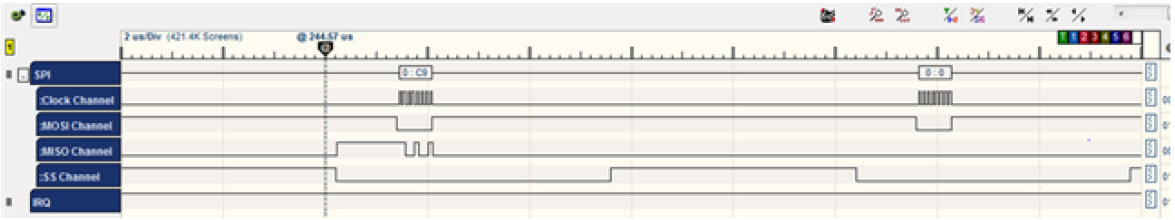
```
sint8 hif_send(uint8 u8Gid,uint8 u8Opcode,uint8 *pu8CtrlBuf,uint16 u16CtrlBufSize,
               uint8 *pu8DataBuf,uint16 u16DataSize, uint16 u16DataOffset)
{
    volatile tstrHifHdr    strHif;
    volatile uint32 reg;
    strHif.u8Opcode    = u8Opcode&(~NBIT7);
    strHif.u8Gid        = u8Gid;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

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

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

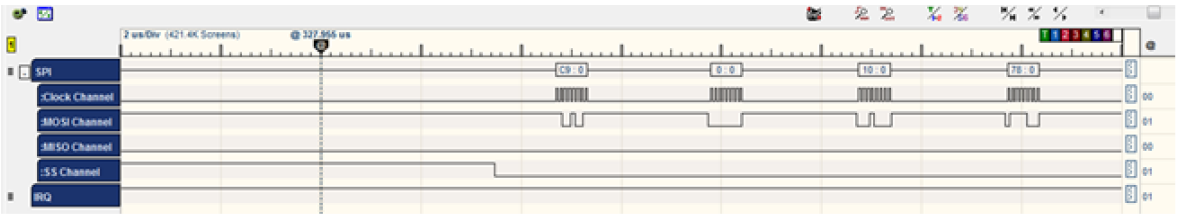


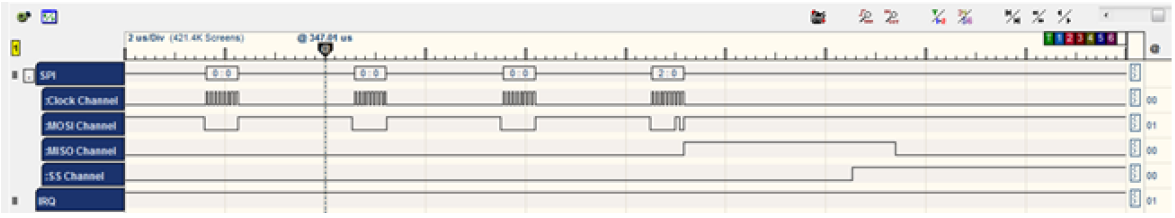


11. WINC 通过发送两个字节[C9] [0]来应答命令。

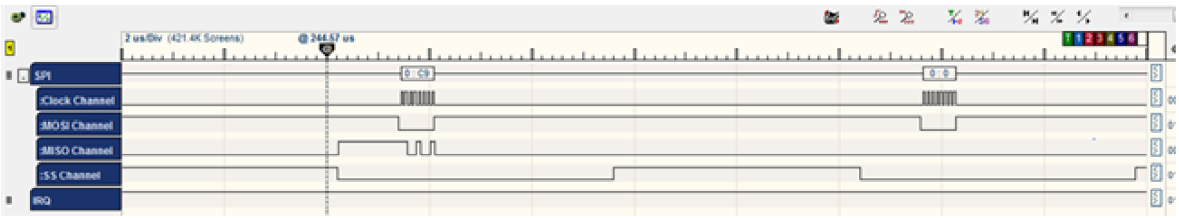



```
Command    CMD_SINGLE_WRITE:0XC9          /*      single word write */
        BYTE [0] = CMD_SINGLE_WRITE
        BYTE [1] = address >> 16;         /*      WIFI_HOST_RCV_CTRL_2address = 0x1078*/
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = u32data >> 24;      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





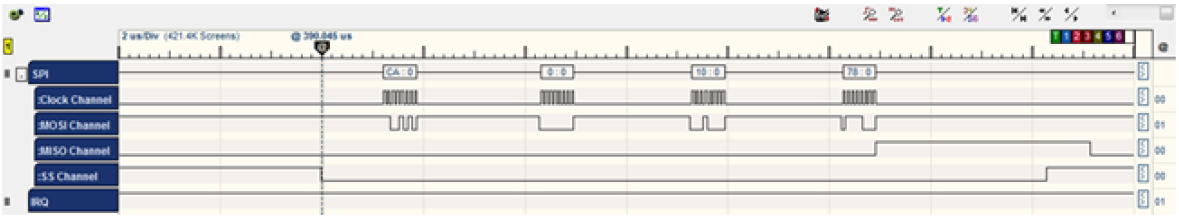
12. WINC 通过发送两个字节[C9] [0]来应答命令。



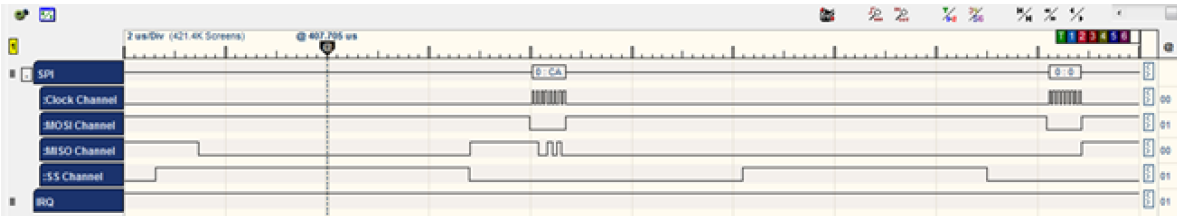
13. 然后 HIF 轮询 DMA 地址。

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

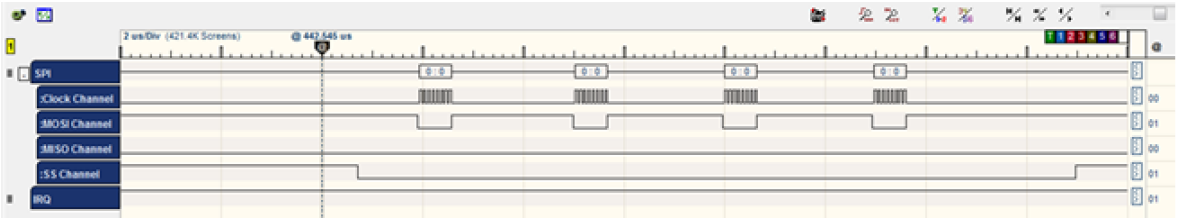
```
Command    CMD_SINGLE_READ:     0xCA         /* single word (4 bytes) read */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;           /* WIFI_HOST_RCV_CTRL_2 address = 0x1078 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



14. WINC 通过发送三个字节[CA] [0] [F3]来应答命令。

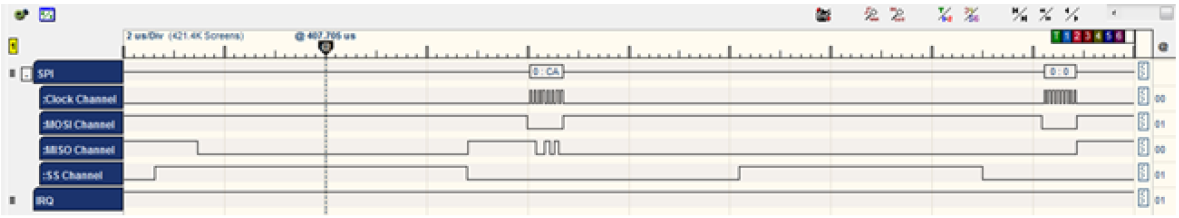


15. WINC 芯片发送寄存器 0x1078 的值（等于 0x00）。

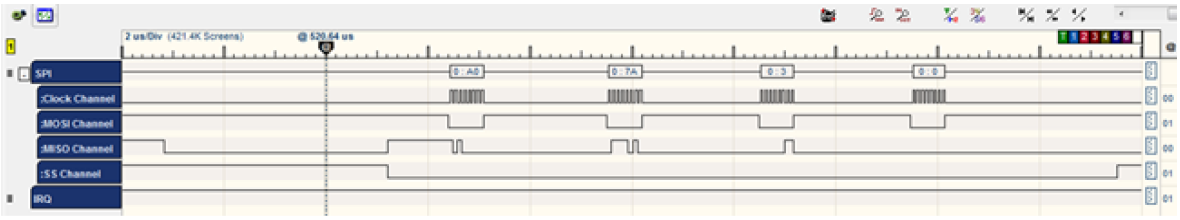



```
Command    CMD_SINGLE_READ:     0xCA         /* single word (4 bytes) read */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;           /* address = 0x1504 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```

16. WINC 通过发送三个字节[CA] [0] [F3]来应答命令。



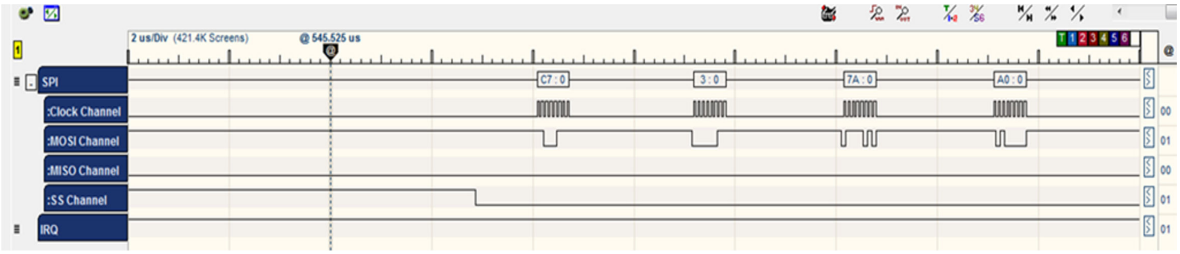
17. WINC 芯片发送寄存器 0x1504 的值（等于 0x037AA0）。

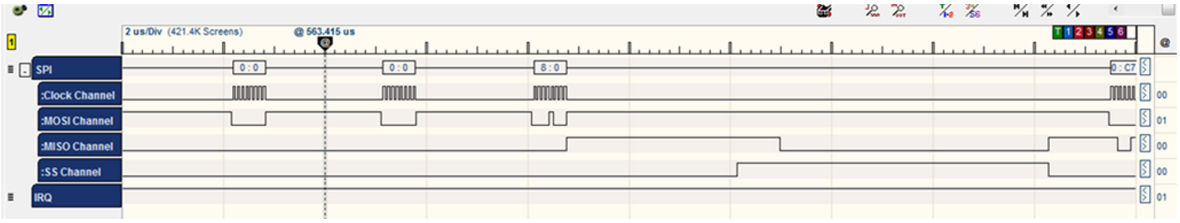


18. WINC 将 HIF 报头写入 DMA 存储器地址。

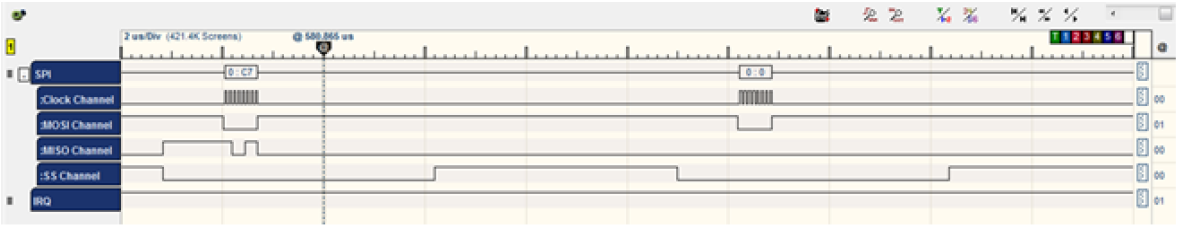
```
u32CurrAddr = dma_addr;
strHif.u16Length=NM_BSP_B_L_16(strHif.u16Length);
ret = nm_write_block(u32CurrAddr, (uint8*)&strHif, M2M_HIF_HDR_OFFSET);
```

