CC3120, CC3220 SimpleLink™ Wi-Fi® and Internet of Things Network Processor

Programmer's Guide

Literature Number: SWRU455E
February 2017–Revised February 2018

Contents

List of Figures

List of Tables

Overview

The CC3120 and CC3220 devices are part of the SimpleLink™ microcontroller (MCU) platform which consists of Wi-Fi®, *Bluetooth*® low energy, Sub-1 GHz and host MCUs, which all share a common, easy-to-use development environment with a single core software development kit (SDK) and rich tool set. A one-time integration of the SimpleLink platform enables you to add any combination of the portfolio's devices into your design, allowing 100 percent code reuse when your design requirements change. For more information, visit www.ti.com/simplelink.

The SimpleLink Wi-Fi Internet-on-a chip™ family of devices from Texas Instruments™ provides a suite of integrated protocols for Wi-Fi and Internet connectivity, to dramatically simplify the implementation of Internet-enabled devices and applications.

This document provides software (SW) programmers with all of the required knowledge for working with the networking subsystem of the SimpleLink Wi-Fi devices. This guide provides basic guidelines for writing robust, optimized networking host applications, and describes the capabilities of the networking subsystem. The guide contains some example code snapshots, to give users an idea of how to work with the host driver. More comprehensive code examples can be found in the formal software development kit (SDK). This guide does not provide a detailed description of the host driver APIs.

This chapter gives a brief introduction to the networking subsystem, lists the key features of the device, and provides an overview of the host driver.

Trademarks

Introduction

The SimpleLink Wi-Fi CC3120 wireless network processor allows the connection of any low-cost, low-power microcontroller (MCU) to the Internet of Things (IoT), using standard communication interfaces such as SPI or UART.

The SimpleLink Wi-Fi CC3220x is a wireless MCU with an integrated high-performance ARM® Cortex®-M4 MCU, built-in Wi-Fi, and a networking subsystem, allowing developers to write an entire application with a single-chip solution.

The CC3120 and CC3220 devices are the second generation of TI's Internet-on-a chip solutions. This generation introduces new features and capabilities that further simplify connectivity of devices to the Internet. The new capabilities include:

- Support for IPv6
- Improved Wi-Fi provisioning
- Improved power consumption
- More concurrent opened BSD and SSL/TLS sockets
- HTTPS – Integrated secure http server
- File system security capabilities
- Image programming
- Wi-Fi access point (AP) with support of up to four stations

Figure 1 shows a block diagram of the SimpleLink Wi-Fi solution at a high level.

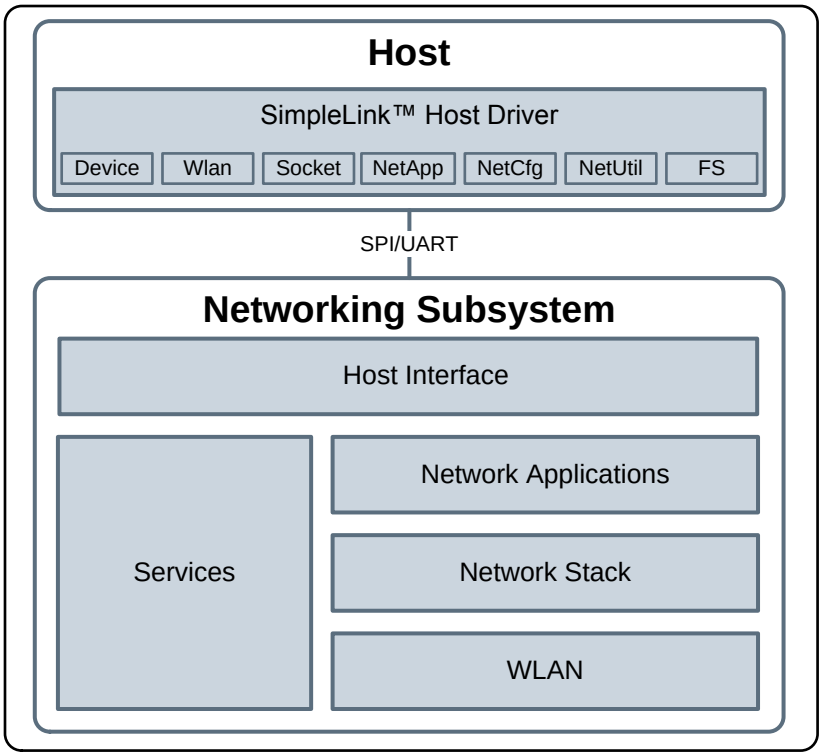

Figure 1. SimpleLink Wi-Fi Solution Block Diagram

In the CC3220 wireless MCU the host is a Cortex-M4 core, the networking subsystem is built into the device as an additional peripheral, and the interface between the Cortex-M4 Core and the networking subsystem is internal.

The host driver is the same for the CC3120 and CC3220, and the networking capabilities are similar for both devices. The network stack is fully implemented in the networking subsystem, thereby offloading the networking activities from the host MCU.

A simple application that only sends a UDP datagram on the local network requires minimum APIs as follows:

```
sl_Start              Start the SimpleLink device in Wi-Fi Station mode
sl_WlanConnect        Connect to a Wi-Fi network
sl_Socket             Create a socket
sl_SendTo             Sends UDP data
sl_Close              Close the socket
sl_Stop               Stops the SimpleLink device
```

NOTE: The target application can choose to use the preferred networks option (profiles), instead of using the sl_WlanConnect command. This option allows the host application to completely offload the entire management of the WLAN connection.

Figure 2 shows a more detailed block diagram of the networking subsystem.

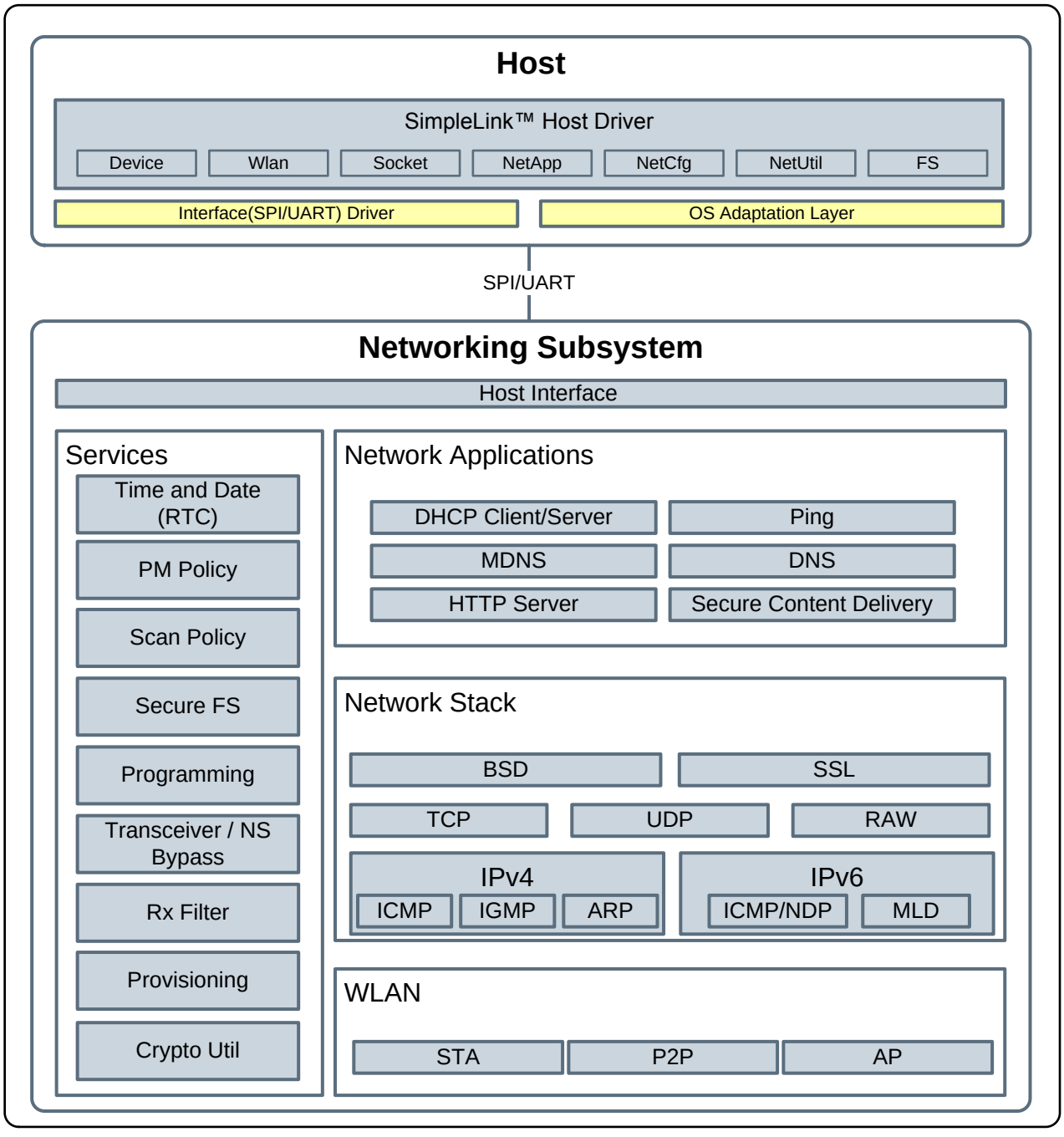

Figure 2. Networking Subsystem Block Diagram

Acronyms and Terminologies

Table 0-1 lists the acronyms and terms used in this document.

Table 0-1. Acronyms and Terminologies

Acronym/Terminology	Description
Host	Host refers to an embedded MCU running the SimpleLink software driver and uses the SimpleLink device as a networking peripheral
AP	Wi-Fi access point
STA	Wi-Fi station
FW	Firmware software
LAN	Local area network
WLAN	Wireless local area network
IE	Information element
OUI	Organization unique identifier
P2P	Wi-Fi Direct® or peer-to-peer (P2P)
GO	Wi-Fi Direct group owner
OS	Operating system

Key Features

Table 0-2 lists the key features of the CC3120 and CC3220 devices.

Table 0-2. Key Features

Feature	Description
Wi-Fi standards	802.11b/g/n station 802.11b/g access point with support for up to four stations Wi-Fi Direct client / group owner
Wi-Fi channels	1–13
Personal and Enterprise Wi-Fi security	WEP, WPA/WPA2 PSK, WPA2 Enterprise (802.1x)
Wi-Fi provisioning	SmartConfig™ technology, Wi-Fi Protected Setup (WPS2), access point mode with internal HTTP web server
IP protocols	IPv4/IPv6
IP addressing	Static IP, LLA, DHCPv4, DHCPv6 with DAD
Cross layer	ARP, ICMPv4, IGMP, ICMPv6, MLD, NDP
Transport	UDP, TCP SSLv3.0/TLSv1.0/TLSv1.1/TLSv1.2 RAW
Network applications and utilities	Ping HTTP/HTTPS web server (including dynamic user call backs and RESTful API support) mDNS DNS-SD DHCP server
Host interface	UART/SPI

Table 0-2. Key Features (continued)

Feature	Description
Security	Secure key storage Trusted root-certificate catalog TI root-of-trust public key File system security Secure boot Secure content delivery Initial secure programming Debug security Software tamper detection Cloning protection
Power management	Enhanced power policy management uses 802.11 power save, and deep-sleep power modes
Other	Transceiver Programmable Rx filters with events trigger mechanism

Host Driver Overview

The SimpleLink Wi-Fi Internet-on-a chip devices provide comprehensive networking functionality that offloads networking activities from the host MCU. TI provides a user-friendly host-software driver to simplify the integration and development of networking applications using the SimpleLink Wi-Fi devices. This host driver can easily be ported to most platforms and operating systems (OSs). The host driver is written in strict ANSI-C (C89) and requires a minimal platform adaptation layer (porting layer).

The driver has a small memory footprint, and can run on 8-, 16-, or 32-bit MCUs with any clock speed (no performance or real-time dependency). Using SPI, both big- and little-endian MCUs are seamlessly supported. With UART, only little endian is supported.

The APIs of the SimpleLink host driver are arranged in several logical and simple modules (silos).

Table 0-3 provides a high-level description of these silos.

Table 0-3. Software Modules of the Host Driver

Silo	Description
Device	Provides interface to hardware and general functionality, such as start/stop or set, and get configurations in the device level
WLAN	Provides interface to WLAN 802.11 protocol-related functionality, such as mode (station, access point, or Wi-Fi Direct), provisioning, connection profiles, and connection policy
Socket	Provides interface to sockets and adheres to BSD sockets. BSD sockets are the most common interface today for internet connectivity.
NetApp	Provides interface to several networking services including the HTTP server service, DHCP server service, and MDNS client\server service
NetCfg	Provides interface to configure different networking parameters, such as setting the MAC address and IP address settings (DHCP/Static)
NetUtil	Provides interface to several network utilities, such as crypto utility, which provides a method for authenticating the device
FS	Provides interface for storing and reading files through a secure file system managed on the serial flash component

Host Interface

The SimpleLink device supports two physical host interfaces: SPI and UART. The same host driver can work with each of these interfaces by using an interface driver adaptation layer.

More information on the adaptation layer is in the host interface section, see Chapter 2 and Chapter 16.

OS versus Non-OS

The same driver can work on platforms running an OS, and platforms without an operating system (non-OS).

An OS adaptation layer is used for binding the host driver and the target OS. The driver already comes with a built-in adaptation layer for platforms running without an OS. Other platforms require a simple OS adaptation layer.

This adaptation layer must wrap two major objects:

- Sync object – Object intended to synchronize between different contexts and interrupt routines
- Lock object – Object intended to protect a shared resource

The driver pre-allocates all the required OS resources (dynamic or static according to the setting) on calling sl_Start. The number of allocated objects is calculated according to the maximum concurrent actions required by the user.

The SimpleLink host driver does not use its own processing context. To bind a context to the driver, the user can implement a spawn mechanism, or use the built-in spawn mechanism provided by the driver. If the built-in mechanism is used, the host application must create dedicated context to the driver and call sl_Task from this context. For platforms without an OS, the application must call to the sl_Task function repeatedly from its main loop.

Quick Reference

Figure 3 shows a quick reference to the entire set of APIs provided by the host driver.

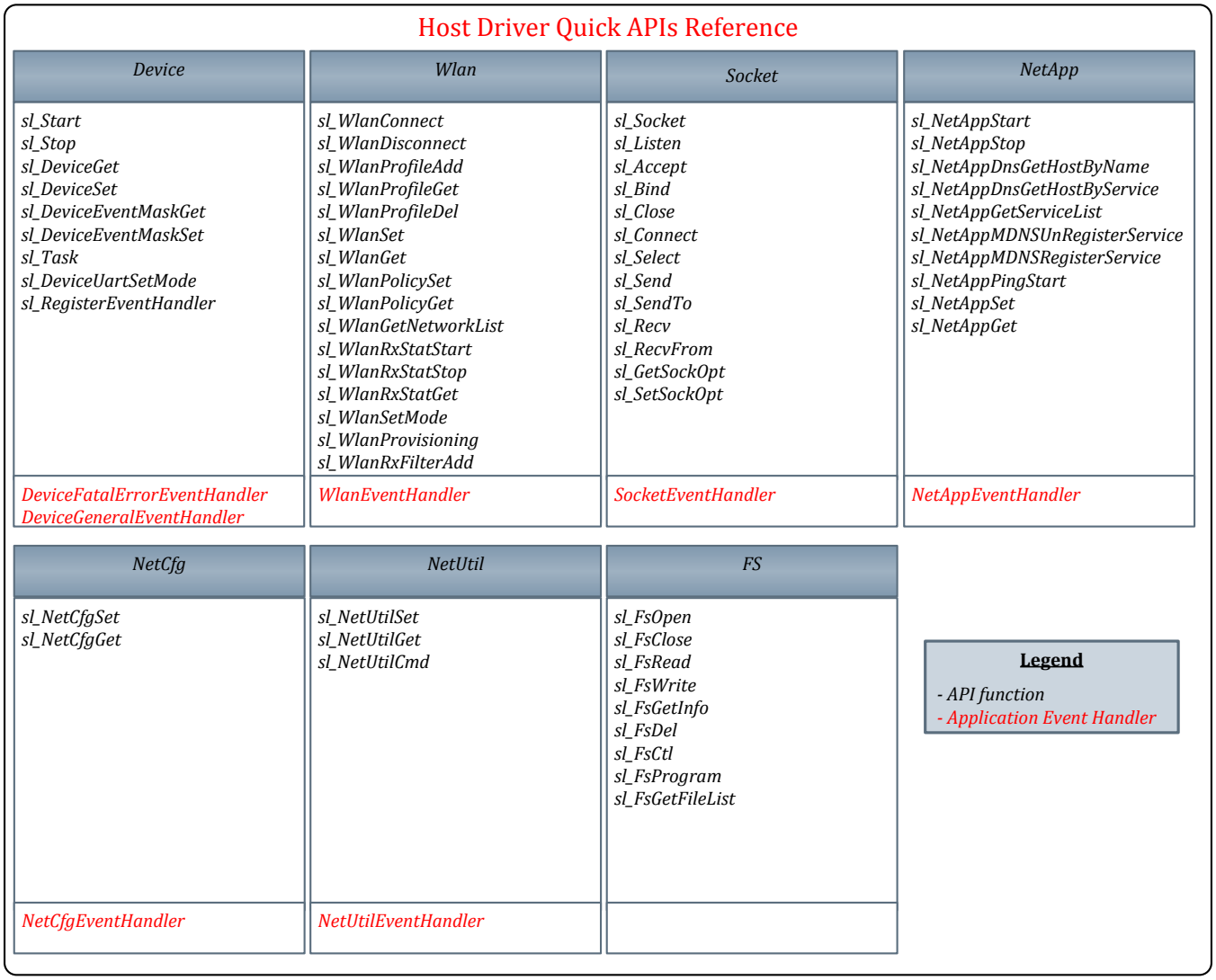

Figure 3. Quick Host APIs Reference

For more information on these APIs, refer Appendix A.

Porting to Different Platforms

To use the driver on different platforms, the host must implement a few adaptation modules.

Figure 4 shows these adaptation modules.

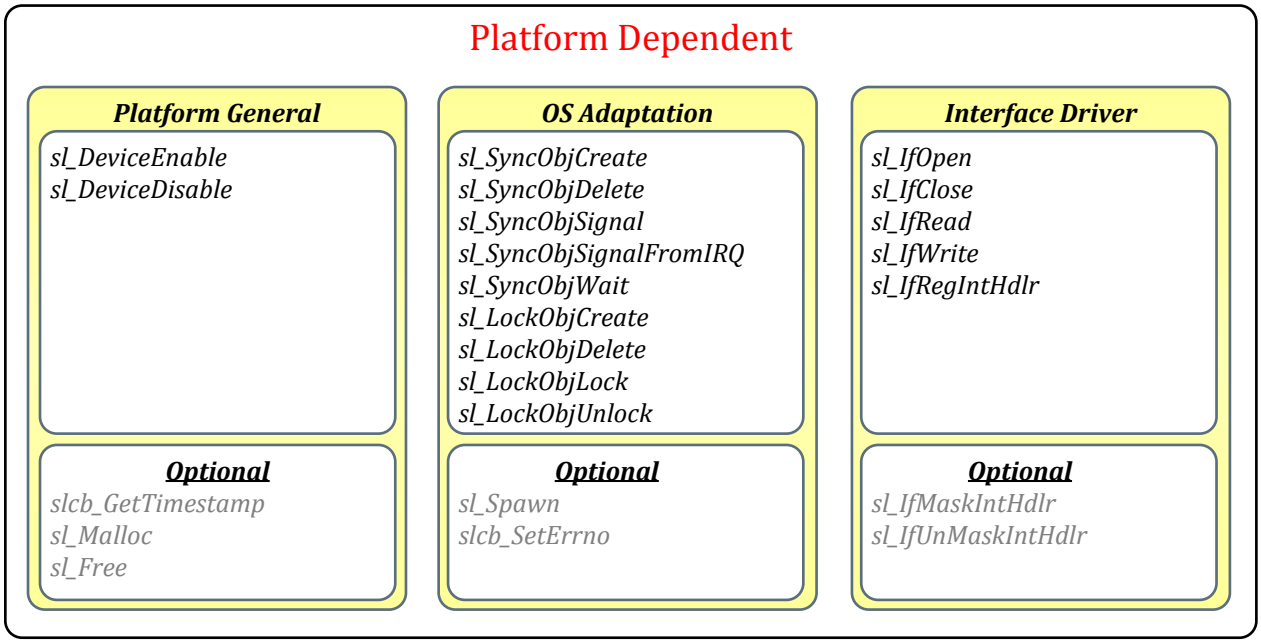

Figure 4. Host Driver Adaptation Modules (Platform-Dependent)

For more information about porting the driver to new platforms, see Chapter 16.

Networking Application

Topic **Page**

1.1 Introduction

This chapter explains the software blocks needed to build robust networking applications, and provides basic guidelines and considerations while designing these applications. Programmers have complete flexibility for using the various software blocks, and should design their own application according to their needs. A robust network application should consider the following aspects during the design:

- Wi-Fi connectivity – What is the connectivity model of the system? Is it always connected or connected on-demand? Wi-Fi connectivity can be used in a wide range of products with different use cases. Some products may be connected through the local Wi-Fi network to the Internet, some may just be connected to the local network or may function as access points, and some products may not be connected to a Wi-Fi network at all (uses Wi-Fi as a radio transceiver).
- Wi-Fi provisioning – What are the possible methods to connect a new device to a Wi-Fi network in the specific target application? Are there any graphical or other interfaces to the system?
- Traffic type – What kind of traffic is expected from the target system? Is it connection-oriented traffic or connectionless-oriented traffic?
- Security – What are the major assets of the system? What kind of protection is needed?
- User experience – What are the major experience factors for the target users? Is it response time? Availability? Or perhaps functionality?
- Power management – Is the system powered by batteries? What is the power budget?
- Data – What kind of data is kept on the system? What is the update frequency?

NOTE: TI highly recommends applying all needed configurations and settings by using the Image Creator tool instead of using host application commands. For more information, refer to the Image Creator user guide (SWRU469).

This chapter discusses the considerations and trade-offs.

1.1.1 Wi-Fi Connectivity

Wi-Fi connectivity can be used in a wide range of products with different use cases and requirements. Figure 1-1 shows some of the available connectivity options and their considerations.

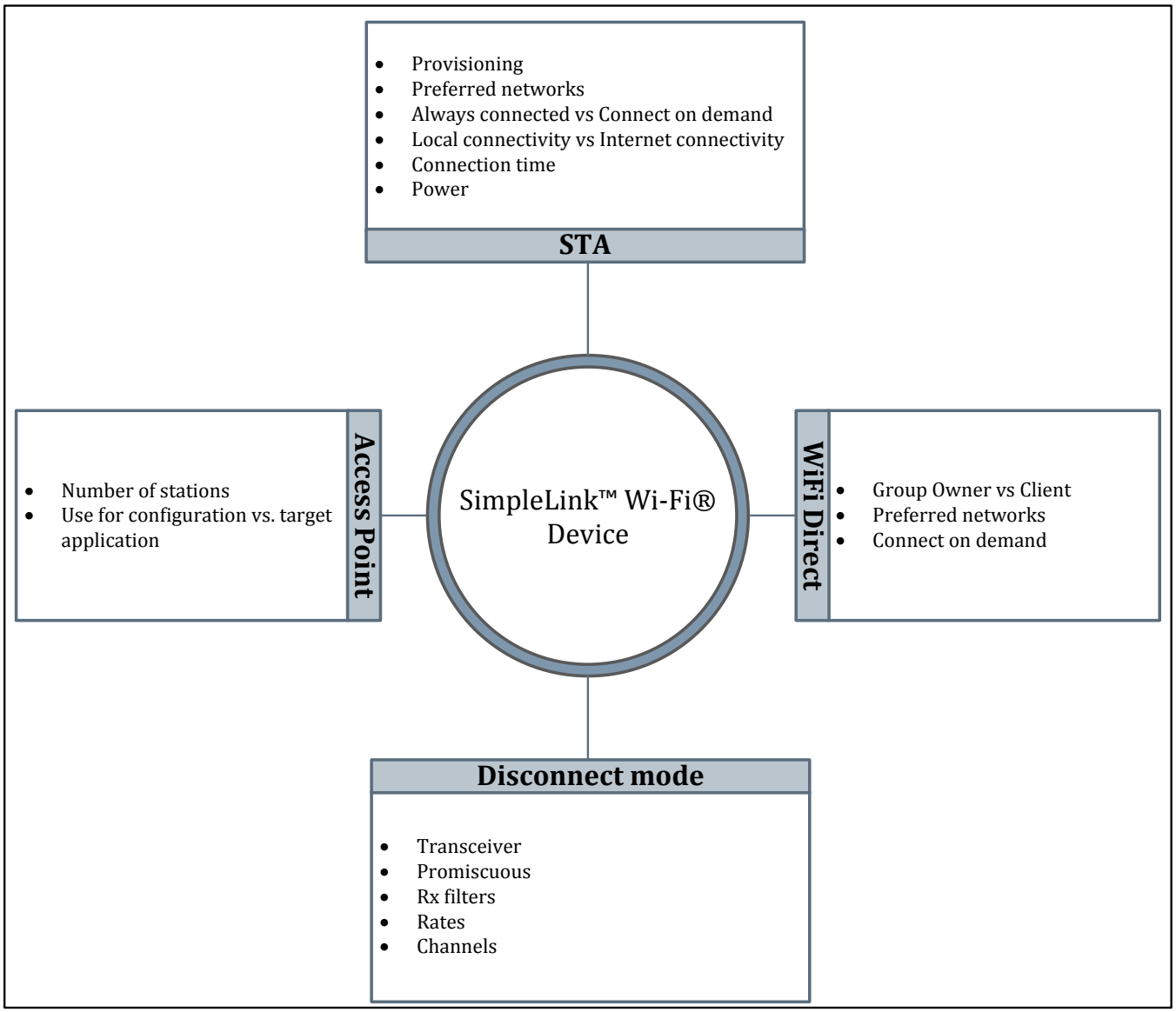

Figure 1-1. Wi-Fi Connectivity

The SimpleLink Wi-Fi device supports different Wi-Fi modes, and the application can move from one mode to another on demand. Moving from mode to mode requires the user to reset the SimpleLink Wi-Fi device. Trade-offs to be considered follow:

- Power consumption
- Response time
- Availability

1.1.2 Traffic Types

Communication protocols are typically divided into two types: connection oriented and connectionless. Connection-oriented protocols require establishing a connection between two entities before any data exchange. The connection is maintained during the connection lifetime, and ensures data is delivered correctly and in order. Connectionless protocols allow data exchanges between any entities, without the need for establishing a connection; however, data integrity and order are not ensured. From a power consumption perspective, connection-oriented protocols may consume more power due to the connection-establishing overhead and connection maintenance.

The SimpleLink Wi-Fi devices support the following communication protocol types:

- Connection oriented –TCP and SSL/TLS
- Connectionless – UDP or RAW

When application designers choose the protocol, the power consumption, reliability, and latency should also be considered. Connectionless protocols are less reliable by nature. However they are more efficient from a power consumption and latency perspective.

Generally, the connection type derives from the services used. For example, many cloud services are based on HTTP or MQTT, which are running over a TCP connection.

1.1.3 Security

The importance of security for IoT devices is crucial due to the sensitive and private nature of the data. This data might include passwords, keys credentials, configuration, and personal information.

The SimpleLink Wi-Fi devices handle the following security aspects:

- Wi-Fi: Secured local network is the first protection layer for the IoT device. SimpleLink Wi-Fi devices support several Wi-Fi security methods, both personal and enterprise. When a SimpleLink device connects to an AP through the profile method, the network encrypted password is stored in the SFLASH, and there is no access to the password, which raises the protection of the local network and the device. More details are in the WLAN chapters.
- Data: Data layer security is a basic requirement for secured local networks, especially when the device is connected to the cloud. SimpleLink devices support the SSL (secured socket layer) standard for data encryption and server verification. More details are in the secure-socket chapter.
- Files: Passwords, configurations, keys, and credentials are private information on the device which must be secured. SimpleLink Wi-Fi devices support secure file systems on an external serial flash, providing a simple API to organize and access the data. More details are in the secure-file system chapters.

1.1.4 User Experience

The IoT refers to a wide range of products with different characterizations. Some of these products must always be available from the cloud with minimal delay (such as smart plugs or security cameras). Other products may connect to a cloud server only on a state change (such as doorbell or fire alarm), and require fast connection with minimal delay. An additional type of product which is not sensitive to delay (such as air conditioning) notifies the time for treatment. The SimpleLink Wi-Fi devices are designed to allow IoT devices to support those characterizations by optimizing power consumption, Wi-Fi connection time, IP acquired time, and more.

1.1.5 Power Consumption

Different applications have different power consumption requirements. Applications which are battery powered are very sensitive to power consumption, because almost every design decision has an impact on total power consumption. The following design decisions have major impacts on power consumption.

- Wi-Fi mode (STA or AP)
- On-demand connection or constant connection
- Traffic type (connection oriented or connectionless)
- Secured socket or not
- Regular socket operation or trigger mode (lets the host enter a deep sleep and is awakened by the SimpleLink Wi-Fi device when data arrives)

More details are in *Chapter 13*.

1.1.6 Provisioning

Provisioning is the process of providing an IoT device with the information needed to connect to a wireless network for the first time (network name, password, and so on). The provisioning process should be fast, easy to use, and not require technical support. More details are in *Chapter 14*.

1.2 Basic Examples

1.2.1 Wi-Fi Doorbell

1.2.1.1 Description

Wireless doorbells may have not only a push-button, but also a microphone or a camera, therefore requiring advanced connectivity options which can be supported by Wi-Fi technology. By using a button push, a smartphone application lets users monitor visitors through video and voice, from any location using internet connectivity. In addition, from time to time the doorbell connects to a server for software updates. The doorbell is usually battery powered.

1.2.1.2 Design Considerations

Table 1-1 lists the design considerations for doorbell applications.

Table 1-1. Design Considerations for Doorbell Applications

Topic	Consideration or Constraint	Recommendations
Wi-Fi connectivity	Wi-Fi connection with home access point, to allow internet access	STA role Configure profile with network name and password during the provisioning process.
Traffic types	Reliable	SSL/TLS
Security	Must secure data: Secure Wi-Fi password Secure credentials Secure user password of the up layer (connect to the server)	Wi-Fi – profile and password are configured during the provisioning process. The password is encrypted and cannot be accessed by the application. Data – encrypt and decrypt data using SSL Credentials and server password – use the SimpleLink secured-file system
Power management	Sensitive, battery powered, or power harvesting	Operation mode – hibernate mode, wakes up the SimpleLink device by pressing a button, returns to hibernate mode quickly Wi-Fi – profile with automatic and fast policies IP – Decrease the number of DNS requests, resolve once and keep the address, if TCP connection fails resolve again
User experience	Connectivity must be fast, with minimal delay	
Provisioning	Easy setup	Easy setup – Use the Smartphone app to perform the provisioning and create the connection profile

1.2.2 Power Socket

1.2.2.1 Description

A power socket connects to the cloud, which lets users control products like air conditioners and boilers. The power socket must be available for smartphone apps at any time with no delay, and are occasionally connected to the server for software updates.

1.2.2.2 Design Constraints

Table 1-2 lists the design constraints for power socket applications.

Table 1-2. Design Considerations for Power Socket Application

Topic	Consideration or Constraint	Recommendations
Wi-Fi connectivity	Wi-Fi connection with home access point, to allow internet access at any time	STA role Configure profile with network name and password by the provisioning process.
Traffic types	Reliable	TCP, also data must be secured
Security	Data must be secured Secure Wi-Fi password Secure credentials Secure user password of up layer (connected to server)	Wi-Fi – profile and password are configured by the provisioning process, the password is encrypted and cannot be accessed by the application. Data – encrypt and decrypt data by SSL Credentials and server password – use SimpleLink secured-file system
Power management	Connected to the power supply	None
User experience	Must be available on smartphone app at any time with no delay	SimpleLink is always on, client TCP secured socket is always connected to the server
Provisioning	Easy setup, without the need of technical support	The provisioning process allows for easy and fast wireless network configuration (network name and password)

1.2.3 Wi-Fi Tag

1.2.3.1 Description

A tag is a tiny device which attaches to expensive assets (such as hospital medical equipment or expensive lab equipment). The tag device occasionally transmits, without being connected to the local network. The transmission allows the central equipment to find the device using a smart-signal algorithm. A tag device can also connect to an AP occasionally to get software updates or send statistical information.

1.2.3.2 Design Consideration

Table 1-3. Design Considerations for Tag Applications

Topic	Consideration or Constrain	Recommendations
Wi-Fi connectivity	Mostly transmitted without being connected, occasionally connect AP for SW updates	Regular: transceiver mode – not connected SW update: STA role
Traffic types	SW update must be reliable	Regular: Propriety data over Wi-Fi frame SW updates: TCP with SSL
Security	Data must be secured: Secure Wi-Fi password Secure credentials Secure user password of up layer (connect to server)	Regular: None For SW update: Wi-Fi – profile and password are configured during the provisioning process. The password is encrypted and cannot be accessed by the application Data – encrypt and decrypt data using SSL Credentials and server password – use the SimpleLink secured-file system
Power management	Sensitive, battery powered	Hibernate or shutdown mode, wakes up the SimpleLink when it is time for tag transmitting or SW updates arrived, returns to hibernate or shutdown quickly. SW update: Wi-Fi: profile with auto and fast policies IP: Decrease the number of DNS requests, resolve once and keep the address, only as a result of TCP connection failure, resolve again
User experience	None	None
Provisioning	None	None

Device

Topic Page

2.1 Introduction

SimpleLink Wi-Fi devices support multiple internal-device configurations and settings such as device initialization, communication interface settings, time and date settings, and more, using simple host-driver commands.

In the following chapters, the word *device* describes the network subsystem.

2.2 Key Features

Table 2-1 lists the key features of the device.

Table 2-1. Key Features

Key Features	Description
Calibration modes	Different types of RF calibration modes to save power
SPI and UART	SPI and UART communication interfaces support
Little- and big-endian auto-detect	Automatic detection of the MCU endian state
Time and date	Support time and date setting, and getting information
Stop time-out	Stop the device with time-out, to allow TX completion
Get device version	Get PHY\FW\NWP\Host versions
Mask asynchronous event	Mask asynchronous events from the device
System-persistent configuration	Set the entire system-persistent configuration

2.3 Start and Stop

2.3.1 Start

From a host perspective, steps to starting the SimpleLink Wi-Fi device include setting the enable pin, opening the communication interface, and waiting for the complete indication of the device initialization. Depending on the hardware design, the enable pin of the CC3120 device can be connected to the nReset or nHibernate pins of the device. The major difference between these modes is that in hibernate mode, the device maintains the value of the real-time clock (RTC), and exiting from this state is faster (more information follows). In both modes when the device completes the initialization (INIT) process, it sends an internal asynchronous event (INIT COMPLETE) to the host.

During the initialization process the host interface (UART or SPI) is determined, and RF calibrations may be performed. A few types of calibration modes are available, and the target application must choose the most compatible mode for its requirements. The mode of the calibration can be changed only by using the Image Creator tool, during the creation of the image. More information about calibrations is in *Chapter 3*, and more information about the Image Creator tool is in the *CC3120, CC3220 SimpleLink Wi-Fi and Internet of Things Image Creator User's Guide*.

The sl_Start API of the host driver can provide a callback function. If the callback function is provided, then the function returns immediately, and the callback is called when the initialization process completes. In this mode, any other APIs should not be called until the initialization completes. If the callback is not provided, sl_Start is blocked until the device initialization completes. This API must be called before any other SimpleLink API is used, or after sl_Stop is called to reinitialize the device and the driver.

This function return value specifies the mode the device is currently running: ROLE_STA, ROLE_AP, or ROLE_P2P. Any other value indicates an error during the initialization process.

2.3.2 Stop

This function clears the enable pin of the device, and closes the communication interface. This function can receive time-out (in milliseconds) as a parameter. This parameter defines the amount of time the device allows for finishing any packet ongoing transmission, reception, or disconnection gracefully before shutting down. This time-out value determines the maximum time the device waits. The function returns when all the activities are performed even before the time-out expires.

Example:

```
_i16 Status, Role;

Role = sl_Start(NULL, NULL, NULL);
if (ROLE_STA == Role)
{
    /* Main application */
}
Status = sl_Stop(100); /* 100 ms Timeout */
if( Status )
{
    /* error */
}
```

2.3.3 Hibernate and Shutdown

Hibernate is the lowest-device power mode which keeps the RTC running. In this mode, the device is powered off, except for the hibernate logic. In this state the volatile memory of the SimpleLink Wi-Fi device is not maintained, but the RTC is maintained, which provides faster boot time and maintains the system date and time. The SimpleLink Wi-Fi device goes into the hibernate state when the correct hardware (HW) lines are set (RESET / HIB) on a call to sl_Stop.

Full shutdown is the lowest power state of the device. In this state both the volatile memory and the RTC are not maintained. The initialization process from full shutdown takes longer compared to initializing from hibernate. The SimpleLink device goes into full shutdown state when the correct HW lines are set on a call to sl_Stop.

2.3.4 Lock State

The device enters a lock state due to one of the following conditions:

- The restore to factory defaults is currently in process, and the device unlocks when the process is finished.
- The device INIT failed and an inaccurate error is sent with the INIT-complete error asynchronous event. The device INIT-complete may then fail due to calibration failure or integrity failure of the file-system data structure
- The security alerts threshold was exceeded. The SimpleLink Wi-Fi device provides a software tampering detection mechanism with a security-alert counter. This procedure detects integrity violation of the following: file-system data, secure-authentication files, or system files. When the device reaches the security alerts threshold (three by default or predefined with Image Creator), it locks.
- A critical security alert occurs.

In the lock state only a few commands are allowed. The list of the enabled commands follows:

- Program a new image
- Restore to factory defaults
- Get current version
- Get storage information (retrieves the number of security alerts and the storage properties)

Any other API issued in locked state returns one of the following error codes:

SL_RET_CODE_DEV_LOCKED (-2011L) //Device was found locked during its init, commands are blocked by the driver

SL_ERROR_NOT_ALLOWED_NWP_LOCKED (-14343L) //Device is currently lock

Recovery from the lock state depends on the reason for the lock. If the lock is due to processing the restore to factory function, then the device automatically unlocks when finished. In all other cases, to recover from the lock state the device can be programed or restored to factory image.

2.3.5 Initialization Sequence

During the INIT sequence, the host driver runs the following key steps:

- Enables communication interface (SPI or UART) with the device
- Registers the asynchronous events handler
- Enables the SimpleLink Wi-Fi device
- Waits for a host IRQ
- Reads the INIT-complete event

The SimpleLink Wi-Fi device determines one active host interface during this phase (SPI or UART) and disables the other.

2.4 Host Interface

The SimpleLink Wi-Fi device provides comprehensive networking functionality. To simplify the integration and development of networking applications a simple, a user-friendly host driver is provided. The SimpleLink Wi-Fi host driver is responsible for the following:

- Provide a simple API to the user application
- Handle communication with the device
- Build and parse commands
- Handle asynchronous events
- Handle flow control for the data path
- Provide serialization of concurrent commands
- Work with the existing UART or SPI physical communication interface drivers
- Provide the ability to work with or without an OS
- Enable porting to any platform

The SimpleLink Wi-Fi host driver is written in strict ANSI C89 for full compatibility with most embedded platforms and development environments.

The following information is relevant for the SimpleLink Wi-Fi CC3120 wireless network processor, which must implement a communication interface with a selected MCU.