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

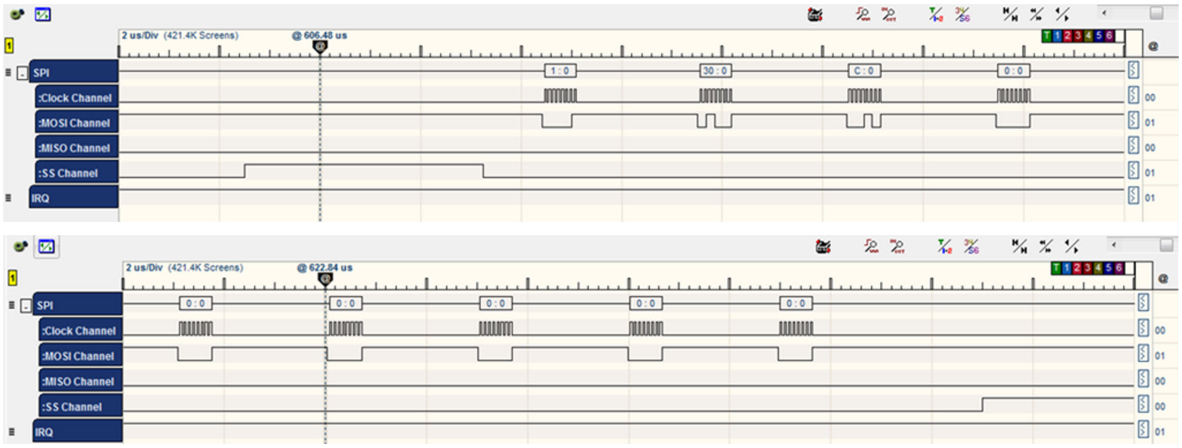




19. WINC 通过发送三个字节[C7] [0] [F3]来应答命令。



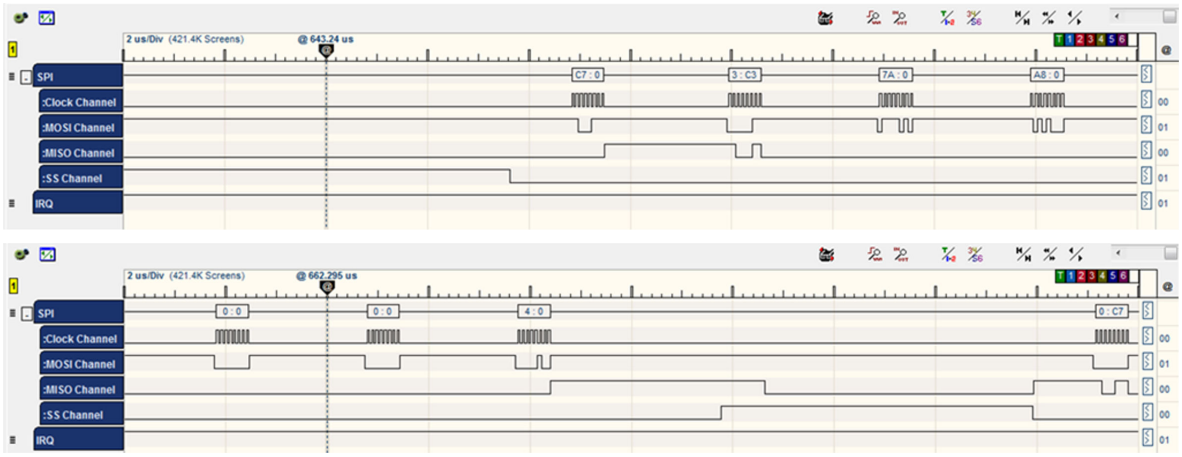
20. HIF 层写入数据。



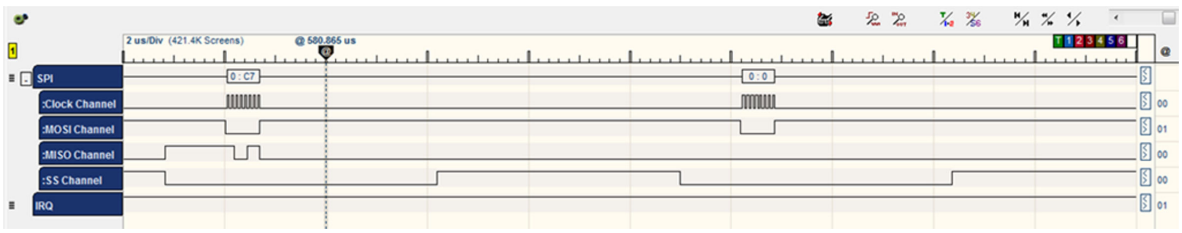
21. HIF 写入控制缓冲区数据（请求帧的一部分）。

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

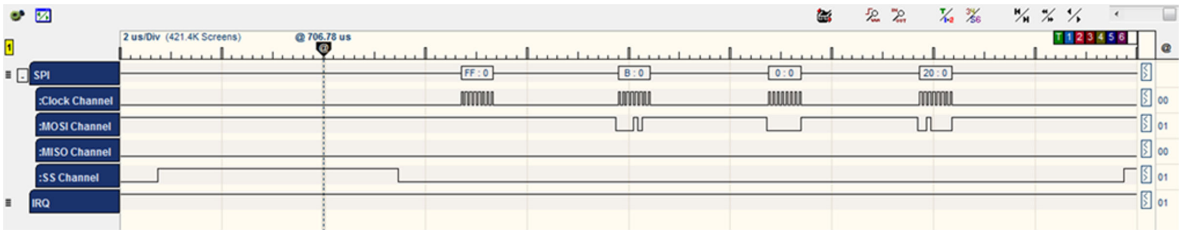
```
Command    CMD_DMA_EXT_WRITE:     0xC7           /* DMA extended write */
        BYTE [0] = CMD_DMA_EXT_WRITE
        BYTE [1] = address >> 16;            /* address = 0x037AA8 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = size >> 16;               /* size =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



22. WINC 通过发送三个字节[C7] [0] [F3]来应答命令。



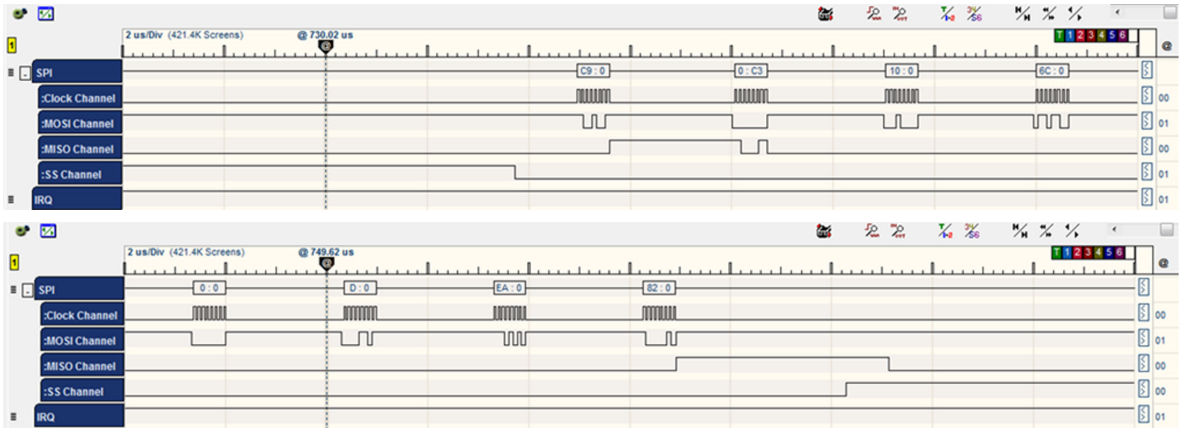
23. HIF 层写入数据。



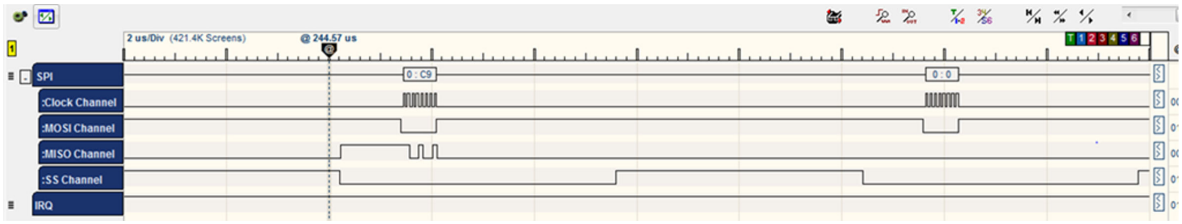
24. HIF 完成将请求数据写入存储器的过程并将中断芯片，通知主机 TX 已完成。

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

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



25. WINC 通过发送两个字节[C9] [0]来应答命令。

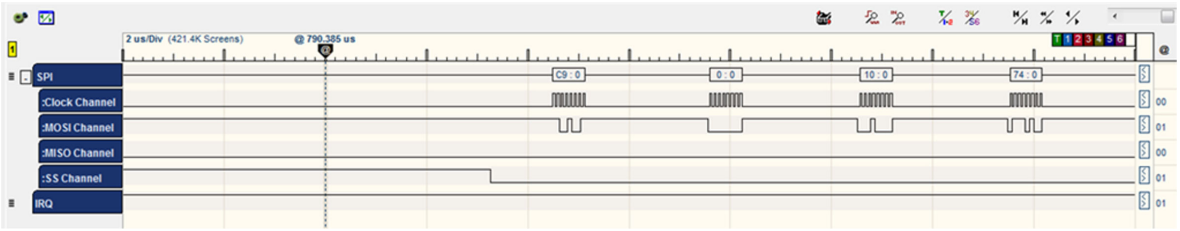


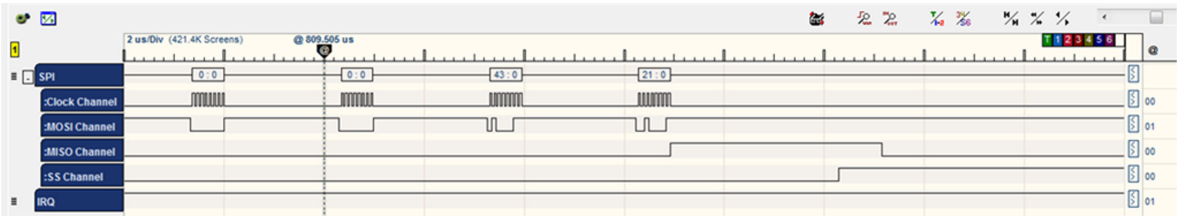
26. HIF 层允许芯片再次进入休眠模式。

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

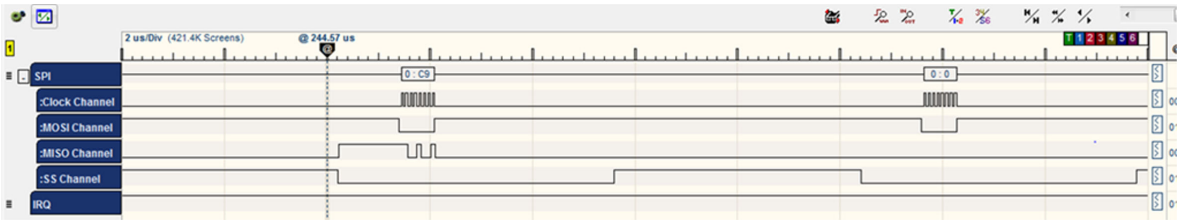
```
    }
}
```

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

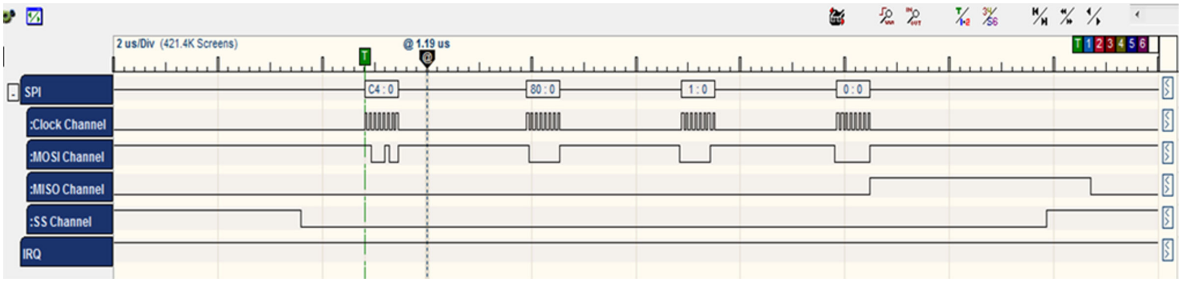




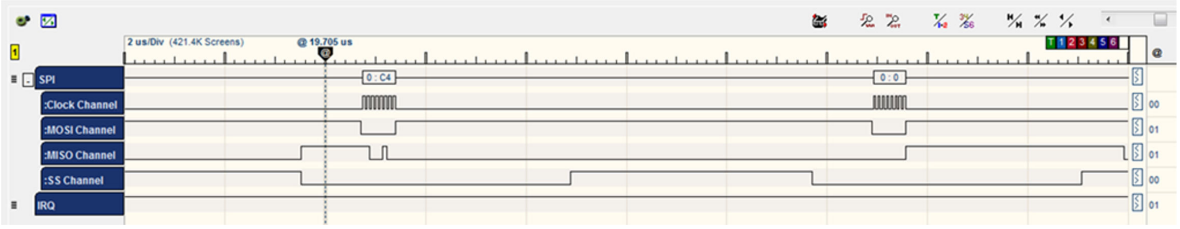
27. WINC 通过发送两个字节[C9] [0]来应答命令。



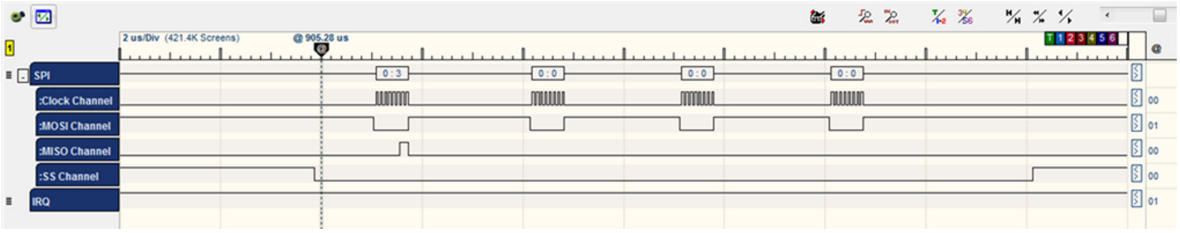