The device supports the SPI and UART standard communication interfaces. Binding the communication interface to the host driver is done by defining the interface functions through the following defines in user.h:

- sl_IfOpen
- sl_IfClose
- sl_IfRead
- sl_IfWrite
- sl_IfRegIntHdlr

More information regarding these functions is in *Chapter 16*.

2.4.1 SPI Interface

The SimpleLink Wi-Fi device runs as a SPI slave and supports a 4-wire SPI interface.

Table 2-2 lists the required SPI settings.

Table 2-2. SPI Configuration

Attribute	Value
Clock rate	Up to 20 MHz
Word length	8-bit, 16-bit, 32-bit
Mode	0 (CPOL=0, CPHA=0)
Other	CS required, and cannot be tight to active state Additional IRQ line required for indicating asynchronous events from the device

In SPI, all communications on the bus are initiated by the SPI master (the host in this case). There is always a single master on the bus. To allow the SimpleLink Wi-Fi device to trigger asynchronous events to the host, an additional I/O must be connected (H.IRQ) between them. This line triggers the host to read a message from the device.

Figure 2-1 shows a typical host setup of the CC3120 wireless network processor using SPI interface.

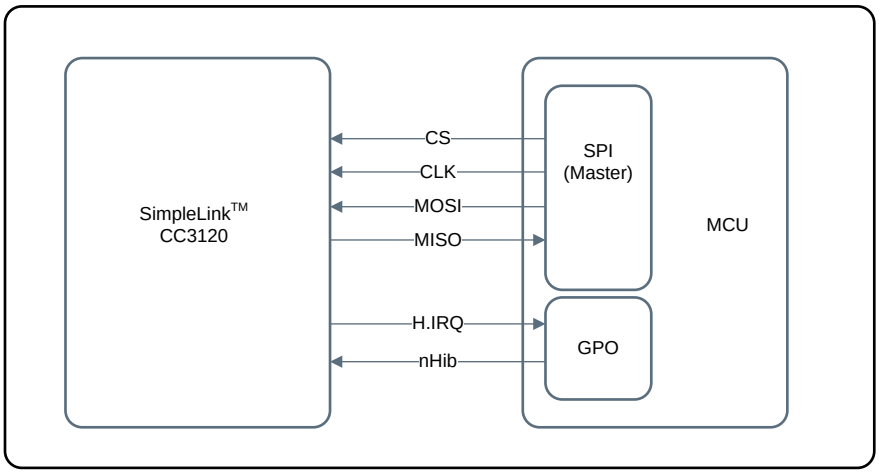

Figure 2-1. Typical CC3120 Setup (SPI)

2.4.2 UART Interface

The SimpleLink Wi-Fi device supports a standard UART interface with a hardware flow control (RTS/CTS). The default baud rate is 115,200 bps and can be increased to 3 Mbps.

Table 2-3 lists the required UART settings.

Table 2-3. UART Settings

Attribute	Value
Baud rate	115,200 bps (can be increased to 3 Mbps)
Flow control	CTS/RTS
Parity	None
Data bits	8
Stop bit	1

Figure 2-2 shows a typical host setup of the SimpleLink Wi-Fi device using UART interface.

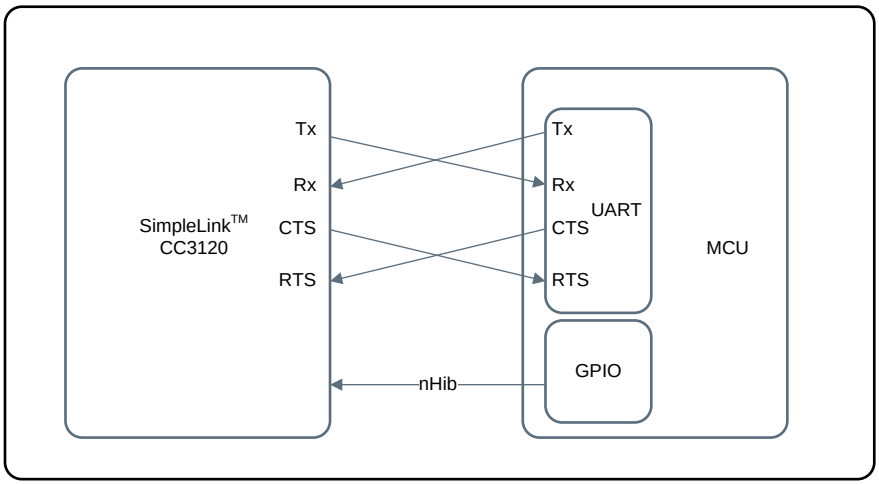

Figure 2-2. Typical CC3120 Setup (UART)

Working with the UART interface requires the use of hardware flow control to avoid data loss. UART hardware flow control works in a way that an entity that is ready to accept data, keeps its RTS line asserted. Before the transmission, the UART peripheral of the second side will read its CTS line, which is connected to the RTS of the first side, to verify if it is allowed to send data over the line. The SimpleLink Wi-Fi device may request to stop transmissions in some scenarios; and therefore, its RTS line must be respected. If the host is fast enough and does not need to stop transmissions from the SimpleLink device at any time, the CTS line of the SimpleLink Wi-Fi device might be tied to a pullup instead.

For UART mode only, the following define should be added in user.h: #define SL_IF_TYPE_UART

2.4.2.1 Change UART Baud Rate

The SimpleLink device does not support automatic baud rate detection; therefore this parameter should be set after every reset. When calling to sl_start, the default baud rate (115,200) must be set as part of the API parameters. If a different baud rate is needed, the host can set it after the initialization process completes by using the API sl_DeviceUartSetMode. This setting is not persistent and must be repeated every time sl_Start is called.

Supported baud rates:

- SL_DEVICE_BAUD_9600
- SL_DEVICE_BAUD_14400
- SL_DEVICE_BAUD_19200
- SL_DEVICE_BAUD_38400
- SL_DEVICE_BAUD_57600
- SL_DEVICE_BAUD_115200
- SL_DEVICE_BAUD_230400
- SL_DEVICE_BAUD_460800
- SL_DEVICE_BAUD_921600

Example:

```
_i16 Status;
_i16 Role;
SlDeviceUartIfParams_t params;
#define COMM_PORT_NUM 24 /* uart com port number */
params.BaudRate = SL_DEVICE_BAUD_115200; /*set default baud rate */
params.FlowControlEnable = 1;
params.CommPort = COMM_PORT_NUM;
Role = sl_Start(NULL, (signed char*)&params, NULL)
params.BaudRate = SL_DEVICE_BAUD_921600; /* set default baud rate 921600 */
Status = sl_DeviceUartSetMode((signed char*)&params);
if( Status )
{
    /* error */
}
```

2.5 Version

The SimpleLink Wi-Fi device offers users the ability to read the internal device firmware version number.

Example:

```
_i16 Status;
SlDeviceVersion_t ver;

pConfigLen = sizeof(ver);
pConfigOpt = SL_DEVICE_GENERAL_VERSION;
Status = sl_DeviceGet(SL_DEVICE_GENERAL,&pConfigOpt,&pConfigLen,(_u8 *)(&ver));
if( Status )
{
    /* error */
}
```

2.6 Event Mask

The SimpleLink Wi-Fi device lets users mask some of the asynchronous events. Masked events do not arrive to the host driver. This setting should apply for each API silo separately and include only the events needed to be masked. By default, none of the events are masked. This configuration is persistent according to the system-persistent configuration.

Example:

```
_i16 Status;

/* Mask WLAN connect and disconnect events */
Status = sl_DeviceEventMaskSet(SL_DEVICE_EVENT_CLASS_WLAN,
(SL_DEVICE_EVENT_BIT(SL_WLAN_EVENT_CONNECT) | SL_DEVICE_EVENT_BIT(SL_WLAN_EVENT_DISCONNECT) ) );
if( Status )
{
    /* error */
}
```

2.7 Time and Date

The SimpleLink Wi-Fi device gives users the option to set, and get time and date configuration from the RTC on the device. The RTC is a continuous counter which is active even during hibernation and resets only after shutdown.

Example:

```
_i16 Status;
SlDateTime_t dateTime= {0};

dateTime.tm_day =   (_u32)23;          /* Day of month (DD format) range 1-31 */
dateTime.tm_mon =   (_u32)6;           /* Month (MM format) in the range of 1-12 */
dateTime.tm_year =  (_u32)2014;        /* Year (YYYY format) */
dateTime.tm_hour =  (_u32)17;          /* Hours in the range of 0-23 */
dateTime.tm_min =   (_u32)55;          /* Minutes in the range of 0-59 */
dateTime.tm_sec =   (_u32)22;          /* Seconds in the range of  0-59 */
Status  = sl_DeviceSet(SL_DEVICE_GENERAL, SL_DEVICE_GENERAL_DATE_TIME, sizeof(SlDateTime_t),
(_u8*)(&dateTime));
if( Status )
{
    /* error */
}
```

2.8 MAC Address

Each SimpleLink Wi-Fi device is manufactured with a unique MAC address. The user can overwrite this default MAC address. The configuration is persistent with no dependency on the system-persistent configuration. Setting a MAC address takes effect only after reset, and can be set by the Image Creator.

NOTE: When configuring a new MAC address, the original MAC address is still used for Image Creator development mode. For more information, refer to the *CC3120/CC3220 SimpleLink Wi-Fi and Internet of Things Image Creator User's Guide* (SWRU469).

Example:

```
_i16 Status;
_u8 MAC_Address[6];
_i16 Role;

MAC_Address[0] = 0x8;
MAC_Address[1] = 0x0;
MAC_Address[2] = 0x28;
MAC_Address[3] = 0x22;
MAC_Address[4] = 0x69;
MAC_Address[5] = 0x31;

Status  = sl_NetCfgSet(SL_NETCFG_MAC_ADDRESS_SET,1,SL_MAC_ADDR_LEN,(_u8 *)MAC_Address);
if( Status )
{
    /* error */
}
Status = sl_Stop(0);
if( Status )
{
    /* error */
}
Role = sl_Start(NULL,NULL,NULL);
```

2.9 Device Name

The device name is used as the common URN name at the WPS, Wi-Fi Direct, MDNS, and DHCPv4 client. The maximum length of the device name is 32 characters, and the following characters are allowed:

- a through z
- A through Z
- 0 through 9
- –

If no device URN name is set, the default name is mysimplelink. If setting the device name with length 0, the device returns to the default name mysimplelink. This configuration is persistent according to the system-persistent configuration.

Example:

```
/* set new device name */
_i16 Status;
_u8 *device_name = "MY-SIMPLELINK-DEV";

Status = sl_NetAppSet (SL_NETAPP_DEVICE_ID,SL_NETAPP_DEVICE_URN, strlen(device_name), (_u8 *)
device_name);
if( Status )
{
    /* error */
}
```

2.10 Domain Name

The domain name is used to access the SimpleLink Wi-Fi device by name, for example accessing the HTTP server in AP mode. If no domain name is set, the default domain name is www.mysimplelink.net or mysimplelink.net. This configuration is persistent according to system-persistent configuration.

Example:

```
/* set new domain name */
_i16 Status;
_u8 *domain_name = "www.myDomain.net";

Status = sl_NetAppSet(SL_NETAPP_DEVICE_ID,SL_NETAPP_DEVICE_DOMAIN,strlen(domain_name),(_u8 *)
domain_name);
if( Status )
{
    /* error */
}
```

2.11 Device Status

The SimpleLink Wi-Fi device provides an option to read the device status according to the last event recorded in the SimpleLink device per API silo. The status is clear on read.

This option has two main return values:

- Device status
- Asynchronous events

Example:

```
_i16 Status;
_u32 statusWlan;
_u8 pConfigOpt;
_u16 pConfigLen;

pConfigOpt = SL_DEVICE_EVENT_CLASS_WLAN;
pConfigLen = sizeof(_u32);
Status  = sl_DeviceGet(SL_DEVICE_STATUS,&pConfigOpt,&pConfigLen,(_u8 *)(&statusWlan));
if (SL_DEVICE_STATUS_WLAN_STA_CONNECTED & statusWlan )
{
        /* The device is connected */
}
if (SL_DEVICE_EVENT_DROPPED_WLAN_RX_FILTERS & statusWlan )
{
        /* RX filer event dropped */
}
```

The full list of possible values for possible device status or asynchronous events can be found in the host driver.

2.12 Persistent Configuration

The SimpleLink Wi-Fi device lets users set the system-persistent configuration. By default, the mode of the system-wide configuration persistence is set to true, and all APIs that follow the system configured persistence maintain their configured settings after reset. If false, all calls to the APIs that follow the system-configured persistence are volatile, and the configurations revert to default after reset.

Example:

```
_i16 Status;
_u8 persistent = 1;

Status  = sl_DeviceSet(SL_DEVICE_GENERAL, SL_DEVICE_GENERAL_PERSISTENT, sizeof(_u8),
(_u8*)(&persistent));
if( Status )
{
    /* error */
}
```

For a full list of parameters and their persistent configuration, refer to *Appendix B*.

NOTE: If system-persistent configuration is enabled, any change in the system settings may result in a serial-flash write operation, and its write endurance must be considered.

2.13 Errors

Errors are indicated by the return value of the API or by an asynchronous event. Asynchronous events can be sent to the host at any time with a specific error indication, and may also include specific data for each event. To listen to these events and conclude the needed information, a handler should be implemented in the user application, and registered under the user.h header file. Each error code is unique. The following errors are common and require user action (a full possible error list is under the file error.h in the host driver):

Table 2-4 lists common errors indicated by asynchronous events.

Table 2-4. Common Asynchronous Error Events

Error	Handler	Comments
SL_DEVICE_EVENT_ERROR	slcb_DeviceGeneralEvtHdlr	General error. Includes the parameters status (specified in the following table and sender, see SlDeviceSource_e).
SL_DEVICE_EVENT_FATAL_DEVICE_ABORT	slcb_DeviceFatalErrorEvtHdlr	Notifies fatal error. The SimpleLink device is asserted. User must perform device restart (call sl_Stop followed by sl_Start).
SL_DEVICE_EVENT_FATAL_DRIVER_ABORT	slcb_DeviceFatalErrorEvtHdlr	Notifies fatal error. The host driver is asserted. User must perform device reset.
SL_DEVICE_EVENT_FATAL_NO_CMD_ACK	slcb_DeviceFatalErrorEvtHdlr	Notifies fatal error. The host driver did not receive the ACK command from the device. User must perform device restart (call sl_Stop followed by sl_Start).
SL_DEVICE_EVENT_FATAL_SYNC_LOSS	slcb_DeviceFatalErrorEvtHdlr	Notifies fatal error. The host driver and SimpleLink device are out of sync. User must perform device restart (call sl_Stop followed by sl_Start).
SL_DEVICE_EVENT_FATAL_CMD_TIMEOUT	slcb_DeviceFatalErrorEvtHdlr	Notifies fatal error. The command time-out has expired. User must perform device restart (call sl_Stop followed by sl_Start).

Table 2-5 lists common errors statuses.

Table 2-5. Common Error Codes

Error	Value	Comments
SL_ERROR_ROLE_STA_ERR	–4107	Initialization failure in the specified mode (sl_Start)
SL_ERROR_ROLE_AP_ERR	–4108	
SL_ERROR_ROLE_P2P_ERR	–4108	
SL_ERROR_CALIB_FAIL	–4110	Calibrations failed
SL_ERROR_FS_CORRUPTED_ERR	–4111	File system is corrupted, restore to factory image or program new image should be invoked (see sl_FsCtl, sl_FsProgram)
SL_ERROR_FS_ALERT_ERR	–4112	Initialization failure due to exceeded number of security alerts (sl_Start); device is locked, restore to factory image or program new image should be invoked (see sl_FsCtl, sl_FsProgram)
SL_ERROR_RET_TO_IMAGE_COMLETE	–4113	Restore to factory image completed, perform reset
SL_ERROR_INCOMPLETE_PROGRAMMING	–4117	Error during programming. Program new image should be invoked (see sl_FsProgram).
SL_ERROR_DEVICE_LOCKED_SECURITY_ALLERT	–28674	Number of security alerts exceeded or system file integrity error; device is locked.

WLAN

Topic Page

3.1 Introduction

The SimpleLink Wi-Fi device supports three WLAN modes: STA, AP, and Wi-Fi-Direct. The device can run one mode at a time. Each mode has specific settings and capabilities which are detailed in the following sections. Using the WLAN modes, connection, scanning networks, and data transmissions are possible. This chapter describes the full set of capabilities of the WLAN system.

3.2 Key Features

Table 3-1 lists the key features and their descriptions.

Table 3-1. Key Features

Key Features	Description
802.11 b/g/n STA	Supports 802.11 b/g/n standards in STA mode
STA power save	Supports STA power save capability
802.11 b/g AP	Supports 802.11 b/g standards in AP mode with up to 4 simultaneously connected stations, built-in DHCP server and DNS server
Manual connection	Supports manual connection to a network by SSID and SSID+BSSID
Preferred networks	Supports up to 7 persistent preferred networks (profiles)
Secured connection	WEP, WPA, WPA2, and WPS security connection types are supported.
Enterprise connection	Multiple EAP methods are supported for enterprise connection.
Connection policy	Connection policy that allows automatic connection to a preferred network in different cases
Wi-Fi Direct	Wi-Fi Direct connection with remote device acting as GO or CLIENT
Scanning	Support scan parameter configuration. Keep up to 30 networks, and the ability to read the results.
WLAN modes	Station, AP (default), and Wi-Fi Direct

3.3 Station (STA)

3.3.1 General Description

Station (STA) is the primary mode of the SimpleLink Wi-Fi device operation. Operating the device in this mode allows the a connection to an AP, obtaining an IP address, transmitting and receiving data over the network, and scanning other network devices. The following sections specify the major settings and modes of operation that are unique to STA mode.

3.3.2 Configurations and Settings

STA configuration is done by using a host driver API while the SimpleLink Wi-Fi device is in STA mode. Some of the configurations are also available through the internal ROM HTTP server (see *Chapter 8* for details and the configuration table).

There are several configurations for each specific use. Some of the configurations are persistent according to the system-persistent configuration, some are persistent and nonpersistent as specified in each configuration specification (more information is in *Appendix B*).

Table 3-2 lists the default parameters in station mode. Not all configurations are mandatory, because the default values of the device are listed in Table 3-2.

Table 3-2. Default Parameters in Station Mode

Configuration	Default Value
Interface	IPv4
Address	DHCP
STA TX power	0 (no back-off, maximum TX power)
Country code	EU (channels 1–13)
Connection policy	Auto and Auto Provisioning
Calibration mode	Normal
Server enterprise authentication	Enabled
Applications	HTTP server and MDNS

3.3.2.1 Set Mode

STA mode is not the initialization mode by default, therefore it must be set by the application or during the image creation. The following API should be called to set the device in STA mode. STA configuration requires a reset and is persistent with no dependency on the system-persistent configuration.

Example:

```
_i16 Role;
_i16 Status;

/* Set the device in STA mode */
Status = sl_WlanSetMode(ROLE_STA);
if( Status )
{
    /* Error setting mode */
}
/* Reset the device */
Status = sl_Stop(0);
Role = sl_Start(NULL,NULL,NULL);
if (ROLE_STA != Role)
{
    /* Role Error */
}
```

3.3.2.2 Set General STA Parameters

STA mode is activated with default configurations. Reconfiguring these settings is possible, but not mandatory. The following configurations are available. These configurations require reset, and are always persistent with no dependency on the system-persistent configuration.

- **STA Transmit (TX) Power**

  Sets the TX power which controls the transmission power level, and can increase or decrease the value, relative to the maximum TX power. The value represents steps from 0 to 15 which reflect as dBm offsets from maximum power (0 means maximum power) according to Figure 3-1.

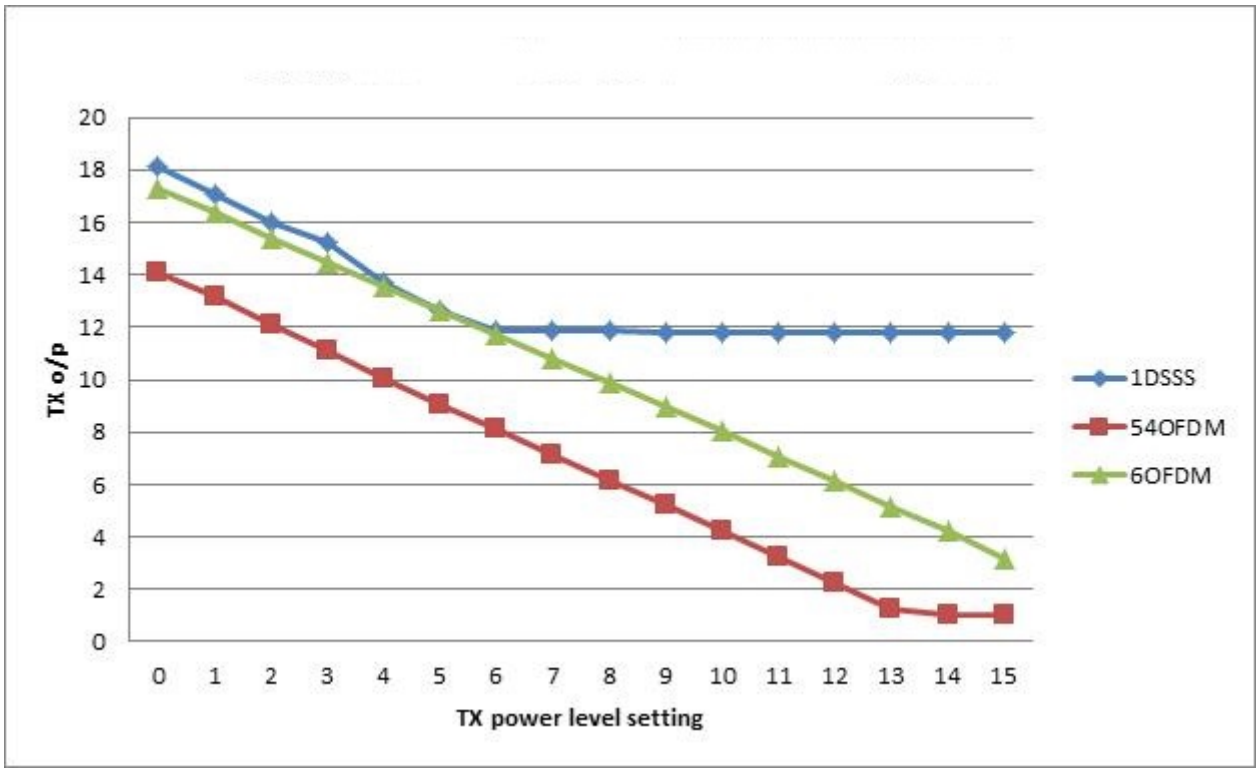

Figure 3-1. Tx Output Power vs Tx Power Settings

Example:

```
_i16 Status;
_u8  StaPower = 3;

Status = sl_WlanSet(SL_WLAN_CFG_GENERAL_PARAM_ID, SL_WLAN_GENERAL_PARAM_OPT_STA_TX_POWER,1,(_u8
*)& StaPower);
if( Status )
{
    /* error */
}
```

- **Set Country Code/Regulatory Domain**

  Sets the country code for STA mode. This setting enables scanning, and connection only to the AP which operates on the chosen channel set. Possible values are US, EU, or JP. Other values are considered an error. Each country code sets different channel ranges:

  – US: Channels 1–11
  – JP, EU: Channels 1–13

Example:

```
_i16 Status;
_u8  Str[] = "US";

Status = sl_WlanSet(SL_WLAN_CFG_GENERAL_PARAM_ID, SL_WLAN_GENERAL_PARAM_OPT_COUNTRY_CODE, 2, Str);
if( Status )
{
    /* error */
}
```

3.3.3 Connection

Connection to a WLAN network is one of the basic capabilities of the SimpleLink Wi-Fi device, and it is the first step required before initializing socket communication. The SimpleLink Wi-Fi device supports two methods of establishing a connection: manual connection and preferred networks.

Each of the following methods is combined with a predefined connection policy which instructs the SimpleLink Wi-Fi device on how to act in different states. Indication of connection completion is performed by an asynchronous event (see *Section 3.3.4*)

3.3.3.1 Connection Policies

There are different ways to define the connection policy behavior of the device. These policies define how the device initiates the connection, and helps to maintain a specific connection configuration after reset, which is appropriate for the desired use case. The WLAN connection policy defines up to four options for connecting the SimpleLink Wi-Fi device to a given AP.

The four options for the connection policy follow:

- Auto – The device tries to connect to an AP from the stored profiles based on priority. Up to seven profiles are supported. On the connection attempt, the device selects the highest priority profile. If several profiles are within the same priority, the decision is made based on the security type (WPA \ WPA2 > WEP > OPEN). If the security type is also the same, the selection is based on the received signal strength.

  Set the Auto policy with the following macro: SL_WLAN_CONNECTION_POLICY(1,0,0,0)
- Fast – The device tries to connect to the last connected AP. In this mode, the probe request is not transmitted before the authentication request, as both the SSID and channel are known from previous successful connection.
  - If the Auto policy is also enabled (Auto and Fast), then a profile exists and previous successful connection was performed to this profile. After reset, the device tries to connect to the same profile with no scan (no probe request transmission). If this connection fails, the device starts scanning according to stored profile priority.
  - If the Fast policy is enabled independently and a previous successful connection exists, after reset the device tries to connect to the same AP with no scan (no probe request transmission). If this connection fails, no further scan is performed.

    Set the Auto and Fast policy with the following macro: SL_WLAN_CONNECTION_POLICY(1,1,0,0)
- AnyP2P – Relevant for Wi-Fi-Direct mode only. The device immediately tries to connect to the first Wi-Fi direct device available, supporting push-button only.

  Set the Auto and AnyP2P with the following macro: SL_WLAN_CONNECTION_POLICY(1,0,1,0)
- Auto Provisioning – The device automatically starts the provisioning process if 2 seconds have passed since reset without receiving any command from the host, while no saved profiles exist. Or the device automatically starts the provisioning process after 2 minutes of disconnection, while saved profiles exist (for more information, refer to *Section 8.4.6*).

  Set the Auto and Auto Provisioning with the following macro: SL_WLAN_CONNECTION_POLICY(1,0,0,1)

  More than one connection policy can be set, for example Auto and Fast and Auto provisioning. The connection policy enabled by default is Auto and Auto Provisioning. Setting the connection policy takes effect immediately. For example, if setting the Auto policy and profiles exists, a connection attempt to the highest priority profile is immediately triggered. This configuration is persistent according to the system-persistent configuration.

  An example of an Auto and Fast and Auto provisioning connection policy follows:

```
_i16 Status;

Status  = sl_WlanPolicySet(SL_POLICY_CONNECTION,SL_WLAN_CONNECTION_POLICY(1,1,0,1),NULL,0);
if( Status )
{
    /* error */
}
```

3.3.3.2 Preferred Networks (Profiles)

The SimpleLink Wi-Fi device provides users the ability to predefine up to seven preferred networks. These preferred networks, or profiles, can be used to establish connection automatically according to the connection policy settings. The profiles are stored in the file system (nonvolatile memory), and therefore are preserved during device reset. Profiles can be added, removed, or modified by using the host driver API or internal web browser. Each profile has a priority which defines how connection order occurs. This means the SimpleLink Wi-Fi device tries to connect to the highest priority profile stored first (see Section 3.3.3.1).

Each profile includes the following information:

- SSID
- BSSID
- Security type
- Password
- EAP parameters (enterprise security type)
- Priority

On successful completion of the provisioning process or the WPS process, a new profile is added. Profiles can be added, removed, edited, viewed, or temporarily suspended by using the following APIs:

- **Add Profile**

  Add a profile to the next available index. The return value is the profile index with a value from 0 to 6. Negative values indicate an error. This index identifies the profile, and should be used when deleting or editing the profile.

  The following are examples of adding a WPA2 secured profile with SSID and BSSID:

```
_u8 MacAddr[] = {0xAA,0xBB,0xCC,0xDD,0xEE,0xFF};
SlWlanSecParams_t SecParams;
_i16 Index;

SecParams.Type = SL_WLAN_SEC_TYPE_WPA_WPA2;
SecParams.Key = "123456789";
SecParams.KeyLen = strlen (SecParams.Key);

Index = sl_WlanProfileAdd("Test_AP", strlen("Test_AP"), MacAddr, &SecParams, NULL, 7 /*
Priority*/, 0);
```

- **Delete Profile**

  A specific profile can be deleted by its index. In addition, all profiles can be deleted at once by using the following value as an index: WLAN_DEL_ALL_PROFILES.

  An example for deleting all profiles:

```
_i16 Status;

Status  = sl_WlanProfileDel(SL_WLAN_DEL_ALL_PROFILES);
if( Status )
{
    /* error */
}
```

- **Get Profile**

  The driver also lets the user read the information of a stored profile by its index. For security reasons, this information includes only the public information of the profile. The password is not accessible from the host.

  An example for getting the information on a profile at index 2 follows:

```
_i16 index, Status;
signed char Name[32];
_i16 NameLength;
unsigned char MacAddr[6];
SlWlanSecParams_t SecParams;
SlWlanGetSecParamsExt_t SecExtParams;
_u32 Priority;
```

```
Index = 2;
Status = sl_WlanProfileGet(Index, Name, &NameLength, MacAddr, &SecParams, &SecExtParams,
&Priority);
if( Status )
{
    /* error */
}
```

- **Edit Profile**

  Adding a profile with an existing SSID, BSSID (if applicable), and security type updates the existing entry. If one of these values is different, it is considered a new profile and is saved as a new entry.

- **Suspend Profiles**

  Specific profiles can be suspended without deletion. This setting is not persistent, and it is deleted after reset.

  An example of suspending a profile with index 1, 4, 6 follows:

```
_u32  SuspendedProfilesMask;
_i16 Status;

SuspendedProfilesMask = INDEX1 | INDEX4 | INDEX 6 ; /* 0x29 */
Status  = sl_WlanSet(SL_WLAN_CFG_GENERAL_PARAM_ID, SL_WLAN_GENERAL_PARAM_OPT_SUSPEND_PROFILES,
sizeof(suspendedProfilesMask),(_u8*)&suspendedProfilesMask);
if( Status )
{
    /* error */
}
```

- **Enterprise Profile**

  Only one enterprise profile is supported. Before adding the profile, write certificate files to the following system files:

  – /sys/cert/ca.der - CA for the server authentication
  – /sys/cert/client.der - Optional, if server requests client authentication
  – /sys/cert/private.key - Optional, if server requests client authentication

  An example of adding an enterprise profile follows:

```
SlWlanSecParams_t SecParams;
SlWlanSecParamsExt_t SecExtParams;
_i16 Index;

SecParams.Type = SL_WLAN_SEC_TYPE_WPA_ENT;
SecParams.Key = "123456789";
SecParams.KeyLen = strlen(SecParams.Key);

SecExtParams.User = "Ent_user";
SecExtParams.UserLen = strlen("Ent_user");
SecExtParams.EapMethod = SL_WLAN_ENT_EAP_METHOD_TTLS_TLS;

Index  = sl_WlanProfileAdd("Test_Ent_AP",strlen("Test_Ent_AP"),0, &SecParams ,& SecExtParams,7 /*
Priority*/,0);
```

3.3.3.3 Manual Connection

Manual connection triggers an immediate connection scan, and tries to establish connection to a specific AP. The connection scan continues until a connection completes or a disconnect command is issued. Manual connection is in higher priority than any other connection type. The connection can be established according to SSID, or SSID and BSSID. The connection command can be applied only by the host driver and returns immediately, before the connection is established. The host application should wait for the connection asynchronous events as in all other connection methods.

Example:

```
SlWlanSecParams_t SecParams;
_i16 Status;

SecParams.Type = SL_WLAN_SEC_TYPE_WPA_WPA2;
SecParams.Key = "123456789";
SecParams.KeyLen = strlen(SecParams.Key);

Status = sl_WlanConnect("Test_Ent_AP",strlen("Test_Ent_AP"),0 ,&SecParams ,0,);
if( Status )
{
    /* error */
}
```

3.3.4 Events and Errors

The host can receive an indication of specific states through events or errors.

Asynchronous events can be sent to the host at any given time with an indication of specific states and specific data for each event. To listen to these events and determine the needed information, a handler must be implemented in the user application and registered under the user.h header file. The following event may be received in relation to a WLAN connection:

- **SL_WLAN_EVENT_CONNECT**

  Indicates the connection is successful and includes the following information:
  - SSID
  - SSID length
  - BSSID
- **SL_WLAN_EVENT_DISCONNECT**

  Indicates the disconnection is successful and includes the following information:
  - SSID
  - SSID length
  - BSSID
  - Disconnect reason code

Errors are indicated by the return value of the API. Each error code is unique. Table 3-3 lists common errors that require user action (a complete list of errors is under the error.h file in the host driver).

Table 3-3. Common Errors

Error	Value	Comments
SL_ERROR_ROLE_STA_ERR	–4107	Initialization failure in STA mode
SL_ERROR_WLAN_INVALID_ROLE	–2050	Action applied does not match the current mode.
SL_ERROR_WLAN_KEY_ERROR	–2049	One of the security parameters or SSID supplied is wrong (invalid length or not supported).
SL_ERROR_WLAN_INVALID_SECURITY_TYPE	–2054	
SL_ERROR_WLAN_PASSPHRASE_TOO_LONG	–2055	
SL_ERROR_WLAN_PASSWORD_ERROR	–2058	
SL_ERROR_WLAN_SSID_LEN_ERROR	–2060	
SL_ERROR_WLAN_ILLEGAL_WEP_KEY_INDEX	–2064	
SL_ERROR_WLAN_EAP_WRONG_METHOD	–2057	One of the EAP security parameters supplied is wrong (invalid length or not supported).
SL_ERROR_WLAN_EAP_ANONYMOUS_LEN_ERROR	–2059	
SL_ERROR_WLAN_USER_ID_LEN_ERROR	–2061	
SL_ERROR_WLAN_PREFERRED_NETWORK_LIST_FULL	–2062	No free profile
SL_ERROR_WLAN_INVALID_POLICY_TYPE	–2066	Invalid policy type. Value is not supported.
SL_ERROR_WLAN_WIFI_ALREADY_DISCONNECTED	–2071	Applying disconnect command when disconnected

Table 3-3. Common Errors (continued)

Error	Value	Comments
SL_ERROR_WLAN_GET_NETWORK_LIST_EAGAIN	–2073	Scan was not enabled, one-shot scan is immediately triggered and the user should fetch the scan results again
SL_ERROR_WLAN_GET_PROFILE_INVALID_INDEX	–2074	Profile index is too high or does not exist

3.4 Access Point

3.4.1 General Description

Access point (AP) is supported by the SimpleLink Wi-Fi device. This mode is used mostly to set the device network configuration. In AP mode, the unprovisioned SimpleLink Wi-Fi device wakes up initially as an AP with an SSID defined by the equipment manufacturer. Before trying to connect to the home network for the first time, the unprovisioned device creates a network of its own, allowing a PC or a smartphone to connect to it directly and facilitate its initial configuration. AP mode supports up to four connected stations and offers a secured connection. Managing the station connection can be done by using host commands (distributes IP address, see connected stations, disconnect stations, add or remove stations from the black list, and so on). Specific settings and modes of operation are unique for AP mode.

3.4.2 Configurations and Settings

The SimpleLink Wi-Fi device AP configuration is done by using the host driver API. Several configurations exist for each specific use case. Some of the configurations are persistent according to the system-persistent configuration, and some are persistent and nonpersistent, as specified in each configuration specification (more information is in annex 2 Persistency). Not all configurations are mandatory because the device has default values, according to Table 3-4.

Table 3-4. AP Default Parameters

Configuration	Default Value
Interface	IPv4
Address	Static with the following parameters: IP 10.123.45.1, Subnet mask: 255.255.255.0 Default gateway: 10.123.45.1, DNS: 10.123.45.1
AP TX power	0 (no back-off, maximum TX power)
Country code	EU (channels 1–13), default channel is 6
Connection policy	N/A
Calibration mode	Normal
Applications	DHCP server and HTTP server and MDNS and DNS server

3.4.2.1 Set Mode

AP mode is the default initial mode of the device. AP configuration is not effective until the device enters into AP mode. This configuration requires a reset, and is persistent with no dependency on the system-persistent configuration. If the device gets set to a different mode, and the AP mode is required again, the following API should be called.

```
_i16 Role;
_i16 Status;

Status = sl_WlanSetMode(ROLE_AP);
if( Status )
{
 /* error */
}
sl_Stop(0);
Role = sl_Start(NULL,NULL,NULL);
```

```
if (ROLE_AP != Role)
{
    /* Role Error */
}
```

3.4.2.2 Set General AP Parameters

AP mode is activated with default configuration, and reconfiguration is not mandatory, although this option exists. The following settings are available, require reset, and are persistent with no dependency on the system-persistent configuration.

- **SSID**

  The SimpleLink Wi-Fi device default SSID is ‘mysimplelink-xxyyzz’ where ‘xxyyzz’ are the last six digits of the device MAC address. Because the MAC address is unique, the SSID is also unique. Still, the SSID configuration exists with maximum length of 32 characters.

  Example:

```
_i16 Status;
_u8  Ssid[] = "Test_AP";

Status=sl_WlanSet(SL_WLAN_CFG_AP_ID, SL_WLAN_AP_OPT_SSID, strlen(Ssid), Ssid);
if( Status )
{
    /* error */
}
```

- **Hidden SSID**

  The device can be configured and not broadcast the SSID inside the Beacon frame. This configuration is disabled by default.

  Example:

```
_i16 Status;
_u8  hidden = TRUE;

 Status =  sl_WlanSet(SL_WLAN_CFG_AP_ID, SL_WLAN_AP_OPT_HIDDEN_SSID, 1, (_u8 *)& hidden);
if( Status )
{
    /* error */
}
```

- **Set Country Code / Regulatory Domain**

  Set the country code for AP mode. Possible values are US, EU, JP, and all others are considered an error. Each country code sets different channel ranges:

  – US: 1–11

  – EU, JP: 1–13

  Example:

```
_i16 Status;
_u8  Str[] = "US";

Status =  sl_WlanSet(SL_WLAN_CFG_GENERAL_PARAM_ID, SL_WLAN_GENERAL_PARAM_OPT_COUNTRY_CODE, 2,
Str);
if( Status )
{
    /* error */
}
```

- **Channel**

  Set the SimpleLink Wi-Fi device AP Operational Channel. Possible values from 1 to 13. Any other value is considered as error. The default channel in AP mode is 6.

  Example:

```
_i16 Status;
_u8  channel = 1;
```

```
Status =  sl_WlanSet(SL_WLAN_CFG_AP_ID, SL_WLAN_AP_OPT_CHANNEL, 1, (_u8 *)& channel);
if( Status )
{
    /* error */
}
```

- **Security Type**

  Set the SimpleLink Wi-Fi device AP network security mode configuration. Possible security types are OPEN, WEP and WPA\WPA2. The default value is Open security.

  Example:

```
_i16 Status;
_u8  val = SL_WLAN_SEC_TYPE_WPA_WPA2;

Status =  sl_WlanSet(SL_WLAN_CFG_AP_ID, SL_WLAN_AP_OPT_SECURITY_TYPE, 1, (_u8 *)&val);
if( Status )
{
    /* error */
}
```

- **Password**

  The SimpleLink Wi-Fi device configured as secured AP includes a security password. This password is used for all secured networks except OPEN. Setting the SimpleLink Wi-Fi device with a WEP security includes a password length of 5 or 10 characters in HEX format, and 13 or 26 characters in ASCII format. For the WPA \ WPA2 security type, set the password length from 8 to 64 characters. The default value is not supplied, and when using a secured network the password must be set.