```
Command     CMD_INTERNAL_READ:     0xC4     /* internal register read     */
        BYTE [0] = CMD_INTERNAL_READ
        BYTE [1] = address >> 8;         /* address = 0x01     */
        BYTE [1] |= (1 << 7);            /* clockless register     */
        BYTE [2] = address;
        BYTE [3] =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



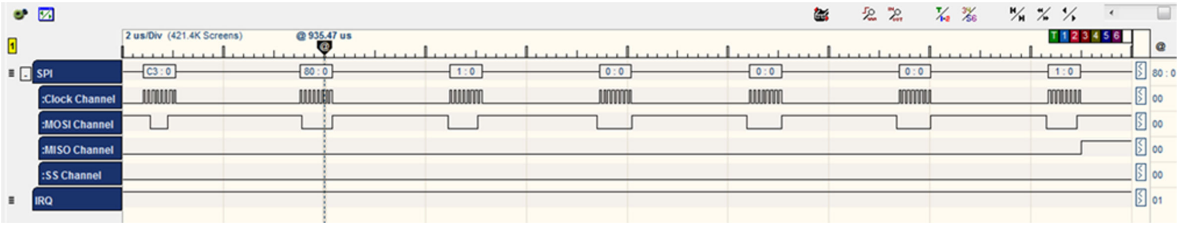
28. WINC 通过发送三个字节[C4] [0] [F3]来应答命令。



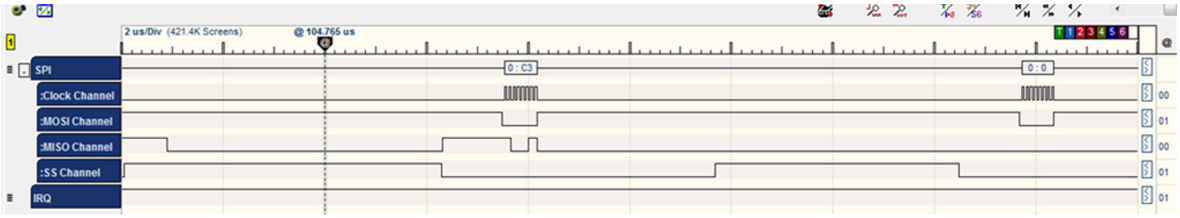
29. WINC 芯片发送寄存器 0x01 的值（等于 0x03）。




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



30. WINC 芯片通过发送两个字节[C3] [0]来应答命令。



31. 至此，HIF 层已完成将扫描 Wi-Fi 请求发布到 WINC 芯片以进行处理。

14.3.2 RX（接收响应）

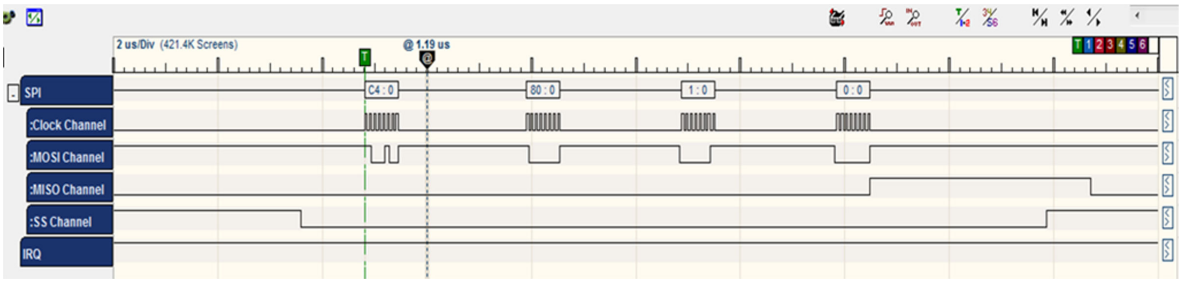
完成所需的操作（扫描 Wi-Fi）后，WINC 将中断主机，以通知其已处理请求。主机将处理该中断以接收响应。

1. `hif_isr` 中的第一步是唤醒 WINC 芯片。

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

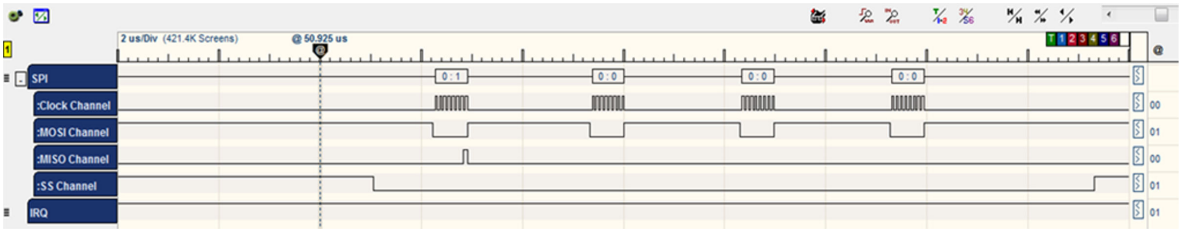
```
    return ret;
}
```

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

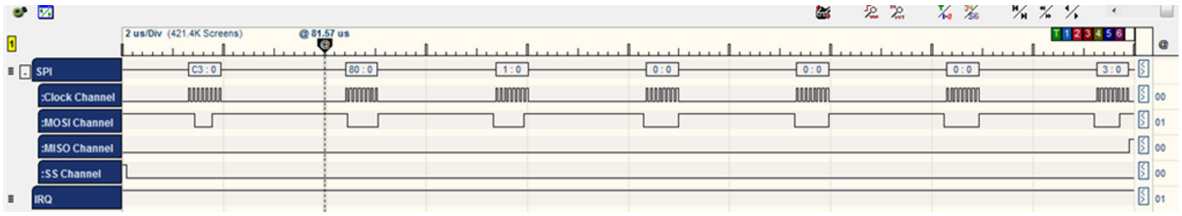


2. WINC 通过发送三个字节[C4] [0] [F3]来应答命令。

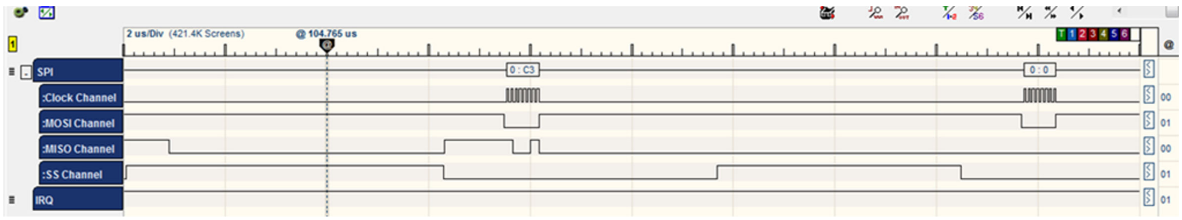
3. WINC 芯片发送寄存器 0x01 的值（等于 0x01）。



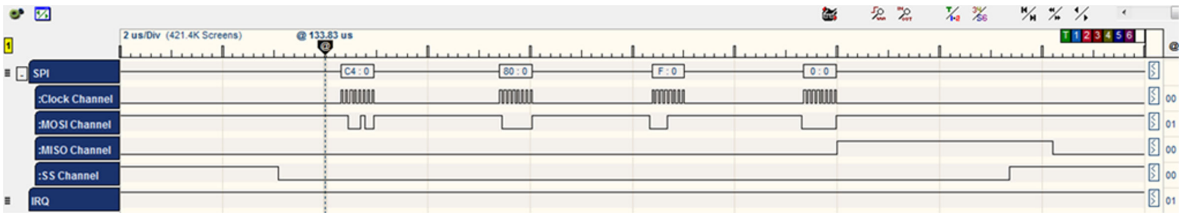

```
Command    CMD_INTERNAL_WRITE:     C3           /*     internal register write */
        BYTE [0] = CMD_INTERNAL_WRITE
        BYTE [1] = address >> 8;                /*     address = 0x01         */
        BYTE [1] |= (1 << 7);                /*     clockless register     */
        BYTE [2] =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



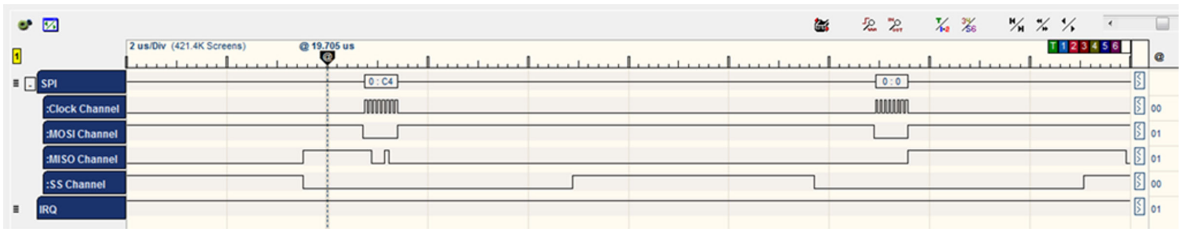
4. WINC 通过发送两个字节[C3] [0]来应答命令。



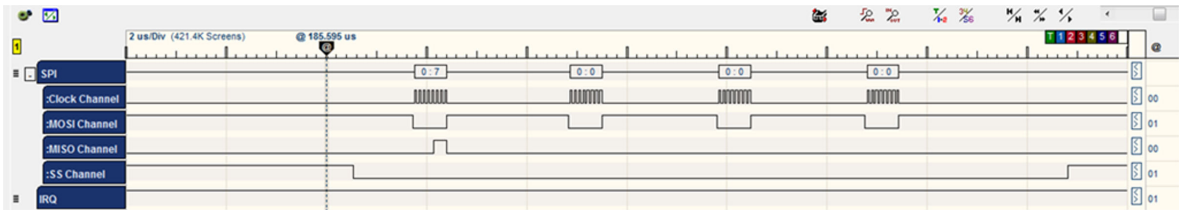

```
Command    CMD_INTERNAL_READ:     0xC4          /*     internal register read    */
        BYTE [0] = CMD_INTERNAL_READ
        BYTE [1] = address >> 8;             /*     address = 0x0F          */
        BYTE [1] |= (1 << 7);                /*     clockless register      */
        BYTE [2] =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



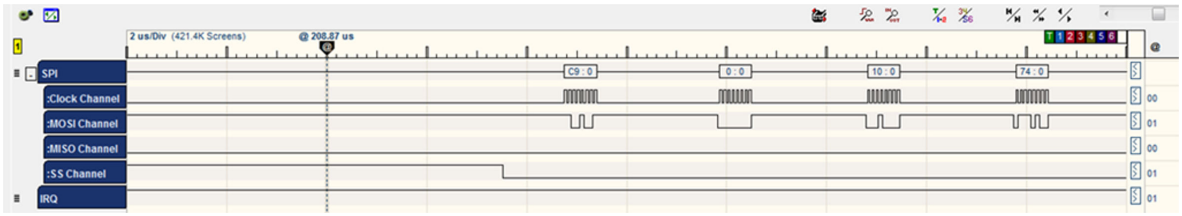
5. WINC 通过发送三个字节[C4] [0] [F3]来应答命令。

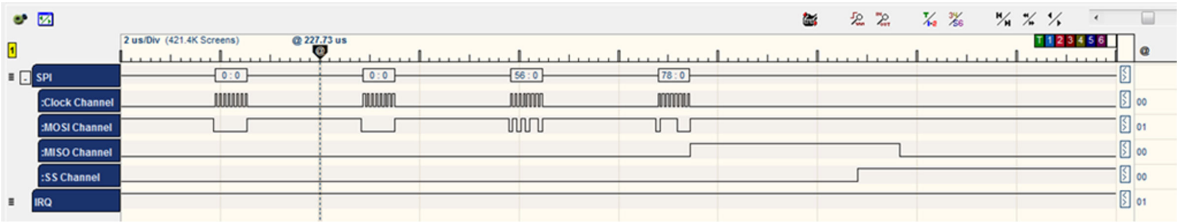


6. 然后 WINC 芯片发送寄存器 0x01 的值（等于 0x07）。

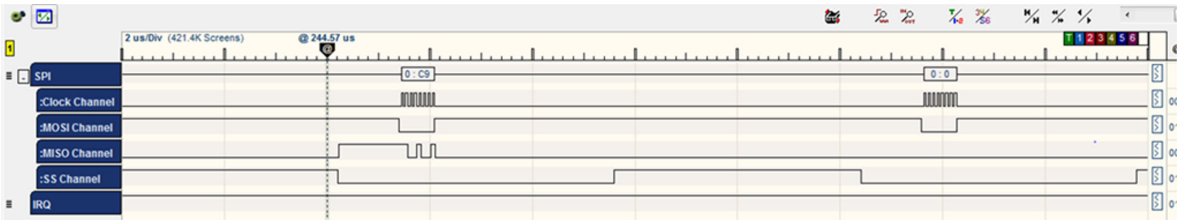



```
Command    CMD_SINGLE_WRITE:0XC9            /* single word write */
        BYTE [0] = CMD_SINGLE_WRITE
        BYTE [1] = address >> 16;        /* WAKE_REG address = 0x1074 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = u32data >> 24;        /* WAKE_VALUE Data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





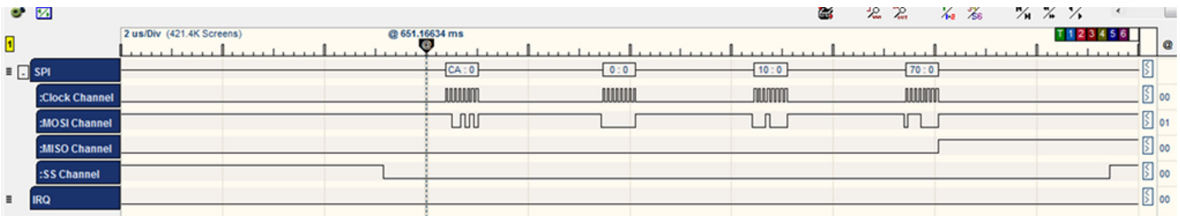
7. 芯片通过发送两个字节[C9] [0]来应答命令。



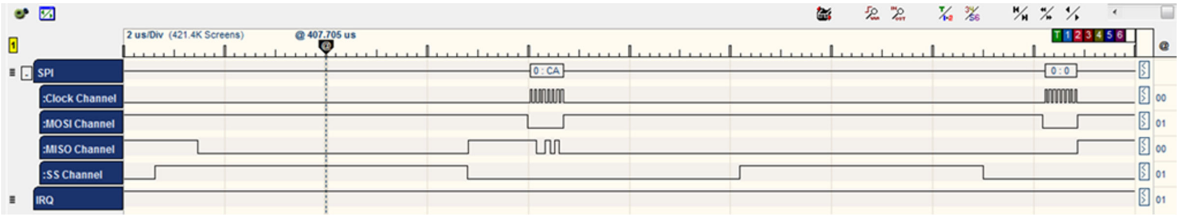
8. 读取寄存器 WIFI_HOST_RCV_CTRL_0 以检查是否有新的中断，并将其清除。

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

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

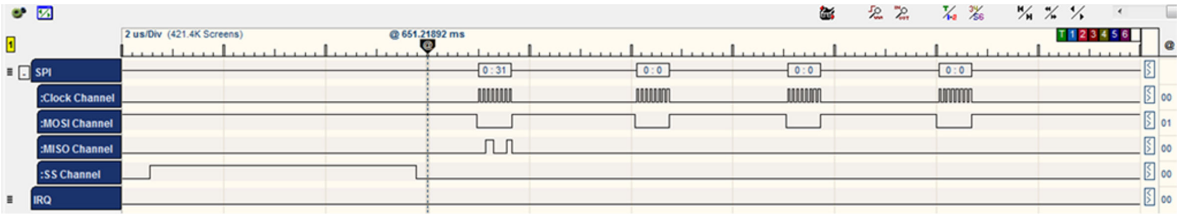
```
Command    CMD_SINGLE_READ:     0xCA         /* single word (4 bytes) read          */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;           /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



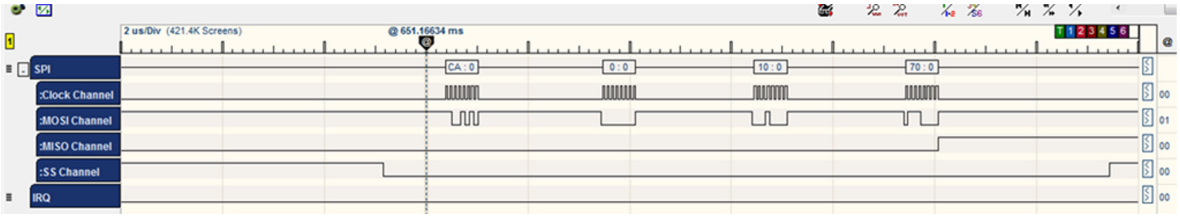
9. WINC 通过发送三个字节[CA] [0] [F3]来应答命令。



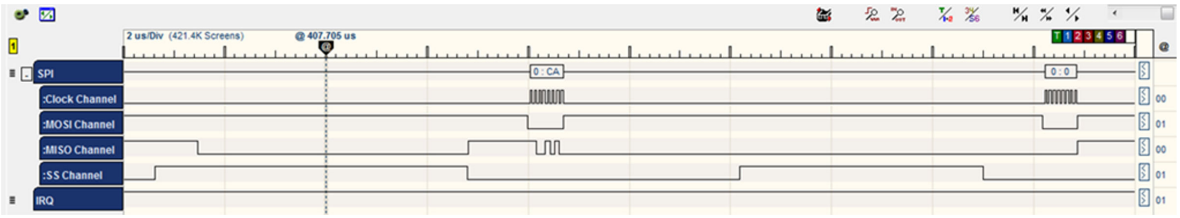
10. WINC 芯片发送寄存器 0x1070 的值（等于 0x31）。



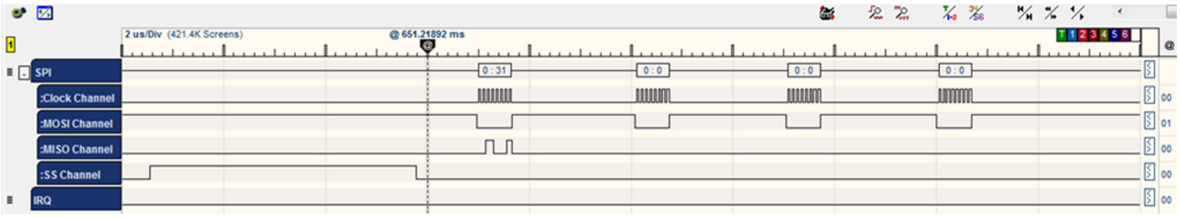

```
Command    CMD_SINGLE_READ:     0xCA         /* single word (4 bytes) read          */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;           /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



11. WINC 通过发送三个字节[CA] [0] [F3]来应答命令。

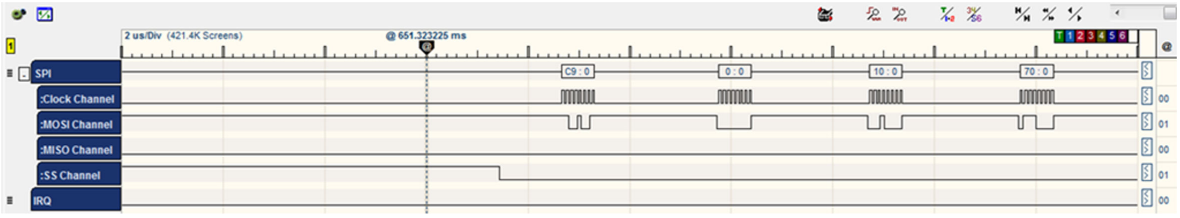


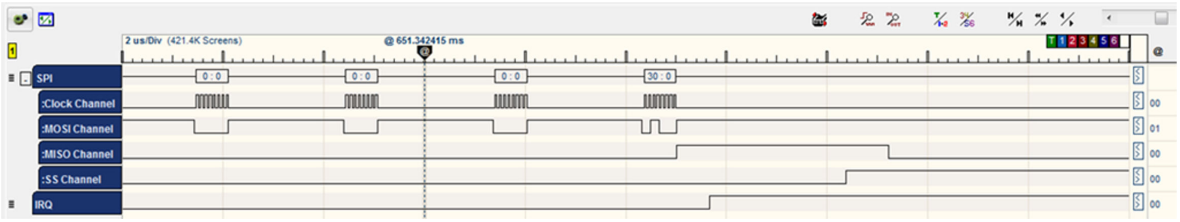
12. WINC 芯片发送寄存器 0x1070 的值（等于 0x31）。



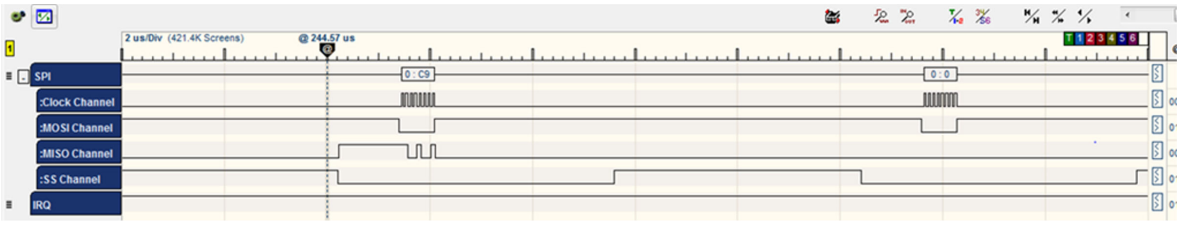
13. 清除 WINC 中断。

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





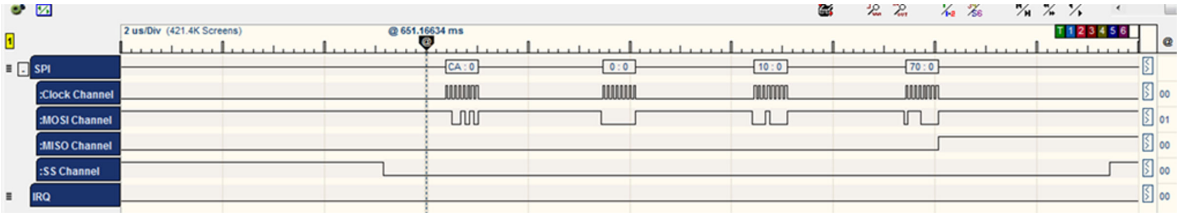
14. 芯片通过发送两个字节[C9] [0]来应答命令。



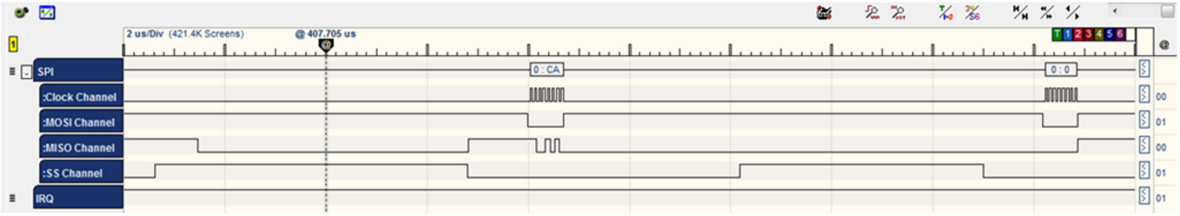
15. HIF 读取数据大小。

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

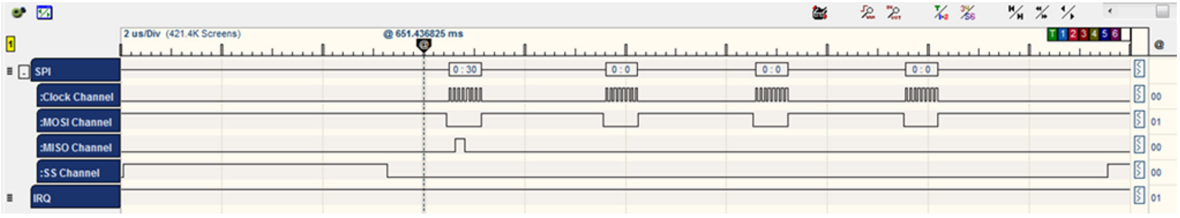
```
Command    CMD_SINGLE_READ:     0xCA         /* single word (4 bytes) read         */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;           /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