  Example:

```
_i16 Status;
_u8  password[] = {"123456789"};
_u16  len = strlen(password);

Status = sl_WlanSet(SL_WLAN_CFG_AP_ID, SL_WLAN_AP_OPT_PASSWORD, len, (_u8 *)password);
if( Status )
{
    /* error */
}
```

- **Maximum Stations Connected**

  The SimpleLink Wi-Fi device lets users configure the value of the maximum connected stations allowed. The available range is from one to four stations. The default value is four stations.

  Example:

```
_i16 Status;
_u8 max_ap_stations = 3;

Status = sl_WlanSet(SL_WLAN_CFG_AP_ID, SL_WLAN_AP_OPT_MAX_STATIONS, sizeof(max_ap_stations), (_u8
*)&max_ap_stations);
if( Status )
{
    /* error */
}
```

- **Station Aging Time**

  The SimpleLink Wi-Fi device lets users set the value of the maximum time before a station is considered inactive. After this time expires, a null data frame is sent to the station. If this frame is not acknowledged and no other frames are received, the station is disassociated. The default value is 60 seconds.

  Example:

```
_i16 Status;
_u16 max_ap_sta_aging = 50;

Status  = sl_WlanSet(SL_WLAN_CFG_AP_ID, SL_WLAN_AP_OPT_MAX_STA_AGING, sizeof(max_ap_sta_aging),
```

```
(_u8 *)&max_ap_sta_aging);
if( Status )
{
    /* error */
}
```

- **AP TX Power**

  The SimpleLink Wi-Fi device lets users set the TX power level in AP mode. The value is from 0 to 15, as dB offset from maximum power (0 is MAX power).

  Example:

```
_i16 Status;
_u8  ApPower = 3;

Status = sl_WlanSet(SL_WLAN_CFG_GENERAL_PARAM_ID, SL_WLAN_GENERAL_PARAM_OPT_AP_TX_POWER,1,(_u8
*)& ApPower);
if( Status )
{
    /* error */
}
```

- **Set Info Elements**

  The SimpleLink Wi-Fi device lets users set up to four custom information (info) elements per mode, AP, or Wi-Fi Direct GO. For AP mode, no more than 300 bytes (SL_INFO_ELEMENT_MAX_TOTAL_LENGTH_AP) can be stored for all info elements (for example, 4 info elements of 75 bytes each). For Wi-Fi Direct GO mode, no more than 160 byes (SL_INFO_ELEMENT_MAX_TOTAL_LENGTH_P2P_GO) can be stored for all info elements (for example, 4 info elements of 40 bytes each). To delete an info element, use the relevant index with length 0.

  Example:

```
_i16 Status;
SlWlanSetInfoElement_t    InfoEle;

InfoEle.Index     = Index;                  /* Index of the info element. range: 0 -
 SL_WLAN_MAX_PRIVATE_INFO_ELEMENTS_SUPPROTED */
InfoEle.Role      = Role;                   /* SL_WLAN_INFO_ELEMENT_AP_ROLE (0) or
SL_WLAN_INFO_ELEMENT_P2P_GO_ROLE (1) */
InfoEle.IE.Id     =  Id;                    /* Info element ID. if
SL_WLAN_INFO_ELEMENT_DEFAULT_ID (0) is set, ID will be set to 221 */
/* Organization unique ID. If all 3 bytes are zero - it will be replaced with 08,00,28 */
InfoEle.IE.Oui[0] =  Oui0;            /* Organization unique ID first Byte */
InfoEle.IE.Oui[1] =  Oui1;            /* Organization unique ID second Byte */
InfoEle.IE.Oui[2] =  Oui2;            /* Organization unique ID third Byte */
InfoEle.IE.Length = Len;              /* Length of the info element. must be smaller than 253
bytes */
InfoEle (infoele.IE.Data, 0, SL_WLAN_INFO_ELEMENT_MAX_SIZE);

if ( Len < = SL_WLAN_INFO_ELEMENT_MAX_SIZE )
{
    memcpy(InfoEle.IE.Data, IE, Len);
    Status =  sl_WlanSet(SL_WLAN_CFG_GENERAL_PARAM_ID, SL_WLAN_GENERAL_PARAM_OPT_INFO_ELEMENT,
    sizeof(SlWlanSetInfoElement_t),(_u8* )&InfoEle);
    if( Status )
    {
        /* error */
    }
}
```

3.4.2.3 Get General AP Parameters

AP mode configuration can be retrieved by host commands. Each set parameter (discussed in the previous section) can be retrieved with the following API, and the same configuration ID and configuration option.

Example:

```
_i16 Status;
_i8 Ssid[33];
_u16 Len = 33;
_u16  Config_opt;

memset(ssid,0,33);
Config_opt = SL_WLAN_AP_OPT_SSID;
Status  = sl_WlanGet(SL_WLAN_CFG_AP_ID, &config_opt , &len, (_u8* )ssid);
if( Status )
{
      /* error */
}
```

3.4.2.4 Black List

The black list lets users filter the stations which can connect to the AP according to their MAC address. The list contains up to eight entries and is persistent. Adding or removing a station to and from the list includes file write. Adding a station to the black list, which is currently connected to the AP, does not disconnect this station from the AP. The host application can enable and disable the black list without erasing the list of stations. By default, the black list is enabled. Removing a station from a list can be done by the MAC address or by the index of the entry.

- **Set Black List Mode**

  The SimpleLink Wi-Fi device allows enabling or disabling the black list mode.

  Example:

```
_i16 Status;
_u8  access_list_mode = SL_WLAN_AP_ACCESS_LIST_MODE_DENY_LIST;

Status = sl_WlanSet(SL_WLAN_CFG_AP_ID, SL_WLAN_AP_ACCESS_LIST_MODE, sizeof(access_list_mode),
(_u8 *)&access_list_mode);
if( Status )
{
    /* error */
}
```

- **Add MAC to the Black List**

  Add a station to the black list. Adding a station to the black list will not disconnect it.

  Example:

```
_i16 Status;
_u8  sta_mac[6] = { 0x00, 0x22, 0x33, 0x44, 0x55, 0x66 };

Status  = sl_WlanSet(SL_WLAN_CFG_AP_ID, SL_WLAN_AP_ACCESS_LIST_ADD_MAC, sizeof(sta_mac), (_u8 *)
&sta_mac);
if( Status )
{
    /* error */
}
```

- **Remove MAC from the Black List**

  Removing a station from the black list can be done using the MAC address or entry index (retrieve the entry index with Get Black option, which is specified as follows).

  Examples of removing entry according to the MAC address:

```
_i16 Status;
_u8  sta_mac[6] = { 0x00, 0x22, 0x33, 0x44, 0x55, 0x66 };

Status  = sl_WlanSet(SL_WLAN_CFG_AP_ID, SL_WLAN_AP_ACCESS_LIST_DEL_MAC, sizeof(sta_mac), (_u8 *)
&sta_mac);
if( Status )
{
    /* error */
```

```
}
```

Examples of removing entry according to the entry index:

```
_i16 Status;
_u8  sta_index = 0;

Status  = sl_WlanSet(SL_WLAN_CFG_AP_ID, SL_WLAN_AP_ACCESS_LIST_DEL_IDX, sizeof(sta_index), (_u8
*)&sta_index);
if( Status )
{
    /* error */
}
```

- **Get Number of Entries in the Black List**

  Get the number of denied stations in the current black list.

  Example:

```
_i16 Status;
_u8 aclist_num_entries;
_u16 config_opt = SL_WLAN_AP_ACCESS_LIST_NUM_ENTRIES;
_u16 len = sizeof(aclist_num_entries);

Status  = sl_WlanGet(SL_WLAN_CFG_AP_ID, &config_opt, &len, (_u8 *)&aclist_num_entries);
if( Status )
{
    /* error */
}
```

- **Get the Black List**

  Get the AP black list from a specific index. The number of entries in the list is extracted from the returned total length, divided by the address size.

  Example:

```
_i16 Status;
_u8 aclist_mac[SL_WLAN_MAX_ACCESS_LIST_STATIONS][MAC_LEN];
unsigned char aclist_num_entries;
unsigned short config_opt;
unsigned short len;
int actual_aclist_num_entries;
unsigned short start_aclist_index;
unsigned short aclist_info_len;
int i;

start_aclist_index = 0;
aclist_info_len = 2*MAC_LEN;
Status =   sl_WlanGet(SL_WLAN_CFG_AP_ACCESS_LIST_ID, &start_aclist_index, &aclist_info_len,
(_u8*)&aclist_mac[start_aclist_index]);
if( Status )
{
    /* error */
}
actual_aclist_num_entries = aclist_info_len / MAC_LEN; /* number of stations in the list */
```

3.4.3 Set Network Configuration

3.4.3.1 Set AP IP Parameters

The SimpleLink Wi-Fi device lets users set the AP static IPv4 parameters (IPv6 is not supported in AP mode). This configuration is persistent, and reset is required for changes to apply. The following parameters can be configured:

- IP – IPv4 static address
- Subnet mask – IPv4 network mask address

- Default gateway – IPv4 default gateway address
- DNS server – IPv4 DNS server address

Example:

```
_i16 Status;
_i16 Role;
SlNetCfgIpV4Args_t ipV4;

ipV4.Ip          = (_u32)SL_IPV4_VAL(10,1,1,201);           /*  IP address               */
ipV4.IpMask      = (_u32)SL_IPV4_VAL(255,255,255,0);        /* Subnet mask               */
ipV4.IpGateway   = (_u32)SL_IPV4_VAL(10,1,1,1);             /* Default gateway address   */
ipV4.IpDnsServer = (_u32)SL_IPV4_VAL(8,16,32,64);           /* _u32 DNS server address   */

Status =
sl_NetCfgSet(SL_NETCFG_IPV4_AP_ADDR_MODE,SL_NETCFG_ADDR_STATIC,sizeof(SlNetCfgIpV4Args_t),(_u8
*)&ipV4);
if( Status )
{
    /* error */
}
/* restart the device */
Status = sl_Stop(0);
Role = sl_Start(NULL,NULL,NULL);
```

3.4.4 Station Management

The SimpleLink Wi-Fi device lets users manage the connected stations by using host commands. Users can enable the device to view the connected stations and disconnect stations according to their MAC address.

3.4.4.1 Get Connected Stations

The SimpleLink Wi-Fi device lets users get the current number of connected stations and get the full list of connected stations.

Example:

```
_i16 Status;
_u8 NumConnectedStations;
_u16 ValueLen  = sizeof(_u8);
_u32 i;
SlNetCfgStaInfo_t ApStaList[4];
_u16 sta_info_len;
_u16 start_sta_index = 0;
_u16 actual_num_sta;


Status  = sl_NetCfgGet(SL_NETCFG_AP_STATIONS_NUM_CONNECTED, NULL, &ValueLen,
&NumConnectedStations);
if( Status )
{
    /* error */
}

/* get list of connected stations */
start_sta_index = 0; /* from index */
sta_info_len = sizeof(ApStaList); /* 4 stations to get */
Status  = sl_NetCfgGet(SL_NETCFG_AP_STATIONS_INFO_LIST, &start_sta_index, &sta_info_len, (_u8
*)ApStaList);
if( Status )
{
    /* error */
}

/* extract actual stations in the response */
```

```
actual_num_sta = sta_info_len / sizeof(SlNetCfgStaInfo_t);
```

3.4.4.2 Disconnect a Station

In AP mode, the SimpleLink Wi-Fi device lets users force disconnect a station by using its MAC address. Disconnecting a station does not add it to the black list, and the station can immediately connect again.

Example:

```
_i16 Status;
_u8  ap_sta_mac[6] = { 0x00, 0x22, 0x33, 0x44, 0x55, 0x66 };

Status  = sl_NetCfgSet(SL_NETCFG_AP_STATION_DISCONNECT,1,SL_MAC_ADDR_LEN,(_u8 *)ap_sta_mac);
if( Status )
{
    /* error */
}
```

3.4.5 Events and Errors

The host can receive indication of specific states through events or errors. Asynchronous events can be sent to the host at any time, with indication of a specific state and specific data for each event. To listen to these events and determine the needed information, a handler should be implemented in the user application, and registered under the user.h to catch the following possible events:

- SL_WLAN_EVENT_STA_ADDED

  Indicates connection is successfully completed and includes the following information: MAC address
- SL_WLAN_EVENT_STA_REMOVED

  Indicates connection is successfully completed and includes the following information: MAC address
- SL_NETAPP_EVENT_IPV4_ACQUIRED

  Indicates connection is successfully completed and includes the following information:
  - IPv4 address
  - Default Gateway address
  - DNS address

Errors are indicated by the return value of the API. Each error code is unique.Table 3-5 lists common errors that require user action (a complete list of possible errors is under the file error.h in the host driver).

Table 3-5. Common Errors

Error	Value	Comments
SL_ERROR_ROLE_AP_ERR	–4108	Initialization failure in AP mode.
SL_ERROR_WLAN_TX_POWER_OUT_OF_RANGE	–2167	Configured TX power is out of range.
SL_ERROR_WLAN_INVALID_ROLE	–2050	Action applied does not match the current mode.
SL_ERROR_WLAN_CANNOT_CONFIG_SCAN_DURING_PROVISIONING	–2052	Illegal action occurred during provisioning.
SL_ERROR_WLAN_INVALID_COUNTRY_CODE	–2464	Invalid country code
SL_ERROR_WLAN_INVALID_AP_PASSWORD_LENGTH	–2168	Configured AP password has invalid length.
SL_ERROR_WLAN_AP_SCAN_INTERVAL_TOO_SHORT	–2176	Scan in AP mode has a minimum interval of 10 seconds.

3.4.6 Limitations

A list of device limitations follows:

- A maximum of four stations can connect to the SimpleLink Wi-Fi device in AP mode.
- Only 802.11bg is supported.
- No power save support in AP mode.

3.5 Wi-Fi Direct

3.5.1 General Description

The SimpleLink Wi-Fi device supports the Wi-Fi Direct standard, which enables the device to connect directly to other devices without an AP. In this mode, one device functions as a GROUP OWNER (AP-like mode) and the other functions as a CLIENT (STA-like mode) by inheriting the entire STA and AP attributes. This mode makes it simple and convenient to establish a connection without joining a traditional home, office, or hotspot network.

3.5.2 Supported Features

A list of supported features follows:

- CLIENT and GROUP OWNER (GO) roles
- Configuring device name, type, listen and operation channels
- Device discovery (FULL/SOCIAL)
- Negotiation with all intents (0 to 15)
- Negotiation initiator policy – Active, Passive, Random Backoff
- WPS method Push-Button, Pin code (keypad and display)
- Join an existing Wi-Fi Direct group
- Device invites to reconnect persistent group (fast-connect)
- Group owner accepts join request
- Persistent group owner, responds to invite requests
- P2P Connect-Disconnect-Connect transition, also between different modes (for example, GO-CL-GO)
- P2P Client Legacy PS and NoA support
- Separate IP Configuration for P2P mode
- Separate Net Applications configuration on top of Wi-Fi Direct CL/GO mode

3.5.3 Configurations and Settings

The SimpleLink Wi-Fi device Wi-Fi Direct settings are configured by using the host driver API which controls the device. Several configurations for each specific use case exist. Some of the configurations are persistent according to the system-persistent configuration, some are persistent and some are nonpersistent, as specified in each configuration specification (more information is at annex 2 Persistency). Not all configurations are mandatory because the device has default values according to Table 3-6, which lists the Wi-Fi Direct default parameters.

Table 3-6. Wi-Fi Direct Default Parameters

Configuration	Default Value
Interface	IPv4
STA Tx power	0 (no backoff, maximum Tx power)
Country code	EU (channels 1 to 13)
Connection policy	Auto and Auto Provisioning
Calibration mode	Normal
Applications	HTTP server
Intent	3
Negotiator	2
CL address	DHCP
GO address	Static with the following parameters: IP 10.123.45.1, Subnet mask: 255.255.255.0 Default gateway: 10.123.45.1, DNS: 10.123.45.1
Device name	mysimplelink_XX (xx = Random 2 characters)

Table 3-6. Wi-Fi Direct Default Parameters (continued)

Configuration	Default Value
Device type	1-0050F204-1
listen channel	Random channel between 1, 6, or 11
Operational channel	Random channel between 1, 6, or 11

3.5.3.1 Configuring Wi-Fi Direct General Parameters

- **Set Mode**

  Wi-Fi Direct mode is not the initialization mode by default, therefore it must be set by the application. The following API should be called to set the device in Wi-Fi Direct mode. Wi-Fi Direct configuration is not effective until the device enters Wi-Fi Direct mode. This configuration requires a reset and persistent with no dependency on the system-persistent configuration.

  Example:

```
_i16 Role;
_i16 Status;

Status = sl_WlanSetMode(ROLE_P2P);
if( Status )
{
    /* error */
}
Status = sl_Stop(0);
Role = sl_Start(NULL,NULL,NULL);
if (ROLE_P2P != Role)
{
    /* error */
}
```

- **Set Network Configuration**

  The network configuration for Wi-Fi Direct mode is similar to the STA and AP modes. For CLIENT use STA network configuration parameters, and for GO use AP network configuration parameters. Persistent:

  – CL – This configuration is persistent according to the system-persistent configuration
  – GO – This configuration is persistent regardless of the system-persistent configuration

  To change the default configuration, the following settings are available:

  - CLIENT – same network confirmation as the STA mode (static or DHCP address)
  - GO – same network confirmation as the AP mode (static address)

  An example of setting CLIENT static IPv4 address:

```
SlNetCfgIpV4Args_t      ipV4;
 _i16 Status;

ipV4.Ip          = (_u32)SL_IPV4_VAL(192,168,0,108);           /* IP address */
ipV4.IpMask      = (_u32)SL_IPV4_VAL(255,255,255,0);           /* Subnet mask for this STA/P2P */
ipV4.IpGateway   = (_u32)SL_IPV4_VAL(192,168,0,1);             /* Default gateway address */
ipV4.IpDnsServer = (_u32)SL_IPV4_VAL(192,168,0,1);             /*  DNS server address */


Status = sl_NetCfgSet(SL_NETCFG_IPV4_STA_ADDR_MODE,
SL_NETCFG_ADDR_STATIC,sizeof(SlNetCfgIpV4Args_t) ,(_u8 *)&ipV4);
if( Status )
{
    /* error */
}
```

- **Set Device Name**

  The device name must be unique because the Wi-Fi Direct connection is device-name based. The

device name is compound of the URN and two random characters. Users can set only the URN; the random characters are internally generated. Default = mysimplelink_XX (xx = random two characters). This configuration is persistent according to the system-persistent configuration.

Example:

```
_u8 device_name[] = "Simple_WiFi_Direct";
_i16 Status;

Status = sl_NetAppSet (SL_NETAPP_DEVICE_ID,SL_NETAPP_DEVICE_URN, strlen(device_name), (_u8 *)
device_name);
if( Status )
{
    /* error */
}
```

- **Set Device Type**

  The following macro is used to set the Wi-Fi Direct device type. The device type is published under P2P I.E. The device type is part of the Wi-Fi Direct discovery parameters, and is used to better recognize the device. The maximum length is 17 characters. Default = 1-0050F204-1. This configuration is persistent according to the system-persistent configuration.

  Example:

```
_i16 Status;
_u8   str[17];
_u16  len = strlen(device_type);

memset(str, 0, 17);
memcpy(str, device_type, len);
Status  = sl_WlanSet(SL_WLAN_CFG_P2P_PARAM_ID, SL_WLAN_P2P_OPT_DEV_TYPE, len, str);
if( Status )
{
    /* error */
}
```

- **Set Listen and Operational Channels**

  The listen channel is used for the discovery state and the value can be 1, 6, or 11. The device stays in this channel when waiting for Wi-Fi Direct probes requests. The operation channel is only used by the GO device. The GO device moves to this channel after the negotiation phase. The default listen channel is randomly assigned between channels 1, 6, or 11. This configuration is persistent according to the system-persistent configuration. The regulatory domain class should be 81 in 2.4 G.

  An example for setting the listen channel to 11 and the operational channel to 6 follows:

```
_u8 channels [4];
_i16 Status;

channels [0] = (unsigned char)11; /* listen channel */
channels [1] = (unsigned char)81; /* listen regulatory class */
channels [2] = (unsigned char)6;  /* operational channel */
channels [3] = (unsigned char)81; /* operational regulatory class */
Status = sl_WlanSet(SL_WLAN_CFG_P2P_PARAM_ID, SL_WLAN_P2P_OPT_CHANNEL_N_REGS,4,channels);
if( Status )
{
    /* error */
}
```

3.5.3.2 Set Wi-Fi Direct Policy

The Wi-Fi Direct connection policy divides into two major configurations:

- **Wi-Fi Direct Intent Value**

  This value indicates in which Wi-Fi Direct mode the device acts (CLIENT, GO, or other). This configuration is done by using the macro SL_WLAN_P2P_POLICY. Three defines can be used when setting the intent:

  1. SL_WLAN_P2P_ROLE_CLIENT (intent 0): Indicates that the device is forced to be CLIENT.

2. SL_WLAN_P2P_ROLE_NEGOTIATE (intent 7): Indicates that the device can be either CLIENT or GO, depending on the Wi-Fi Direct negotiation tie-breaker. This tie-breaker is the system default.
3. SL_WLAN_P2P_ROLE_GROUP_OWNER (intent 15): Indicates that the device is forced to be P2P GO.

NOTE: This configuration is persistent according to the system-persistent configuration.

- **Negotiation Initiator**

  This value determines whether the SimpleLink Wi-Fi device first initiates the negotiation, or passively waits for the remote side to initiate the negotiation. This configuration must be used when working with two SimpleLink Wi-Fi devices. In general, the user does not have a GUI to start the negotiation by pressing a button or by entering a pin key. Therefore, this option is given to avoid starting the negotiation at the same time by both devices after the discovery process.

1. SL_WLAN_P2P_NEG_INITIATOR_ACTIVE: When the remote peer is found after the discovery process, the device immediately sends the negotiation request to the peer device.
2. SL_WLAN_P2P_NEG_INITIATOR_PASSIVE: When the remote peer is found after the discovery process, the device passively waits for the peer to start the negotiation, and only responds after.
3. SL_WLAN_P2P_NEG_INITIATOR_RAND_BACKOFF: When the remote peer is found after the discovery process, the device triggers a random timer (1 to 6 seconds). During this period, the device passively waits for the peer to start the negotiation. If the timer expires without negotiation, the device immediately sends the negotiation request to the peer device. This is the system default, and also the recommendation for working with two SimpleLink Wi-Fi devices out-of-the box, because no negotiation synchronization must be done.

NOTE: This configuration is persistent according to the system-persistent configuration.

Example:

```
_i16 Status;

Status  = sl_WlanPolicySet(SL_WLAN_POLICY_P2P,  SL_WLAN_P2P_POLICY( SL_WLAN_P2P_ROLE_NEGOTIATE,

              SL_WLAN_P2P_NEG_INITIATOR_RAND_BACKOFF) , NULL,0);
if( Status )
{
    /* error */
}
```

3.5.3.3 Configure Connection Policy

This policy is used for automatic connection. The system tries to connect to a peer automatically after reset, or after disconnect operation by the remote peer. There is a general mechanism for the peer profile and peer profile configuration which is not described in this section, though an example of how to add a profile is explained in a later section. The mechanism described here explains how the device uses these profiles in relation to the Wi-Fi Direct automatic connection. The same connection policy can also be configured in STA mode, use the same setting parameters, and be applied in both modes, but it has slight differences.

The four connection policy options follow:

- Auto – This policy is similar to Auto Start in STA mode. The device starts the Wi-Fi Direct find process, and searches for all Wi-Fi Direct profiles stored on the device, then tries to find the best candidate to start negotiating. If at least one candidate is found, the connection attempt is triggered. If more than one device is found, the best candidate according to profiles parameter is chosen.
- Fast – In Wi-Fi Direct mode, this policy is the equivalent to the Wi-Fi Direct persistent group, but it has a different meaning between GO and CLIENT. This option is very useful for making fast connection after reset, but it is dependent on the last connection state. This option is active only if there was a successful connection before the device was reset, because the last connection parameters are saved and used by the fast connection option. If the device was CLIENT in its last connection (before reset or remote disconnect) then following the reset, users must send the p2p_invite to the previously

connected GO, to perform fast reconnecting. If the device was GO in its last connection (before reset or remote disconnect) then following reset, users must reinvoke the p2p_group_owner, and wait for the previously connected peer to reconnect to the device.

- AnyP2P policy – This policy creates a connection to any Wi-Fi Direct peer device found during discovery. This option does not use any stored profiles and is relevant for negotiation with push-button only.
- Auto Provisioning – This policy is not relevant in Wi-Fi Direct mode. Each option in this macro should be sent or set as true or false. Multiple options can be used. This configuration is persistent according to the system-persistent configuration.

Example:

```
_i16 Status;

Status = sl_WlanPolicySet(SL_WLAN_POLICY_CONNECTION,SL_WLAN_CONNECTION_POLICY(1,1,0,1),NULL,0);
if( Status )
{
    /* error */
}
```

3.5.4 Connection

- **Discovering Remote Wi-Fi Direct Peers**

  This section describes how to start a Wi-Fi Direct search or discovery, and how to view the discovered remote Wi-Fi Direct devices. The scan policy must be set to start the Wi-Fi Direct find process, and to discover remote Wi-Fi Direct peers. This process is done by setting a scan policy for Wi-Fi Direct mode.

  NOTE:
  - Setting the scan policy should be done while the device is in Wi-Fi Direct mode.
  - Wi-Fi Direct discovery is performed as a part of any connection, but it can be activated using SCAN_POLICY as well.
  - This configuration is not persistent.

  Example:

```
_u32 intervalInSeconds = 20;
_i16 Status;

Status  = sl_WlanPolicySet(SL_WLAN_POLICY_SCAN, SL_WLAN_SCAN_POLICY(1,1),
(_u8*)&intervalInSeconds,sizeof(intervalInSeconds));
if( Status )
{
    /* error */
}
```

- **Retrieve Remote Wi-Fi Direct Peers**

  There are two ways to see and get Wi-Fi Direct remote devices that were discovered during the Wi-Fi Direct find and search operation:

  – Listening to the event SL_WLAN_EVENT_P2P_DEVFOUND:

    This event is sent asynchronously to the host when a remote Wi-Fi Direct is found, and contains the MAC address, device name, and length of the device name. By listening to this event, the user can immediately find each remote Wi-Fi Direct device that exists in their neighborhood, and issue a connect or add profile command.

  – Calling to API sl_WlanGetNetworkList:

    By calling to this API the user receives a list of remote peers that were found during the scan and are saved in the device cache memory. By receiving the network list, the user can immediately find any remote Wi-Fi Direct device and issue a manual connection or add profile command.

  Example:

```
SlWlanNetworkEntry_t netEntries[30];

_i16 resultsCount = sl_WlanGetNetworkList(0,30,&netEntries[0]);
```

- **Wi-Fi Direct Remote Connection**

  Enabling the scan policy sets the device to be discoverable for other devices. The two following options are available to complete the connection:

  – Combine the scan policy first with the connection policy AnyP2P, and allow the remote device to find and complete the connection without any action from the user side (PBC only).

  – Listen to the SL_WLAN_EVENT_P2P_REQUEST event. This event holds information about the remote device that initiated the connection such as the device name, name length, MAC address, and WPS method. To complete the connection issue, connect or add profile command with the correct parameters.

- **Negotiation Method**

  The following are two different Wi-Fi Direct negotiation methods which indicate the WPS phase that follows to the negotiation:

  – Push-button

    Both sides negotiate with PBC method. Define: SL_WLAN_SEC_TYPE_P2P_PBC.

  – **Pin Code Connection**

    Divided to two options:

    - PIN_DISPLAY – this side looks for this pin to be written by its remote peer. Define: SL_WLAN_SEC_TYPE_P2P_PIN_DISPLAY
    - PIN_KEYPAD – this side sends a pin code to its remote peer. Define: SL_WLAN_SEC_TYPE_P2P_PIN_KEYPAD

These parameters influence the negotiation method and are supplied during the manual connection API command that comes from the host or by setting the profile for automatic connection. The negotiation method is performed by the device without user interference.

NOTE: If no pin code is entered in the display side, the NWP auto-generates the pin code from the device MAC using the following method:

1. Take the 7 LSB decimal digits in the device MAC address.
2. Add the checksum of the 7 LSB decimal digits to the LSB (8 digits total).

For example, if the MAC address is 03:4A:22:3B:FA:42, convert to it decimals (059:250:066); 7 LSB decimal digits are: 9250066, and the WPS pin checksum digit is 2. The default pin code for this MAC is 92500662.

Configure the negotiation method by setting the security type in the security structure when issuing a connect or add profile command.

- **Push Button**: secParams.Type = SL_WLAN_SEC_TYPE_P2P_PBC
- **Pin Code Keypad**:
  – secParams.Type = SL_WLAN_SEC_TYPE_PIN_KEYPAD
  – secParams.Key = “12345670”
- **Pin Code Display:**
  – secParams.Type = SL_WLAN_SEC_TYPE_PIN_ DISPLAY
  – secParams.Key = “12345670”
- **Manual Connection**

  After finding a remote Wi-Fi Direct device, the host can instruct the device to connect to it by issuing a simple connect command. This command performs immediate Wi-Fi Direct discovery, and once the remote device is found, the negotiation phase is started according to the negotiation initiator policy, method, and intent selected.

NOTE:

- The connection parameters are not saved to flash memory so in case of disconnection or reset no reconnection will be done, unless fast-connect policy is on.
- This connection is in higher priority than connection through profiles. It means that if there is already an existing Wi-Fi Direct connection in the system, the current connection will be disconnected and the manual connection is operated.
- At the beginning of the discovery phase, full scan cycle on all channels is performed to find Autonomous GO which can operate on every channel.

Example:

```
_i16 Status;
SlWlanSecParams_t SecParams;

Status  = sl_WlanConnect("my-tv-p2p-device", 16, NULL, &SecParams ,0);
if( Status )
{
    /* error */
}
```

- **Manual Disconnection**

  The manual disconnect option lets the user disconnect from the remote peer by a host command. This command performs Wi-Fi Direct group.

  Example:

```
_i16 Status;

Status  = sl_WlanDisconnect();
if( Status )
{
     /* error */
}
```

- **Wi-Fi Direct Profiles**

  The purpose of profile configuration is to make an automatic Wi-Fi Direct connection after reset, or after disconnection from the remote peer device. The add profile command stores the Wi-Fi Direct remote device parameters in flash as a new profile, along with profile priority. These profiles are similar to the STA mode profiles and have the same automatic connection behavior. The connection is dependent on the profile policy configuration (see the connection policy section). If the Auto policy is on, a Wi-Fi Direct discovery is performed, and if one or more of the found remote devices matches one of the profiles, a negotiation phase is started according to the negotiation initiator policy, method, and intent selected. The chosen profile is the one with the highest-priority profile.

NOTE: If a manual connection is sent during a profile connection, the profile connection is stopped, and the manual connection is started.

Example:

```
_u8 val = 1;
_u8 policyVal;
_i16 Role, Status;
_u8  my_p2p_device[33];
_u8  remote_p2p_device[33];
_u8 bssidEmpty[6] = {0,0,0,0,0,0};
SlWlanSecParams_t SecParams;

Role = sl_Start(NULL, NULL, NULL);
if( Role !=  ROLE_P2P)
{
   /* Set P2P as active mode */
   Status = sl_WlanSetMode(ROLE_P2P);
}

/* Set Wi-Fi Direct client dhcp enable (assuming remote GO running DHCP server) */
```

```
Status = sl_NetCfgSet(SL_NETCFG_IPV4_STA_ADDR_MODE, SL_NETCFG_ADDR_DHCP,0,0);
if( Status )
{
    /* error */
}
/* Set Device Name */
strcpy(my_p2p_device,"sl_p2p_device");

Status = sl_NetAppSet (SL_NETAPP_DEVICE_ID, SL_NETAPP_DEVICE_URN, strlen(my_p2p_device), (_u8 *)
my_p2p_device);
if( Status )
{
    /* error */
}
/* set connection policy Auto-Connect and Fast*/
Status = sl_WlanPolicySet(SL_WLAN_POLICY_CONNECTION, SL_WLAN_CONNECTION_POLICY
(1/*Auto*/,1/*Fast*/, 0/*OpenAP*/,0/*AnyP2P*/,0/*auto provisioning*/), NULL, 0 );

/* set P2P Policy - intent 0, random backoff */
Status = sl_WlanPolicySet( SL_WLAN_POLICY_P2P, SL_WLAN_P2P_POLICY(SL_WLAN_P2P_ROLE_CLIENT/*Intent
0 - Client*/,
    SL_WLAN_P2P_NEG_INITIATOR_RAND_BACKOFF/*Negotiation initiator – random backoff*/),NULL,0);
SecParams.Type = SL_WLAN_SEC_TYPE_P2P_PBC;
SecParams.Key = "";
SecParams.KeyLen = 0;
strcpy(remote_p2p_device,"Remote_GO_Device_XX");
Status = sl_WlanProfileAdd(remote_p2p_device, strlen(remote_p2p_device),bssidEmpty,&SecParams
,NULL ,7,0);
if( Status )
{
    /* error */
}
//restart the device
Status = sl_Stop(100);
if( Status )
{
    /* error */
}

Role = sl_Start(NULL, NULL, NULL);
```

3.5.5 Events and Errors

The host can receive indication of specific states through events or errors. Asynchronous events can be sent to the host at any given time with indication of the specific state and specific data for each event. To listen to these events and determine the needed information, a handler should be implemented in the user application, and registered under the user.h file. The following events may be received:

- SL_WLAN_EVENT_P2P_CONNECT

  Indicates that a Wi-Fi Direct connection was successfully completed. The device is Wi-Fi Direct CLIENT and contains the remote device parameters:

  – SSID
  – SSID length
  – BSSID
  – Go device name
  – Go device name length

- SL_WLAN_EVENT_P2P_DISCONNECT

  Indicates that Wi-Fi Direct disconnect is successfully completed. The device is Wi-Fi Direct CLIENT and contains the remote device parameters:

  – SSID

  - SSID length
  - BSSID
  - Go device name
  - Go device name length
- SL_WLAN_EVENT_P2P_CLIENT_ADDED

  Indicates that Wi-Fi Direct connection was successfully completed. The device is Wi-Fi Direct GO and contains the remote device parameters:
  - Client MAC address
  - Client device name
  - Client device name length
  - Own device name
  - Own device name length
- SL_WLAN_EVENT_P2P_CLIENT_REMOVED

  Indicates that a Wi-Fi Direct client was disconnected successfully. The device is P2P GO and contains the remote device parameters:
  - Client MAC address
  - Client device name
  - Client device name length
  - Own device name
  - Own device name length
- SL_WLAN_EVENT_P2P_DEVFOUND

  Indicates that a Wi-Fi Direct device was found during the scan and it contains the remote device parameters:
  - Device name
  - Device name length
  - Device MAC address
  - WPS Method
- SL_WLAN_EVENT_P2P_REQUEST

  Indicates that a negotiation request was received from a Wi-Fi Direct remote device and it contains the remote device parameters:
  - Device name
  - Device name length
  - Device MAC address
  - WPS Method
- SL_WLAN_EVENT_P2P_CONNECTFAIL

  This event is sent if the connection failed with the failure reason.

Errors are indicated by the return value of the API. Each error code is unique. Table 3-7 lists common errors that require user action (a complete list of possible errors is under the file error.h in the host driver).

Table 3-7. Common Errors

Error	Value	Comments
SL_ERROR_ROLE_P2P_ERR	-4109	Initialization failure in Wi-Fi Direct mode
SL_ERROR_NET_APP_P2P_ROLE_IS_NOT_CONFIGURED	-6210	Wi-Fi Direct mode is not configured yet, and should be CL or GO to execute the command.
SL_ERROR_WLAN_INVALID_ROLE	-2050	Action applied does not match the current mode.

Table 3-7. Common Errors (continued)

Error	Value	Comments
SL_ERROR_WLAN_KEY_ERROR	-2049	One of the security parameters or SSID supplied is wrong (invalid length or not supported).
SL_ERROR_WLAN_INVALID_SECURITY_TYPE	-2054	
SL_ERROR_WLAN_PASSPHRASE_TOO_LONG	-2055	
SL_ERROR_WLAN_PASSWORD_ERROR	-2058	
SL_ERROR_WLAN_SSID_LEN_ERROR	-2060	
SL_ERROR_WLAN_PREFERRED_NETWORK_LIST_FULL	-2062	No free profile
SL_ERROR_WLAN_INVALID_POLICY_TYPE	-2066	Invalid policy type. Value is not supported.
SL_ERROR_WLAN_WIFI_ALREADY_DISCONNECTED	-2071	Applying disconnect command when disconnected
SL_ERROR_WLAN_GET_NETWORK_LIST_EAGAIN	-2073	Scan was not enabled, one-shot scan is immediately triggered, and user should fetch the scan results again.
SL_ERROR_WLAN_GET_PROFILE_INVALID_INDEX	-2074	Profile index is too high or does not exist

3.5.6 Limitations

- Service discovery is not supported.
- GO-NOA is not supported.
- No provisioning support for Wi-Fi Direct mode
- Autonomous group is not supported.
- P2P Group Owner mode supports single peer (client) connected.
- Connection search is infinite, meaning if the remote device is not found the device keeps searching for it.

3.6 WLAN Security

The SimpleLink Wi-Fi device supports a secured connection to the AP. A secured connection can be used when establishing the connection manually or by profiles, and depends on the settings of the AP.

3.6.1 Personal Security

The SimpleLink Wi-Fi device supports all Wi-Fi security types, commonly known as AES, TKIP, and WEP. The personal security type and personal security key are set both in manual connection API or profiles connection API. Table 3-8 lists the supported security types.

Table 3-8. Supported Personal Security Types

Value	Description	Password Length	Supported Mode
SL_WLAN_SEC_TYPE_OPEN	No security		STA, AP
SL_WLAN_SEC_TYPE_WEP	WEP open security	5 or 10 characters in HEX format 13 or 26 characters in ASCII format	STA, AP
SL_WLAN_SEC_TYPE_WEP_SHARED	WEP shared security	5 or 10 characters in HEX format 13 or 26 characters in ASCII format	STA

Table 3-8. Supported Personal Security Types (continued)

Value	Description	Password Length	Supported Mode
SL_WLAN_SEC_TYPE_WPA_WPA2	WPA \ PSK and WPA2 \ PSK security types, or a mixed mode of WPA \ WPA2 PSK security type (TKIP, AES, mixed mode)	8 to 63 characters	STA, AP
SL_WLAN_SEC_TYPE_WPS_PBC	WPS push-button security (for more information refer to the WPS section)		STA
SL_WLAN_SEC_TYPE_WPS_PIN	WPS pin code security (for more information refer to the WPS section)		STA
SL_WLAN_SEC_TYPE_WPA_ENT	Enterprise security (for more information refer to the enterprise security section)		STA
SL_WLAN_SEC_TYPE_P2P_PBC	Relevant for Wi-Fi Direct mode, push button security (for more information refer to the Wi-Fi Direct section)		Wi-Fi Direct
SL_WLAN_SEC_TYPE_P2P_PIN_KEYPAD	Relevant for Wi-Fi Direct mode, pin code keypad security (for more information refer to the Wi-Fi Direct section)		Wi-Fi Direct
SL_WLAN_SEC_TYPE_P2P_PIN_DISPLAY	Relevant for Wi-Fi Direct mode, pin code display security (for more information refer to the Wi-Fi Direct section)		Wi-Fi Direct

An example of adding a WPA2 secured profile:

```
SlWlanSecParams_t SecParams;
_i16 Index;

SecParams.Type = SL_WLAN_SEC_TYPE_WPA_WPA2;
SecParams.Key = SEC_SSID_KEY;
SecParams.KeyLen = strlen(SEC_SSID_KEY);

Index  = sl_WlanProfileAdd((_i8*)SEC_SSID_NAME, strlen(SEC_SSID_NAME), 0, &secParams, 0, 7, 0);
```

3.6.2 Enterprise Security

The SimpleLink Wi-Fi device supports Wi-Fi enterprise connection according to 802.1x authentication process. Enterprise connection requires an authentication of the STA by the radius server behind the AP. Enterprise connection can be invoked from manual connection or a profile. Only one enterprise profile is supported. The following authentication methods are supported:

- EAP-TLS
- EAP-TTLS with MSCHAP
- EAP-TTLS with TLS
- EAP-TTLS with PSK
- EAP-PEAP0 with TLS
- EAP-PEAP0 with MSCHAP
- EAP-PEAP0 with PSK
- EAP-PEAP1 with TLS
- EAP-PEAP1 with PSK
- EAP-FAST AUTH PROVISIONING
- EAP-FAST UNAUTH PROVISIONING
- EAP-FAST NO PROVISIONING)

When the station has been authenticated, the AP and the station negotiate with the WPA/WPA2 security. The enterprise connection can require up to three files to complete the process (to authenticate the radius server and client according to the device and server authentication settings).