16. WINC 通过发送三个字节[CA] [0] [F3]来应答命令。



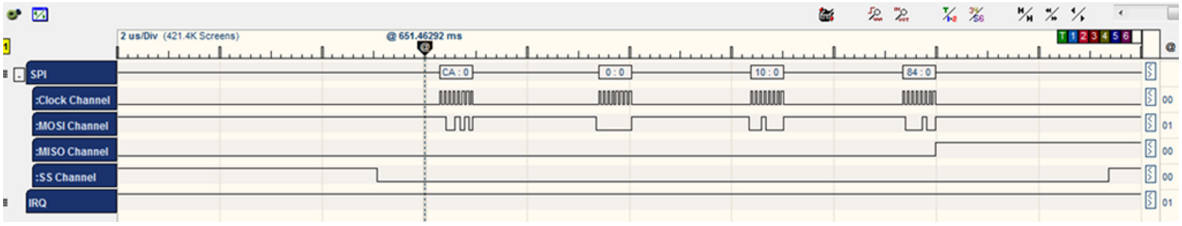
17. WINC 芯片发送寄存器 0x1070 的值（等于 0x30）。



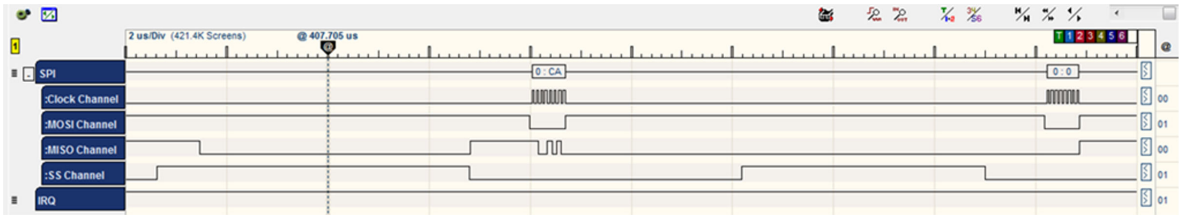
18. HIF 读取 hif 报头地址。

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

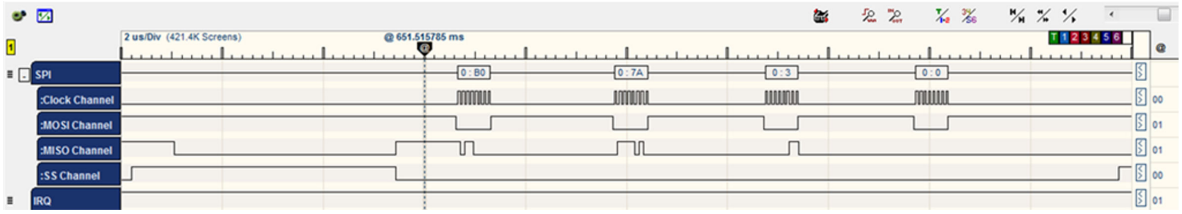
```
Command    CMD_SINGLE_READ:     0xCA        /* single word (4 bytes) read */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_1 address = 0x1084 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



19. WINC 通过发送三个字节[CA] [0] [F3]来应答命令。



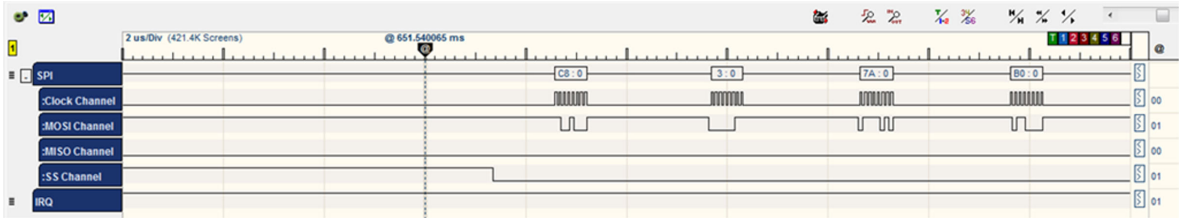
20. WINC 芯片发送寄存器 0x1078 的值（等于 0x037AB0）。

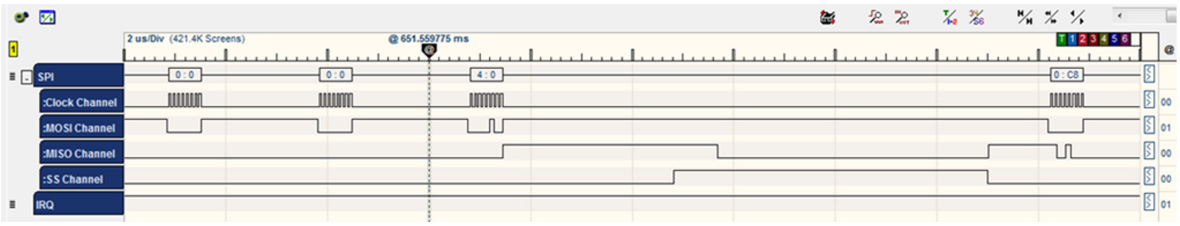


21. HIF 读取 hif 报头数据（作为块）。

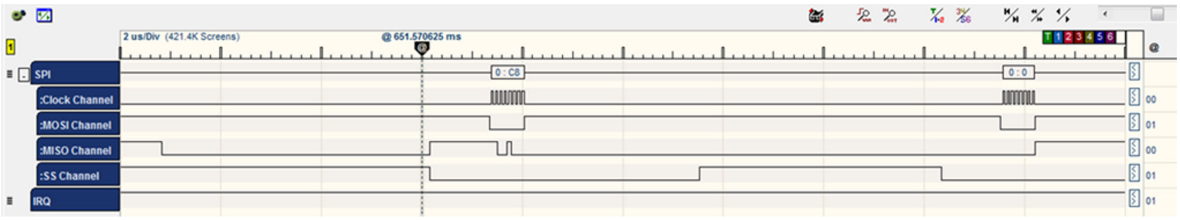
```
ret = nm_read_block(address, (uint8*)&strHif, sizeof(tstrHifHdr));
```

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

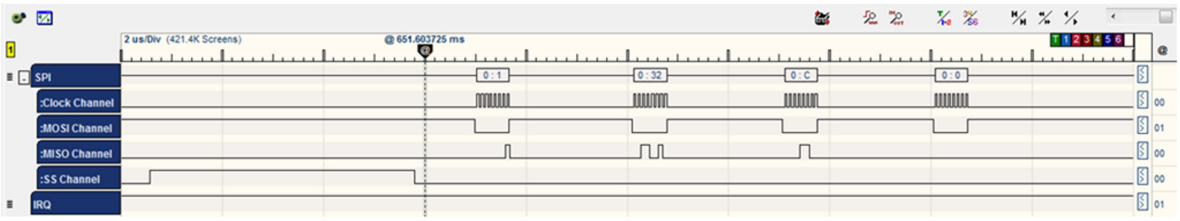




22. WINC 通过发送三个字节[C8] [0] [F3]来应答命令。



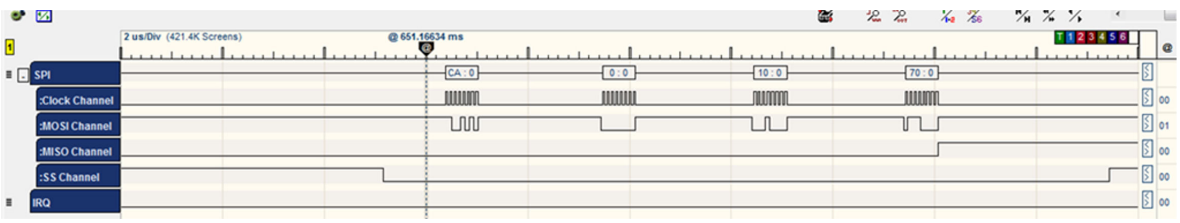
23. WINC 发送数据块（四个字节）。



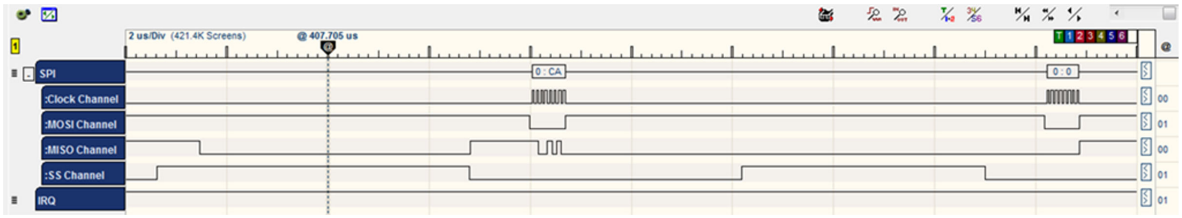
24. HIF 根据收到的尝试接收响应数据有效负载的 hif 报头来调用适当的处理程序。
注： `hif_receive` 获取额外数据。

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

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

}
```

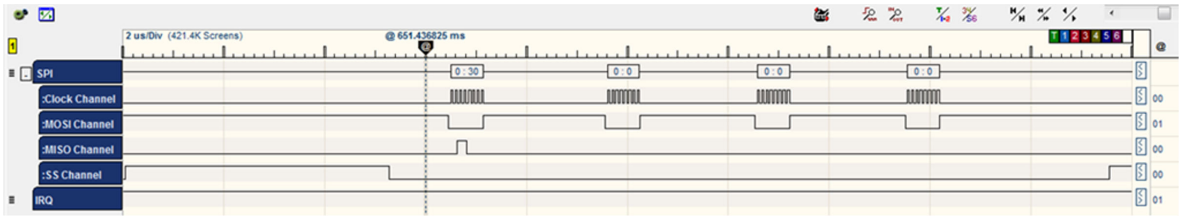
```
Command    CMD_SINGLE_READ:     0xCA        /* single word (4 bytes) read         */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;        /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



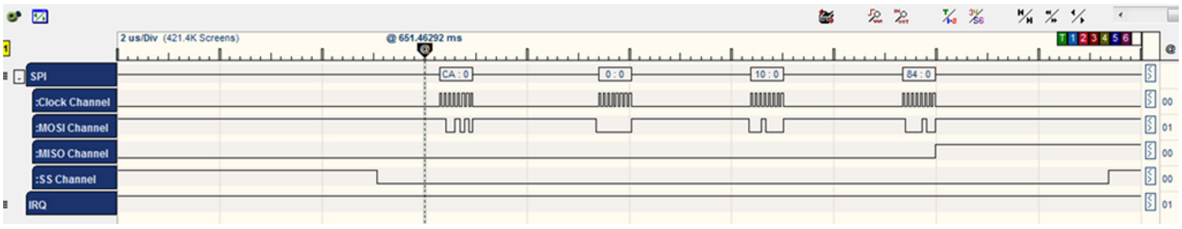
25. WINC 通过发送三个字节[CA] [0] [F3]来应答命令。



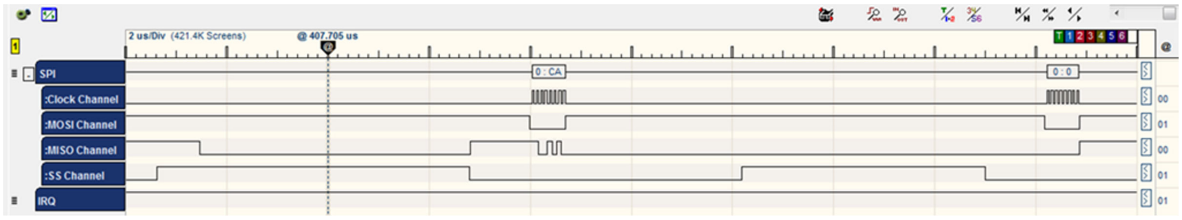
26. WINC 芯片发送寄存器 0x1070 的值（等于 0x30）。



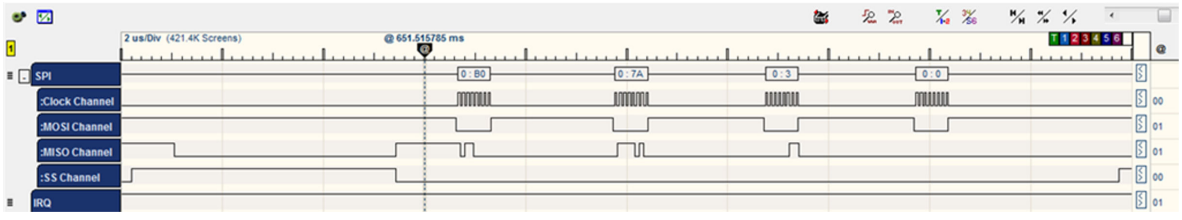

```
Command    CMD_SINGLE_READ:     0xCA         /* single word (4 bytes) read          */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;           /* WIFI_HOST_RCV_CTRL_1 address = 0x1084 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



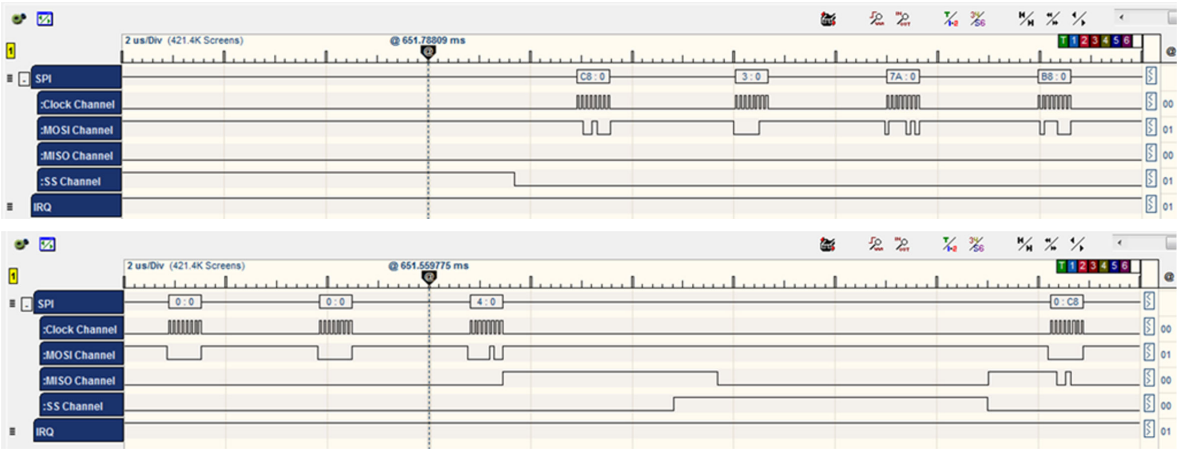
27. WINC 通过发送三个字节[CA] [0] [F3]来应答命令。



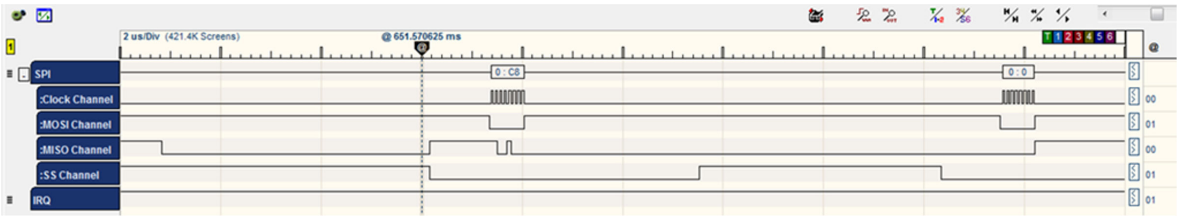
28. WINC 芯片发送寄存器 0x1078 的值（等于 0x037AB0）。



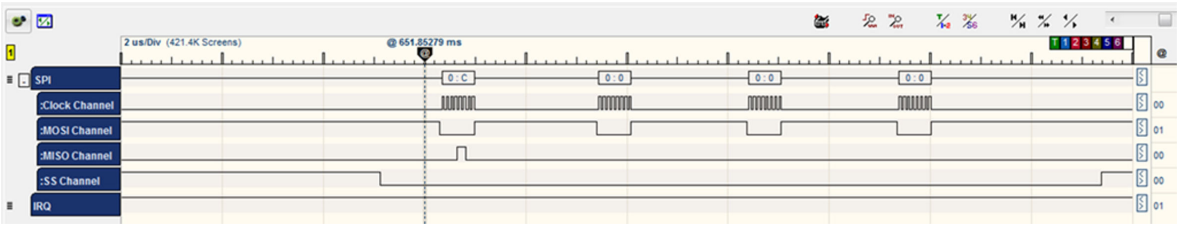

```
Command    CMD_DMA_EXT_READ:     C8         /* dma extended read */
        BYTE [0] = CMD_DMA_EXT_READ
        BYTE [1] = address >> 16;          /* address = 0x037AB8*/
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = size >> 16;
        BYTE [5] = size >>;
     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



29. WINC 通过发送三个字节[C8] [0] [F3]来应答命令。



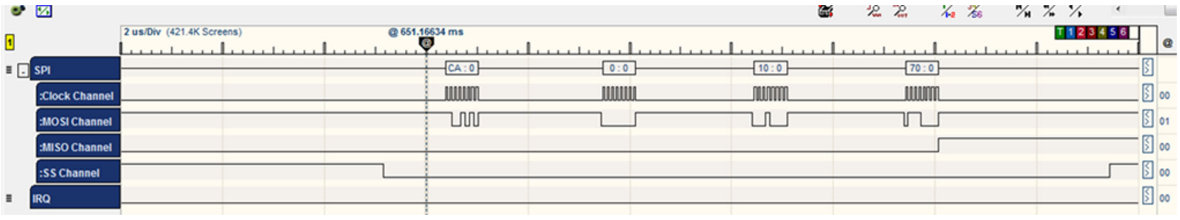
30. WINC 发送数据块（四个字节）。



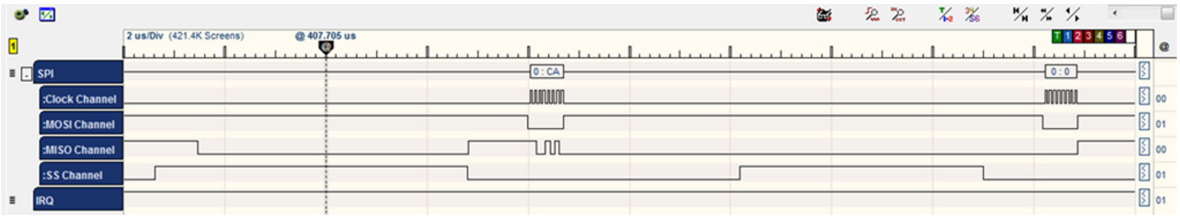
31. 在 HIF 层接收到响应之后，它将中断芯片以发送关于主机 RX 完成的通知。

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

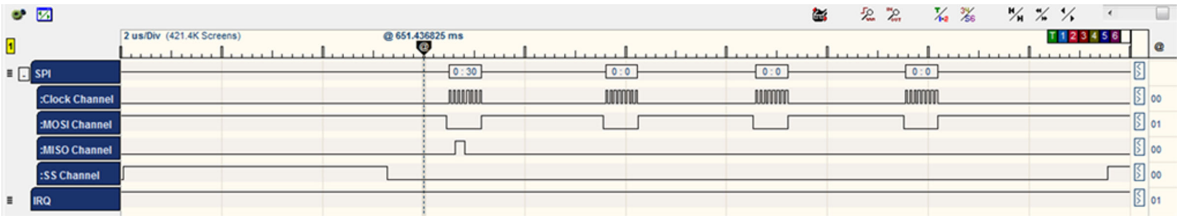
```
Command    CMD_SINGLE_READ:     0xCA         /* single word (4 bytes) read         */
        BYTE [0] = CMD_SINGLE_READ
        BYTE [1] = address >> 16;          /* WIFI_HOST_RCV_CTRL_0 address = 0x1070 */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
```



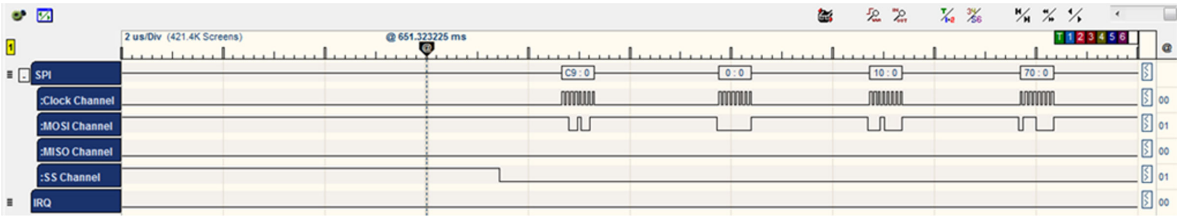
32. WINC 通过发送三个字节[CA] [0] [F3]来应答命令。

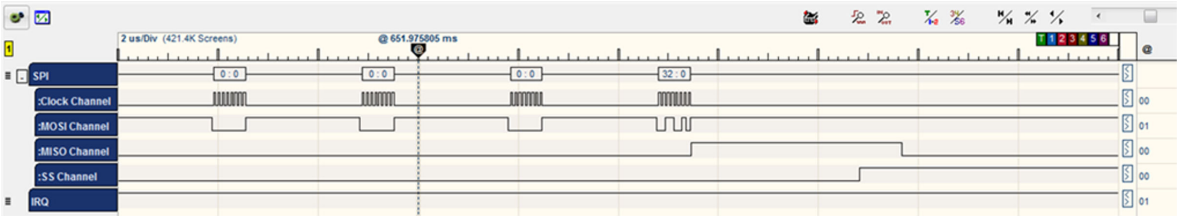


33. WINC 芯片发送寄存器 0x1070 的值（等于 0x30）。

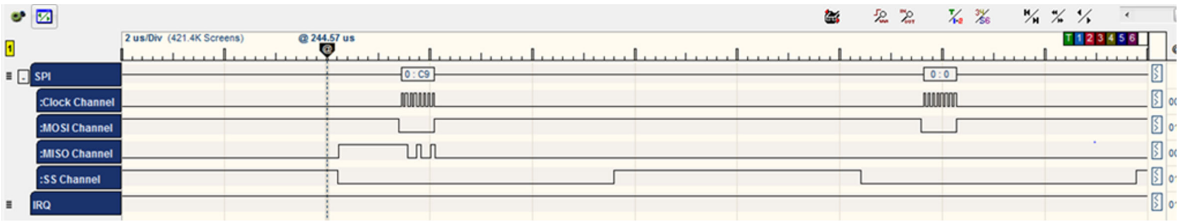



```
Command    CMD_SINGLE_WRITE:0XC9              /* single word write */
        BYTE [0] = CMD_SINGLE_WRITE
        BYTE [1] = address >> 16;         /* WIFI_HOST_RCV_CTRL_0 address = 0x1070  */
        BYTE [2] = address >> 8;
        BYTE [3] = address;
        BYTE [4] = u32data >> 24;         /*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