- **Client Authentication**

  If the server requires client authentication, the following files are required:

  - Private Key – Station (client) RSA private key file in PEM format
  - Client Certificate – Certificate of the client, given by the authenticating network (has the public key matches to the private key) in PEM format

- **Server Authentication**

  The SimpleLink Wi-Fi device requires server authentication by default and the following file is required:

  Server Root CA file – This file must be in PEM format. The demand for server authentication can be canceled through the WLAN setting. Canceling this authentication is valid for a single manual connection only.

  Example:

```
_i16 Status;
_u8 param;

_u8 param = 1; /* 1 means disable the server authentication */
Status  =
sl_WlanSet(SL_WLAN_CFG_GENERAL_PARAM_ID,SL_WLAN_GENERAL_PARAM_DISABLE_ENT_SERVER_AUTH,1,&param);
if( Status )
{
    /* error */
}
```

Those files must be programmed with the following names:

- Root CA – sys/cert/ca.der
- Client certificate – sys/cert/client.der
- Private key – sys/cert/private.key

Manual enterprise connection and preferred network enterprise connection both include the same security information needed to complete enterprise connection.

The following information is required according to the server demands:

- User – Enterprise identity name. Maximum length is 64 bytes.
- Anonymous user – Anonymous EAP identity. Maximum length is 64 bytes.
- EAP method – defines the EAP methods.

Configure to one of the following values according to the target authentication method:

- SL_WLAN_ENT_EAP_METHOD_TLS
- SL_WLAN_ENT_EAP_METHOD_TTLS_TLS
- SL_WLAN_ENT_EAP_METHOD_TTLS_MSCHAPv2
- SL_WLAN_ENT_EAP_METHOD_TTLS_PSK
- SL_WLAN_ENT_EAP_METHOD_PEAP0_TLS
- SL_WLAN_ENT_EAP_METHOD_PEAP0_MSCHAPv2
- SL_WLAN_ENT_EAP_METHOD_PEAP0_PSK
- SL_WLAN_ENT_EAP_METHOD_PEAP1_TLS
- SL_WLAN_ENT_EAP_METHOD_PEAP1_PSK
- SL_WLAN_ENT_EAP_METHOD_FAST_AUTH_PROVISIONING
- SL_WLAN_ENT_EAP_METHOD_FAST_UNAUTH_PROVISIONING
- SL_WLAN_ENT_EAP_METHOD_FAST_NO_PROVISIONING

The SimpleLink Wi-Fi supports only one enterprise profile and requires using the above-specified file names.

An example of manual connection to an enterprise network:

```
SlWlanSecParams_t SecParams;
SlWlanSecParamsExt_t  SecExtParams;
_i16 Status;

SecParams.Type = SL_WLAN_SEC_TYPE_WPA_ENT;
```

```
SecParams.Key = KEY;
SecParams.KeyLen = strlen(KEY);

SecExtParams.User = IDENTITY;
SecExtParams.UserLen = strlen(IDENTITY);
SecExtParams.AnonUser = ANONYMOUS;
SecExtParams.AnonUserLen = strlen(ANONYMOUS);
SecExtParams.EapMethod = SL_WLAN_ENT_EAP_METHOD_PEAP0_MSCHAPv2;

Status  = sl_WlanConnect((_i8*)SSID,strlen(SSID),0,&SecParams ,&SecExtParams);
if( Status )
{
    /* error */
}
```

3.6.3 WPS

The SimpleLink Wi-Fi device provides users the ability to create a secure connection by using Wi-Fi Protected Setup (WPS). WPS is one of the provisioning ways to connect the device to the network (for more information on provisioning, see *Chapter 14*). WPS allows an easy and secure method to provision devices without knowing the network name and without typing long passwords. The standard defines two mandatory methods for WPS-enabled APs. The SimpleLink device support both methods:

- Push-Button Connect (PBC) – Push the physical WPS button in the AP, or if the button is unavailable start the WPS process using the GUI of the AP. The AP enters the WPS provisioning process for 2 minutes. During this period, the SimpleLink device also enters the provisioning process by calling the sl_WlanConnect API with WPS parameters. If the connection successfully completes, a profile with the network name and security parameters is automatically added.
- Personal Identification Number (PIN) – Enter the PIN code generated by the host using the GUI of the AP. The AP enters the WPS provisioning process for 2 minutes. During this period, the SimpleLink device also enters the WPS provisioning process by calling the sl_WlanConnect API with WPS parameters. If the connection successfully completes, a profile with the network name and security parameters is automatically added.

When the WPS process successfully completes, a connection with the AP is established in the correct security setting according to the configuration of the AP (WPA/WPA2). The connection parameters are saved as a profile. According to the connection policy, allow a reconnection after a reset.

An example of initiating WPS with the PBC method:

```
_i16 Status;
SlWlanSecParams_t SecParams;

SecParams.Type = SL_WLAN_SEC_TYPE_WPS_PBC;
SecParams.KeyLen = 0;
SecParams.Key = "";
Status  = sl_WlanConnect("WPS_AP",strlen("WPS_AP"),NULL,&SecParams ,NULL);
if( Status )
{
    /* error */
}
```

An example of initiating WPS with the PIN Code:

```
_i16 Status;
SlWlanSecParams_t SecParams;

SecParams.Type = SL_WLAN_SEC_TYPE_WPS_PIN;
SecParams.KeyLen = strlen("11361435");
SecParams.Key = "11361435":
Status  = sl_WlanConnect("WPS_AP",strlen("WPS_AP"),NULL, &SecParams ,NULL);
if( Status )
{
    /* error */
}
```

3.7 Scan

3.7.1 General Description

The SimpleLink device can be enabled to perform scans and discover remote devices. The device returns up to 30 scan results. The device performs three types of scan:

- Connection scan – This scan is performed when the device tries to connect to an AP by issuing a manual connection command, or using stored profiles with the Auto connection policy enabled. The scan is an active scan (sends broadcast probe requests).
- Scan policy – Setting the scan policy triggers an immediate active scan (with no connection purpose), and the scan is performed on the enabled channels with a desired interval between scan cycles.
- One-shot scan – This single scan is performed on the enabled channels.

All of the previously mentioned scan types update the scan result and are supported in STA, AP, and P2P modes. This section describes the scan policy and one-shot scan. The connection scan is not a user task, it is activated internally when the connection attempt is performed.

3.7.2 Configuration (AP/STA)

- **Start Scan Policy**

  To enable or disable the scan policy, sl_WlanPolicySet should be called with enable or disable parameter and a desired scan interval. The interval value is in seconds.

  An example of setting a scan policy with a hidden SSID scan and an interval of 20 seconds:

```
_u32 intervalInSeconds = 20;
_i16 Status;

Status  = sl_WlanPolicySet(SL_WLAN_POLICY_SCAN, SL_WLAN_SCAN_POLICY(1,1),
(_u8*)&intervalInSeconds,sizeof(intervalInSeconds));
if( Status )
{
    /* error */
}
```

- **Setting Scan Parameters**

  The SimpleLink device lets users set the scan parameters. Two parameters must be configured before activating the scan policy:

  – RSSI threshold – Set the minimum RSSI threshold. Results with RSSI below this value are not presented. The default value is –95 dBm

  – Channel mask – Scan specific channels. Scans performed only on the desired channels and networks that operate on this specific channel are presented. The default value is 0x1FFF (channel 1 to 13)

  An example of setting the minimum RSSI to –70 dBm and scan channels to 1, 6, and 11:

```
_i16 Status;
SlWlanScanParamCommand_t ScanParamConfig;

ScanParamConfig.RssiThershold = -70;
ScanParamConfig.ChannelsMask = 0x421; /* channels 1,6,11 */

Status  = sl_WlanSet(SL_WLAN_CFG_GENERAL_PARAM_ID, SL_WLAN_GENERAL_PARAM_OPT_SCAN_PARAMS,
sizeof(ScanParamConfig), (_u8* )& ScanParamConfig);
if( Status )
{
    /* error */
}
```

- **Getting Scan Results**

  Scan results can be retrieved after setting the scan policy. Each scan cycle updates the results (added,

updated, or removed in case of aging). Scan results can include up to 30 entries. Each entry includes the following parameters:

- SSID
- BSSID RSSI
- Security type and cipher (hidden is part of the type)
- Channel

sl_WlanGetNetworkList triggers a one-shot scan if there are no scan results in the system, or if the scan results which exist are old (aging is defined as 20 seconds if the scan policy is disabled, or twice the scan interval if the policy is enabled).

An example of getting scan results from index 0 to 29:

```
SlWlanNetworkEntry_t netEntries[30];

_i16 resultsCount = sl_WlanGetNetworkList(0,30,&netEntries[0]);
```

3.7.3 Usage

Scan can be used to find nearby networks before issuing a connection command.

3.7.4 Miscellaneous

- Scan policy configuration is persistent according to the system-persistent configuration, except for Wi-Fi Direct mode, where the scan policy cannot be persistent.
- In Wi-Fi Direct mode, setting the scan policy scans only Wi-Fi Direct devices.
- Scan runs regardless of the connection state (runs in disconnect mode as well).
- Scan results are being updated while reading them, so when trying to retrieve specific indexes, duplicates and other problems may occur.
- The scan results are not used by the system, but they can be changed by some activities (for example, the connect activity does not use existing results in the table before it starts a new scan, but the table is changed during the connection process).
- If more than one network has the same SSID, but different BSSID, each BSSID is stored in a different entry.
- When the scan policy is enabled during the connection scan, the scan policy is activated only after the connection scan is done (after a successful connection, disconnect command, connection policy change, or profile deletion), because the connection scan has a higher priority.

3.8 Calibrations

The SimpleLink device performs calibration of the Wi-Fi physical layer. The system supports three different calibration modes to optimize this process with the required use case. The default calibration mode is triggered. Setting the calibration mode can be done only by the Image Creator tool during the creation of the image.

- Normal calibration mode is used to achieve the best RF performance, or when the environment of the device is prone to changes (temperature changes).
- Triggered calibration mode is used for lowest power consumption. Calibrations are done once on the first boot, and kept for consecutive boots. Recalibration is done on TX power change, or during the restore-to-factory process. Triggered mode can issue calibrations when updating to a new service pack that includes radio changes, after any TX power level change, or if the calibration file in the serial flash is corrupted.
- One-time calibration mode is similar to Triggered mode, but recalibration is never done under any circumstances. One-time is used when the system power source is not able to handle the peak calibration current. In this mode, user actions that trigger recalibration in Triggered mode are blocked.

NOTE: For low power applications, TI recommends choosing Triggered mode over One-Time calibration mode, unless current peak limit is an absolute constraint. Triggered mode does not issue calibrations unless absolutely necessary, or manually triggered.

Calibration failure:

When the device fails to calibrate, the device INIT complete fails and the INIT complete async event has the error: SL_ERROR_CALIB_FAIL.

- For a calibration error of Normal or Triggered calibration, power/hibernate cycle invokes recalibration.
- For One-Time calibration mode, the calibration is made once on the first power/hibernate cycle after the device programming; the user should verify that on the first power/hibernate cycle of the network subsystem, the INIT-complete succeeded. During a calibration failure the device should be reprogrammed.

Table 3-9 describes the differences between these modes.

Table 3-9. Calibration Modes

	First Time INIT	Exit from Reset	Exit from Hibernate	TX Power Change	Calibration Assessment	Restore to Factory Defaults and Image
Normal	Calibrate	Calibrate	No calibration	Calibrate on next power/hibernate cycle. Until the next power cycle, the power change is ignored.	Calibrate if needed (subset calibration, no peak current)	Calibration data is deleted. Calibrate on next power/hibernate cycle.
Triggered	Calibrate	No calibration	No calibration	Calibrate on next power/hibernate cycle. Until the next power cycle, the power change is ignored.	No runtime calibration	Calibration data is deleted. Calibrate on next power/hibernate cycle.
One Time	Calibrate	No calibration. Corrupted or missing data leads to INIT failure (lock state) but no re-calibration.	No calibration. Corrupted or missing data leads to INIT failure (lock state) but no re-calibration.	Invalid operation. In this mode, setting the TX power is allowed only by the Image Creator tool.	No runtime calibration	Calibration data is kept – no re-calibration

Network Addresses

Topic **Page**

4.1 Introduction

The SimpleLink Wi-Fi device has a built-in integrated network stack that offloads network activities from the host MCU, and decreases its code size and memory consumption. The network stack supports IPv4, IPv6, TCP, UDP, SSL, TLS, and a suite of network applications that are required by IoT and internet-enabled devices. This chapter provides the basic information and feature list of this network stack.

The host requires integrating only a small software driver, which provides a simple and slim API set for the networking activities. The traffic APIs performed by the socket layer adhere to the Linux variant of the Berkeley Sockets (BSD). *Chapter 5* describes this layer in more detail.

The SimpleLink device implements a dual network stack, which allows access to IPv4 and IPv6 networks simultaneously. IPv4 is enabled by default in all Wi-Fi modes: STA, AP, and Wi-Fi Direct. IPv6 is supported only in STA mode, disabled by default, and can be enabled if needed.

4.2 Key Features

Table 4-1 describes the major features of this network stack.

Table 4-1. Key Features

Key Features	Description
IP protocols	IPv4, IPv6
IP addressing	LLA, DHCPv4, DHCPv6, static, stateless
Cross layer	DAD, NDP, ARP, ICMPv4, ICMPv6
Application	DNS server, DNS client, DHCP server

4.3 Addressing

The SimpleLink Wi-Fi device supports multiple IP address-acquiring methods. For Wi-Fi station and Wi-Fi Direct client modes, IP acquiring processes start after successful Wi-Fi connection. For AP and Wi-Fi Direct GO, the IP address is static and predefined. Changing addressing configuration requires device reset, as shown in Table 4-2.

Table 4-2. Addressing

	Wi-Fi Station	Wi-Fi AP	Wi-Fi Direct
IPv4	Always enabled	Static	Client – like station Group Owner – Like AP
	One IP address: • DHCP • LLA • Static		
IPv6	Disabled (default)	Not supported	Not supported
	Up to two IP addresses: • Local (mandatory): o Stateless – Link-Local Address (FE80::/64) o Statefull (DHCPv6) o Static		
	• Global (optionally): o Stateless o Statefull (DHCPv6) o Static		

NOTE:

- Ipv4 is always enabled. IPv6 can also be enabled, but it cannot work without IPv4.
- During a DHCP IPv4 failure, the SimpleLink device acquires the IPv4 address by using Local-Link Address protocol (LLA)
- For power-sensitive systems, TI recommends disabling IPv6.
- All addressing configurations are persistent and available through the host interface.

4.3.1 IPv4 Addresses

The SimpleLink device allows the following IPv4 acquisition methods:

- **Stateful (DHCPv4) with Stateless (LLA) Fallback –** In this mode, the device starts by trying to acquire the IPv4 address from a DHCP server. LLA is acquired only after a DHCPv4 client time-out expires. The default time-out is 25 seconds, and this time can be configured.

NOTE:

- LLA allows communicating with devices on the local network only.
- The LLA IP address range is from 169.254.1.0 to 169.254.254.255. The default gateway and DNS address are not configured.

Example:

```
_i16 Status;

Status = sl_NetCfgSet(SL_NETCFG_IPV4_STA_ADDR_MODE,SL_NETCFG_ADDR_DHCP_LLA,0,0);
if( Status )
{
    // error
}
```

- **Stateful (DHCPv4) Only –** In this mode, the device tries to acquire the IPv4 address from a DHCP server with no time-out.

Example:

```
_i16 Status;

Status = sl_NetCfgSet(SL_NETCFG_IPV4_STA_ADDR_MODE,SL_NETCFG_ADDR_DHCP,0,0);
if( Status )
{
    // error
}
```

- **Static –** In this mode the IPv4 address of the device is preconfigured.

Example:

```
_i16 Status;
SlNetCfgIpV4Args_t ipV4;

ipV4.Ip          = (_u32)SL_IPV4_VAL(10,1,1,201);      // _u32 IP address
ipV4.IpMask      = (_u32)SL_IPV4_VAL(255,255,255,0);   // _u32 Subnet mask for this STA/P2P
ipV4.IpGateway   = (_u32)SL_IPV4_VAL(10,1,1,1);        // _u32 Default gateway address
ipV4.IpDnsServer = (_u32)SL_IPV4_VAL(8,8,8,8);         // _u32 DNS server address

Status =
sl_NetCfgSet(SL_NETCFG_IPV4_STA_ADDR_MODE,SL_NETCFG_ADDR_STATIC,sizeof(ipV4),(_u8*)&ipV4);
if( Status )
{
    // error
}
```

4.3.2 IPv6 Addresses

To enable IPv6, the host application must configure an IPv6 LLA. Configuration of an IPv6 global address is optional.

Example:

```
_u32 IfBitmap = 0;
_i16 Status;

IfBitmap = SL_NETCFG_IF_IPV6_STA_LOCAL | SL_NETCFG_IF_IPV6_STA_GLOBAL;
Status = sl_NetCfgSet(SL_NETCFG_IF,SL_NETCFG_IF_STATE,sizeof(IfBitmap),&IfBitmap);
if( Status )
{
    // error
}
```

4.3.2.1 Local Link

IPv6 local link must consist of the following prefix: Fe80::/64. The following IPv6 link-local acquisition methods are allowed:

- **Stateless Auto Configuration –** The least significant 64 bits are filled with the device MAC address in EUI-64 format. The Duplicate Address Detection (DAD) algorithm is used to verify that the address is unique on the local link. When DAD failure occurs, this procedure continues with random numbers on the least significant 64 bits.

  Example:

```
_i16 Status;

Status = sl_NetCfgSet(SL_NETCFG_IPV6_ADDR_LOCAL,SL_NETCFG_ADDR_STATELESS,0,0);
if( Status )
{
    // error
}
```

- **Stateful (DHCPv6) –** IPv6 LLA is acquired from the DHCPv6 server. The DAD algorithm is used to verify that the address is unique on the local link. When DAD failure occurs, stateless auto-configuration is used instead.

  Example:

```
_i16 Status;

Status = sl_NetCfgSet(SL_NETCFG_IPV6_ADDR_LOCAL,SL_NETCFG_ADDR_STATEFUL,0,0);
if( Status )
{
    // error
}
```

- **Static –** In this mode the IPv6 address of the device is preconfigured. The DAD algorithm is used to verify that the address is unique on the local link. When DAD failure occurs the address is not valid, and notification is sent to the host.

  Example:

```
_i16 Status;
SlNetCfgIpV6Args_t ipV6;

memset(&ipV6, 0, sizeof(ipV6));
ipV6.Ip[0] = 0xfe800000;
ipV6.Ip[1] = 0x00000000;
ipV6.Ip[2] = 0x00004040;
ipV6.Ip[3] = 0x0000ce65;

Status = sl_NetCfgSet(SL_NETCFG_IPV6_ADDR_LOCAL,SL_NETCFG_ADDR_STATIC,sizeof(ipV6),(_u8*)&ipV6);
if( Status )
{
    // error
```

```
}
```

4.3.2.2 Link-Global

The SimpleLink device allows the following IPv6 global address which must consist of the prefix 2000::/3. The following acquisition methods are allowed:

- **Stateless:** The most significant 64 bits acquired from the RA messages (router advertisement message that is sent periodically by an IPv6 router). The least significant 64 bits are filled with a MAC address in EUI-64 format. The DAD algorithm is used to verify that the address is unique on the link. When DAD failure occurs, the global address is invalid and the device cannot communicate outside the local network.
- **Stateful (DHCPv6):** The IPv6 global address is learned from the DHCPv6 server. The DAD algorithm is used to verify that the address is unique on the link. When DAD failure occurs, the global address is invalid and the device cannot communicate outside the local network.
- **Static:** The user configures the IPv6 global address and single IPv6 DNS server address. The DAD algorithm is used to verify that the address is unique on the link. When DAD failure occurs, the global address is invalid and the device cannot communicate outside the local network.

4.3.3 DNS Addresses

The SimpleLink device supports IPv4 and IPv6 protocols. Each interface can support up to two DNS servers:

- In DHCP mode, the SimpleLink device can receive up to two DNS server addresses. The host application can temporarily overwrite the second address. However, this address is effective until the next IP acquire.
- In static address mode, the host application can configure two DNS server addresses. The first address is persistent, and the second address is effective until the next IP acquire

One DNS request is supported at a time, the default time-out is 18 seconds per DNS server, and it can be configured by the host.

4.4 DHCPv4 client

4.4.1 Modes

The SimpleLink device supports some enhanced DHCP modes for IP acquisition after connection to a Wi-Fi network:

- **Full Renew Process** – If the lease time of the acquired IP address has not expired, the device starts by trying to renew this address. Failure to renew the last address invokes a full DHCP process. This mode is enabled by default and occurs only if the lease time is greater than 1 hour (otherwise the full DHCP process occurs).

  Example:

```
_i16 Status;

Status = sl_NetCfgSet(SL_NETCFG_IPV4_STA_ADDR_MODE,SL_NETCFG_ADDR_FAST_RENEW_MODE_WAIT_ACK,0,0);
if( Status )
{
    // error
}
```

- **Opportunistic Renew Process** – This mode is similar to the full renew process mode but the host is notified on IP acquired immedicably and the traffic enabled even before the ACK has been received from the DHCP server. In case of renew failure, an IP loss event is triggered and the traffic is blocked until a new IP address is acquired by a full DHCP process. This mode allows the host to communicate with devices faster than other DHCP modes.

  Example:

```
_i16 Status;
```

```
Status =
sl_NetCfgSet(SL_NETCFG_IPV4_STA_ADDR_MODE,SL_NETCFG_ADDR_FAST_RENEW_MODE_NO_WAIT_ACK,0,0);
if( Status )
{
    // error
}
```

- **Full DHCP Process** – The entire DHCP sequence is processed with every connection to the network. Example:

```
_i16 Status;

Status = sl_NetCfgSet(SL_NETCFG_IPV4_STA_ADDR_MODE, SL_NETCFG_ADDR_DISABLE_FAST_RENEW,0,0);
if( Status )
{
    // error
}
```

Figure 4-1 shows the differences between the modes.

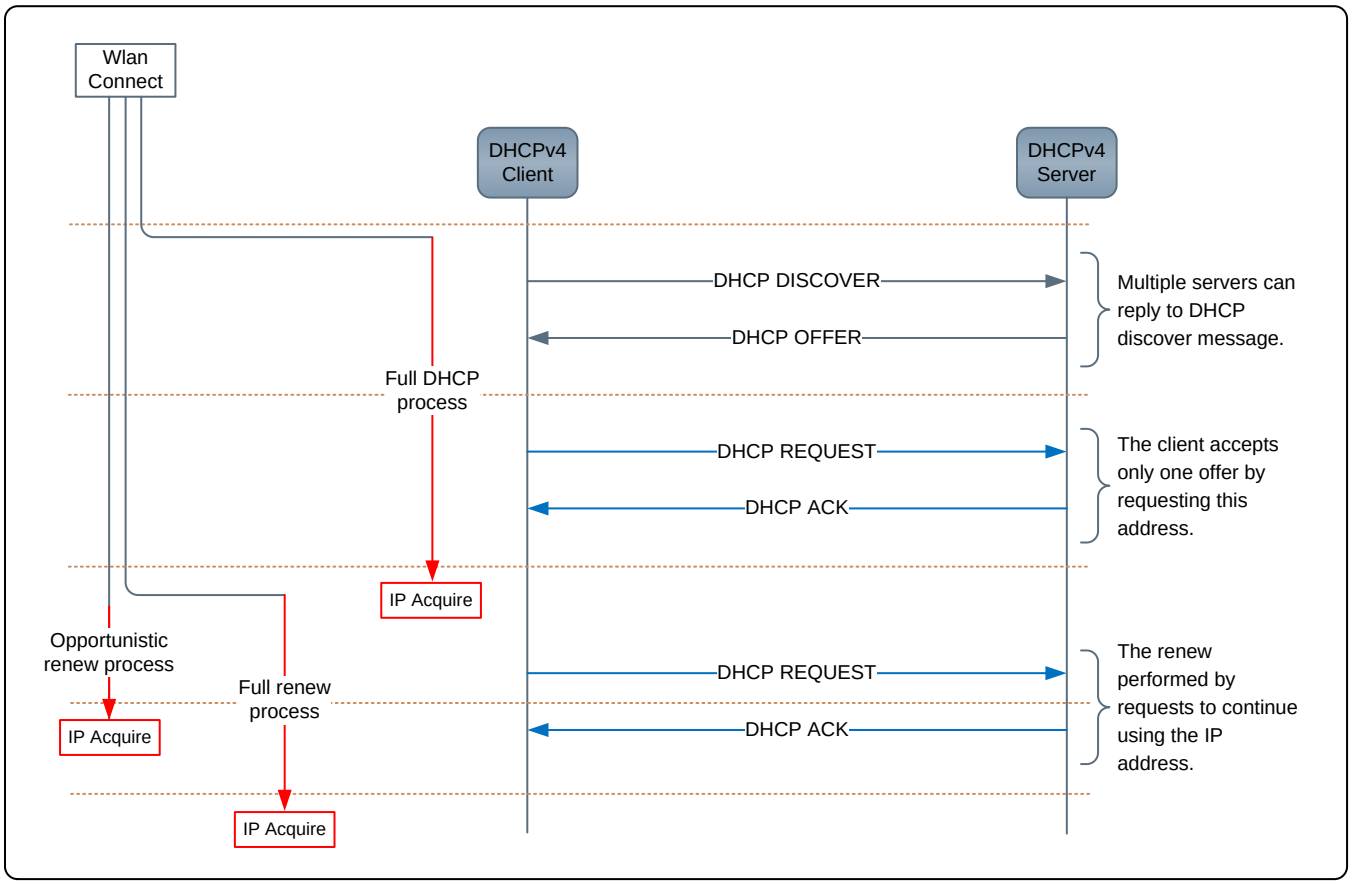

Figure 4-1. DHCPv4 IP Acquisition Modes

4.4.2 Address Release

By default the SimpleLink device does not release the DHCP address when Wi-Fi disconnect is requested. However, in some use cases the release is required, even if the lease time is short due to limited address range at the DHCP server. The SimpleLink device enables a special mode that releases the IP address on requested WiFi disconnect commands from the host application. This mode is not enabled by default.

Example:

```
_i16 Status;

Status = sl_NetCfgSet(SL_NETCFG_IPV4_STA_ADDR_MODE,SL_NETCFG_IF_ENABLE_DHCP_RELEASE,0,0);
if( Status )
{
    // error
}
```

4.5 DHCPv4 Server

The SimpleLink device includes an internal DHCPv4 server which is supported in AP mode and Wi-Fi Direct mode (group owner). The DHCPv4 server allocates IP addresses for connected stations. The range and lease time of the IP address can be configured by the host driver APIs. The AP/GO IP and DHCP server addresses range should have the same class C subnet. Station-leased IP address information is not persistent, and all addresses are considered as available for lease after the SimpleLink device reset. Table 4-3 shows the DHCP server defaults.

Table 4-3. DHCP Server Defaults

DHCP Server	Default
Mode	Enabled
Gateway IP address	10.123.45.1
IP range	10.123.45.2 to 10.123.45.254
Lease time	86,400 seconds
32 Address	Maximum lease addresses

4.5.1 Enable and Disable the DHCP Server

The SimpleLink device lets users enable or disable the DHCP server. In AP mode, the DHCP server is enabled by default. This configuration is persistent according to the system-persistent configuration.

An example of enabling the DHCP server:

```
_i16 Status;

Status  = sl_NetAppStart(SL_NETAPP_DHCP_SERVER_ID); //enable the DHCP server
if( Status )
{
    // error
}
```

An example of disabling the DHCP server:

```
_i16 Status;

Status  = sl_NetAppStop(SL_NETAPP_DHCP_SERVER_ID); //disable the DHCP server
if( Status )
{
    // error
}
```

4.5.2 Set DHCP Server Parameters

The SimpleLink device lets users set the DHCP server parameters. The following parameters can be set:

- **Address range** – First and last IP address for addressed allocation. The following macro can be used: SL_IPV4_VAL(192,168,1,10)
- **Lease time** – Lease time (in seconds) of the IP address.

The range of the DHCP server addresses must be in the subnet of the AP IP address. This configuration is persistent. The configuration should be performed when the DHCP server is down.

Example:

```
_i16 Status;
SlNetAppDhcpServerBasicOpt_t dhcpParams;
_u8 outLen = sizeof(SlNetAppDhcpServerBasicOpt_t);

dhcpParams.lease_time      = 4096;                        // lease time (in seconds) of the IP
Address
dhcpParams.ipv4_addr_start =  SL_IPV4_VAL(192,168,1,10);   // first IP Address for allocation
dhcpParams.ipv4_addr_last  =  SL_IPV4_VAL(192,168,1,16);   // last IP Address for allocation.

Status  = sl_NetAppStop(SL_NETAPP_DHCP_SERVER_ID);                 // Stop DHCP server before
settings
if( Status )
{
    // error
}
Status  = sl_NetAppSet(SL_NETAPP_DHCP_SERVER_ID, SL_NETAPP_DHCP_SRV_BASIC_OPT, outLen, (_u8*
)&dhcpParams);
if( Status )
{
    // error
}
Status  = sl_NetAppStart(SL_NETAPP_DHCP_SERVER_ID);                // Start DHCP server with new
settings

if( Status )
{
    // error
}
```

4.6 DNS Server

The SimpleLink device has an internal DNS server which runs in AP mode and Wi-Fi Direct mode (GO). The DNS server is enabled by default and can be disabled. The DNS server resolves the SimpleLink device IPv4 address. The default domain name is *mysimplelink* and it can be configured.

Example:

```
_i16 Status;
Status  = sl_NetAppStop(SL_NETAPP_DNS_SERVER_ID);   // Stop DNS server
if( Status )
{
    // error
}
```

4.7 Errors and Asynchronous Events

Table 4-4 summarizes the major asynchronous events which are part of the NetApp silo event handler (slcb_NetAppEvtHdlr).

Table 4-4. Major Asynchronous Events in NetApp Silo

Event	Description	STA Role	AP Role
SL_NETAPP_EVENT_IPV4_ACQUIRED	IPv4 interface is available for traffic. The event includes IPv4 parameters such as gateway mask and DNS server address.	After Wi-Fi connection, two options: • Immediate event: static configuration or DHCPv4 opportunistic renew configuration. • Delay between the connection and the event: DHCPv4, DHCPv4, fast renew or LLA.	Immediate

Table 4-4. Major Asynchronous Events in NetApp Silo (continued)

Event	Description	STA Role	AP Role
SL_NETAPP_EVENT_IPV6_ACQUIRED	IPv6 local address or global interface is available for traffic. The event includes IPv6 parameters such as IP address and DNS server address.	After Wi-Fi connection and DAD successfully complete	IPv6 is not supported.
SL_NETAPP_EVENT_DHCPV4_LEASED	IPv4 DHCP client acquired IPv4 address from the internal DHCP server. Event includes IPv4 address, lease time, and client MAC address.	DHCPv4 server is not supported.	DHCPv4 server must be enabled (default).
SL_NETAPP_EVENT_DHCPV4_RELEASED	Client IPv4 address released. Event includes IPv4 address, client MAC address, and reason.	DHCPv4 server is not supported.	DHCPv4 server is enabled (default).
SL_NETAPP_EVENT_IPV4_LOST	The acquired IPv4 address is no longer available.	Supported	Not supported
SL_NETAPP_EVENT_DHCP_IPV4_ACQUIRE_TIMEOUT	Acquiring the IPv4 address by DHCP is too long and not completed yet, acquiring by DHCP still continues.	After Wi-Fi connection and DHCP configuration	Not supported
SL_NETAPP_EVENT_IP_COLLISION	IPv4 address conflict, two stations connected, one station acquired IPv4 address by the SimpleLink DHCP server and the second station has static IPv4 address with the same IP address. Event includes IPv4 address and two MAC addresses.	DHCPv4 server is not supported.	DHCPv4 server is enabled (default)
SL_NETAPP_EVENT_IPV6_LOST	Global or Local acquired IPv6 address is no longer available. Event includes IPv6 address.	Supported	IPv6 is not supported

Table 4-5 summarizes the major asynchronous events that are part of the NetCfg silo event handler (slcb_DeviceGeneralEvtHdlr).

Table 4-5. Major Asynchronous Events in NetCfg Silo

Event	Description	STA Role	AP Role
SL_ERROR_STSTIC_ADDR_SUBNET_ERROR	Ipv4 static configuration. IPv4 address is not in the same subnet of the gateway.	Supported	Supported

Table 4-6 describes the major error codes that may be returned while calling sl_NetCfgSet.

Table 4-6. Major Errors While Calling sl_NetCfgSet

Event	Description	STA Role	AP Role
SL_ERROR_INCORRECT_IPV6_STATIC_LOCAL_ADDR	IPv6 local address static configuration, address prefix is not the local address prefix.	Supported	IPv6 is not supported
SL_ERROR_INCORRECT_IPV6_STATIC_GLOBAL_ADDR	IPv6 global address static configuration, address prefix is not the global address prefix.	Supported	IPv6 is not supported
SL_ERROR_IPV6_LOCAL_ADDR_SHOULD_BE_SET_FIRST	The local IPv6 address must be enabled when the global IPv6 address is enabled.	Supported	IPv6 is not supported

NOTE:

- On the SL_NETAPP_EVENT_IPV4_LOST and SL_NETAPP_EVENT_IPV6_LOST events, TI highly recommends closing the relevant sockets.
- On the SL_NETAPP_EVENT_IPV4_ACQUIRED or SL_NETAPP_EVENT_IPV4_ACQUIRED events, if the new IP is different from the previous IP, TI highly recommends closing the relevant sockets, and opening new sockets before any transmit and receive occurs.

Socket

Topic Page

5.1 Introduction

Sockets allow the communication between two or more peers in the network. The SimpleLink device complies with the BSD, which is a common IP connection interface in the industry. This chapter describes the socket layer of the SimpleLink device. The socket layer provides a set of simple API for sending and receiving data. The SimpleLink device implements a subset of the BSD API which complies with the Linux variant.

5.2 Key Features

Table 5-1 lists the key features of the socket.

Table 5-1. Key Features

Key Features	Description
Max Sockets	16 sockets including up to 6 connected secured sockets
Socket Types	SL_SOCK_STREAM (TCP)
	SL_SOCK_DGRAM (UDP)
	SL_SOCK_RAW
	SL_IPPROTO_TCP (TCP RAW socket)
	SL_IPPROTO_UDP (UDP RAW socket)
	SL_IPPROTO_RAW (IP RAW socket)
	SL_SEC_SOCKET (secure socket – SSL/TLS)
Address Families	SL_AF_INET (IPv4)
	SL_AF_INET6 (IPv6)
	SL_AF_RF (transceiver)
	SL_AF_PACKET
Connection Types	Client
	Server
Modes	Blocking
	Non-blocking
	Trigger
Dual Stack Mode	IPv6 server allows IPv4 client connections.
UDP Packet Boundary	Enable and disable (disable by default)
Select	Select on receive, accept, and connect
GetHostByName	Retrieve the IPv4/IPv6 address according to the host name.
Multicast	Up to eight Multicast sockets

5.3 Socket Types

The socket layer of the SimpleLink device supports the following socket types:

- **UDP** sockets provide users a basic transport service, with no guarantee of delivery and packet ordering. UDP also allows more than two hosts to exchange data through a multicast group.
- **TCP** sockets enable two hosts to establish a connection and exchange streams of data with a guarantee of delivery and packet ordering.
- **RAW** sockets provide users access to the underlying communication protocols with socket abstractions. RAW sockets are datagram oriented (packet boundary). The SimpleLink device allows RAW sockets to the following layers:
  - Layer 1: Physical (available only if the device is not connected to a wireless network).
  - Layer 2: Data Link (MAC)
  - Layer 3: Network
  - Layer 4: Transport

- **Secure** sockets provide users the ability to establish encrypted data transport (SSL and TLS). For more information, see *Chapter 6*.

5.4 BSD API

The SimpleLink driver provide two sets of socket API: the SimpleLink API and a BSD-compliant API. The major differences are:

- The SimpleLink APIs return informative error codes instead of using the errno method
- The BSD-compliant API is an additional socket layer which allows the user to use an errno mechanism implemented by the operating system. If this mechanism is not available in the target operating system, the host driver offers an internal errno mechanism.

Table 5-2 describes a list of BSD socket APIs and their corresponding SimpleLink API.

Table 5-2. BSD APIs

BSD	SimpleLink	Server or Client	TCP or UDP	Description
socket()	sl_Socket()	Both	Both	Creates an endpoint for communication.
bind()	sl_Bind()	Both	Both	Assigns an IP address and a port to a socket. If the socket is not bound, a port is chosen automatically.
listen()	sl_Listen()	Server	TCP	Listens for connections on a socket.
connect()	sl_Connect()	Client	Both	Initiates a connection on a socket.
accept()	sl_Accept()	Server	TCP	Accepts an incoming connection on a socket.
send(), recv()	sl_Send(), sl_Recv()	Both	Both	Writes and reads data. (On UDP, connect API which sets the default address must be called before sl_Send).
write(), read()	Not supported			
sendto(), recvfrom()	sl_SendTo(), sl_RecvFrom()	Both	UDP	Writes/reads data to/from a UDP socket.
close()	sl_Close()	Both	Both	Causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.
select()	sl_Select()	Both	Both	Select allows a program to monitor multiple sockets, waiting until one or more sockets become ready. Only a single select is supported at a time.
				SimpleLink supports: • Readfds: On data socket: data arrived. On listen socket: indicating new client connected • Writefds: only on TCP connect, must configure nonblocking socket • Exceptfds: not supported
gethostbyname()	sl_NetAppDnsGetHostByName()	None	None	This is not a socket operation. It is preliminary to a socket operation, to retrieve host IP information corresponding to a host name.
poll()	Not supported			
getsockopt()	sl_SockOpt()	Both	Both	Retrieves the current value of a particular socket option for the specified socket.
setsockopt()	sl_SetSockOpt()	Both	Both	Sets a particular socket option for the specified socket.
htons(), ntohs()	sl_Htons(), sl_Ntohs()	Both	Both	Reorders the bytes of a 16-bit unsigned value from processor order to network order.
htonl(), ntohl()	sl_Htonl(), sl_Ntohl()	Both	Both	Reorders the bytes of a 32-bit unsigned value from processor order to network order.

The following examples demonstrate the differences between these APIs:

```
/* Send using BSD API and checking errno value */
if (send(sock , pBuff, sizeof(pBuff), 0) == -1)
{
      int errsv = errno;
      printf("send() failed\n");
      if (errsv == ...) { ... }
```

```
}

/* Send using SimpleLink API and checking the return value */
Status = sl_Send(sock , pBuff, sizeof(pBuff), 0);
if ( Status < 0)
{
   printf("send() failed\n");
   if (Status == ...) { ... }
}
```

5.5 Socket Working Flow

Two main categories of sockets exist: datagram sockets (connectionless) and stream sockets (connection oriented). Datagram sockets or connectionless sockets allow for data exchange between entities without establishing a connection before any data delivery. In this category the data integrity and packet order are not ensured.

Stream sockets or connection-oriented sockets require establishing a connection between the two entities before any data exchange. While the connection is maintained, data integrity and the order are ensured. Programmers should choose between connection-oriented transport protocol and connectionless transport protocol according to the requirements of their applications. For example, VoIP applications, which are sensitive to delays, may require the connectionless transport protocols. File transfer applications may require connection-oriented transport protocol due to the guaranty of data integrity and packet ordering.

5.5.1 TCP

TCP is a connection-oriented transport protocol. The TCP client initiates the connection to a TCP server, and after establishing the connection successfully, the socket provides a bidirectional tunnel between the client and the server.

Figure 5-1 describes the general flow of TCP between a server and a client.

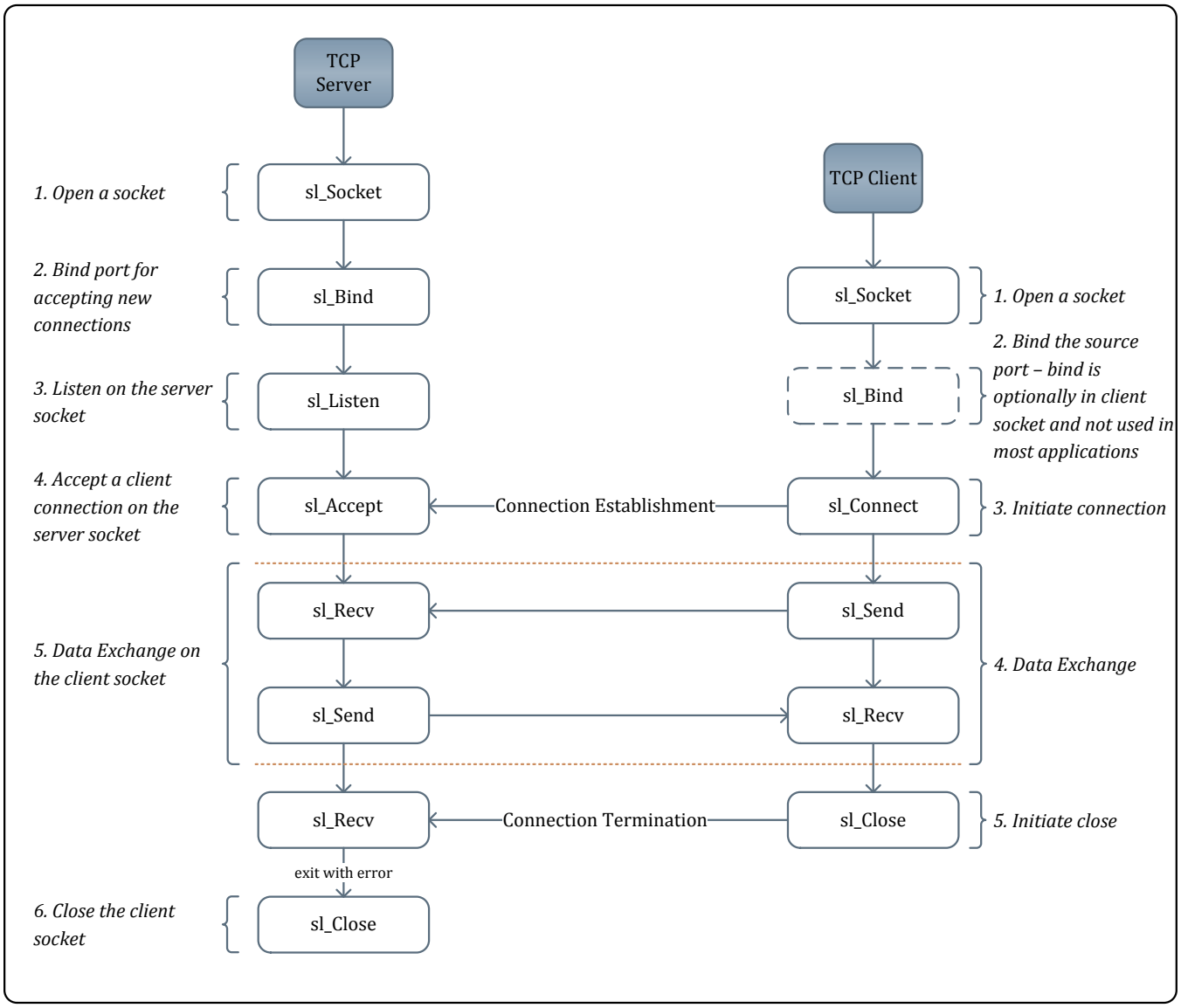

Figure 5-1. TCP Socket Flow

5.5.1.1 Client Side

1. Open the TCP socket. Use family type: SL_AF_INET for IPv4, and SL_AF_INET6 for IPv6.
2. Bind the source port. This step is optional for the client socket. If the sl_Bind API is not called, the SimpleLink device internally binds a random source port. Binding the port is performed in the same way a server socket binds a port (see the following example).
3. Initiate a connection to the server. The TCP IPv6 client can also connect to the IPv4 server. In this case, when the IPv6 socket is connecting to the IPv4 server, the IPv4 destination address is mapped to IPv6 format (for example, ::00:ffff:ipv4).
4. Send and receive the data.
5. Close the socket. By default the sl_Close API returns immediately and the close process is done internally. There are two ways to confirm that all the data was transmitted and the socket closed gracefully:
   - By default: the sl_Close API returns immediately, while the close process is done internally. The socket is closed only after all queued packets successfully transmit. If the device failed to transmit

all queued packets, the host application is notified through an asynchronous error event (SL_SOCKET_TX_FAILED_EVENT).

- The common option in BSD: use the SO_LINGER option. When a socket is set as linger, the sl_Close API does not return until all queued packets successfully transmit, or earlier if the linger configured time-out expires with an appropriate error indication.

Example:

```
_i16 Status;
_i16 Sd;
SlSockAddrIn_t  Addr;
_i8  SendBuf[] = "Hello World !!!";
_i8  RecvBuf[1460];

Addr.sin_family      = SL_AF_INET;
Addr.sin_port        = sl_Htons(5001);
Addr.sin_addr.s_addr = sl_Htonl(SL_IPV4_VAL(192,168,1,31));

Sd = sl_Socket(SL_AF_INET, SL_SOCK_STREAM, 0);
if( 0 > Sd )
{
    // error
}
Status = sl_Connect(Sd, ( SlSockAddr_t *)&Addr, sizeof(SlSockAddrIn_t));
if( Status )
{
    // error
}
Status = sl_Send(Sd, SendBuf, strlen(SendBuf), 0 );
if( strlen(SendBuf) != Status )
{
// error
}
Status = sl_Recv(Sd, RecvBuf, 1460, 0);
if( 0 > Status )
{
// error
}
Status = sl_Close(Sd);
if( Status )
{
    // error
}
```

5.5.1.2 Server Side

1. Open the TCP socket. Use family type: SL_AF_INET for IPv4, and SL_AF_INET6 for IPv6. The socket is the public socket of the server.
2. Bind the public port of the server. The host application must set a specific port for the server to allow clients to connect.
3. Listen. This stage marks the socket as a server socket. Here an additional socket is allocated for this specific server socket to reserve a socket for the next client connection (from this point the server socket is ready to accept new connections even if the host still did not call to sl_Accept).
4. Accept a client connection. This step extracts a connection request from the queue of pending connections on the server socket, and creates a new connected socket for data exchange between the server and the client side. The original public socket is not affected by this call, and an additional accept could be called on the public socket to accept additional clients. Each newly created client decreases the number of available sockets in the system by one. IPv6 server sockets bound to any interface, can accept IPv6 and IPv4 clients. When accepting IPv4 clients, the returned client IP address is IPv4 mapped to IPv6 format (for example, :00:ffff:ipv4).
5. Send and receive the data. Use the client socket descriptor to send and receive data. This step is done in the same way as in a client socket.
6. Close the data socket. To close a connection with a specific client, the close operation should be called

with the client socket. The close is performed in a similar way to closing a client socket. For more information regarding linger, see the close section of client socket.

7. Close the server socket. When there is no need to accept any new client connections, call the close API on the server socket. The client sockets are not affected by closing the public socket, and only new connections cannot be accepted. If the host application is required to close the clients and the server, TI recommends closing the client socket first.

5.5.1.3 TCP Keep Alive

The keep-alive option is relevant for TCP connection only and is enabled by default. If there were no messages between the client and the server in the time-out period, a keep-alive message is sent. This option can be disabled by calling sl_SetSockOpt with the option SL_SO_KEEPALIVE. The keep-alive time-out is also configurable using the option SL_SO_KEEPALIVETIME. The default keep-alive time-out of a new socket is 5 minutes. The value is set in seconds.

An example of disabling the keep-alive command:

```
_i16 Status;
SlSockKeepalive_t enableOption;
enableOption.KeepaliveEnabled = 0;

Status = sl_SetSockOpt(Sd,SL_SOL_SOCKET,SL_SO_KEEPALIVE, (_u8
*)&enableOption,sizeof(enableOption));
 if( Status )
{
    // error
}
```

An example of setting the keep-alive time-out:

```
_i16 Status;
_u32 TimeOut = 120;

Status = sl_SetSockOpt(Sd, SL_SOL_SOCKET, SL_SO_KEEPALIVETIME,( _u8*) &TimeOut, sizeof(TimeOut));
 if( Status )
{
    // error
}
```

Example:

```
_i16 Status;
SlSockAddrIn_t  Addr;
_i16 ClientSd;
SlSockAddrIn_t  Addr;
_i16 AddrSize = sizeof(SlSockAddrIn_t);

Addr.sin_family      = SL_AF_INET;
Addr.sin_port        = sl_Htons(6000);
Addr.sin_addr.s_addr = SL_INADDR_ANY;

Status = sl_Bind(Sd, ( SlSockAddr_t *)&Addr, sizeof(SlSockAddrIn_t));
if( Status )
{
// error
}
Status = sl_Listen(Sd, 1);
if( Status )
{
    // error
}
ClientSd = sl_Accept( Sd, ( SlSockAddr_t *)&Addr, &AddrSize);
if(0 > ClientSd)
{
// error
}
```

5.5.2 UDP

UDP is a connectionless transport protocol. It does not require establishing a connection with a peer socket, and each packet is individually managed. However, the SimpleLink device lets the host application use UDP either as a connectionless or a connection-oriented protocol. In the connection-oriented mode, received packets with a different source than the connect source are dropped. In UDP there are no client and server sides. Both sides can initiate data exchange or wait for reception of data. In most applications one side waits for data reception and one side initiates the data exchange. The side that waits for data reception is related as a server and the side that initiates the data exchange is related as a client.

Figure 5-2 shows these two methods:

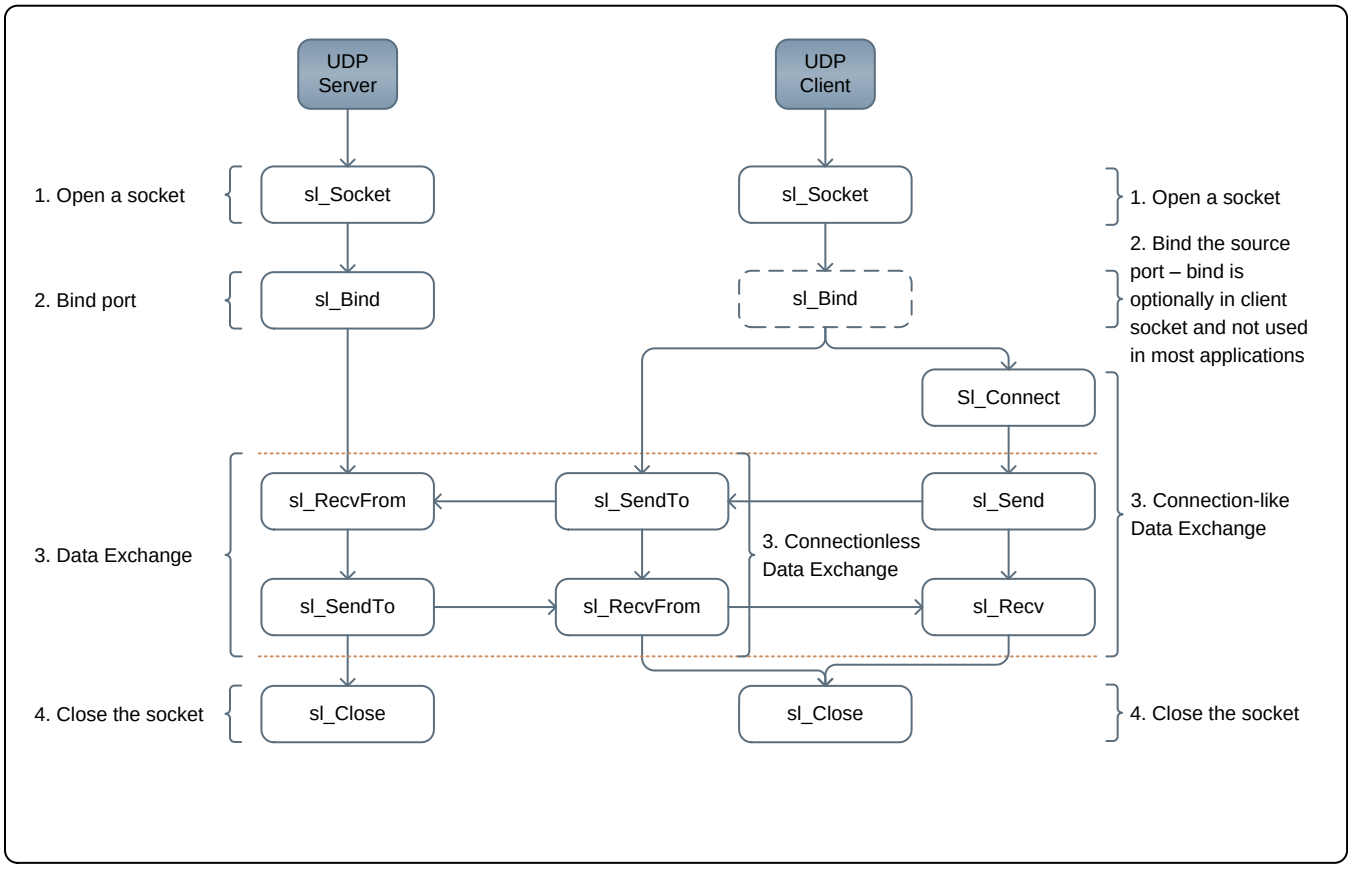

Figure 5-2. UDP Socket Flow

1. Open the UDP socket. Use family type: SL_AF_INET for IPv4, use SL_AF_INET6 for IPv6.
2. Bind the source port. This step is optional. If sl_Bind is not called, the SimpleLink device automatically binds a random source port. In practice, the server side must bind the port to define the destination port to the other side.
3. Data Exchange
   - Connectionless
     - Send Data – The host application must provide the destination address and port. To send data from the IPv6 socket to an IPv4 socket, the IPV4 destination address in sl_SendTo must be mapped to IPv6 format (for example, ::00:ffff:ipv4).
     - Receive Data – The host application must provide the source address and port. The IPv6 socket can receive data from the IPv4 socket, by mapping the source address to IPv6 format (for example, ::00:ffff:ipv4).
   - Connection-Oriented
     - Connect – Calling the API sl_Connect for UDP sockets defines the destination address. By

calling to sl_Send (in connection-oriented mode the host application calls sl_Send instead of sl_SendTo), the address of the remote peer is defined and datagrams from other addresses are dropped.

- Send Data – Send a datagram to the address that was defined during the connect process.
- Receive Data – Receive a datagram from the address that was defined during the connect process.

4. Close the socket. The close API returns immediately. LINGER has no meaning for connectionless socket.

Example:

```
_i16 Sd;
_i16 Status;
SlSockAddrIn_t  Addr;
_i8 SendBuf[] = "Hello World !!!";
_i8  RecvBuf[1460];

Sd = sl_Socket(SL_AF_INET, SL_SOCK_DGRAM, 0);
if( 0 > Sd )
{
    // error
}
Addr.sin_family      = SL_AF_INET;
Addr.sin_port        = sl_Htons(5001);
Addr.sin_addr.s_addr = SL_INADDR_ANY;

Status = sl_Bind(Sd, ( SlSockAddr_t *)&Addr, sizeof(SlSockAddrIn_t));
if( Status )
{
    // error
}
Addr.sin_family      = SL_AF_INET;
Addr.sin_port        = sl_Htons(5001);
Addr.sin_addr.s_addr = sl_Htonl(SL_IPV4_VAL(192,168,1,31));

Status = sl_SendTo(Sd, SendBuf, strlen(SendBuf), 0, (SlSockAddr_t*)&Addr,sizeof(SlSockAddr_t));
if( strlen(SendBuf) != Status )
{
    // error
}
AddrSize = sizeof(SlSockAddrIn_t);
Status = sl_RecvFrom(Sd, RecvBuf, 1460, 0, ( SlSockAddr_t *)&Addr, &AddrSize);
if( 0 > Status )
{
    // error
}
Status = sl_Close(Sd);
if( Status )
{
    // error
}
```

5.5.2.1 Multicast

IPv4 and IPv6 multicasts allow for one-to-many communication over an IP network. If a device is interested in receiving multicasts which are sent to a specific group of devices, it may join or leave the group by sending join or leave messages. The UDP socket that joined a group receives group multicast packets in addition to the regular unicast packets. Television is a good example of multicasting, where each channel is transmitted on a different multicast group. When a user changes a channel, the UDP socket leaves the multicast group and joins another multicast group.

The SimpleLink device supports IPv4 IGMPv2 and IPv6 MLDv1 protocols for joining and leaving groups. Users can support up to eight IPv4 multicast groups and up to eight IPv6 multicast groups. Two UDP sockets which join the same group decrease the available multicast group only by one.

To join or leave a group, use sl_SetSockOpt with the options listed in Table 5-3.

Table 5-3. Multicast

SL_IP_ADD_MEMBERSHIP	Join IPv4 group
SL_IP_DROP_MEMBERSHIP	Leave IPv4 group
SL_IPV6_ADD_MEMBERSHIP	Join IPv6 group
SL_IPV6_DROP_MEMBERSHIP	Leave IPv6 group

An example of joining the IPv4 multicast group:

```
_i16 Status;
SlSockIpMreq_t MulticastIp;

MulticastIp.imr_multiaddr.s_addr = sl_Htonl(SL_IPV4_VAL(224,0,1,200));
MulticastIp.imr_interface.s_addr = SL_INADDR_ANY;
Status = sl_SetSockOpt(Sd, SL_IPPROTO_IP, SL_IP_ADD_MEMBERSHIP,(char*) &MulticastIp,
sizeof(MulticastIp));
if( Status )
{
    // error
}
```

An example of leaving the IPv4 multicast group:

```
_i16 Status;
SlSockIpMreq_t MulticastIp;

MulticastIp.imr_multiaddr.s_addr = sl_Htonl(SL_IPV4_VAL(224,0,1,200));
MulticastIp.imr_interface.s_addr = SL_INADDR_ANY;
Status = sl_SetSockOpt(Sd, SL_IPPROTO_IP, SL_IP_DROP_MEMBERSHIP,(char*) &MulticastIp,
sizeof(MulticastIp));
if( Status )
{
    // error
}
```

5.5.2.2 Packet Boundary

By default the Rx boundary is kept. When the host application reads only a part of the data, the rest is dropped. The host application can disable the Rx boundary by using sl_SetSockOpt with SL_SO_RX_NO_IP_BOUNDARY. Here reading only a part of the data does not drop the rest of the data. This is a propriety option for UDP sockets only, which enables the host with limited buffering resources to read data in small chunks.

```
_i16 Status;
SlSockRxNoIpBoundary_t enableOption;

enableOption.RxIpNoBoundaryEnabled = 1;
Status = sl_SetSockOpt(Sd,SL_SOL_SOCKET,SL_SO_RX_NO_IP_BOUNDARY, (_u8*)&enableOption
,sizeof(enableOption));
 if( Status )
{
    // error
}
```

5.5.3 RAW

RAW sockets provide access to the underlying communication protocols with socket abstractions. The working flow is very similar to a connectionless socket (UDP).

5.5.3.1 Layer 4: Transport

RAW sockets in layer 4 let the host application send and receive packets, which include the IP header. Opening a RAW socket with TCP/UDP protocol means that all packets are forwarded directly to the RAW socket, and if any other TCP/UDP socket is open, it does not receive any of these packets. RAW sockets can work with any desired protocol, which should be specified when opening the socket.

By default all received packets include the IP header. If the IP header of the packet is not needed, it can be removed by calling the API sl_SetSockOpt with the option SL_IP_RAW_RX_NO_HEADER.

Example:

```
#define MY_PROTOCOL  90
_i16 Sd, Protocol = MY_PROTOCOL;

Sd = sl_Socket(SL_AF_INET /* SL_AF_INET6 */, SL_SOCK_RAW, Protocol);
if( 0 > Sd )
{
    // error
}
```

5.5.3.2 Layer 3: Network

RAW sockets in layer 3 let the host application send and receive packets, which include the network header. When opening a RAW socket with UDP/TCP protocol, TCP/UDP packets are forwarded directly to the RAW socket, and any other UDP/TCP sockets are useless. Calling sl_SetSockOpt with the option SL_IP_HDRINCL must contain an IP header. IPv4 checksum is calculated and set by the SimpleLink device. The received packet includes the IP header. This socket type is not supported for IPv6.

```
#define MY_PROTOCOL  90
_i16 Sd, protocol = MY_PROTOCOL , Status;
_u32 IncludeIpHeader = 1;

Sd = sl_Socket(SL_AF_INET, SL_SOCK_RAW, protocol);
if( 0 > Sd )
{
    // error
}
Status = sl_SetSockOpt(Sd, SL_IPPROTO_IP, SL_IP_HDRINCL, & IncludeIpHeader,
sizeof(IncludeIpHeader));
if( Status )
{
    // error
}
```

5.5.3.3 Layer 2: Data Link (Transceiver Mode, Not Connected)

The SimpleLink transceiver mode lets the host transmit Wi-Fi frames in disconnected mode only. The SimpleLink network stack can be bypassed by using the layer 2 RAW socket. Layer 2 lets hosts implement their own network stack and applications. For more information, see *Chapter 11*.

```
i16 Sd;

Sd = sl_Socket(SL_AF_RF, SL_SOCK_DGRAM, Channel);
if( 0 > Sd )
{
    // error
}
```

The SimpleLink transceiver mode lets the host transmit Wi-Fi frames in disconnected mode only. For more detailed information, see *Chapter 12*.

Example:

```
_i16 Sd, Channel = 11;

Sd = sl_Socket(SL_AF_RF, SL_SOCK_RAW, Channel);
if( 0 > Sd )
{
    // error
}
```

5.6 DNS

Hosts are mostly identified by their name and not their IP address, because the IP address might change, but the host name remains the same. Even in cases where the IP address is reserved permanently, it is common to remember names, and not IP addresses. For example, the IP address of Google® is not familiar to users even though it is reserved permanently. On the contrary, socket APIs use IP addresses not names, and sl_gethostbyName APIs are designed to bridge that gap. If successful, sl_gethostbyName resolves the IP address. To resolve an IP address, sl_gethostbyName sends the UDP DNS request several times, and with every retry the time-out increases. The number of retries and time-out parameters are configurable. The command sl_gethostbyName is a blocking command so if failure occurs it may take some time to return.

An example of the host application setting sl_gethostbyName parameters:

```
_i16 Status;
SlNetAppDnsClientTime_t Time;

Time.MaxResponseTime = 2000; // Max DNS retry timeout, DNS request timeout changed every retry,
start with 100Ms up to MaxResponseTime Ms
Time.NumOfRetries = 30; // number DNS retries before sl_NetAppDnsGetHostByName failed
Status = sl_NetAppSet(SL_NETAPP_DNS_CLIENT_ID, SL_NETAPP_DNS_CLIENT_TIME, sizeof(Time), (_u8
*)&Time);
if( Status )
{
    // error
}
```

An example of resolving the IPv4 address:

```
_i16 Status;
_u32 Ipv4Addr = 0;

Status =
sl_NetAppDnsGetHostByName("www.google.com",strlen("www.google.com"),&Ipv4Addr,SL_AF_INET);
if( Status )
{
   // error
}
```

An example of resolving the IPv6 address:

```
 _i16 Status;
_u32 Ipv6Addr[4] = {0};

Status =
sl_NetAppDnsGetHostByName("www.facebook.com",strlen("www.facebook.com"),Ipv6Addr,SL_AF_INET6);
if( Status )
{
   // error
}
```

5.7 Operation Modes

By default network bound APIs are blocking (network-bound APIs are APIs that trigger networking transactions and waits for their completion). For some implementations, especially on non-OS platforms, nonblocking operations are essential to allow other activities during these periods. For use cases, the SimpleLink device supports the standard nonblocking method of BSD sockets, and also a proprietary mode (trigger mode). In non-blocking mode, it is the responsibility of the application to poll the relevant API until the operation is completed. However, in trigger mode, instead of polling the API, the host receives an event when the operation is completed, and only then should call the API again.

Table 5-4 describes the different modes of the relevant APIs.

Table 5-4. Operational Modes

API	TCP, UDP, RAW	Blocking Mode	Non-Blocking Mode	Trigger Mode
sl_Connect	TCP	Blocked until connect success, or connect time-out.	Supported. SL_ERROR_BSD_EALREADY error code means not connected yet; poll again.	Not supported
sl_Recv/sl_RecvFrom	TCP	Blocked until data arrives. Recv Time-out can be set by sl_SetSockOpt.	Supported. SL_ERROR_BSD_EAGAIN error code means data has not arrived; poll again.	Not supported
	UDP			
	RAW			
sl_Send/sl_SendTo	TCP	Blocked until the internal buffer is available.	Supported. SL_ERROR_BSD_EAGAIN error code means no internal buffer available; try to send again.	Not supported
	UDP			
	RAW			
sl_Accept	TCP	Blocked until client connects.	Supported. SL_ERROR_BSD_EAGAIN error code means no client connection; try to accept again.	Not supported
sl_Select	TCP	Blocked until one or more registered sockets become ready.	Supported	Supported
	UDP			
	RAW			

5.7.1 Nonblocking Mode

In nonblocking mode, operations return immediately even if the data does not exist, or a connection is not established yet. It is the responsibility of the application to poll the operation until completion. When a server socket is configured as nonblocking, the accepted private socket inherits the nonblocking attribute. If there are several nonblocking sockets, TI recommends using sl_Select with time-out 0, instead of polling each socket separately.

The commands sl_Recv/ and sl_RecvFrom are unique, and allow nonblocking operation although the socket is in blocking mode. Two options are available for this mode.

- A single call to sl_Recv or sl_RecvFrom in nonblocking mode by using the SL_MSG_DONTWAIT flag. The API returns immediately with data if it exists or with the error SL_ERROR_BSD_EAGAIN. This action does not affect any socket settings or the following calls to sl_Recv/ and sl_RecvFrom.
- Setting receives a time-out. This setting applies for all the following calls to sl_Recv/ and sl_RecvFrom. When time-out expires, sl_Recv and sl_RecvFrom returns with SL_ERROR_BSD_EAGAIN, or earlier if the data arrives.

An example of setting the socket as non-blocking:

```
_i16 Status;
SlSockNonblocking_t BlockingOption;
BlockingOption.NonBlockingEnabled = 1;

// Enable or disable non-blocking mode
Status =
sl_SetSockOpt(Sd,SL_SOL_SOCKET,SL_SO_NONBLOCKING,(_u8*)&BlockingOption,sizeof(BlockingOption));
if( Status )
if( Status )
```

```
{
    // error
}
```

An example of a non-blocking TCP connect:

```
_i16 Status;
SlSockAddrIn_t  Addr;

Addr.sin_family      = SL_AF_INET;
Addr.sin_port        = sl_Htons(5001);
Addr.sin_addr.s_addr = sl_Htonl(SL_IPV4_VAL(192,168,1,31));

Status = SL_ERROR_BSD_EALREADY;
while( 0 > Status )
{
    Status = sl_Connect(Sd, ( SlSockAddr_t *)&Addr, sizeof(SlSockAddr_t));
    if( 0 > Status )
{
        if( SL_ERROR_BSD_EALREADY != Status )
        {
            // error
            break;
        }
    }
}
```

An example of receiving data with no wait flag:

```
_i16 Status;
_i8  RecvBuf[1460];

 Status = sl_Recv(Sd, RecvBuf, 1460, SL_MSG_DONTWAIT);
if( (0 > Status) && (SL_ERROR_BSD_EAGAIN != Status) )
{
    // error
}
```

An example of setting the receive data timeout:

```
_i16 Status;
struct SlTimeval_t TimeVal;

TimeVal.tv_sec =  5;             // Seconds
TimeVal.tv_usec = 0;             // Microseconds. 10000 microseconds resolution
Status = sl_SetSockOpt(Sd,SL_SOL_SOCKET,SL_SO_RCVTIMEO, (_u8 *)&TimeVal, sizeof(TimeVal));    //
Enable receive timeout
if( Status )
{
    // error
}
```

5.7.2 Trigger Mode

The trigger mode enables host applications to be triggered by the SimpleLink device when network activity is detected, without using the blocking mode or polling the socket. This mode is useful when the power consumption is extremely sensitive and the host processor is able to enter a deep sleep, recover fast, and retain memory. The trigger mode is implemented by calling sl_Select. The host enters a deep sleep and wakes up due to an event, when one or more sockets become ready. After the host wakes up, sl_Select must be called again to identify the network activity. All blocking socket operations can be monitored by sl_Select, called with time-out values set to 0 (sec and µs), which allow application flexibility to implement many communication use cases. Only one select operation is supported simultaneously.

To define the host application in trigger mode follow these steps:

- Define host IRQ as the host wake up source.
- Ensure slcb_SocketTriggerEventHandler is registered under user.h and handle the trigger

asynchronous event SL_SOCKET_TRIGGER_EVENT_SELECT.

- Detecting select events should notify the main task trigger event of arrival, call select again.

Figure 5-3 describes a general flow of using trigger mode for accept on a server socket.

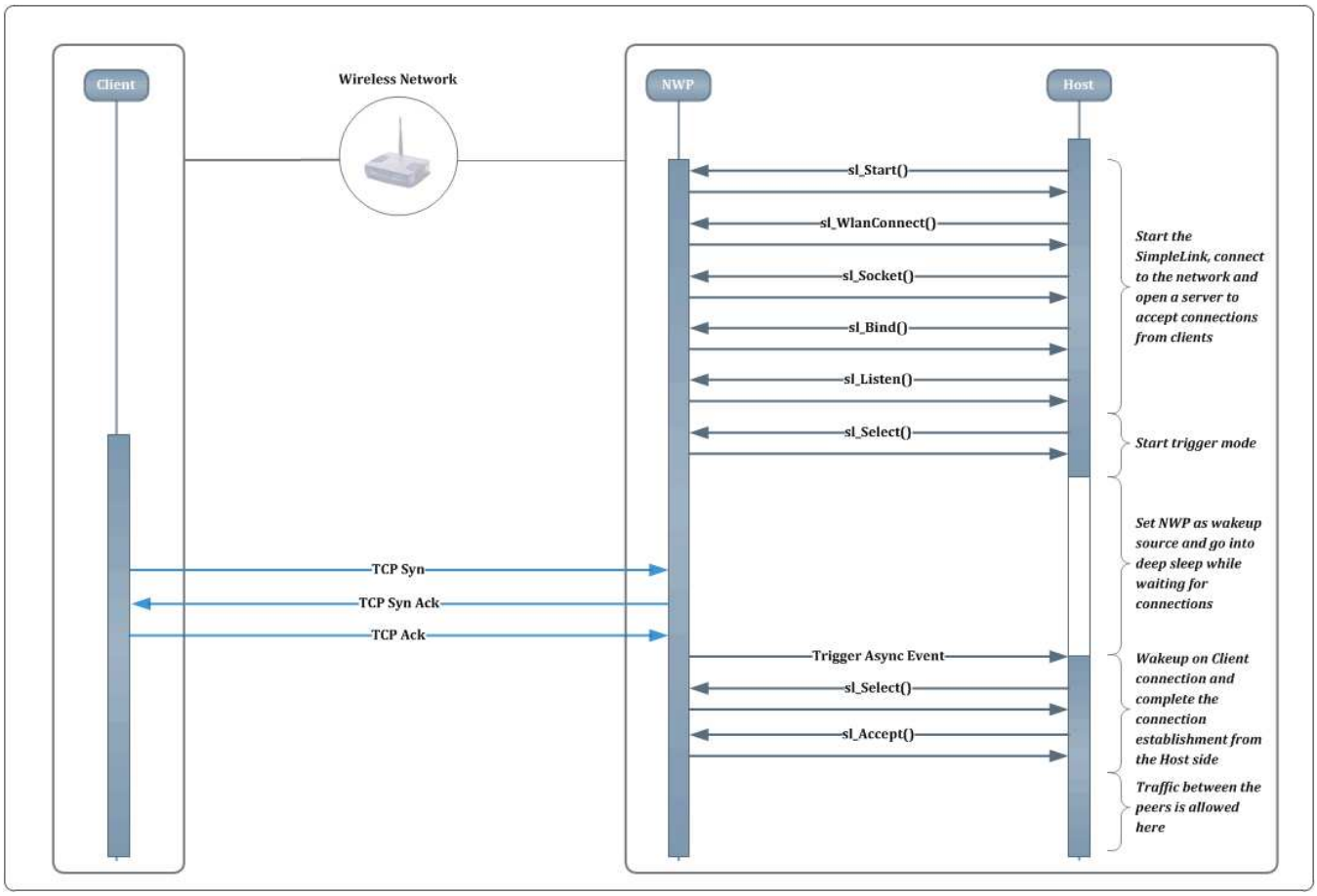

Figure 5-3. Trigger Mode Flow

An example of select trigger event handle:

```
void SimpleLinkSocketTriggerEventHandler(SlSockTriggerEvent_t    *pSlTriggerEvent)
{
    switch (pSlTriggerEvent ->Id)
    {
        case SL_SOCKET_TRIGGER_EVENT_SELECT:
     {
         //Notify main task trigger event arrive, wake up and call select again
          break;
     }
     default:
      break;
    }
}
```

5.7.2.1 Trigger Mode for Accept

1. Open the TCP Server Socket and call sl_Select on the following socket.

```
_i16 Status,Sd,LocalSd;
_u16 nfds;
SlSockAddrIn_t  LocalAddr,Addr;
SlTimeval_t timeVal;
SlFdSet_t  rxSet;
```

```
LocalAddr.sin_family = SL_AF_INET;
LocalAddr.sin_port = sl_Htons(5001);
LocalAddr.sin_addr.s_addr = 0;

timeVal.tv_sec = 0;
timeVal.tv_usec = 0;

//Open TCP server socket
Sd = sl_Socket(SL_AF_INET,SL_SOCK_STREAM, 0);
if( Status )
{
    // error
}
//Bind the server socket
Status = sl_Bind(Sd, (SlSockAddr_t *)&LocalAddr, sizeof(SlSockAddrIn_t));
if( Status )
{
    // error
}
//Listen
Status = sl_Listen(Sd, 0);
if( Status )
{
    // error
}
nfds = Sd + 1;
SL_SOCKET_FD_ZERO( &rxSet );
SL_SOCKET_FD_SET( Sd, &rxSet );
Status = sl_Select( nfds, &rxSet, NULL, NULL, &timeVal );
if( Status )
{
    // error
}
```

2. The host now can enter deep sleep until triggered by the select event.
3. After the SL_SOCKET_TRIGGER_EVENT_SELECT event is received, the host wakes up and calls sl_Select to identify which socket has network activity.

```
//Call select again since the trigger event has arrived (see handler example above)
SL_SOCKET_FD_ZERO( &rxSet );
SL_SOCKET_FD_SET( Sd,& rxSet );
Status = sl_Select( nfds, &rxSet, NULL, NULL, &timeVal );
if (SL_SOCKET_FD_ISSET(Sd, &rxSet))
{
    //socket is marked, call accept
    LocalSd = sl_Accept(Sd, (SlSockAddr_t*)&Addr, (SlSocklen_t*) (sizeof(SlSockAddrIn_t)));
}
```

5.7.2.2 Trigger Mode for Data Reception

1. Open the TCP client socket and call sl_Select on the following socket.

```
_i16 Status, Sd;
_u16 nfds;
SlSockAddrIn_t Addr;
SlTimeval_t timeVal;
SlFdSet_t  rxSet;
_i8  RecvBuf[1460];

timeVal.tv_sec = 0;
timeVal.tv_usec = 0;

//Open TCP client socket
Sd = sl_Socket(SL_AF_INET,SL_SOCK_STREAM, 0);
if( Status )
```

```
{
    // error
}
Addr.sin_family = SL_AF_INET;
Addr.sin_port = sl_Htons(5001);
Addr.sin_addr.s_addr = sl_Htonl(SL_IPV4_VAL(192,168,1,31));;
Status = sl_Connect(Sd, ( SlSockAddr_t *)&Addr, sizeof(SlSockAddr_t));
if( Status )
{
    // error
}
Status = sl_Select( nfds, NULL, &rxSet, NULL, &timeVal );
//Sleep until triggered by the select event
```

2. The host can now enter deep sleep until triggered by the select event
3. After the SL_SOCKET_TRIGGER_EVENT_SELECT event is received, the host wakes up and calls sl_Select to identify which socket has network activity.

```
//Call select again since the trigger event has arrived
SL_SOCKET_FD_ZERO( &rxSet );
SL_SOCKET_FD_SET( Sd, &rxSet );
Status = sl_Select( nfds, NULL, &rxSet, NULL, &timeVal );
if (SL_SOCKET_FD_ISSET(Sd, &rxSet))
{
//socket is marked, call receive
Status = sl_Recv(Sd, RecvBuf, 1460, 0);
if( Status )
{
    // error
}
}
```

5.8 IP Fragmentation

IP fragmentation is a method of breaking the IP packet into smaller messages compatible with the Maximum Transmission Unit (MTU) size, and reassembling them on the receive side. IPv4 routers fragment packets according to the MTU of the link. IPv6 routers do not fragment, and it is the responsibility of the device to fragment the packets. When receiving data, the SimpleLink device supports reassembling of the received IP fragmented packets for both IPv4 and IPv6. When the host application sends data which is bigger than the MTU size, the SimpleLink device splits this data into packets compliant with the MTU size without using IP fragmentation. For TCP, the size has no effect because TCP ensures byte ordering. However, for UDP the size may cause packet reordering, therefore, TI recommends that host application sends UDP data up to the MTU size (1472 bytes for IPv4 and 1452 bytes for IPv6), or verify data integrity in higher layers.