34. 芯片通过发送两个字节[C9] [0]来应答命令。

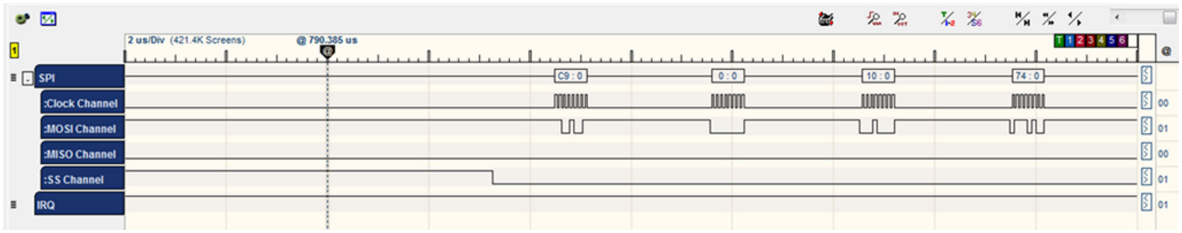


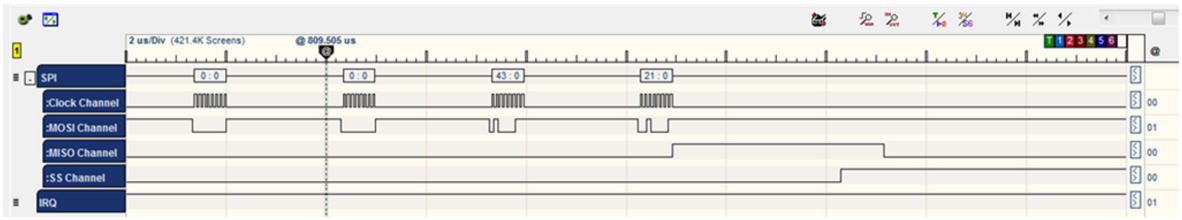
35. HIF 层允许芯片再次进入休眠模式。

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

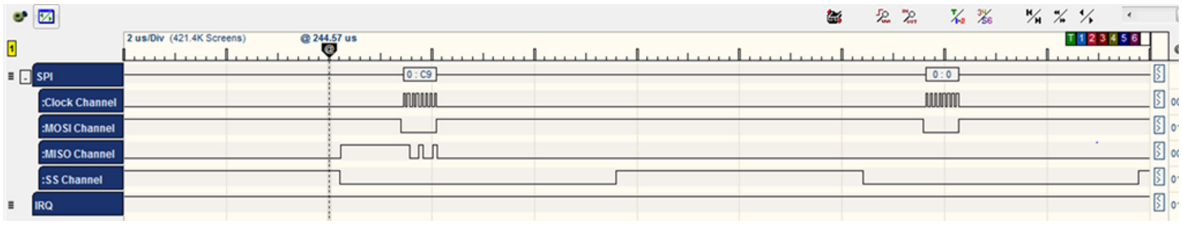
```
        reg &=~(1 << 1);
        ret = nm_write_reg(0x1, reg);
    }
}
```