The SimpleLink device response to a fragmented ping, the maximum ping packet payload is 19,232 bytes for Ipv4 and 27,976 bytes for IPv6.

5.9 Errors

One of the main differences between BSD sockets and SimpleLink sockets implementation is that error codes are returned directly, and not through the errno method (as in Linux). Errors are indicated by the return value of the API, or by asynchronous events. Asynchronous events can be sent to the host at any given time with a specific error indication and include specific data for each event. To listen to these events and conclude the needed information, a handler should be implemented in the user application and registered under the user.h header file. Each error code is unique. The following errors are common and require user action (full possible error list is under the file error.h in the host driver).

Table 5-5 lists errors indicated by asynchronous events.

Table 5-5. Asynchronous Error Events

Error	Handler	Comments
SL_SOCKET_TX_FAILED_EVENT	slcb_SockEvtHdlr	Socket error – include the parameters status (specified in Table 5-6and socket ID)

Table 5-6 lists common errors status codes.

Table 5-6. Common Error Status Codes

Error	Value	Comments
SL_ERROR_BSD_SOC_ERROR	–1	General socket error
SL_ERROR_BSD_INEXE	–8	Socket command in execution
SL_ERROR_BSD_EBADF	–9	Bad file number
SL_ERROR_BSD_ENSOCK	–10	The system limit on the total number of open sockets has been reached.
SL_ERROR_BSD_EAGAIN	–11	Try again
SL_ERROR_BSD_ECLOSE	–15	Close socket operation failed to transmit all queued packets.
SL_ERROR_BSD_EINVAL	–22	Invalid argument
SL_ERROR_BSD_EPROTOTYPE	–91	Protocol wrong type for socket
SL_ERROR_BSD_EADDRINUSE	–98	Address is already in use
SL_ERROR_BSD_ENETUNREACH	–101	Network is unreachable
SL_ERROR_BSD_ETIMEDOUT	–110	Connection timed out
SL_ERROR_BSD_ECONNREFUSED	–111	Connection refused
SL_ERROR_BSD_EALREADY	–114	Nonblocking connect in progress, try again

Secure Socket

Topic **Page**

6.1 Introduction

The SimpleLink device provides a secured socket layer using the SSL and TLS protocols (both referenced as SSL in this document), which are cryptographic protocols designed to provide communications security over a TCP connection. For common systems, the SSL is a layer on top of the transport layer. To simplify the use, the SSL is embedded into the BSD layer in the SimpleLink device. SSL operations are easily done by using the BSD commands with unique parameters and options. The SimpleLink device supports up to six SSL sockets connected at a time. The SSL uses separate execution environment by design, to better secure the keys and flows in the SimpleLink device. Hardware accelerators are used to offload the MCU in arithmetic calculation of cryptography algorithms.

6.2 Key Features

Table 6-1 lists the key features of the secure socket.

Table 6-1. Key Features

Key Features	Description	Client	Server
SSL server	Open SSL servers and accept up to six peers (six is the maximum SSL connections, it depends on how many clients are connected).		
SSL client	Open SSL client and connect up to six peers (six is the maximum SSL connections, it depends on how many servers are connected).		
Certificates	Support certificates and root CAs according to x509 standard.	√	√
BSD commands	The SSL layer is embedded into the BSD commands to ease the usage.	√	√
Server verification	Support full chain of trust verification while the SimpleLink device is in client mode.	√	N/A
Domain verification	Support domain verification in client mode, to help against MITM attack.	√	X
Client verification	Support client authentication, both in server mode to authenticate a client that is trying to connect to the server, and in client mode, when a remote server is asking for client certificate.	N/A	√
Time and Date verification	Support time and date verification of server/client cert according to the time and date configured in the SimpleLink device.	√	X
Cryptography	Support the following cryptographic algorithms – RC4,AES|GCM|CBC,CHACHA20,SHA1|256|384|512,MD5,POLY 1305,RSA,DHE,ECDSA,ECDHE.	√	√
STARTTLS	Start SSL handshake on a regular TCP socket. Usually used for SMTP on port 587.	√	√
ALPN	Support Application Layer Protocol Names List; this is a limited list with HTTP1.1 and H2 drafts.	√	X
DER/PEM file formats	Certificate files and keys can be programmed to the file system in either DER or PEM formats. Certificate chain must be in PEM format. Certificate chain is only available in server mode.	√	√
Trusted root-certificate catalog	Mechanism to determine if a root CA is known and trusted by TI or if a certificate is revoked.	√	X
Server name indication (SNI)	Setting a domain name verification enables the SNI extension in the client hello message, according to RFC 6066	√	X

6.3 Opening a Secure Socket

This section provides information on how to establish secured socket session with BSD API. A secured socket is a TCP socket, which encrypts and decrypts data. The BSD flow is the same as regular TCP socket BSD, excluding specific secured socket options.

There are two ways to open secured socket:

- sl_Socket(SL_AF_INET, SL_SOCK_STREAM, SL_SEC_SOCKET) – This command opens a secured socket. The first two parameters are typical TCP socket parameters, and the last parameter enables the security. After the socket has been created, it is possible to use the standard *BSD commands

(sl_Close, sl_Listen, sl_Accept, sl_Bind, sl_SetSockOpt, and so forth).

- Use STARTTLS to upgrade a regular connected TCP socket to a secured one (used mainly for SMTP port 587), according to the following flow:
  1. sl_Socket(SL_AF_INET, SL_SOCK_STREAM,0) – Opens a regular TCP socket.
  2. Use sl_Accept (in server mode) or sl_Connect to establish a connection.
  3. May transfer unsecured data using sl_Send and sl_Recv.
  4. Upgrade a socket to STARTTLS using sl_SetSockOpt with the SL_SO_STARTTLS option.

When the connection is established, it is possible to use sl_Recv and sl_Send to transact data between the peers, exactly like in an unsecured TCP socket.

NOTE: Some dedicated SSL configurations (performed by calling sl_SetSockOpt) must be applied after opening the socket, and not after sl_Connect in client mode or sl_Listen in server mode, as described in Section 6.5.

6.4 Trusted Root-Certificate Catalog

The trusted root-certificate catalog is a file, provided by TI, containing a list of known and trusted root CAs by TI. The certificate store holds the common trusted root CAs in the market, such as VeriSign, GoDaddy, GeoTrust, and so forth.

The trusted root-certificate catalog also holds a list of revoked certificates known to TI. The trusted root-certificate catalog is used only in client mode. Servers use a proprietary root CA to authenticate clients, and therefore cannot use the certificate store. The trusted root-certificate catalog gives the user the confidence that the CA is trusted and known. When a root CA does not exist in the catalog, the sl_Connect command returns the error SL_ERROR_BSD_ESECUNKNOWNROOTCA, which means the connection is successfully done, but the root CA used to verify the server chain of trust is unknown. When a revoked certificate is received during the SSL connection (all of the certificate chain is checked) or if the root CA set by the user is revoked, the handshake fails, and the error SL_ERROR_BSD_ESECCERTIFICATEREVOKED returns from the sl_Connect command.

6.5 Options and Features Use

Options are used to enable or disable features, or to set some configurations to the SSL socket. To change the options, use the BSD sl_SetSockOpt with unique options.

If no options were set, the following defaults take effect:

- All SSL versions are enabled (handshake starts with the highest – TLS1.2, but the server could peek lower versions).
- All cipher suites are enabled.
- Files which are required for the SSL connection (in server mode, some of the files are mandatory to complete the handshake) remain blank.
- Trusted root-certificate catalog is used by default.

The socket settings (specified in *Section 6.5.1*) must be called before the sl_Connect or sl_Listen commands to take effect. In server mode, those settings are inherited to the child socket, and cannot be applied directly on the child socket.

NOTE: Setting the server certificate and private key are mandatory when opening an SSL server.

6.5.1 Set SSL Version

Set specific SSL versions for the socket. This should be called before sl_Connect or sl_Listen.

- SL_SO_SEC_METHOD_SSLV3
- SL_SO_SEC_METHOD_TLSV1
- SL_SO_SEC_METHOD_TLSV1_1

- SL_SO_SEC_METHOD_TLSV1_2
- SL_SO_SEC_METHOD_SSLv3_TLSV1_2 – all enabled

Example:

```
SlSockSecureMethod_t method;
_i6 status;

method.SecureMethod = SL_SO_SEC_METHOD_TLSV1 | SL_SO_SEC_METHOD_TLSV1_2;
status = sl_SetSockOpt(sd,SL_SOL_SOCKET,SL_SO_SECMETHOD,&method,sizeof(SlSockSecureMethod_t));
```

6.5.2 Set Cipher Suites

Set the socket to use specific cipher suites. This should be called before sl_Connect, or sl_Listen.

- SL_SEC_MASK_SSL_RSA_WITH_RC4_128_SHA
- SL_SEC_MASK_SSL_RSA_WITH_RC4_128_MD5
- SL_SEC_MASK_TLS_RSA_WITH_AES_256_CBC_SHA
- SL_SEC_MASK_TLS_DHE_RSA_WITH_AES_256_CBC_SHA
- SL_SEC_MASK_TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
- SL_SEC_MASK_TLS_ECDHE_RSA_WITH_RC4_128_SHA
- SL_SEC_MASK_TLS_RSA_WITH_AES_128_CBC_SHA256
- SL_SEC_MASK_TLS_RSA_WITH_AES_256_CBC_SHA256
- SL_SEC_MASK_TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- SL_SEC_MASK_TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- SL_SEC_MASK_TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- SL_SEC_MASK_TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- SL_SEC_MASK_TLS_RSA_WITH_AES_128_GCM_SHA256
- SL_SEC_MASK_TLS_RSA_WITH_AES_256_GCM_SHA384
- SL_SEC_MASK_TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
- SL_SEC_MASK_TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
- SL_SEC_MASK_TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- SL_SEC_MASK_TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- SL_SEC_MASK_TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- SL_SEC_MASK_TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- SL_SEC_MASK_TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
- SL_SEC_MASK_TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
- SL_SEC_MASK_TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256

Example:

```
SlSockSecureMask_t mask;
_i16 status;

mask.SecureMask = SL_SEC_MASK_TLS_RSA_WITH_AES_256_CBC_SHA |
SL_SEC_MASK_TLS_RSA_WITH_AES_256_CBC_SHA;
status = sl_SetSockOpt(sd,SL_SOL_SOCKET,SL_SO_SECURE_MASK,&mask,sizeof(SlSockSecureMask_t));
```

6.5.3 Set Certificates, Root CA, Private Key, and DH Files

Set filenames to be used during the SSL handshake. The files must be programmed to the NWP file system. The files should be in PEM or DER format. The client can successfully connect to a server that does not require client authentication, without any files (the server would not be verified, because no root CA is programmed). The server must provide a server certificate during the SSL handshake, and therefore must use this sl_SetSockOpt command to provide the certificate and private key of the server.

DH files are Diffie Hellman parameters files. These parameter files contain parameters for generating a DH key when using DHE cipher suites in server mode.

In server mode; if there is no DH file, the DH cipher suites are not available, even if a secured mask is used to peek certain cipher suites with DH. If an ECDSA signature is used in the server certificate, the RSA ciphers are not available, and vice versa.

Table 6-2. Related Files

File	Client	Server
Root CA file Format: PEM/DER. The self-signed certificate that signed the other peer chain	Validates the remote peer (the remote server) If file does not exist, connection success with error SL_ERROR_BSD_ESECSNOVERIFY	Enables client verification when programmed (not mandatory). If programmed and peer did not send its certificate, a socket asynchronous event is raised with error SL_ERROR_BSD_ESEC_NO_PEER_CERT.
Cert Format: PEM/DER. A certificate issued to this peer side.	Client Cert or certificate chain if server requires client authentication. Chain can only be programmed in a PEM format, where the client certificate is the first, followed by all the intermediate CAs. If file does not exist, and the server requires client authentication, the server returns ALERT of peer verify error in the sl_Connect command. The user must program private key with this file, or else connection fails with SL_ERROR_BSD_ESECBADPRIVATEFILE.	Server certificate or certificate chain. Chain could only be programmed in PEM format. The server cert should be the first in the list. The file must be configured. If not configured, error SL_ERROR_BSD_ESECBADCERTFILE occurs.
Private Key Format: PEM/DER. RSA or ECDSA key.	Client private key if server requires client auth. The user must program cert with this file, or else connection fails with SL_ERROR_BSD_ESECBADCERTFILE.	The private key of the server. Must be configured. If not configured, error SL_ERROR_BSD_ESECBADPRIVATEFILE is raised.
DH (server) or PEER Cert (client) Format: PEM/DER. Other side certificate or DH parameters.	Configuring this file enables the domain verification by full server cert comparison. This file is the server expected cert. This is being compared to the server certificate that was received from the server during the handshake phase, to validate that this is truly the domain to connect to (stronger than the domain name verification).	DH file –Diffie Hellman parameters file. Contains parameters for generating DH key when using DHE cipher suites in server mode. Enables the DH ciphers.

Binding a file to a socket is done using sl_SetSockOpt, before the sl_Connect or sl_Listen commands.

- SL_SO_SECURE_FILES_PRIVATE_KEY_FILE_NAME
- SL_SO_SECURE_FILES_CERTIFICATE_FILE_NAME
- SL_SO_SECURE_FILES_CA_FILE_NAME
- SL_SO_SECURE_FILES_PEER_CERT_OR_DH_KEY_FILE_NAME

Example:

```
_i16 status;

status =
sl_SetSockOpt(sd,SL_SOL_SOCKET,SL_SO_SECURE_FILES_CA_FILE_NAME,"ca.der",strlen("ca.der"));
```

6.5.4 Disable the Use of the Trusted Root-Certificate Catalog

The user can disable the use of the trusted root-certificate catalog if a personal unknown root CA is used. This is done by using this sl_SetSockOpt, before the sl_Connect or sl_Listen commands.

Example:

```
_u32 dummyVal;
_i16 status;
```

```
status = sl_SetSockOpt(SockID,SL_SOL_SOCKET, SL_SO_SECURE_DISABLE_CERTIFICATE_STORE,

&dummyVal,sizeof(dummyVal));
```

6.5.5 Set ALPN List

ALPN is a list of application protocols negotiated in the handshake. The client sends the desired ALPN list, and the server picks one and notifies the client.

The supported protocols are:

- SL_SECURE_ALPN_H1 – http 1.1
- SL_SECURE_ALPN_H2 – http 2
- SL_SECURE_ALPN_H2C – http 2 draft c
- SL_SECURE_ALPN_H2_14 – http 2 draft 14
- SL_SECURE_ALPN_H2_16 – http 2 draft 16
- SL_SECURE_ALPN_FULL_LIST

This list is only available in client mode. The list is not set by default if this option is not used. To retrieve the selected protocol after the handshake, use sl_GetSockOpt with the SL_SO_SSL_CONNECTION_PARAMS option. This option should be called before sl_Connect or sl_Listen.

Example:

```
SlSockSecureALPN_t alpn;
_i16 status;

alpn.SecureALPN = SL_SECURE_ALPN_H1 | SL_SECURE_ALPN_H2_16;
status = sl_SetSockOpt(SockID,SL_SOL_SOCKET,SL_SO_SECURE_ALPN,&alpn,sizeof(SlSockSecureALPN_t));
```

6.5.6 Set Domain Name for Verification and SNI

Set the domain name to verify the desired domain during the SSL handshake. The domain verification is used to help against “man in the middle attack,” where a third party could buy a fake certificate from the same root CA that signed the certificate of the server, and redirect the traffic to their server. Besides the full chain verification, TI recommends checking the domain name as well. This option is only available for client mode. This option should be called before sl_Connect or sl_Listen. Setting a domain name also enables the SNI extension in the client hello message, according to RFC 6066.

Example:

```
_i16 status;

Status = sl_SetSockOpt(SockID, SL_SOL_SOCKET,_SO_SECURE_DOMAIN_NAME_VERIFICATION,
    "www.google.com",strlen("www.google.com"));
```

6.5.7 Upgrade Nonsecured Socket to Secured

When connecting a regular TCP socket to a peer, the TCP socket can be upgraded to an SSL socket by using the STARTTLS option, depending on the application layer of the other peer. The other peer also must support such an upgrade. The upgrade is basically the initialization of an SSL handshake between the peers, while in a connected session.

The most common use case is the SMTP protocol, on port 587. The client connects to an SMTP server, several packets may transact unencrypted, and then the client initiates a STARTTLS request to the server (each application protocol has its own STARTTLS byte string, and therefore should be sent by the host application). At this point the handshake starts with a GO AHEAD message sent by the server, responded to by a HELLO message from the client.

Calling sl_SetSockOpt with the STARTTLS option triggers the NWP, in client mode, to send the client HELLO message, and in server mode to wait until the client HELLO message is received. When the handshake is finished, the user gets a socket asynchronous event which indicates success or failure, and in case of failure, a specific error code.

Example:

```
void slcbSockEvtHdlr(SlSockEvent_t* pSlSockEvent)
{
    char *CAname;
    if(SL_SOCKET_ASYNC_EVENT == pSlSockEvent->Event)
    {
        /* debug print "an event received on socket %d\n",
           pSlSockEvent->SocketAsyncEvent.SockAsyncData.Sd */
        switch(pSlSockEvent->SocketAsyncEvent.SockAsyncData.Type)
        {
        case SL_SSL_NOTIFICATION_CONNECTED_SECURED:
            /* indicate handshake successful ok */
            break;
        case SL_SSL_NOTIFICATION_HANDSHAKE_FAILED:
            /* retrieve an error from pSlSockEvent->SocketAsyncEvent.SockAsyncData.Val; */
            break;
        default:
            break;
        }
    }
}

void ClientSTARTTLSExample()
{
    SlSockAddrIn_t  Addr;
    SlSockSecureMethod_t method;
    _i32 sd,len,dummyVar;
    _i16 status;
    _u32 DestinationIP = SL_IPV4_VAL(192,168,1,31);  /* An SMTP server's IP */
    _i16 AddrSize;
    _i8 buf[100];

    Addr.sin_family = SL_AF_INET;
    Addr.sin_port = sl_Htons(587);
    Addr.sin_addr.s_addr = sl_Htonl(DestinationIP);
    AddrSize = sizeof(SlSockAddrIn_t);

    /* Open SSL socket */
    sd = sl_Socket(SL_AF_INET,SL_SOCK_STREAM,0);
    if(sd < 0)
    {
        /* error... */
    }

    method.SecureMethod = SL_SO_SEC_METHOD_SSLv3_TLSV1_2;
    status = sl_SetSockOpt(sd,SL_SOL_SOCKET,SL_SO_SECMETHOD,&method,sizeof(SlSockSecureMethod_t));
    if(status < 0)
    {
        /* error... */
    }

    /* set a CA filename to be used to verify the SMTP server
       certificate when the handshake will take place */
    status = sl_SetSockOpt(sd,SL_SOL_SOCKET,SL_SO_SECURE_FILES_CA_FILE_NAME,
                           "smtpCa.der",strlen("smtpCa.der"));
    if(status < 0)
    {
        /* error... */
    }

    status = sl_Connect(sd, ( SlSockAddr_t *)&Addr, AddrSize);
    if(status < 0)
    {
        /* error... */
    }
```

```
    /* unsecured transaction */
    len = sl_Recv(sd,buf,100,0);
    if(len < 0)
    {
        /* error... */
    }

    len = sl_Send(sd,"HELO server",strlen("HELO server"),0);
    if(len < 0)
    {
        /* error... */
    }

    /*...
      ...
      ... */

    len = sl_Send(sd,"STARTTLS",strlen("STARTTLS"),0);
    if(len < 0)
    {
        /* error... */
    }

    len = sl_Recv(sd,buf,100,0);
    if(len < 0)
    {
        /* error... */
    }

    if(strcmp(buf,"GO AHEAD") == 0)
    {
        /* we got a green light, we can start the SSL handshake */
        status = sl_SetSockOpt(sd,SL_SOL_SOCKET,SL_SO_STARTTLS,&dummyVar,sizeof(dummyVar));
        if(status < 0)
        {
            /* error... */
        }

        /* wait for the flag to update from slcbSockEvtHdlr async event
           and handle it, if an error occurs
           ...
           ...
           ... */
    }

    /*...
      ...
      ... */

    status = sl_Close(sd);
    if(status < 0)
    {
        /* error... */
    }
}
```

6.5.8 Get Connection Parameters

Get the connection parameters after a successful handshake completes. The received parameters include server certificate parameters, chosen SSL version and cipher suite, and more.

Due to a large amount of data, only 16 bytes of the issuer and subject common name are kept. The XORed hash of those names, plus the 16-byte name strings, are helpful in verifying a wanted name.

Example:

```
SlSockSSLConnectionParams_t conPa;
```

```
_i16 status;

SlSocklen_t len = sizeof(SlSockSSLConnectionParams_t);
status = sl_GetSockOpt(SockID,SL_SOL_SOCKET,SL_SO_SSL_CONNECTION_PARAMS,&conPa,&len);
```

6.6 Supported Cryptographic Algorithms

Table 6-3 lists the supported cryptographic algorithms.

Table 6-3. Cryptographic Algorithms

Algorithm	Hardware or Software	Usage	Key Length
ECDSA	Software	Signature algorithm	Dynamically generated Named curves – secp160r1 secp192r1 secp224r1 secp256r1 secp384r1 secp521r1
ECDHE	Software	Key exchange	Dynamically generated Named curves as ECDSA
DH	Software	Key exchange	Dynamically generated
RSA Key < 4096	Hardware	Signature algorithm/Key exchange	128, 256
RSA Key > 4096	Software	Signature algorithm/Key exchange	512,1024
SHA1	Hardware	Signature algorithm/Message authentication code	20
SHA256	Hardware	Signature algorithm/Message authentication code	32
SHA384	Software	Signature algorithm	48
SHA512	Software	Signature algorithm	64
MD5	Hardware	Signature algorithm/Message authentication code	16
POLY1305	Software	Message authentication code	16
AES CBC	Hardware	Data encryption	16, 32
AES GCM	Hardware	Data encryption/Message authentication code	16, 32
RC4	Software	Data encryption	16
CHACHA20	Software	Data encryption	16
TRNG	Hardware	Random numbers	

6.7 Common Errors and Asynchronous Events

In most cases, the socket API returns the error code as a return value of the API. In other cases, where the error occurs during a live process, the error or notification returns in a socket asynchronous event.

6.7.1 Using Socket Asynchronous Events in SSL

SSL asynchronous events which provide information about the connection:

- SL_OTHER_SIDE_CLOSE_SSL_DATA_NOT_ENCRYPTED – The remote side closed the SSL layer, and the socket is not secured anymore; data can still transfer but is not encrypted.
- SL_SSL_ACCEPT – An error occurred during an SSL accepting, but the socket is ready to accept again with no need to call accept again. A good example of that is a time-out during the handshake.
- SL_SSL_NOTIFICATION_WRONG_ROOT_CA – This event is only available in client mode, and it goes along with the SL_ERROR_BSD_ESEC_ASN_NO_SIGNER_E error received during the sl_Connect command. This event indicates that a certificate in the certificate chain could not be verified because the CA programmed to the file system is not the right CA that signed the chain. This

event gives the CommonName of the CA root expected to verify the certificate.

Example:

```
void slcbSockEvtHdlr(SlSockEvent_t* pSlSockEvent)
{
char *CAname;
if(SL_SOCKET_ASYNC_EVENT == pSlSockEvent->Event)
    {
/* debug print "an event received on socket %d\n",pSlSockEvent-
>SocketAsyncEvent.SockAsyncData.Sd */
    switch(pSlSockEvent->SocketAsyncEvent.SockAsyncData.Type)
        {
    case SL_SSL_NOTIFICATION_CONNECTED_SECURED:
        break;
    case SL_SSL_NOTIFICATION_HANDSHAKE_FAILED:
        break;
    case SL_SSL_ACCEPT:
        break;
    case SL_OTHER_SIDE_CLOSE_SSL_DATA_NOT_ENCRYPTED:
        break;
    case SL_SSL_NOTIFICATION_WRONG_ROOT_CA:
        break;
    default:
        break;
        }
    }
}
```

6.7.2 Common Errors

Table 6-4 lists the common errors.

Table 6-4. Common Errors

Error	Client	Server
SL_ERROR_BSD_ESECSNOVERIFY	Connected without verifying the peer. Use sl_SetSockOpt to set the CA to verify the peer.	N/A
SL_ERROR_BSD_ESECNOCAFILE	The CA filename used in the sl_SetSockOpt is not in the file system. Use the correct filename, or program the file in the name desired.	The CA filename used in the sl_SetSockOpt is not in the file system. Use the correct filename, or program the file in the name desired.
SL_ERROR_BSD_ESECBADCAFILE SL_ERROR_BSD_ESECBADCERTFILE SL_ERROR_BSD_ESECBADPRIVATEFILE SL_ERROR_BSD_ESECBADDHFILE	The file is not valid. Check if it is a valid DER/PEM CA file.	The file is not valid. Check if it is a valid DER/PEM CA file.
SL_ERROR_BSD_ESECT00MANYSSLOPENED	Exceed maximum SSL connections. The SimpleLink supports only six SSL connected sockets.	Exceed maximum SSL connections. The SimpleLink supports only 6 SSL connected sockets.
SL_ERROR_BSD_ESECDATEERROR	Connected but with error verifying time and date on the certificates error of the server. Set the time and date on the device or check the certificate date of the other side.	N/A
SL_ERROR_BSD_ESEC_SOCKET_ERROR	TCP socket was disconnected during the SSL handshake. This usually occurs when the other side closed the connection. Investigate the peer.	TCP socket was disconnected during the SSL handshake. This usualyl occurs when the other side closed the connection. Investigate the peer.

Table 6-4. Common Errors (continued)

Error	Client	Server
SL_ERROR_BSD_ESEC_ASN_NO_SIGNER	Could not verify one of the certificates in the peer's cert. This usually occurs when using a wrong CA to verify the peer. Use the SL_SSL_NOTIFICATION_WRONG_ROOT_CA to get the desired CA CommonName.	Could not verify one of the certificates in the peer's cert.
SL_ERROR_BSD_ESECUNKNOWNROOTCA	Connected but the root CA used to verify the peer is unknown to TI. That means it does not appear in the trusted root-certificate catalog.	N/A

File System

Topic Page

7.1 Introduction

This chapter describes the capabilities of the file system and the host interface, and provides usage recommendation.

The SimpleLink Wi-Fi device maintains a nonvolatile file system which stores the data on an external serial flash (through SPI). The file system provides the ability to organize data and access resources using a simple host interface. In addition, the file system is used to store the networking subsystem configuration. The need for secure storage is a major factor in the IoT world, where devices are more vulnerable to security attacks. The SimpleLink Wi-Fi device provides a file system with security features that protects the device from cloning. Secure files are kept encrypted on the external storage device (external SFLASH). File authentication is also supported.

The SimpleLink Wi-Fi networking subsystem uses the file system to store the system configuration files and for storing the service pack. The CC3220 device uses the storage to store the host application.

The file system provides features to protect the stored files from manipulation by unknown or malicious users, such that the files cannot be read or modified freely by third parties.

In addition, the SimpleLink Wi-Fi device supports cloning protection; moving or cloning a SFLASH that was written by one device to other does not work . One device cannot use a file system or read secure files created by another device.

File integrity monitoring is an internal process that performs the act of validating the integrity of the file system and stored files, using a verification method between the current file state and the known, good baseline. This comparison method involves calculating a known cryptographic checksum of the original baseline of the file, and comparing it with the calculated checksum of the current state of the file.

The SimpleLink internal process for software tamper detection monitors the use of the file and detects attempts to tamper the file system; it detects operations such as accessing a file without the correct credentials, or writing a file by an unauthenticated user.

The SimpleLink Wi-Fi device also provides a recovery mechanism; it enables to rollback the system configuration to the factory settings.

7.2 Key Features

Table 7-1 lists the key security features.

Table 7-1. Key Features

Feature	Description
Maximum number of files	240 (50 files should be reserved for system files)
Maximum file size	Unlimited
Maximum file name length	180 bytes
Maximum SFLASH size	16 MB
Type of files	Regular, secure, secure + authenticate
File functions	Create, open for write, open for read, rename a file, get file information
Get a files list	Retrieves the file list and attributes
Get storage usage	Retrieve general information regarding the storage status: free space, allocated space, number of security alerts.
File commit/rollback	Methods for downloading a single file and in case of failure rollback to the former file image
Bundle commit/rollback	Methods for downloading a group of files and in case of failure rollback all the files (as single transaction) to the former files image
Programming	The method to first install the device with the required configuration and files.
Programming by third party	Program the SFLASH and assemble it to the device
Restore to factory image/defaults	Return to the programmed image
Security alerts	
Development/production format	Option to create a development image which can run on specific MAC.

7.3 File System Characteristics

The following list describes the file system characteristics:

- Supported number of files is 240, including system files.
- The maximum number of system files is 50 files; however, most of the system files are created only if they are required by the application.
- File size is unlimited.
- Filename can be up to 180 bytes. Choose short names due to the limited allocated size for filenames.

  Filenames are kept on a best-effort basis; the size allocated for filenames is 3136 bytes. If the total length of the filenames (calculated without the null terminator) exceeds the allocated size, the file name is not kept, that is, the name is displayed in the file list as “name not kept”. Files where names were not kept have all of file system functions working correctly, can open files, read and write, close, and so forth.
- File can be opened for either a read or a write; file cannot be read and written, correspondingly.
- During the programming or restore to factory process, no file operation can be executed; when trying to read or write a file, an error of SL_ERROR_FS_PROGRAMMING_IN_PROCESS is received. In this case, the file system function can be re-invoked after the programming process is finished.
- Trying to invoke a file system function when the file system is not yet formatted results in the error SL_ERROR_FS_NO_DEVICE_IS_LOADED.
- Some of the file system functionality is only available for secure devices; Table 7-2 describes the different functionality between secure and nonsecure devices.

Table 7-2. Secure Files

	CC3220S, CC3220SF, CC3120	CC3220R
Create encrypt programming image	+	–
Setting alarms threshold	+	–
Store the CC3220 host app as secure	+	–
Write secure user files	+	Only certificate files and private keys can be created secure. These files could not be read by the host.
Read secure user files	+	–
Secure programming	+	–

- The common way to create the service pack and the trusted root-certificate catalog is by the UniFlash Image Creator tool. The Image Creator tool supports the functionality of setting the correct creation attributes for the system files. If the following files are first created by the host, the described creation attributes should be used:
  - Trusted root-certificate catalog, name: /sys/certstore.lst ,maxsize : 7000 bytes. Creation flags: SL_FS_CREATE_SECURE |SL_FS_CREATE_PUBLIC_WRITE| SL_FS_CREATE_FAILSAFE
  - Service pack, name: /sys/servicepack.ucf ,maxsize : 131072 bytes. Creation flags: SL_FS_CREATE_SECURE |SL_FS_CREATE_PUBLIC_WRITE| SL_FS_CREATE_FAILSAFE

NOTE: When closing (after open for write) the service pack file or the trusted root-certificate catalog file, a signature must be supplied with a null certificate.

7.4 Write a File

7.4.1 Introduction

To write a file, the file must first be opened for write; at the end it should be closed. The following is a description of the writing procedure:

1. Open the file for create or write, function sl_FsOpen().
2. Write the file, function sl_FsWrite (); can be called several times.

3. Close the file, function sl_FsClose().

The next subsections provide detailed descriptions of the functions involved in the write file process.

7.4.2 Create a File versus Open for Write

The host provides different functions for creating a new file, or open for write an existing file. The following is a list of open file methods.

1. Open-Create-Write: By default, if the file does not exist, the device creates a new file and opens it for write; otherwise the device opens the file for write.

   Example:

```
&_i32             FileHdl;
unsigned char     DeviceFileName[180];
_u32              MaxSize, MasterToken;

FileHdl =  sl_FsOpen(unsigned char *)DeviceFileName,
SL_FS_CREATE|SL_FS_CREATE_SECURE |SL_FS_OVERWRITE |
SL_FS_CREATE_NOSIGNATURE | SL_FS_CREATE_MAX_SIZE( MaxSize ),
&MasterToken);
if(FileHdl < 0 )
{
  /*error */
}
```

2. Open-Create: Creates a new file; the open function returns an error if a file with the same name already exists.

   Example:

```
_i32              FileHdl;
unsigned char      DeviceFileName[180];
_u32               MaxSize, MasterToken;

FileHdl =  sl_FsOpen(unsigned char *)DeviceFileName,
               SL_FS_CREATE|SL_FS_CREATE_SECURE |
               SL_FS_CREATE_NOSIGNATURE | SL_FS_CREATE_MAX_SIZE( MaxSize ),
               &MasterToken);
if(FileHdl < 0 )
{
    /*error */
}
```

3. Open-Write: Opens an existing file for write; the open function returns an error if the file does not exist.

   Example:

```
_i32              FileHdl;
unsigned char    DeviceFileName[180];
_u32              MaxSize, MasterToken;


FileHdl =  sl_FsOpen(unsigned char *)DeviceFileName,
               SL_FS_OVERWRITE | SL_FS_CREATE_SECURE |
               SL_FS_CREATE_NOSIGNATURE | SL_FS_CREATE_MAX_SIZE( MaxSize ),
               &MasterToken);
if(FileHdl < 0 )
{
    /* error */
}
```

7.4.3 Create a File

The open-create or open-create-write creates a new file, and as part of the function the device allocates the storage for the file. The size of the allocated storage is determined by the maximum size parameter. The close function makes the opened file valid.

A file created but not closed has allocated storage (according its maximum size), but does not have a valid copy.

Because the process of creating a file involves updating the file allocation table on the SFLASH, TI recommends minimizing the creation of files. If it is required to update the file content, open the file for write rather than to delete and create it.

For a secure file, the default behavior of the file creation function is to generate the file tokens (including the master): the master token is returned as the function output, and all other tokens can be retrieved using the sl_FsGetInfo() function. Further information regarding the file tokens can be find in *CC3120/CC3220 SimpleLink Wi-Fi Devices Built-In Security Features* (SWRA509).

The sl_FsOpen() function tests that the file storage can be allocated, that the file does not exist, and that the creation flags are valid. If in error, the function returns a negative value and represents the error number; the following is a partial list of errors that might be returned by the creation function:

- SL_ERROR_FS_NOT_ENOUGH_STORAGE_SPACE, no available storage for the file
- SL_ERROR_FS_FILE_ALREADY_EXISTS, file with the same name already exists
- SL_ERROR_FS_NO_AVAILABLE_NV_INDEX, number of opened files exceeded
- SL_ERROR_FS_FILE_INVALID_FILE_SIZE, the maximum file size is set to 0

The create file input parameters are:

- Filename: The filename is a string of up to 180 bytes; TI recommends using short filenames (explained in *Section 7.3*); the file name is not case-sensitive.
- Maximum file size
  - When creating a file, the storage for the file is allocated according the requested maximum file size. For an existing file, the maximum file size cannot be changed; thus, when defining the maximum size of a file, the future growth of the file should be considered.
  - Creating a file with the FAILSAFE flag creates the file with a copy, thus the allocated storage size for the file is doubled.
  - Because the smallest erase unit of a SFLASH is 4096 bytes, the file system allocates storage size, which is aligned to 4096 bytes.
- File tokens: The token is the key for accessing a secure file; for a nonsecure file, it is set to zero. The file creation function returns the file token. By default, the device generates the file master token and returns it to the host. The default behavior of the token creation can be overridden by special creation flags.
- Creation flags: The creation flags are set during the file creation and cannot be changed afterward. The following is a list of creation flags:
  - SL_FS_CREATE_FAILSAFE: A file opened with failsafe has double copies, but only one copy is considered to be active at a time. Each time file is opened for write, the file storage is erased. If the system is powered off while writing a file with no failsafe, the file content is lost. If the system is powered off while writing a file with failsafe, the old content becomes the active one. Using the FAILSAFE doubles the file allocated storage. If the bundle feature (used for OTA) is used with the file, this flag is mandatory.
  - SL_FS_CREATE_SECURE: A file created as secure has its content encrypted on the SFLASH. Access to the file is limited, and requires a file token. See the security application notes file for more information about secure files.
  - SL_FS_CREATE_NOSIGNATURE: The flag is relevant only for secure files. By default, a secure file has a signature, which authenticates the file creator. See the security application notes for more information about how to create file signature.
  - SL_FS_CREATE_STATIC_TOKEN: Relevant only for secure files. This flag changes the default behavior of the file tokens creation: with this flag, the file tokens are not changed each time a file is opened for write.
  - SL_FS_CREATE_VENDOR_TOKEN: Relevant only for secure files. This flag changes the default behavior of the file tokens creation: with this flag, the file master tokens are set by the host.
  - SL_FS_CREATE_PUBLIC_WRITE: Relevant only for secure files. This flag changes the default behavior of the file tokens creation: with this flag, the file can be written without a token, but for a read operation a token is required.
  - SL_FS_CREATE_PUBLIC_READ: Relevant only for secure files. This flag changes the default behavior of the file tokens creation; with this flag, the file can be read without a token, but for a

write operation a token is required.

- Flags: The following flags are not creation flags, and can be set when creating or opening an existing file for write.
  - SL_FS_WRITE_BUNDLE_FILE: Used for the bundle commit feature; for new files, the FAILSAFE flag is not a precondition for this flag.
  - SL_FS_WRITE_ENCRYPTED: Used for secure content download.

If the application creates a file once, it can then be created by the Image Creator tool with the default content. The application can then update the file when required.

7.4.3.1 Secure File Creation Notes

When creating a secure file, the file resides encrypted on the SFLASH, and any access to the secure file requires a token. The default behavior is that the open for create function returns the master token of the file, the token is kept by the host application, and is then used for the file operation (read/write/delete).

To prevent a situation in which the host application was powered off before the received token is kept, use one of the following methods:

- Create the file with the SL_FS_CREATE_VENDOR_TOKEN flag and set the required token; in this way, the token is kept in the host application code.
- Create the file with the SL_FS_CREATE_PUBLIC_WRITE and SL_FS_CREATE_PUBLIC_READ flags; in this way, the secure file can be write/read without a token. To delete the file, a token is required, so this method is ideal for a file which is created once and never deleted.
- Combine both methods mentioned in the previous bullets; create a secure file with the vendor and public write and public read flags. In this case, no token is required for read and write, and deleting the file requires the vendor token.

7.4.3.2 Forced Creation Flags

For security reasons, some of the system files must be created with specific flag. Table 7-3 lists the files and their required creation flags.

Table 7-3. Creation Flags

Filename	CC3120, CC3220S, CC3220SF	CC3220R	Remark
/sys/servicepack.ucf /sys/certstore.lst	Secure signed by TI + public write + Fail-safe	Secure signed by TI	Those files are delivered by TI. The service pack contains fixes to the device code; the trusted root-certificate catalog contains the root CAs supported by TI and a revoked certificate list. TI might deliver a new version for those files when required. TI highly recommends designing the host to support future updates of these files.

/sys/mcuimg.bin
//CC3220R/CC3220S
/sys/mcuflashimg.bin
//CC3220SF
	Secure signed	Not secure	The file contains the host program.

/sys/cert/private.key /sys/cert/client.der /sys/cert/ca.der	Secure	Secure, blocked for read	The files contain the key and certificate for SSL connection.

NOTE: Downgrading the trusted root-certificate catalog is not possible.

NOTE: The service pack is a special file which already contains the signature. When writing the service pack by the host, the sl_FsClose() function should receive a NULL certificate name and a NULL signature.

7.4.4 Open a File for Write

Opening an existing file for write is the preferred way to update the file content (rather than to delete it and recreate it). A file which was not closed or aborted cannot be opened for write.

Appending content to an existing file is not supported; when the file is opened for write, the storage of the file is erased.

If the file was created with a FAILSAFE flag, the storage of the nonactive content is erased; thus, if the device was powered off before the file closure, the file contains the last valid content.

If the file was created without a FAILSAFE flag, powering off the device before the file is closed results in no valid copy.

The operation of opening a file for write does not involve updating the file allocation table, unlike the create file function.

For a secure file, the default behavior is that when opening a file for write, all the tokens except the master are regenerated. The new tokens can be retrieved by the sl_FsGetInfo function.

If the file is already open (for write or read) or does not exist, sl_FsOpen() returns an error.

The following is a partial list of errors that might be returned by the creation function:

- SL_ERROR_FS_FILE_IS_ALREADY_OPENED: The file is already opened for read or write.
- SL_ERROR_FS_INVALID_TOKEN_SECURITY_ALERT: For a secure file, the input token is not valid; this triggers security alerts if the device is secured.
- SL_ERROR_FS_FILE_NOT_EXISTS: The file does not exist.

The following parameters are required to create a file:

- Filename: The filename is not case-sensitive.
- File tokens: The token is the key for accessing a secure file; for nonsecure files it is set to zero. For a secure file with no public write permission, the host should supply a token with write permission (master token, write token, or read-write token).
- The function returns a token on the same permission level as the input one.
- Flags: The noncreation flags that can be set:
  - SL_FS_WRITE_MUST_COMMIT – used for the file commit feature, and can be used only if the file was created with a FAILSAFE flag.
  - SL_FS_WRITE_BUNDLE_FILE – used for the bundle commit feature, and can be used only if the file was created with a FAILSAFE flag.
  - SL_FS_WRITE_ENCRYPTED – used for secure content delivery.

7.4.5 Write an Opened File

The host can invoke the write command for each file opened for write. For nonsecure files, the write command can be done to random offsets. For secure files, the write operation also encrypts the file, thus writing secure files requires writing to sequential offsets, or writing to a 16-byte-aligned offsets buffer, which is also 16-byte-aligned in size.

The file system sets the actual file size as the higher offset that was written; the actual file size can be retrieved by the sl_FsGetInfo() function.

The return value of the write function is the number of bytes written, and a negative value is an error number.

The following is a partial list of errors that might be returned by the write function:

- SL_ERROR_FS_OFFSET_OUT_OF_RANGE: The file can be written to offsets which are less than the maximum file size, and trying to write a file to an offset which exceeded the maximum file size results in an error.
- SL_ERROR_FS_INVALID_HANDLE: The input file handle is illegal.
- SL_ERROR_FS_OFFSET_NOT_16_BYTE_ALIGN: For a secure file, when trying to write to an offset, this is not sequential.

- SL_ERROR_FS_FILE_ACCESS_IS_DIFFERENT: Trying to read a file which was opened for read.

Example:

```
_i32              FileHdl;
_i32              Status;
_u32              Offset = 0;
unsigned char     pData[100];
_u32              Len = 0;

Status = sl_FsWrite( FileHdl, Offset, pData, Len );
if( Status < 0 )
{
    /* error */
    /* abort */
    sl_FsClose(FileHdl,0,'A',1);
}
```

7.4.6 Close an Opened (for Write) File

Closing or aborting an opened file is mandatory. Closing the file frees the file resources from the device memory and sets the last file copy (if one exists) as the active one.

If the host application decides not to write an opened file due to an error, use abort instead of close.

The abort function requires the file handle. How to abort a file without using the file handle is described in the file commit-rollback function.

A file that was not closed or aborted appears in the file system; its storage is allocated, but it might have no valid copy. Such files can be observed by the SL_FS_INFO_NOT_VALID flag, and the file flags can be retrieved by the sl_FsGetFileList () or sl_FsGetInfo () functions.

For a file opened with the FAILSAFE flag that has a valid copy (from a previous write operation), invoking the abort function sets the nonactive copy as the active one, so that the next read operation reads the valid copy and not the one that has been aborted.

In the case of an unexpected or sudden shutdown, each file opened for write that has not been closed is treated as if abort has been called for this file.

The function returns 0 for success, and a negative number for an error.

The following is a partial list of errors that might be returned by the close after write function:

- SL_ERROR_FS_INVALID_HANDLE: The input file handle is illegal.
- SL_ERROR_FS_CERT_CHAIN_ERROR_SECURITY_ALERT: For a secure signed file, testing the certificate chain of trust failed, and a security alert is triggered.
- SL_ERROR_FS_CERT_IN_THE_CHAIN_REVOKED_SECURITY_ALERT: For a secure signed file, the certificate chain of trust exists in the revoked list, and a security alert is triggered.
- SL_ERROR_FS_WRONG_SIGNATURE_SECURITY_ALERT: For a secure signed file, the signature test failed, and a security alert is triggered.
- SL_ERROR_FS_WRONG_CERTIFICATE_FILE_NAME: For a secure signed file, if one of the certificates in the chain of trust is missing, it does not trigger security alerts.

Close nonsigned file example:

```
_i32              FileHdl;
_i16              Status;
const _u32        SignatureLen;
_u8*              pSignature, pCeritificateFileName;

pCeritificateFileName = 0;
pSignature = 0;
SignatureLen = 0;
Status = sl_FsClose(FileHdl,pCeritificateFileName,pSignature,SignatureLen);
if( Status < 0 )
{
    /* error */
```

```
    /* abort */
    sl_FsClose(FileHdl,0,'A',1);
}
```

Abort file example:

```
_i32             FileHdl;
_i16             Status;
const _u8        Signature;
const _u32       SignatureLen;
_u8*             pCeritificateFileName;

pCeritificateFileName = 0;
Signature = 'A';
SignatureLen = 1;
Status = sl_FsClose(FileHdl,pCeritificateFileName,Signature,SignatureLen);
if( Status < 0 )
{
    /*error */
}
```

7.4.7 Close an Opened (for Write) Secure-Signed File

To create a file signature:

1. The vendor generates a public and private RSA key pair, supported: RSA 128 or 256 bytes, the generated files are of public.pem and private.pem.
2. A known CA creates a signed certificate which contains the public key.
3. Using the private key, the file digital signature is generated. The signature is a standard digital signature; the algorithm first calculates the SHA of the file content and then the SHA result is encrypted using the private key. The supported signature types are: PKCS#1, RSA 256 or 128 bytes, SHA_1 (the signature length is 256 or 128 bytes). The signature for the file can be created by standard tools, or by the UniFlash Image Creator tool (using the private key).
4. The close function receives the file signature as an input, and the signed certificate (in DER encoding).

Notes:

- All the chained certificates should exist in the SFLASH when the close function is called.
- The supported encoding for the certificate files is DER.
- The supported certificates are:
  - RSA 1024 to 4096
  - SHA 1-512
- The signed certificates filename should be created in the device with the name as it appears under the “issued to” property of the certificate (the exact name should be given).
- The trusted root-certificate catalog delivered by TI contains the list of supported and revoked certificates. For a list of supported CAs, see the security application document.

Example for secure-signed files close function:

```
_i32             FileHdl;
_i16             Status;
const _u8        CeritificateFileName[180];
const _u8        Signature[256];
const _u32       SignatureLen;

SignatureLen = sizeof(Signature);
Status = sl_FsClose(FileHdl, CeritificateFileName, Signature, SignatureLen);
if( Status < 0 )
{
    /* error */
    /* abort */
    sl_FsClose(FileHdl,0,'A',1);
}
```

7.5 Read a File

To read a file, it should first be opened for read. The following functions are involved in the read file procedure:

- Open-read a file, function sl_FsOpen()
- Read the file, function sl_FsRead()
- Close the file, function sl_FsClose()

7.5.1 Open a File for Read

Open a file for read only succeeds if the file has been closed or aborted. The open-for-read function does not involve any SFLASH updates, and it has no effect on the SFLASH endurance.

The open-for-read function returns a negative value in case of an error.

The following is a partial list of errors that might be returned by the open-for-read function:

- SL_ERROR_FS_FILE_IS_ALREADY_OPENED: The file is already opened for read or write.
- SL_ERROR_FS_INVALID_TOKEN_SECURITY_ALERT: For a secure file, the input token is not valid; this triggers security alerts if the device is secured.
- SL_ERROR_FS_FILE_NOT_EXISTS: The file does not exist.
- SL_ERROR_FS_DEVICE_NOT_SECURED: Reading the secure file can only be done in a secure device type.
- SL_ERROR_FS_WRONG_SIGNATURE_SECURITY_ALERT: For secure-signed files, each time the file is opened for read, the file integrity is tested. If the test is failed, an error and a security alert is raised.

Example:

```
_i32              FileHdl;
_u8               DeviceFileName[180];
_u32              MasterToken;

FileHdl =  sl_FsOpen(unsigned char *)DeviceFileName, SL_FS_READ, &MasterToken);
if( FileHdl < 0 )
{
    /*error */
}
```

7.5.2 Read an Opened File

To read a file, the host requires the file handle, the offset to read, and the output buffer. A file can be read from random offsets.

The sl_FsRead returns the actual bytes read, or a negative value which represents an error.

The following is a partial list of errors that might be returned by the open-for-read function:

- SL_ERROR_FS_OFFSET_OUT_OF_RANGE: The file system set the actual file size as a higher offset than was written. Trying to read a file from an offset which is higher than the actual file size results in an error.
- SL_ERROR_FS_NO_MEMORY: The read operation requires a system resource (RX-socket). The system may return that if there are not available resources (sockets) for the operation; in this case, the host can repeat the read operation after a while (or reduce the traffic overload while reading a file).

Example:

```
_i32              FileHdl;
_i32              Status;
_u32              Offset = 0;
unsigned char     pData[100];
_u32              Len = 0;

Status = sl_FsRead( FileHdl, Offset, pData, Len );
if( Status < 0 )
```

```
{
    /*error */
    /* abort */
    Status = sl_FsClose(FileHdl,0,'A',1);
}
```

7.5.3 Close an Opened (for Read) File

When the read is completed, the host must close the file. Closing the file releases the file resources from the device memory.

An abort file command can be invoked without using the file handle; the reference can be found in the file commit-rollback function.

An abort file command open for read has the same functionality as the close function. In case of an unexpected or sudden shutdown, each file opened for read that is not closed is treated as if it is aborted.

The following is a partial list of errors that might be returned by the close after read function:

- SL_ERROR_FS_INVALID_HANDLE: The input file handle is illegal.

Close file example:

```
_i32             FileHdl;
_i16             Status;
const _u32       SignatureLen;
_u8*             pSignature, pCeritificateFileName;

pCeritificateFileName = 0;
pSignature = 0;
SignatureLen = 0;
Status = sl_FsClose(FileHdl,pCeritificateFileName,pSignature,SignatureLen);
if( Status < 0 )
{
    /* error */
    /* abort */
    sl_FsClose(FileHdl,0,'A',1);
}
```

Abort file example:

```
i32             FileHdl;
_i16             Status;
const _u8        Signature;
const _u32       SignatureLen;
_u8*             pCeritificateFileName;

pCeritificateFileName = 0;
Signature = 'A';
SignatureLen = 1;
Status = sl_FsClose(FileHdl,pCeritificateFileName,Signature,SignatureLen);
if( Status < 0 )
{
    /*error */
}
```

7.6 Delete a File

Deleting of a file removes its storage from the file system and updates the files allocation table. Deleting is done by the host function sl_FsDel().

On successful delete, the file allocation storage on the SFLASH is removed, and can be used by other files. For secure files, the delete requires the file master token.

TI recommends reducing the delete file operations to the minimum, because it involves updating the allocation table.

The following is a partial list of errors that might be returned by the delete function:

- SL_ERROR_FS_FILE_IS_ALREADY_OPENED: File that is opened cannot be deleted. The file is

expected to be closed or aborted to be deleted; trying to delete a file opened for write/read results in this error.

- SL_ERROR_FS_FILE_NOT_EXISTS: Trying to delete a file which does not exist results in this error.

Example:

```
_i16             Status;
_u8              DeviceFileName[180];
_u32             MasterToken;

Status = sl_FsDel( DeviceFileName, MasterToken );
if( Status < 0 )
{
    /*error */
}
```

7.7 Rename a File

This function renames an existing file; for secure files, the rename requires the master token.

The following is a partial list of errors that might be returned by the rename function:

- SL_ERROR_FS_FILE_IS_ALREADY_OPENED: File that is opened cannot be renamed. The file is expected to be closed or aborted to be renamed.
- SL_ERROR_FS_FILE_NAME_RESERVED: System file cannot be renamed, and file cannot be renamed to a system filename.
- SL_ERROR_FS_FILE_NAME_EXIST: Renaming a file to a filename that already exists results in an error.

Example:

```
_i32             Status;
_u8              DeviceFileName[180], NewDeviceFileName[180];
_u32             Token;

Status = sl_FsCtl(SL_FS_CTL_RENAME, Token, DeviceFileName, NewDeviceFileName, 0, NULL, 0, NULL );
if( Status < 0 )
{
    /*error */
}
```

7.8 File System Helper Functions

Some functions are used for observing the file system state. This section describes those functions.

7.8.1 Get File Information

The function retrieves information regarding a specific file. For secure files, the file requires a token. For secure files where the input token is zero, only a part of the information of the file is retrieved, because the tokens and the creation token flags are hidden.

Trying to request information for a file that does not exist results in an error (FILE_NOT_EXISTS). Trying to request information for a file which has no valid copy results in retrieving the file information, but the return value will be an error (SL_FS_INFO_NOT_VALID_EXISTS).

Example:

```
_i16             Status;
_u8              DeviceFileName[180];
_u32             Token;
SlFsFileInfo_t   FsFileInfo;

Status = sl_FsGetInfo( DeviceFileName, Token, &FsFileInfo);
if( Status < 0 )
{
    /*error */
}
```

7.8.2 Get Storage Information

This function retrieves information about the storage usage and the file system state. The function output contains information regarding the number of security alerts, the number of allocated files user/system, the configured storage size, the format type, and so forth.

Example:

```
_i32                                    Status;
SlFsControlGetStorageInfoResponse_t     GetStorageInfoResponse;

Status = sl_FsCtl( ( SlFsCtl_e)SL_FS_CTL_GET_STORAGE_INFO, 0, NULL , NULL , 0, (_u8
*)&GetStorageInfoResponse, sizeof(SlFsControlGetStorageInfoResponse_t), NULL );
if( Status < 0 )
{
    /*error */
}
```

7.8.3 Get List of Files

Retrieves the file list, names, and their main attributes, and observes how many blocks (= subsector, 4096 bytes) each file occupies on the SFLASH.

This function is an iterative function; the host retrieves an iterator that can be used to retrieve the next bulk of files.

Example:

```
_i32 NumOfEntriesOrError = 1;
_i32 Index = -1;
slGetfileList_t File[COUNT];
_i32  i;
_i32 Status = 0;

while( NumOfEntriesOrError > 0 )
{
    NumOfEntriesOrError = sl_FsGetFileList( &Index, COUNT, (_u8)(SL_FS_MAX_FILE_NAME_LENGTH +
         sizeof(SlFileAttributes_t)), (unsigned char*)File, SL_FS_GET_FILE_ATTRIBUTES);
    if (NumOfEntriesOrError < 0)
    {
            Status = NumOfEntriesOrError;//error
            break;
    }
    for (i = 0; i < NumOfEntriesOrError; i++)
    {
        /* print
        File[i].fileName
        File[i].attribute.FileAllocatedBlocks
        File[i].attribute.FileMaxSize,(_u16)File[i].attribute.Properties
            */
    }
}
return Status;//0 means O.K
```

7.9 Bundle Protection

Bundling changes content to a group of files and then accepts or rejects the changes for all the files in the group simultaneously. The bundle is used by the OTA process, which downloads a group of files and needs the ability to first test the files and then to accept or reject the downloaded content.

Table 7-4 shows a common work flow of using the bundle.

Table 7-4. Bundle Protection

Step	Operation	System State After Operation
1	Create (with failsafe flag), write, close n files	Bundle STOPPED

Table 7-4. Bundle Protection (continued)

Step	Operation	System State After Operation
2	For each of the n files, open the file for write with the bundle flag, write the file and close it.	Bundle STARTED, reading the files results in their old copy. Before the files closure they are on state OPEN_BUNDLE_COMMIT. After the file closure they are in state PENDING_BUNDLE_COMMIT.
3	Call Sl_Stop(X>0) and Sl_Start(). On this step, the host tests the system, to make sure that the downloaded content is functioning as expected.	Bundle state is PENDING_COMMIT, Reading the files results in their new copy. Each n file is in PENDING_BUNDLE_COMMIT state.
4	In case the system test passed successfully, the host approves the bundle (=call to bundle commit), else initiate rollback of the files (=call the bundle rollback).	Bundle state is STOPPED. Rollback of the bundle files makes their old copy the valid one. Commit of the bundle files makes their new copy the valid one.
	In case of power failure before the content approval (= failure during Step 2 or 3), the device automatically calls the bundle rollback.	Bundle state is STOPPED. Reading the files results in their old copy.

7.9.1 Bundle File States

To update a file as part of a bundle, the file should be opened for write with the bundle flag (SL_FS_WRITE_BUNDLE_FILE). Open a new file as part of a bundle that has no pre-conditions.

To open an existing file as part of a bundle, the file should fulfill the following conditions:

- The file was created with the FAILSAFE flag.
- The file has a valid copy (meaning that the file was successfully written at least once).

The device manages the state of a file. The file state can be viewed by retrieving the file flags, and the file flags can be retrieved by the FsGetFileList() function and the sl_FsGetInfo() function.

The following is a list of the possible file states.

- Standard: A file which is not part of a bundle.
  - If a bundle is committed or rolled back, the bundle files state is changed to standard file.
  - A file opened as a bundle file, but was aborted instead of being closed, changes its state to standard file.
- SL_FS_INFO_BUNDLE_FILE
  - The file is currently open with the bundle flag, but has not been closed yet.
  - If the host invokes abort instead of close, the file state changes to standard file.
- SL_FS_INFO_PENDING_BUNDLE_COMMIT
  - The file is currently open with the bundle flag, but has been closed.
  - The file in this state cannot be opened for write until the bundle is committed or rolled back. Trying to open this file for write results with the error: SL_ERROR_FS_FILE_IS_PENDING_COMMIT.
  - The file on this state can be opened for read; the file copy that will be read is depended on the bundle state.

Table 7-5 is a summary of the possible file states related to the bundle state (the bundle states are described in the following section).

Table 7-5. Bundle States

Bundle State	Possible Files State
STOPPED	All files are in normal state
STARTED	• Normal • BUNDLE_FILE • PENDING_BUNDLE_COMMIT
PENDING_COMMIT	• Normal • PENDING_BUNDLE_COMMIT

7.9.2 Bundle States

The bundle can be in one of three states:

- STOPPED
- STARTED
- PENDING_COMMIT

The following subsections describe the various bundle states.

7.9.2.1 STOPPED

No bundle exists.

7.9.2.2 STARTED

The bundle changes its state to STARTED when the first bundle file is opened for write.

In this state, the host is writing the bundle files and keeps the order of the files updated (a certificate should be written before the file that uses it is closed). Opening the files that belong to a bundle in the STARTED state for read results in the content of the old file copy.

Transition from this state to the PENDING_COMMIT state is executed if all the following conditions are fulfilled:

- sl_Stop (x > 0) and sl_Start() is called.
- All the bundle files are in the PENDING_BUNDLE_COMMIT state.

Transition from this state to the STOPPED state is executed if the following condition is fulfilled:

- sl_Start() is called without calling sl_Stop (x > 0). In this scenario, the bundle is automatically rolled back by the device, or the rollback function was invoked by the host.

7.9.2.3 PENDING_COMMIT

This state is used to enable the host to run test code to decide if the downloaded bundle files are working as expected.

While in this state, open files for read returns the content of the new files copy.

While in this state, files cannot be opened for write with the bundle flags; in such cases, the device returns an error: SL_ERROR_FS_BUNDLE_NOT_IN_CORRECT_STATE.

Transition from this state to the STOPPED state is executed if one of the following scenarios is fulfilled:

- On a successful host test, the host invokes the commit bundle function: sl_FsCtl (SL_FS_CTL_COMMIT...).
- On a failed host test, the host resets the device (hibernate or POR) or invokes the rollback bundle: sl_FsCtl (SL_FS_CTL_ROLLBACK...), after calling the command reboot is required.
- The sl_Start() function is called; automatic rollback is triggered by the device.

7.9.3 Commit a Bundle

Committing the bundle approves all the files which belong to the bundle. At the end of the process, all the bundle files are in the standard file state, and the bundle state is in STOPPED state. In addition, the newly downloaded content of the files becomes the active one.

The commit process is fail-safe; that is, if the device has been shut down during the bundle commit procedure, on power-up the device automatically continues the bundle commit process.

7.9.4 Rollback a Bundle

Rolling back the bundle rejects all the new bundle file content. At the end of the process, all the bundle files are in the standard file state, and the bundle state is in STOPPED state. In addition, the newly downloaded content of the files is ignored, and the old copy becomes the active one.

The rollback process is fail-safe; that is, if the device has been shut down during the bundle rollback procedure, on powerup the device automatically continues the bundle rollback process.

7.9.5 Retrieve the Bundle and Files State

To view the current state of the bundle, use the function sl_FsCtl (SL_FS_CTL_GET_STORAGE_INFO...).

To retrieve the bundle state of a specific file, use the sl_FsGetFileList() function or the sl_FsGetInfo() function.

7.9.6 CC3220 Bundle Aspects

The M4 program can be created as a bundle file.

When the bundle is in the PENDING_COMMIT state, hardware WDT is automatically activated (if configured in the mcubootinfo.bin). If the WDT expires, automatic reboot is triggered and the bundle files are automatically rolled back.

When invoking the commit function, there are two options:

- Continue the session as it is. In this case, there is no need to stop the WDT. Use PRCMPeripheralReset(PRCM_WDT).
- Do a clean reboot. In this case, the recommended way is to use PRCMHibernateCycleTrigger(). This also stops the WDT (this method should be also used after rollback).

When invoking rollback (when the bundle is in the pending commit state), a clean reboot is required (PRCMHibernateCycleTrigger()).

NOTE: To configure the WDT, set the mcubootinfo file.

The WDT resets (hibernate-reset) the system after two time-out events.

BootInfo.ulStartWdtTime is a 32-bit field that contains the number of clock ticks; because the WDT runs at 80 MHz, the maximum time-out possible is approximately (53 sec × 2)

Example of how to set the WDT for the CC3220 device:

```
#define APPS_WDT_START_KEY          0xAE42DB15

typedef struct sBootInfo
{
  _u8   ucActiveImg;
  _u32  ulImgStatus;
  _u32  ulStartWdtKey;
  _u32  ulStartWdtTime;
}sBootInfo_t;

_sBootInfo_t       sBootInfo;
_u32              MasterToken = 1234;
i32               FileHdl;
_i32              Status;
```

```
_i16             StatusClose;

//Open file "/sys/mcubootinfo.bin" for write
FileHdl =  sl_FsOpen((unsigned char *)("/sys/mcubootinfo.bin",
               SL_FS_CREATE|SL_FS_OVERWRITE |
               SL_FS_CREATE_SECURE | SL_FS_CREATE_PUBLIC_WRITE |
               SL_FS_CREATE_NOSIGNATURE | SL_FS_CREATE_VENDOR_TOKEN
               SL_FS_CREATE_MAX_SIZE(sizeof(sBootInfo)), &MasterToken);
if(FileHdl < 0 )
{
    /*error */
    /* abort */
    sl_FsClose(FileHdl,0,'A',1);
}
memset(&sBootInfo,0,sizeof(sBootInfo_t));
sBootInfo.ulStartWdtKey = APPS_WDT_START_KEY;
sBootInfo.ulStartWdtTime = 80000000;
Status = sl_FsWrite(FileHdl,0,(_u8*)&sBootInfo, sizeof(sBootInfo_t));
if( Status < 0 )
{
    /*error */
}
StatusClose = sl_FsClose(FileHdl,0,0,0);
if( StatusClose < 0 )
{
    /*error */
    /* abort */
    sl_FsClose(FileHdl,0,'A',1);
}
```

7.10 File Commit Feature

The file commit feature updates a single file and then commits it or rolls it back. A file opened with the commit flag that was closed is blocked for write operations.

The file is blocked until it is committed or rolled back.

File rollback can also be invoked on files opened without the commit flag.

To open a file with the commit flag, the file should fulfill the following requirements:

- The file was created with the FAILSAFE flag.
- The file has a valid copy (meaning that the file was successfully written at least once).

The following is a common file-commit work flow:

1. Create a file, write, and close.
2. Open the file for write with the commit flag (SL_FS_WRITE_MUST_COMMIT), write the file, and close.
3. The host tests the system: if the test passed successfully, the host commits the file, else rolls it back.

The following is a description of the file-commit states:

- Standard file
- SL_FS_INFO_MUST_COMMIT: The file was opened for write with the commit flag, and has not been closed.
- SL_FS_INFO_PENDING_COMMIT: The file was opened with the commit flag and has been closed. The file is waiting for the host to invoke the file commit or rollback operation. A file in this state cannot be opened for write. A file in this state can be opened for read; the file image that is read is the latest image. If the file is committed or rolled back, the file state is changed to standard file.

7.10.1 File Commit Process

Committing a secure file requires a file token with at least write permission.

Committing a file approves the new content of the file; at the end of the process, the file state is changed to standard file.

The function interface for committing a file is sl_FsCtl (SL_FS_CTL_COMMIT).

7.11 File Rollback Process

Rolling back a secure file requires a file token with write permission.

Rolling back the file makes the old file copy the active one; at the end of the process, the file state is changed to standard file.

The function interface for rolling back a file is sl_FsCtl (SL_FS_CTL_ROLLBACK..). For secure files, the rollback also rolls back the file tokens.

Rolling back files that are currently in the standard file state acts as file abort, but with the filename as input rather than the file handle.

7.12 Programming

For a fast and smooth production line, the SimpleLink Wi-Fi device offers a programming interface. This process involves two major steps:

- Creation of the programming image
- Programming the device with the image

The same image can be used to program many devices. At the end of the programming process, the device is configured and contains the packed files.

7.12.1 Creation of the Programming Image

The programming image is a packed file which contains the service pack, the system configuration files, the user files, and the host program (for the CC3220 device).

The process of creating the programming image is an offline process.

For creating the programming image, the SimpleLink Wi-Fi package contains the Image Creator tool, a web application (it also has CLI interface) which lets the user easily create the programming image and supports programming the device.

The UniFlash Image Creator tool creates three types of files:

- .sli, the file format is TI proprietary structure and is used for the Image Creator tool and host programming.
- .ucf, same as the s.li, used for the host programming
- .bin, the standard binary file for external flash programming
- .hex, the standard Intel Hex file for external flash programming

The programming image can be created as encrypted (AES-CTR 128); the key used for the encryption is supplied during the image programming.

7.12.1.1 Programming Image Types

The programming image can be defined (by the Image Creator) as one of the following types:

- Production
- Development

The development image is intended for development and debugging; the development image can be created only for a specific device (by using its MAC address). For a device programmed with the development image, the Image Creator tool can retrieve the file list from the device and edit the files (online). For the CC3220S/CC3220SF device, the development mode also enables the JTAG interface.

7.12.1.2 Program the Device

The image can be programmed to the device using one of the following methods:

- Image Creator tool (by UART lines)

- Host (not relevant to the CC3220 device)
- External (third party) flash programmer

The programming time depends on the image size and the SFLASH type. During the programming, no file operation can be executed; trying to read or write a file results in an error of SL_ERROR_FS_PROGRAMMING_IN_PROCESS.

The programming involves two internal steps:

1. Download the image.
2. Extract the image packed file.

7.12.1.2.1 Image Creator Tool (UART) Programming

The UniFlash Image Creator tool is a method of programming that uses the device UART interface. The tool is the common programming method.

The device starts the extraction procedure when the last portion of the image file is received.

At the end of the programming method, a success status is returned to the caller, and the device is operational.

7.12.1.2.2 Host Programming

The host has an interface function for programming the device. For the CC3220/R/F/S devices, the host programming method is not applicable (because the host application is part of the programming image).

The programming file used by the host interface is created by the UniFlash Image Creator tool.

For non-secure programming, use the file programming.ucf. For secure programming, use the file programming.encrypt.ucf.

```
_i32  Status
_u8   DataBuf[1000];
_u16  Len;
_u8   Key[16];

Status = sl_FsProgram ((const _u8*)DataBuf , Len , &Key ,  0 );
if( Status < 0 )
{
    /*error */
}
```

For programming, the function receives the .ucf file and the image key (or null key). The device starts the extraction procedure when the last portion of the file is received. The function returns after the extraction is complete.

When the function completes, a hibernate-reset is required (sl_Stop, sl_Start).

7.12.1.2.3 External Tool Programming

Programming the SFLASH by an external tool requires using the .bin or .hex programming files. The programming files used by the external tool are created by the UniFlash Image Creator tool.

For non-secure programming, use the programming.bin/.hex. For secure programming, use the programming.encrypt.bin/.hex.

The .bin and .hex (Intel Hex) files are standard file formats.

At the end of the programming process, the device is operational and configured.

The external programming steps are:

1. Program the SFLASH with the third-party programming tool.
   - Important: The entire SFLASH should be erased before programming the image. The extraction process considers that the SFLASH is erased, except to the programming image storage.
2. Assemble the SFLASH to the device.

3. For a nonsecure image:
   a. For the CC3220/R/S/F devices: set the SOP to 000.
   b. POR the device.
   c. The image-extracting process is started automatically by the device.
4. For a secure image (secure image can be created for the CC3120, CC3220S, CC3220F) :
   a. Set the SOP to UART-programming mode (010).
   b. Set the encryption key using the Image Creator tool (after setting the key, it resets the device). The device extracting process starts automatically. Setting the encryption key is done by the Image Creator tool using the UART interface.
   c. For the CC3220/S/F device: set the SOP to 000
   d. POR the device.

NOTE: If the device was POR during the extracting process (after setting the SOP to 000 in Step 3.a. or 4.c.), the process continues and the device automatically triggers an additional hibernate-reset.

7.13 Restore to Factory

The SimpleLink Wi-Fi device has an internal recovery mechanism, in which the level of recovery can be set by the Image Creator tool; it is kept as part of the programming image.

Three levels of recovery are supported:

- None – no recovery level is supported.
- Restore-to-factory default
- Restore-to-factory image
- Restore-to-factory image using SOP, but only if the restored-to-factory image is enabled

If one of the recovery methods is enabled, the programming image is kept on the SFLASH, and the recovery process uses the kept programming image.

Restore to factory image procedure:

- All the files are rolled back to the image configuration; files that do not exist in the image are deleted.
- Can be invoked by the host, or by SOP

Restore-to-factory default:

- All the files are rolled back to the image configuration, except the service pack and the host program (CC3220 host).
- Can be invoked by the host

The process of restore-to-factory is fail-safe: the process has two stages:

- Preparation, which takes about 0.3 seconds.

  If the device is being reset during this stage, the file system will not change.
- Extraction depends on the vendor programming image size and the SFLASH type.

  If the device was reset during the extraction, the extracting process continues when the device is powered up.

NOTE: If the Wi-Fi calibration mode is configured as one time, the Wi-Fi calibration is not regenerated when invoking the restore-to-factory-default method (the current calibration is used).

While the restore-to-factory operation is in process, the networking subsystem is in lock state. Most of the functions are blocked and will return an error SL_ERROR_INCOMPLETE_PROGRAMMING.

The only procedure requiring a SOP setting of 010 is the UART-programming.

For the CC3220/R/S/F device, after the programming stage the SOP setting should be 000, because it has implication on the restore-to-factory function.

For the CC3220/R/S/F device, if the SOP is configured as UART-programming mode, which is 010(SOP2=1, SOP1=0, SOP0=0), the following scenario may occur:

1. Restore to factory is called.
2. Device-reset occurs during the restore-to-factory extraction stage.
3. The device is halted.
   - The following steps overcome the halt situation:
     a. Set the SOP to 000.
     b. POR the device, and the restore to factory is completed successfully.

7.13.1 Restore to Factory by the Host

To trigger the restore-the-factory image by SOP, the following steps are required:

1. From the host, invoke the restore function as in the following example. The function is synchronous; it returns when the process is finished.
2. Hibernate-reset the device:
   - CC3120: sl_Stop, sl_Start
   - CC3220/R/S/F: sl_Stop, PRCMHibernateCycleTrigger()

Examples:

```
_i32 slRetVal;
SlFsRetToFactoryCommand_t RetToFactoryCommand;
_i32 Status, ExtendedStatus;

 RetToFactoryCommand.Operation = SL_FS_FACTORY_RET_TO_IMAGE;
 Status = sl_FsCtl( (SlFsCtl_e)SL_FS_CTL_RESTORE, 0, NULL , (_u8 *)&RetToFactoryCommand ,
sizeof(SlFsRetToFactoryCommand_t), NULL, 0 , NULL );
 if ((_i32)Status < 0)
 {
     /*error*/
     //Status is composed from Signed error number & extended status
     Status = (_i16)Status>> 16;
     ExtendedStatus = (_u16)slRetVal& 0xFFFF;
     break;
}
//Reset
sl_Stop(10);
sl_Start(NULL, NULL, NULL);
```

```
_i32 slRetVal;
SlFsRetToFactoryCommand_t RetToFactoryCommand;
_i32 Status, ExtendedStatus;

 RetToFactoryCommand.Operation = SL_FS_FACTORY_RET_TO_DEFAULT;
 Status = sl_FsCtl( (SlFsCtl_e)SL_FS_CTL_RESTORE, 0, NULL , (_u8 *)&RetToFactoryCommand ,
sizeof(SlFsRetToFactoryCommand_t), NULL, 0 , NULL );
 if ((_i32)Status < 0)
 {
     /*error*/
```

```
        //Status is composed from Signed error number & extended status
        Status = (_i16)Status>> 16;
        ExtendedStatus = (_u16)slRetVal& 0xFFFF;
        break;
    }
    //Reset
    sl_Stop(10);
    sl_Start(NULL, NULL, NULL);
```

7.13.2 Restore to Factory by Using the SOP

To trigger the restore the factory image by SOP, follow the methods outlined in the following sections.

7.13.2.1 CC3120

The recommended method for the CC3120 device is to invoke the restore to factory by the host, because restore to factory by the SOP requires POR (see Step 1) and it is not a good practice to reset the device while the host is still running.

The following steps are required:

1. Set the SOP to 011(SOP2=0, SOP1=1, SOP0=1) and perform POR (power on reset). This initiates the restore to factory operation.
2. Revert the SOP to 000.
3. When the restore-to-factory operation is done, the device sends Init complete event with a LOCKED + Factory restored indication.
4. Perform a POR to clear the SOP indication in the host.

   If the user pressed the POR during Step 3, the restore process continues and the device automatically triggers an additional hibernate-reset when finished.
5. The device sends an Init complete event to the host.

7.13.2.2 CC3220/R/S/F

The recommended method for the CC3220 device is to invoke the restore to factory by the host, because restore to factory by the SOP requires POR (see Step 1) and it is not a good practice to reset the device while the host is still running.

The following steps are required:

1. Set the SOP to 011(SOP2=0, SOP1=1, SOP0=1) and perform POR. This initiates the restore to factory operation (the LaunchPad has a button for it).
2. Revert the SOP to 000 (the host SOP indication is not yet cleared).
3. When the restore to factory operation is done, the device initiates a hibernate-cycle and the host application starts.
4. When the host program detects the SOP indication, the host program requests the user to POR (it must not call sl_Start beforehand).
5. Perform a POR to clear the SOP indication in the host.

   If the user pressed the POR during Step 3, the restore process continues and the device automatically triggers an additional hibernate-reset when finished.

NOTE: While the restore to factory function is in process, the networking subsystem is in lock state.

CC3220: On Step 4, detection of the SOP indication(011) : (HWREG(0x4402FC18) & 0x80)

CC3220: If the user sets the SOP to 010 in Step 2, the scenario described in Step 5 will not work.

7.14 Security Alerts

The SimpleLink Wi-Fi device provides a software tamper detection procedure with a security-alert counter. This procedure detects an integrity violation of file system data, the content of a secure-authenticate file, and system files. This procedure also detects unauthorized operations, such as trying to read a secure file with an invalid token.

When detecting data tampering, the device data-tampering procedure increases the system security-alert counter, and when the system reaches the security-alert (configured) threshold, the device is locked. In addition, the host receives a lock asynchronous event (SL_ERROR_DEVICE_LOCKED_SECURITY_ALERT), and each call from the host to a file system interface results in SL_ERROR_FS_FILE_SYSTEM_IS_LOCKED or SL_RET_CODE_DEV_LOCKED.

A locked device has a limited access; to recover from a locked device (if the reason is a security alert), the device can reprogram or recover using the restore to factory method. The security-alert counter is a persistent counter, and can be set to zero only by the programming or recovery functions.

The default security alerts threshold is set by the UniFlash Image Creator. The host can retrieve the current number of security alerts and the defined threshold using the function sl_FsCtl(SL_FS_CTL_GET_STORAGE_INFO..). This function is also enabled when the device is locked.

There are two kinds of security alerts:

- Explicit Alerts – Critical: the device is locked immediately regardless the alert counters. Explicit alerts are created when detecting the following tamper events:
  - File system data integrity violation
  - System configuration files integrity violation
- Implicit Alerts – The device is locked when the alert counter crosses the alerts threshold. Implicit alerts are created when detecting the following tamper events:
  - Trying to make an operation on a secure file without a valid token
  - Detecting an integrity violation when a secure-authenticate file is opened for read
  - Setting an invalid signature or invalid certificate when changing a secure-authenticate file

7.15 Design Consideration

7.15.1 Choosing SFLASH Type

Choosing the correct SFLASH for the application is an important step. The section describes the factors to consider when choosing the SFLASH. A list of recommended SFLASH types is published on the TI website.

In general, SFLASH types are varied in the following factors:

- The operating voltage: the Wi-Fi subsystem operating voltage should never be dropped to a level lower than the SFLASH-required operating voltage. See the TI wiki for further explanation regarding the required design.
- Power removal: all systems using serial flash are vulnerable to the effects of sudden power removal. The TI wiki describes how to minimize the potential for serial flash corruption due to power removal.
- Access time: the time for erases, reads, and writes is different among types of SFLASH. Faster SFLASH results in faster access of the SimpleLink Wi-Fi device to the file system.
- SFLASH write endurance: a typical serial flash ensures a data endurance of 100K write cycles per sector, and 20 years data retention.
- Size of the SFLASH: the SimpleLink Wi-Fi device supports SFLASH up to 16MB.

7.15.2 Software Design Consideration

Writing to the system with SFLASH requires consideration in the software design to maximize the SFLASH capabilities and expand the SFLASH life-time.

The following is a list of design recommendations:

- Minimum file updates: invoking of file system functions, which triggers erase or write storage commands, should be minimized. Such commands are triggered with each update of a file, and each system function has documentation about whether it triggers a file update.
- File allocation table: the file system allocation table resides on the SFLASH. New file creation and the deletion of existing files involves updating the allocation table. TI recommends rewriting a file rather than recreating (that is, deleting and creating) the file.
- System configuration and user file creation: the system configuration is stored on the SFLASH, and setting the system configuration triggers a file update operation. TI recommends setting the system configuration and user file creation as part of the programming image, and then programming the image. The programming method is working in an efficient way, to keep the number of SFLASH accesses to the minimum. During the programming procedure, regardless the number of the programmed files, the file system is written only twice.
- The SFLASH storage type that the SimpleLink Wi-Fi device supports has a minimal block (=subsector) size of 4096 bytes; this is the minimal unit with which the file system can work. Thus, the file system software rounds the files sizes to a multiple of 4096 bytes. For example, creating a file with maximum size of 20 bytes results in a file of 4096 bytes. For optimal consumption of the SFLASH, create files where their maximum size is a multiple of 4096 minus 500 bytes (for each file the file system allocates a header of 500 bytes).
- The file system does not handle fragmentation; sometimes changing the order in which the files are created may result in more space. Also, changing the order of user file creation in the programming image may affect the SFLASH usage.
- To reduce the SFLASH endurance, create a file which requires frequent updates with the FAILSAFE flag; the number SFLASH writes is reduced in half because the system switches between the file images.