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



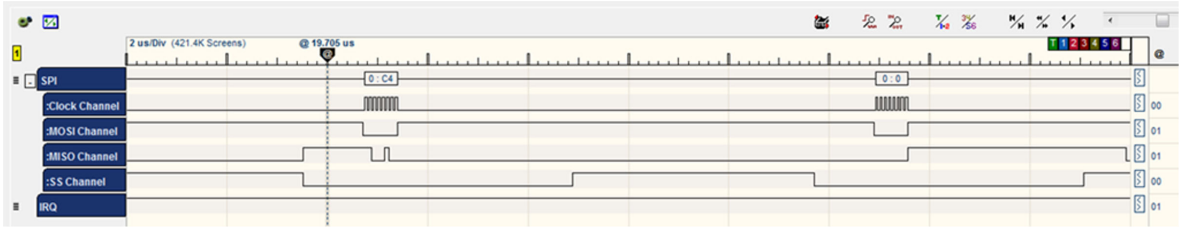


36. WINC 通过发送两个字节[C9] [0]来应答命令。

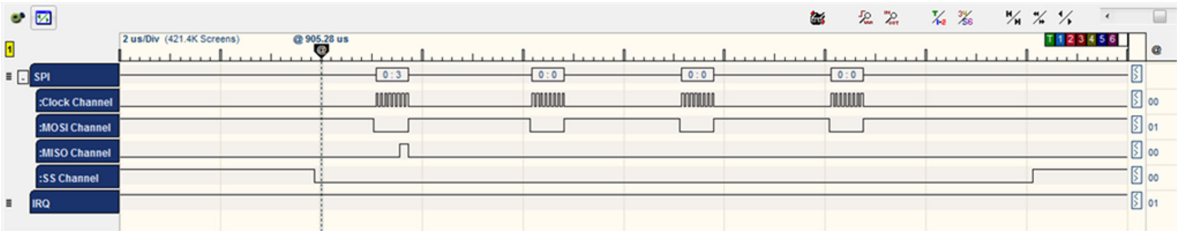



```
Command    CMD_INTERNAL_READ:     0xC4        /* internal register read     */
        BYTE [0] = CMD_INTERNAL_READ
        BYTE [1] = address >> 8;              /* address = 0x01         */
        BYTE [1] |= (1 << 7);                 /* clockless register     */
        BYTE [2] = address;
  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

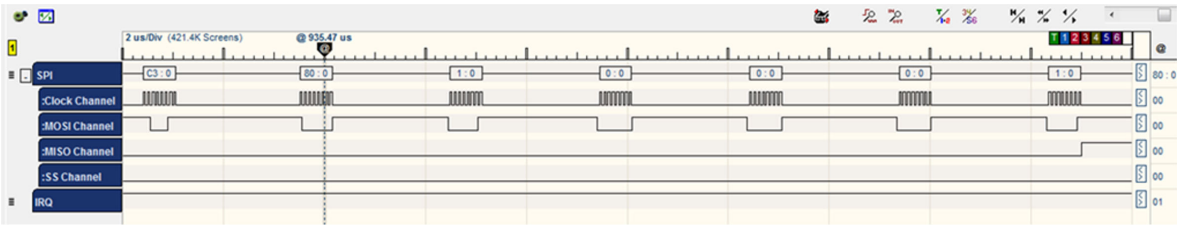
37. WINC 通过发送三个字节[C4] [0] [F3]来应答命令。



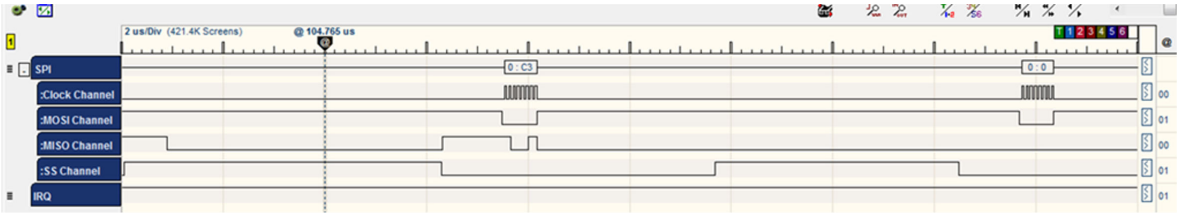
38. 然后 WINC 芯片发送寄存器 0x01 的值（等于 0x03）。




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



39. WINC 芯片通过发送两个字节[C3] [0]来应答命令。



40. 扫描 Wi-Fi 请求将发送到 WINC 芯片，而响应将成功发送到主机。

15. 附录 A. 如何生成证书

15.1 简介

本章介绍使用 OpenSSL 创建和签署自定义证书时需遵循的步骤。要使用本指南，必须在机器上安装 OpenSSL。

OpenSSL 是 SSL 和 TLS 协议的开源实现。核心库采用 C 编程语言编写，不但实现了基本的加密功能，而且提供多种实用功能。

用户可通过以下 URL 下载 OpenSSL：https://www.openssl.org/related/binaries.html。

15.2 步骤

安装 OpenSSL 后，打开 CMD 提示符并导航到 OpenSSL 的安装目录（例如：`C:\OpenSSL-Win64\bin`）。

1. 为 CA（证书颁发机构）生成密钥。要生成 4096 位长的 RSA（创建一个新文件 `CA_KEY.key` 来存储随机密钥），请使用以下命令（CMD）：

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

2. 创建自签名根 CA 证书 `CA_CERT.crt`；您需要使用以下命令（CMD）为根证书提供一些数据：

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

3. 创建自定义证书，该证书由先前创建的 CA 根证书签名。首先，使用以下命令（CMD）生成 `Custom.key`：

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

4. 要使用此生成的密钥生成证书申请文件（CSR），请使用以下命令（CMD）：

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

5. 使用以下命令（CMD）处理证书申请并由根 CA 签名：

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

15.3 限制

以下是 `BigInt_ModExp()` API 的限制。

1. 不支持大于 2048 位的 DHE。
2. 大于 2048 位的 RSA 签名验证通过软件完成；假设典型的公钥为 2^16+1，则每次验证 4096 位需要 4 秒。
3. 不支持大于 2048 位的 RSA 签名生成。

16. 附录 B. X.509 证书格式和转换

16.1 简介

对于 X.509 数字证书，使用最为广泛的两种编码格式为 PEM 和 DER 格式。

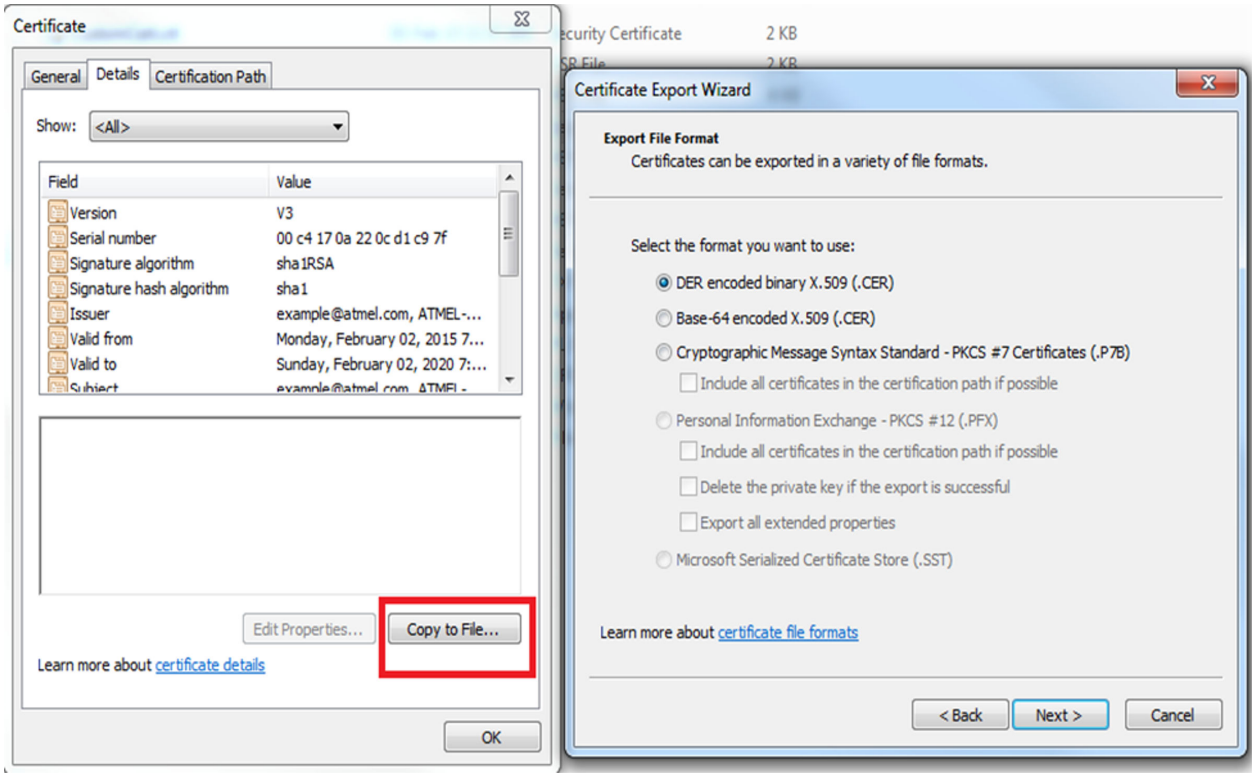
PEM 格式是 DER 的 base64 编码，首尾报文分别为“-----BEGIN CERTIFICATE-----”和“-----END CERTIFICATE-----”。

16.2 不同格式之间的转换

WINC root_certificate_downloader 的当前实现仅支持 DER 格式。如果证书不是 DER 格式，则必须先进行转换。不同格式之间的转换有以下几种方法：

16.2.1 使用 Windows®

自 Windows 7 起，双击 .crt 证书文件，然后转到 Details（详细信息）选项卡，并单击“Copy to File”（复制到文件）。遵循 Certificate Export Wizard（证书导出向导）操作，最后单击 **Finish**（完成）按钮。



16.2.2 使用 OpenSSL

OpenSSL 用于通过以下命令进行证书转换。

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

16.2.3 在线转换

网上有一些实用的在线工具可以实现不同证书格式之间的转换，使用“OpenSSL”等关键字在线搜索即可找到。

17. 参考资料

如需进一步研究，可参考以下文档：

- ATWINC15x0 Wi-Fi Network Controller Software Programming Guide
- ATWINC15x0-MR210xB Data Sheet

如需进一步研究 API，可以参考以下网页：

- Atmel Software Framework for ATWINC1500 (Wi-Fi)

18. 文档版本历史

版本 B（2018 年 10 月）

章节	更改
6.2.3.12 setsockopt	增加了 SOL_SSL_SOCKET 的信息与示例。
10. 无线升级	从本章中删除了“不支持 HTTPS”。
8.5 AP 模式代码示例	增加了“节能”注释。
文档	删除了与 Wi-Fi 直连模式和 Wi-Fi 嗅探模式相关的内容。
4.2 WINC 工作模式	更新了 WINC 工作模式。
8.1 概述 和 8.2 设置 WINC AP 模式	更新了与 WINC1500 v19.6.1 固件对应的“Wi-Fi AP 模式”一章。
5. Wi-Fi 站模式	增加了对 Wi-Fi 站模式的加密凭证存储、简单漫游、多个增益表和主机文件下载的支持。

版本 A（2017 年 5 月）

章节	更改
文档	• 从 Atmel 模板更新为 Microchip 模板。 • 分配了新的 Microchip 文档编号。先前版本为 Atmel 42420 版本 B。 • 增加了 ISBN 编号。

Microchip 网站

Microchip 网站 http://www.microchip.com/为客户提供在线支持。客户可通过该网站方便地获取文件和信息。只要使用常用的互联网浏览器即可访问，网站提供以下信息：

- **产品支持**——数据手册和勘误表、应用笔记和示例程序、设计资源、用户指南以及硬件支持文档、最新的软件版本以及归档软件
- **一般技术支持**——常见问题（FAQ）、技术支持请求、在线讨论组以及 Microchip 顾问计划成员名单
- **Microchip 业务**——产品选型和订购指南、最新 Microchip 新闻稿、研讨会和活动安排表、Microchip 销售办事处、代理商以及工厂代表列表

变更通知客户服务

Microchip 的变更通知客户服务有助于客户了解 Microchip 产品的最新信息。注册客户可在他们感兴趣的某个产品系列或开发工具发生变更、更新、发布新版本或勘误表时，收到电子邮件通知。

欲注册，请登录 Microchip 网站 http://www.microchip.com/。在“支持”（Support）下，点击“变更通知客户”（Customer Change Notification）服务后按照注册说明完成注册。

客户支持

Microchip 产品的用户可通过以下渠道获得帮助：

- 代理商或代表
- 当地销售办事处
- 应用工程师（FAE）
- 技术支持

客户应联系其代理商、代表或应用工程师（FAE）寻求支持。当地销售办事处也可为客户提供帮助。本文档后附有销售办事处的联系方式。

也可通过以下网站获得技术支持： http://www.microchip.com/support

Microchip 器件代码保护功能

请注意以下有关 Microchip 器件代码保护功能的要点：

- Microchip 的产品均达到 Microchip 数据手册中所述的技术指标。
- Microchip 确信：在正常使用的情况下，Microchip 系列产品是当今市场上同类产品中最安全的产品之一。
- 目前，仍存在着恶意、甚至是非法破坏代码保护功能的行为。就我们所知，所有这些行为都不是以 Microchip 数据手册中规定的操作规范来使用 Microchip 产品的。这样做的人极可能侵犯了知识产权。
- Microchip 愿意与关心代码完整性的客户合作。
- Microchip 或任何其他半导体厂商均无法保证其代码的安全性。代码保护并不意味着我们保证产品是“牢不可破”的。

代码保护功能处于持续发展中。Microchip 承诺将不断改进产品的代码保护功能。任何试图破坏 Microchip 代码保护功能的行为均可视为违反了《数字器件千年版权法案（Digital Millennium Copyright Act）》。如

果这种行为导致他人在未经授权的情况下，能访问您的软件或其他受版权保护的成果，您有权依据该法案提起诉讼，从而制止这种行为。

法律声明

提供本文档的中文版本仅为了便于理解。请勿忽视文档中包含的英文部分，因为其中提供了有关 Microchip 产品性能和使用情况的有用信息。Microchip Technology Inc.及其分公司和相关公司、各级主管与员工及事务代理机构对译文中可能存在的任何差错不承担任何责任。建议参考 Microchip Technology Inc.的英文原版文档。

本出版物中所述的器件应用信息及其他类似内容仅为您提供便利，它们可能由更新之信息所替代。确保应用符合技术规范，是您自身应负的责任。Microchip 对这些信息不作任何明示或暗示、书面或口头、法定或其他形式的声明或担保，包括但不限于针对其使用情况、质量、性能、适销性或特定用途的适用性的声明或担保。Microchip 对因这些信息及使用这些信息而引起的后果不承担任何责任。如果将 Microchip 器件用于生命维持和/或生命安全应用，一切风险由买方自负。买方同意在由此引发任何一切伤害、索赔、诉讼或费用时，会维护和保障 Microchip 免于承担法律责任，并加以赔偿。除非另外声明，否则在 Microchip 知识产权保护下，不得暗中或以其他方式转让任何许可证。

商标

Microchip 的名称和徽标组合、Microchip 徽标、Adaptec、AnyRate、AVR、AVR 徽标、AVR Freaks、BesTime、BitCloud、chipKIT、chipKIT 徽标、CryptoMemory、CryptoRF、dsPIC、FlashFlex、flexPWR、HELDO、IGLOO、JukeBlox、KeeLoq、Kleer、LANCheck、LinkMD、maXStylus、maXTouch、MediaLB、megaAVR、Microsemi、Microsemi 徽标、MOST、MOST 徽标、MPLAB、OptoLyzer、PackeTime、PIC、picoPower、PICSTART、PIC32 徽标、PolarFire、Prochip Designer、QTouch、SAM-BA、SenGenuity、SpyNIC、SST、SST 徽标、SuperFlash、Symmetricom、SyncServer、Tachyon、TempTrackr、TimeSource、tinyAVR、UNI/O、Vectron 及 XMEGA 均为 Microchip Technology Inc.在美国和其他国家或地区的注册商标。

APT、ClockWorks、The Embedded Control Solutions Company、EtherSynch、FlashTec、Hyper Speed Control、HyperLight Load、IntelliMOS、Libero、motorBench、mTouch、Powermite 3、PrecisionEdge、ProASIC、ProASIC Plus、ProASIC Plus 徽标、Quiet-Wire、SmartFusion、SyncWorld、Temux、TimeCesium、TimeHub、TimePictra、TimeProvider、Vite、WinPath 和 ZL 均为 Microchip Technology Inc.在美国的注册商标。

Adjacent Key Suppression、AKS、Analog-for-the-Digital Age、Any Capacitor、AnyIn、AnyOut、BlueSky、BodyCom、CodeGuard、CryptoAuthentication、CryptoAutomotive、CryptoCompanion、CryptoController、dsPICDEM、dsPICDEM.net、Dynamic Average Matching、DAM、ECAN、EtherGREEN、In-Circuit Serial Programming、ICSP、INICnet、Inter-Chip Connectivity、JitterBlocker、KleerNet、KleerNet 徽标、memBrain、Mindi、MiWi、MPASM、MPF、MPLAB Certified 徽标、MPLIB、MPLINK、MultiTRAK、NetDetach、Omniscient Code Generation、PICDEM、PICDEM.net、PICkit、PICtail、PowerSmart、PureSilicon、QMatrix、REAL ICE、Ripple Blocker、SAM-ICE、Serial Quad I/O、SMART-I.S.、SQI、SuperSwitcher、SuperSwitcher II、Total Endurance、TSHARC、USBCheck、VariSense、ViewSpan、WiperLock、Wireless DNA 和 ZENA 均为 Microchip Technology Inc.在美国和其他国家或地区的商标。

SQTP 为 Microchip Technology Incorporated 在美国的服务标记。

Adaptec 徽标、Frequency on Demand、Silicon Storage Technology 和 Symmcom 为 Microchip Technology Inc.在除美国外的国家或地区的注册商标。

ISBN： 978-1-5224-4546-3

质量管理体系

有关 Microchip 质量管理体系的更多信息，请访问 www.microchip.com/quality。

全球销售及服务网点

美洲	亚太地区	亚太地区	欧洲
公司总部 2355 West Chandler Blvd. 钱德勒，亚利桑那州 85224-6199 电话：480-792-7200 传真：480-792-7277 技术支持： http://www.microchip.com/support 网址： www.microchip.com **亚特兰大** 德卢斯，佐治亚州 电话：678-957-9614 传真：678-957-1455 **奥斯汀，德克萨斯州** 电话：512-257-3370 **波士顿** 韦斯特伯鲁，马萨诸塞州 电话：774-760-0087 传真：774-760-0088 **芝加哥** 艾塔斯卡，伊利诺伊州 电话：630-285-0071 传真：630-285-0075 **达拉斯** 阿迪森，德克萨斯州 电话：972-818-7423 传真：972-818-2924 **底特律** 诺维，密歇根州 电话：248-848-4000 **休斯顿，德克萨斯州** 电话：281-894-5983 **印第安纳波利斯** 诺布尔斯维尔，印第安纳州 电话：317-773-8323 传真：317-773-5453 电话：317-536-2380 **洛杉矶** 米慎维荷，加利福尼亚州 电话：949-462-9523 传真：949-462-9608 电话：951-273-7800 **罗利，北卡罗来纳州** 电话：919-844-7510 **纽约，纽约州** 电话：631-435-6000 **圣何塞，加利福尼亚州** 电话：408-735-9110 电话：408-436-4270 **加拿大 - 多伦多** 电话：905-695-1980 传真：905-695-2078	**澳大利亚 - 悉尼** 电话：61-2-9868-6733 **中国 - 北京** 电话：86-10-8569-7000 **中国 - 成都** 电话：86-28-8665-5511 **中国 - 重庆** 电话：86-23-8980-9588 **中国 - 东莞** 电话：86-769-8702-9880 **中国 - 广州** 电话：86-20-8755-8029 **中国 - 杭州** 电话：86-571-8792-8115 **中国 - 香港特别行政区** 电话：852-2943-5100 **中国 - 南京** 电话：86-25-8473-2460 **中国 - 青岛** 电话：86-532-8502-7355 **中国 - 上海** 电话：86-21-3326-8000 **中国 - 沈阳** 电话：86-24-2334-2829 **中国 - 深圳** 电话：86-755-8864-2200 **中国 - 苏州** 电话：86-186-6233-1526 **中国 - 武汉** 电话：86-27-5980-5300 **中国 - 西安** 电话：86-29-8833-7252 **中国 - 厦门** 电话：86-592-2388138 **中国 - 珠海** 电话：86-756-3210040	**印度 - 班加罗尔** 电话：91-80-3090-4444 **印度 - 新德里** 电话：91-11-4160-8631 **印度 - 浦那** 电话：91-20-4121-0141 **日本 - 大阪** 电话：81-6-6152-7160 **日本 - 东京** 电话：81-3-6880- 3770 **韩国 - 大邱** 电话：82-53-744-4301 **韩国 - 首尔** 电话：82-2-554-7200 **马来西亚 - 吉隆坡** 电话：60-3-7651-7906 **马来西亚 - 槟榔屿** 电话：60-4-227-8870 **菲律宾 - 马尼拉** 电话：63-2-634-9065 **新加坡** 电话：65-6334-8870 **台湾地区 - 新竹** 电话：886-3-577-8366 **台湾地区 - 高雄** 电话：886-7-213-7830 **台湾地区 - 台北** 电话：886-2-2508-8600 **泰国 - 曼谷** 电话：66-2-694-1351 **越南 - 胡志明市** 电话：84-28-5448-2100	**奥地利 - 韦尔斯** 电话：43-7242-2244-39 传真：43-7242-2244-393 **丹麦 - 哥本哈根** 电话：45-4450-2828 传真：45-4485-2829 **芬兰 - 埃斯波** 电话：358-9-4520-820 **法国 - 巴黎** 电话：33-1-69-53-63-20 传真：33-1-69-30-90-79 **德国 - 加兴** 电话：49-8931-9700 **德国 - 哈恩** 电话：49-2129-3766400 **德国 - 海尔布隆** 电话：49-7131-72400 **德国 - 卡尔斯鲁厄** 电话：49-721-625370 **德国 - 慕尼黑** 电话：49-89-627-144-0 传真：49-89-627-144-44 **德国 - 罗森海姆** 电话：49-8031-354-560 **以色列 - 若那那市** 电话：972-9-744-7705 **意大利 - 米兰** 电话：39-0331-742611 传真：39-0331-466781 **意大利 - 帕多瓦** 电话：39-049-7625286 **荷兰 - 德卢内市** 电话：31-416-690399 传真：31-416-690340 **挪威 - 特隆赫姆** 电话：47-72884388 **波兰 - 华沙** 电话：48-22-3325737 **罗马尼亚 - 布加勒斯特** 电话：40-21-407-87-50 **西班牙 - 马德里** 电话：34-91-708-08-90 传真：34-91-708-08-91 **瑞典 - 哥德堡** 电话：46-31-704-60-40 **瑞典 - 斯德哥尔摩** 电话：46-8-5090-4654 **英国 - 沃金厄姆** 电话：44-118-921-5800 传真：44-118-921-5820