7.15.3 Retrieving Info Regarding SFLASH Usage

The SimpleLink Wi-Fi device provides counters of the number of SFLASH write operations. The counters can be obtained by:

- Getting the storage info command returns the file allocation table writes counter. Though the file allocation write counter is increased during the programming process, the actual count during the programming is only two SFLASH write operations.
- Getting the file info command returns the file write counter. For files created with the fail-safe flag, the retrieved count should be divided in 2.

7.15.4 SFLASH Size

The SimpleLink Wi-Fi device file system supports an SFLASH size from 1MB up to 16MB.

The required storage size depends on the size of the vendor files, and the requirements of the target system.

The minimum sizes recommended for the CC3220/R/S device is 2MB to 4MB, 4MB for the CC3220F, and 1MB to 2MB for the CC3120.

The following subsections describe the usage and the sizes that should be considered when determining the required size of the SFLASH.

7.15.4.1 Restore to Factory is Disabled

If the implementation disables the restore-to-factory feature, the following storage units are allocated:

- 5 blocks, file system allocation table
- 32 blocks, reserved for system files
- 7 blocks, TI information file
- 66 blocks, service pack
- Host size blocks, for the CC3220 device: storage for the host program.
- Blocks for the vendor files

- Temporary storage for the image, depending on the image size

NOTE: The size of the programming image depends on the size of the stored files.

During the programming, temporary storage for keeping the programming image is required; at the end of the programming, the temporary storage can be used for vendor files.

The term block is related to subsector(4096 bytes), which is the smallest erase unit of the SFLASH.

7.15.4.2 Restore to Factory is Enabled

If the implementation enables the restore-to-factory feature, the following storage units are allocated:

- 4 blocks, file system allocation table
- 32 blocks reserved for system files
- 7 blocks, TI information file
- 66 blocks, service pack
- Host size blocks, for the CC3220 device: storage for the host program.
- Blocks for the vendor files
- Temporary storage for the image, depending on the image size, rounded to 32 blocks

NOTE: The size of the image depends on the size of the stored files.

7.15.5 Storage Usage Information

The required size for the programming image can be observed in the UniFlash Image Creator log. The Image Creator tool operates a log which is displayed during the image creation; the log displays how many blocks are allocated for each file, and an estimation of the total required storage size.

After the SFLASH is programmed, the file list function (host driver) retrieves information about the existing files and the number of allocated blocks per file.

The get storage info function contains information about the device usage, information about the device capacity, the largest available gap, and so forth.

Figure 7-1 is an example of the Image Creator log, which is displayed during the image generation.

```
Total size of image( 72 blocks )
        Total size of user files after extraction( 7 blocks )
        Total size of FileSystem( 4 blocks )

        Total System files after extraction(includes reserved space for system files)( 106 blo
cks )
                Total reserved for system files ( 106 blocks )
                        Service pack after extraction size ( 66 blocks )
                        Application code  after extraction size ( 0 blocks )
                        Kept for system files ( 32 blocks )
                System files after extraction size (includes the service-pack) ( 76 blocks )
        Reserved size for the Image (includes image protecting) ( 96 blocks )
         ==> After the extraction the set will require total size of 213 blocks.<==
         ==> During the extraction process the set will require total size of 213 blocks.<==

Non-encrypted files generated successfully !!!
```

Figure 7-1. Image Creator Log

HTTP Server

Topic Page

8.1 Introduction

The SimpleLink Wi-Fi device includes a built-in HTTP server that lets the end-user remotely communicate with the device. This chapter describes the internal HTTP server capabilities and the relevant API. The SimpleLink HTTP server consists of the following:

- HTML pages stored on the file system
- Content generated on the fly by the host
- Hard coded configuration pages permanently stored in the ROM of the device

8.1.1 Built-in Configuration Pages

These web pages are stored in the SimpleLink device ROM, and allow for changing and reading many of its setting through a web interface commonly used in many routers and access points, as shown in Figure 8-1. The pages are completely self-contained, and no host involvement is necessary for their function.

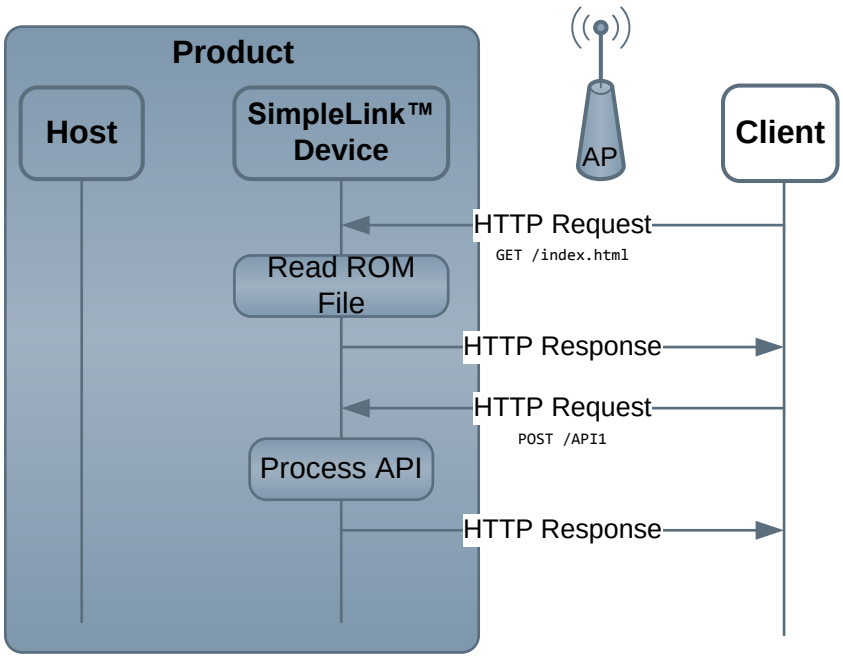

Figure 8-1. Configuration Pages

8.1.2 RESTful APIs

A reserved set of resource names may be used to configure various parameters in the SimpleLink Wi-Fi device directly, through HTTP requests with no host application involvement.

8.1.2.1 Changing Configuration

Settings are changed through HTTP POST requests to hard coded resource names. This is handled by the device and transparent to the host, as shown in Figure 8-2.

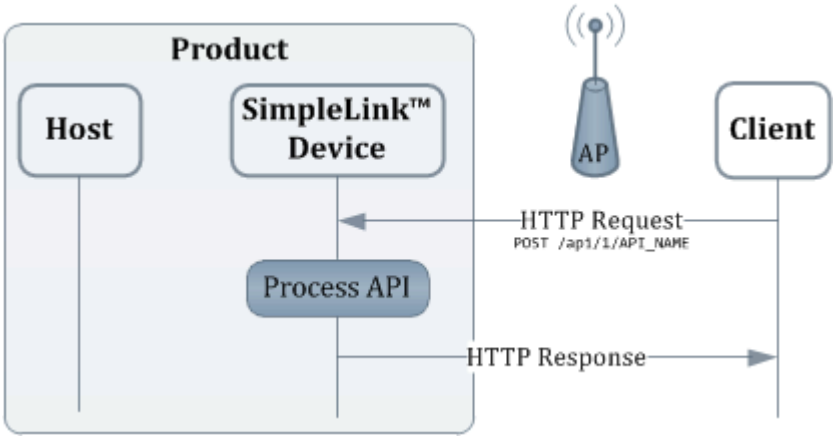

Figure 8-2. Changing Configuration

For details, see *Section 8.4*.

8.1.2.2 Reading Configuration

Settings can be read through HTTP GET requests to various token names. This is handled by the device and is transparent to the host, as shown in Figure 8-3.

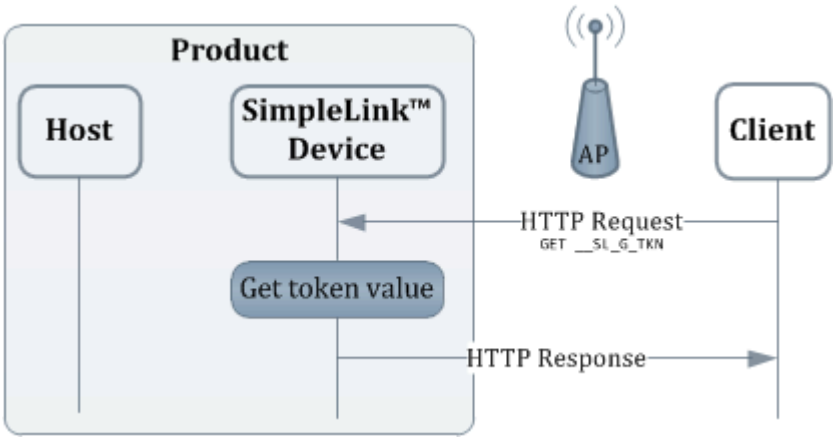

Figure 8-3. Reading Configuration

For details, see *Section 8.5*.

8.1.3 Custom Static Pages

User pages are any content stored under the /www/ or /www/safe/ path on the file system. The content is placed by the application, either as part of the programming process or using the host application file system API.

NOTE: The previously mentioned paths are used only to organize files on the file system, and must not be part of a URL when linking to resources. For example, the URL of a file named example.html which is placed at */www/user_directory/example.html* will be *http://deviceIP/user_directory/example.html*.

Placing any file under the /www/ path of the file system makes it a resource which the HTTP server can serve to its clients. These resources are static, as their content remains constant. Serving these resources is handled completely by the HTTP server and is transparent to the host as seen in Figure 8-4.

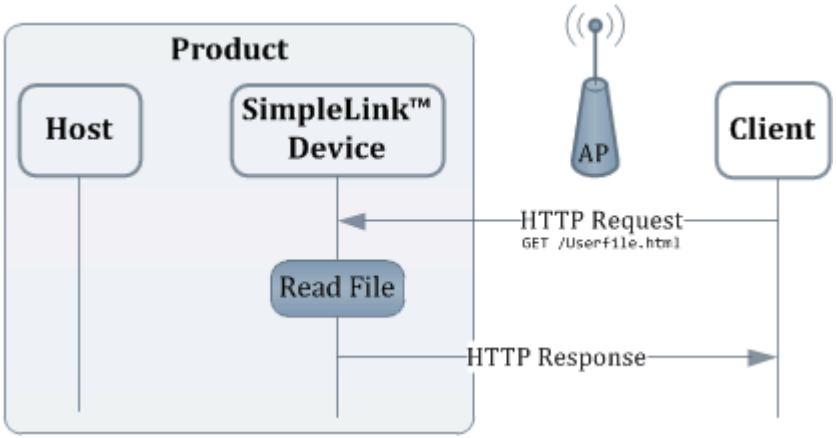

Figure 8-4. Static Pages

8.1.3.1 Custom Pages With Device Tokens

Device Tokens are special text strings inside a resource, which the HTTP server substitutes with values just before serving the resource to the requesting client (see Section 8.5 for details). These tokens are updated by the SimpleLink device every time a resource is served, which lets users create pages with some dynamic content (various parameters of the SimpleLink device) without any involvement from the host, as shown in Figure 8-5.

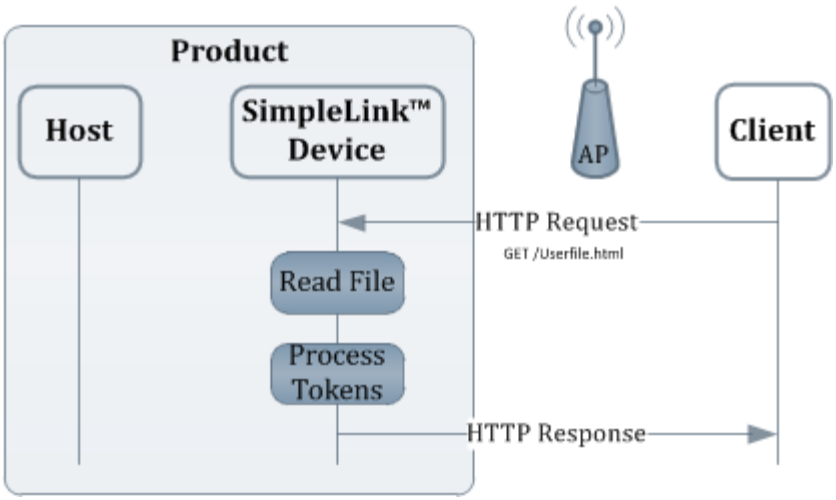

Figure 8-5. Custom Pages With Device Tokens

8.1.3.2 Static Pages With Host Tokens

Host tokens are similar to device tokens, except that the substitution of each token is deferred to the host through an asynchronous event. This lets users create pages containing dynamic content with minimal host involvement, because the major and static part of the web page is stored on the file system and only small dynamic parts are handled by the host. This can be seen in Figure 8-6.

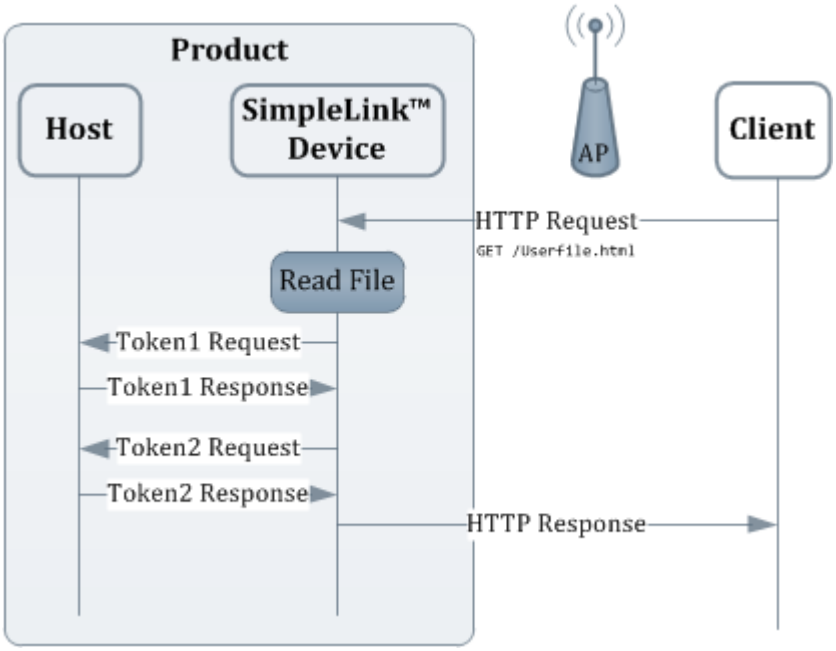

Figure 8-6. Static Pages With Host Tokens

8.1.4 Host Application Interface

If the served content is highly dynamic, defer the request completely to the host using the mechanism described in Section 8.7.2. In this case, the entire content of the response must be generated by the host on each request, as seen in Figure 8-7.

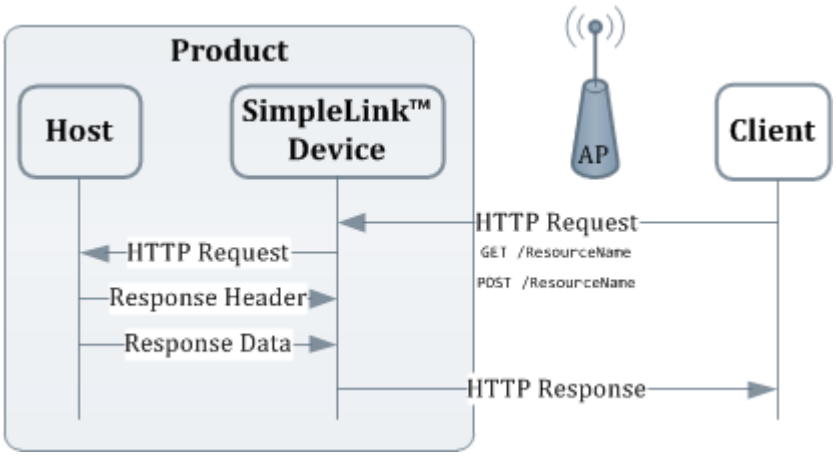

Figure 8-7. Host Application Interface

8.2 Key Features

Table 8-1 lists the key features of the HTTP server.

Table 8-1. Key Features

Feature	Description
HTTP	Support version 1.0, single client, GET&POST requests
HTTPS	SSL connections to the server are supported.

Table 8-1. Key Features (continued)

Feature	Description
Serve HTML pages from file system	Serve any resource that can be placed on the file system.
RESTful APIs	Execute various APIs through simple POST requests.
Built-in configuration pages	Built-in default page that provides device configuration, status, and analyzing tools
Host callbacks	HTTP requests can be handled by the host through a simple callback mechanism.
HTTP port configuration	Default is port 80.
HTTP web server authentication	Includes authentication name, password, and realm, which are configurable. Can be enabled or disabled (disabled by default).
Domain name configuration	Supported in AP mode.
Redirect mechanism	Redirect a nonsecure connection to secured .
Host-controlled resource transfer size	Host is able to select desired packet size for sending and receiving resources.

8.3 Configurations and Settings

The HTTP server is active by default on all device modes (STA, AP, and Wi-Fi Direct). It can be disabled or enabled per specific device mode using the sl_NetAppStart / sl_NetAppStop API. Domain names can be set by using the HTTP server options, and are configured through the sl_NetAppSet API with SL_NETAPP_HTTP_SERVER_ID as the App ID. For the configuration to take effect, the server must be restarted (either by stopping and restarting the service or by restarting the entire network subsystem).

Table 8-2 describes the available configuration options.

Table 8-2. Configuration Options

Option Name	Function	Notes	Default Value
SL_NETAPP_HTTP_PRIMARY_PORT_NUMBER	Port on which the server accepts new connections		80
SL_NETAPP_HTTP_PRIMARY_PORT_SECURITY_MODE	Enables the secure socket connection (SSL) on the primary server port		Disabled
SL_NETAPP_HTTP_AUTH_CHECK	Enable or disable the client authentication.		Disabled
SL_NETAPP_HTTP_AUTH_NAME	Authentication username	Maximum length is 20 characters	Admin
SL_NETAPP_HTTP_AUTH_PASSWORD	Authentication password	Maximum length is 20 characters	Admin
SL_NETAPP_HTTP_AUTH_REALM	Authentication realm	Only one realm is supported. Maximum length is 20 characters	SimpleLink CC31xx
SL_NETAPP_HTTP_ROM_PAGES_ACCESS	Enable access to the configuration pages stored in the ROM of the device and processing of the RESTful APIs.		Enabled
SL_NETAPP_HTTP_SECONDARY_PORT_NUMBER	Secondary port on the HTTP server accepts connections.		80
SL_NETAPP_HTTP_SECONDARY_PORT_ENABLE	Enable or disable secondary port.		Disabled
SL_NETAPP_HTTP_PRIVATE_KEY_FILENAME	Public or private pair used for key exchange when secure socket is enabled		N/A
SL_NETAPP_HTTP_DEVICE_CERTIFICATE_FILENAME	Public or private pair used for key exchange when secure socket is enabled		N/A
SL_NETAPP_HTTP_CA_CERTIFICATE_FILE_NAME	Certificate file name which will be used for client authentication (if present).		N/A

8.4 RESTful API Processing

The SimpleLink HTTP server recognizes dedicated resource names and treats them as APIs. A POST request to these names executes the API without any involvement from the host application. All HTTP API requests must have the encoding of *application/x-www-form-urlencoded*. Most APIs require one or more parameters. These parameters are passed as part of the message body and have a rigid structure. They begin with the prefix "__SL_P_", followed by three characters for the parameter ID, followed by an equal sign, and then by the parameter value (such as __SL_P_T.A=192.168.10.10).

Several parameters can be chained together with the ampersand operator (such as __SL_P_T.A=192.168.10.10& postPingSetPktSize=64). Blank spaces that are not part of the parameter value are not allowed. All parameters relevant to an API should be provided in the body of the same request. However, if a parameter was omitted, its previously known value is used. This feature is enabled by default (see Section 8.3 for details).

8.4.1 Ping

The device has a built-in ping utility for testing and troubleshooting network connectivity issues. Ping is started by posting the following parameters to */api/1/netapp/ping*, as shown in Table 8-3.

Table 8-3. Ping Options

Name (code)	Description	Example
Target IP (__SL_P_T.A)	IPv4 target address of ping requests.	(__SL_P_T.A=192.168.10.10)
Ping packet size (__SL_P_T.B)	Size of the ping payload in bytes (from 1 to 1472).	(__SL_P_T.B=1024)
Packets to send (__SL_P_T.C)	Number of packets to send (from 1 to 255).	(__SL_P_T.C=4)

For example, the following request will send 4 ping packets, each of size 1024 bytes, to IP 10.123.45.2:

```
POST /api/1/netapp/ping HTTP/1.1
Host: mysimplelink.net
Content-Type: application/x-www-form-urlencoded

__SL_P_T.C=4&__SL_P_T.B=1024&__SL_P_T.A=10.123.45.2
```

The ping process stops automatically when the requested number of packets is sent. To manually stop it beforehand, a post request should be sent to */api/1/netapp/ping_stop* (no parameters are necessary). The results can be retrieved by requesting the token __SL_G_T.D (see Appendix A for details).

8.4.2 IP Configuration

Many IP settings can be configured from the HTTP interface by sending a POST request to either */api/1/netapp/netcfg_sta*, */api/1/netapp/netcfg_sta_ipv6*, or */api/1/netapp/netcfg_ap* URLs with some (or all) of the parameters listed in Table 8-4.

Table 8-4. IP Configurations

Name (code)	Description	Example
STA mode IP (__SL_P_N.A)	Device IP in station mode	(__SL_P_N.A=192.168.10.10)
AP mode IP (__SL_P_N.P)	Device IP in AP mode	(__SL_P_N.P=192.168.10.10)
STA mode netmask (__SL_P_N.B)	Device subnet mask in station mode	(__SL_P_N.B=255.255.255.0)
AP mode netmask (__SL_P_N.Q)	Device subnet mask in access point mode	(__SL_P_N.Q=255.255.255.0)
STA Gateway (__SL_P_N.C)	Network gateway IP in station mode	(__SL_P_N.C=192.168.10.1)
AP Gateway (__SL_P_N.T)	Network gateway IP in AP mode	(__SL_P_N.T=192.168.10.1)
Address of primary STA DNS server (__SL_P_N.H)	Address of primary DNS server in station mode	(__SL_P_N.H=8.8.8.8)
Address of primary AP DNS server (__SL_P_N.U)	Address of primary DNS server in AP mode	(__SL_P_N.U=8.8.8.8)

Table 8-4. IP Configurations (continued)

Name (code)	Description	Example
IPv4 mode (__SL_P_N.D)	IP acquisition mode for IPv4 address. Options are: LLA DHCP, DHCP, and Static	(__SL_P_N.D=DHCP)
IPv6 Local mode (__SL_P_I.S)	IP acquisition mode for local IPv6 local address. Options are: Stateless, Static, and Statefull	(__SL_P_I.S=Disable, __SL_P_I.S =Stateless, __SL_P_I.S =Static, __SL_P_I.S =Statefull)
IPv6 Local address (__SL_P_I.L)	Set the IPv6 link-local address (if local mode is set to Static)	(__SL_P_I.S=fe80::ccaf:9519:0002:a5fd)
IPv6 Global mode (__SL_P_I.G)	IP acquisition mode for global IPv6 local address. Options are: Stateless, Static, and Statefull	(__SL_P_I.G=Disable, __SL_P_I.G =Stateless, __SL_P_I.G =Static, __SL_P_I.G =Statefull)
IPv6 Global address (__SL_P_I.B)	Set the IPv6 global address (if global mode is set to Static)	(__SL_P_I.B==2001:0db8:3c4d:0015:0000:0000:1a2f:1a2b)
IPv6 DNS address (__SL_P_I.K)	Set IPv6 primary DNS server	(__SL_P_I.K= 2001:4860:4860::8888)

For example, the following request sets the AP mode IP address to 10.10.10.10 without DHCP (Static):

```
POST /api/1/netapp/netcfg_ap HTTP/1.1
Host: mysimplelink.net
Content-Type: application/x-www-form-urlencoded

__SL_P_N.P=10.10.10.10&__SL_P_N.D=Static
```

8.4.3 URN Configuration

The device URN (uniform resource name) can be set by posting to */api/1/netapp/set_urn* the parameters listed in Table 8-5. The maximum size of the URN is 33 characters (not including the null terminator).

Table 8-5. URN Configurations

Name (code)	Description	Example
Device URN (__SL_P_S.B)	Must not exceed 33 characters.	__SL_P_S.B=mysimplelink1.net

For example, the following request changes the device URN to my-urn:

```
POST /api/1/netapp/set_urn HTTP/1.1
Host: mysimplelink.net
Content-Type: application/x-www-form-urlencoded

__SL_P_S.B=my-urn
```

8.4.4 WLAN Profiles

WLAN connection profiles can be added by posting to either */api/1/wlan/profile_add* or */api/1/wlan/profile_p2p* the parameters listed in Table 8-6 (all are case sensitive).

Table 8-6. WLAN Profiles

Name (code)	Description	Example
SSID (__SL_P_P.A)	The SSID of the desired AP. Must not exceed 32 characters.	__SL_P_P.A=TargetSSID
Security (__SL_P_P.B)	Security type for the connection. 0-Open, 1-WEP, 2-WPA1, 3-WPA2, 6-WPS/Push-button, 7-WPS/Pin Keypad, 8-WPS/Pin Display.	__SL_P_P.B=3
Security key (__SL_P_P.C)	Security key or PIN code. Must not exceed 64 characters.	__SL_P_P.C=MySecurePassword
Priority (__SL_P_P.D)	Priority of the profile. Must be from 0 to 15.	__SL_P_P.D=1

For EAP connections, the URL for posting is */api/1/wlan/profile_eap*, and the parameters are listed in Table 8-7 (all are case sensitive).

Table 8-7. WLAN EAP Profiles

Name (code)	Description	Example
SSID (__SL_P_P.H)	The SSID of the desired AP. Must not exceed 32 characters.	__SL_P_P.H=TargetSSID
Identity (__SL_P_P.I)	User identity. Must not exceed 64 characters.	__SL_P_P.I=MyIdentity
Anonymous Identity (__SL_P_P.J)	Anonymous user identity. Must not exceed 64 characters.	__SL_P_P.J=MyAnonymousIdentity
Password (__SL_P_P.K)	Connection password. Must not exceed 63 characters.	__SL_P_P.K=MySecurePassword
Priority (__SL_P_P.L)	Priority of the profile. Must be from 0 to 15.	__SL_P_P.L=1
EAP Method (__SL_P_P.M)	Can be TLS / TTLS / PEAP0 / PEAP1 / FAST.	__SL_P_P.M=TLS
Pahse2 Authentication (__SL_P_P.N)	Can be None / TLS / MSCHAPV2 / PSK	__SL_P_P.N=None
EAP Provisioning Type (__SL_P_P.O)	Can be None / 0 / 1 / 2	__SL_P_P.O=0

A post to */api/1/wlan/profile_del* with the parameters listed in Table 8-8 erases it from the file system.

Table 8-8. Erase Profiles

Name (code)	Description	Example
Delete profile (__SL_P_PRR)	Delete the profile with the specified index such that 0 < Index < 9.	__SL_P_PRR=2

A post to */api/1/wlan/profile_del_all* deletes all profiles stored on the file system (this includes all profiles and is not limited to those created through the HTTP interface). Information on the existing profiles can be accessed using the __SL_G_PN1 to __SL_G_PP7 tokens (see *Section 8.5*).

For example, the following request adds a profile for connecting to a secure (3) network with SSID mySSID, password 0123456789, and priority 5:

```
POST /api/1/wlan/profile_add HTTP/1.1
Host: mysimplelink.net
Content-Type: application/x-www-form-urlencoded

__SL_P_P.B=3&__SL_P_P.A=mySSID&__SL_P_P.C=0123456789&__SL_P_P.D=5
```

8.4.5 WLAN Scan

A WLAN scan for nearby access points may be triggered by posting to */api/1/wlan/en_ap_scan* using the parameters listed in Table 8-9.

Table 8-9. WLAN Scan

Name (code)	Description	Example
Number Of Scan Cycles (__SL_P_SC2)	Number of scan cycles to execute. Must be greater than zero and smaller than 2^32.	__SL_P_SC2=64
Time Between Scan Cycles (__SL_P_SC1)	Time (in seconds) to wait between each two cycles.	__SL_P_SC1=10

For example, the following request triggers 3 scan cycles with 10-second intervals between them:

```
POST /api/1/wlan/en_ap_scan HTTP/1.1
Host: mysimplelink.net
Content-Type: application/x-www-form-urlencoded
```

```
__SL_P_SC2=3&__SL_P_SC1=10
```

The scan results can be accessed with the __SL_G_NW0 and __SL_G_NW1 tokens (see Section 8.5).

8.4.6 Provisioning Confirmation

Posts to */api/1/wlan/en_ap_scan/confirm_req* are handled as described in *Chapter 14*.

8.4.7 Connection Policy

The connection policy of the device can be set by posting to */api/1/wlan/policy_set*. Any combination of the parameters listed in Table 8-10 and present in the request turns on their associated option. Options with parameters that are not preset are turned off. No values are provided after the equal sign; the options are chained together with the ampersand operator. See Section 3.3.3 for details on each option.

Table 8-10. Connection Policies

Name (code)	Description	Example
Enable Auto Connect (__SL_P_P.E)	Auto connect policy	__SL_P_P.E=
Enable Fast Connect (__SL_P_P.F)	Fast connect policy	__SL_P_P.F=
Enable P2P Any Connect (__SL_P_P.G)	AnyP2P connect policy – relevant for Wi-Fi Direct only	__SL_P_P.G=
Enable Auto Provisioning (__SL_P_P.Q)	Auto provisioning policy	__SL_P_P.Q=

For example, the following request enables auto-connect and fast-connect policies:

```
POST /api/1/wlan/policy_set HTTP/1.1
Host: mysimplelink.net
Content-Type: application/x-www-form-urlencoded

__SL_P_P.E=&__SL_P_P.F=
```

8.4.8 Station Action

When the device is in AP mode, posts to */api/1/wlan/en_ap_scan/station_action* can be performed to disconnect stations from the device. The station to disconnect is given by the parameters listed in Table 8-11.

Table 8-11. Station Action

Name (code)	Description	Example
Station number to disconnect (__SL_P_CRR)	The number of the station to disconnect. Must be from 1 to the maximum number of station as set by the user.	__SL_P_CRR=1

For example, the following request disconnects station 1:

```
POST /api/1/wlan/en_ap_scan/station_action HTTP/1.1
Host: mysimplelink.net
Content-Type: application/x-www-form-urlencoded

__SL_P_CRR=1
```

8.4.9 AP Black List

When the device is in AP mode, posts to */api/1/wlan/en_ap_scan/ap_aclist* can be made to control the black list, which indicates stations are not allowed to connect to the device. Table 8-12 lists these parameters.

Table 8-12. AP Control

Name (code)	Description	Example
Blacklist Filter Enable (__SL_P_C.M)	Enables the station MAC address blacklist filter: 0-Filter disabled 1-Filter enabled	__SL_P_C.M=1
Add station to the blacklist filter (__SL_P_CRL)	Adds the station of the specified index to the blacklist filter (if filter is enabled it will not be allowed to connect again).	__SL_P_CRL=1
Remove station from the blacklist filter (__SL_P_CRS)	Removes the station of the specified index from the blacklist filter (station can connect).	(__SL_P_CRS=1)
Remove station from the blacklist filter (__SL_P_C.B)	Sets the maximum number of simultaneousl -connected stations. Must be smaller than 5.	(__SL_P_C.B=3)

For example, the following request enables the filter and adds a station at index 1, preventing it from connecting to the device:

```
POST /api/1/wlan/en_ap_scan/ap_aclist HTTP/1.1
Host: mysimplelink.net
Content-Type: application/x-www-form-urlencoded

__SL_P_C.M=1&__SL_P_CRL=1
```

AP black list information can be accessed using the __SL_G_SR1 to __SL_G_CL8 tokens (see Section 8.5).

8.4.10 Date and Time

The device time and date can be set by posting to */api/1/wlan/en_ap_scan/set_time* the parameters listed in Table 8-13.

Table 8-13. Date and Time

Name (code)	Description	Example
Set date and time (__SL_P_S.J)	This parameter sets the time and date according to the following format: yyyy,mm,dd,hh,mm,ss (year,month,day,hours,minutes,seconds) Each number must not contain more than four characters.	__SL_P_S.J=2016,01,01,12,45,30

For example, the following request sets the date to 30/5/2016 and the time to 13:45:00:

```
POST /api/1/wlan/en_ap_scan/set_time HTTP/1.1
Host: mysimplelink.net
Content-Type: application/x-www-form-urlencoded

__SL_P_S.J=2016,05,30,13,45,00
```

8.5 Device Parameter Querying Through HTTP (Device Tokens)

The SimpleLink HTTP server supports querying various device parameters through a mechanism called device tokens. These tokens can be requested directly through an HTTP GET request, or embedded inside any serveable resource where they are replaced by their value when it is served. The token name has a rigid convention of "__SL_G_" followed by three characters of the parameter ID (for example, __SL_G_T.A).

8.5.1 Retrieving Tokens Through GET Request

A token value may be retrieved by an HTTP GET request whose target is the token name. These requests must have the encoding of *application/x-www-form-urlencoded*. Only one parameter can be queried in each HTTP GET request.

8.5.2 Embedded Tokens

The HTTP server automatically replaces token names with their values when it serves files from the file system or ROM. For example, if a text file is created on the file system under the path */www/example.txt* with the content:

```
Device hardware version: __SL_G_V.D
Device network version: __SL_G_V.A
```

Then a GET request to *mysimplelink.net /example.txt* returns the following text:

```
Device hardware version: 20000000
Device network version:  3.92.1.1
```

The tables in the following sections specify all supported tokens.

8.5.3 System Information

Table 8-14 lists the system information tokens.

Table 8-14. System Information Tokens

Token	Name	Value and Usage
__SL_G_S.A	System Up Time	Returns the system up time since the last reset in the following format: 000 days 00:00:00
__SL_G_S.B	Device Name (URN)	Returns device name
__SL_G_DNP	Device Name	Returns device name + MAC address (as string) if the default device name is set.
__SL_G_S.C	Domain Name	Returns domain name
__SL_G_S.D	Device Mode (role)	Returns device role. Values: Station, Access Point, P2P
__SL_G_S.E	Device Role Station	Drop-down menu select/not select Returns selected if device is station, else it returns not_selected.
__SL_G_S.F	Device Role AP	Drop-down menu select/not select Returns selected if device is AP, else it returns not_selected.
__SL_G_S.G	Device Role P2P	Drop-down menu select/not select Returns selected if device is in P2P, else it returns not_selected.
__SL_G_S.H	Device Name URN (truncated to 16 bytes)	Returns the URN string name with up to 16 bytes length. Longer names will be truncated.
__SL_G_S.I	System requires reset (after parameters change)	If system requires reset, return value will be the following string: "-- Some parameters were changed, System may require reset --" else it returns an empty string. (Every internal post that was handled will cause this token to return TRUE.)
__SL_G_S.J	Get System Time and Date	Returned value is a string with the following format: year, month, day, hours, minutes, seconds
__SL_G_S.K	Safe Mode Status	If device is in safe mode – return Safe Mode, if not return empty string.

8.5.4 Version Information

Table 8-15 lists the version information tokens.

Table 8-15. Version Information Tokens

Token	Name	Value and Usage
__SL_G_V.A	NWP version	Returns string with the version information
__SL_G_V.B	MAC version	Returns string with the version information
__SL_G_V.C	PHY version	Returns string with the version information
__SL_G_V.D	HW version	Returns string with the version information
__SL_G_REV	Revision	R2.0

8.5.5 Network Information

Table 8-16 lists the network information tokens.

Table 8-16. Network Information Tokens

Token	Name	Value and Usage
Station (and P2P Client)		
__SL_G_N.A	STA IPv4 Address	String format: xxx.yyy.zzz.ttt
__SL_G_N.B	STA IPv4 Subnet Mask	
__SL_G_N.C	STA IPv4 Default Gateway	
__SL_G_N.D	MAC Address	String format: 00:11:22:33:44:55
__SL_G_N.E	STA IPv4 DHCP State	Returned value: Enabled / Disabled
__SL_G_N.F	STA IPv4 DHCP Disable State	If DHCP is disabled, returns Checked, else returns Not_Checked. Used in the DHCP radio button
__SL_G_N.G	STA IPv4 DHCP Enable State	If DHCP is enabled, returns Checked, else returns Not_Checked. Used in the DHCP radio button
__SL_G_N.L	STA IPv4 LLA Enable State	If LLA option is enabled, returns Checked, else returns Not_Checked.
__SL_G_N.H	STA IPv4 DNS Server	String format: xxx.yyy.zzz.ttt
__SL_G_LV6	STA IPv6 Enable	If IPv6 interface is enabled, returns Checked, else returns Not_Checked.
__SL_G_LSC	STA IPv6 Local Address Type	Returns Checked if IPv6 local address mode is static.
__SL_G_LSS		Returns Checked if IPv6 local address mode is stateless.
__SL_G_LSF		Returns Checked if IPv6 local address mode is statefull.
__SL_G_N.Z	STA IPv6 Global Address Type	Returns Checked if IPv6 global address is disabled.
__SL_G_N.R		Returns Checked if IPv6 global address mode is static
__SL_G_N.O		Returns Checked if IPv6 global address mode is statefull
__SL_G_N.S		Returns Checked if IPv6 global address mode is stateless
__SL_G_LSK	STA Current IPv6 Local Address	Returns the address in the format xxxx:xxxx:xxxx:xxxx: xxxx:xxxx:xxxx:xxxx
__SL_G_LSG	STA Current IPv6 Global Address	
__SL_G_LSP	STA IPv6 DNS Server	
__SL_G_LSO	STA IPv6 Local Address Mode	Returns Disabled / Static / Stateless / Statefull according to the configured local address mode

Table 8-16. Network Information Tokens (continued)

Token	Name	Value and Usage
__SL_G_LSD	STA IPv6 Global Address Mode	Returns Disabled / Static / Stateless / Statefull according to the configured global address mode.
DHCP server		
__SL_G_N.I	DHCP Start Address	String format: xxx.yyy.zzz.ttt
__SL_G_N.J	DHCP Last Address	
__SL_G_N.K	DHCP Lease Time	String of the lease time in seconds
AP (and P2P Go)		
__SL_G_N.P	AP IP Address	String format: xxx.yyy.zzz.ttt
__SL_G_N.Q	AP Subnet Mask	
__SL_G_N.T	AP Gateway Address	
__SL_G_N.U	AP DNS Address	
__SL_G_W.A	Channel # in AP mode	
__SL_G_W.B	SSID	
__SL_G_W.I	Is SSID Public	If SSID is public (visible), returns Checked, else returns Not_Checked. Used in the security type radio button check/not checked.
__SL_G_W.J	Is SSID Hidden	If SSID is hidden (invisible), returns Checked, else returns Not_Checked. Used in the security type radio button check/not checked.
__SL_G_W.C	Security Type	Returned values: Open, WEP, WPA.
__SL_G_W.D	Security Type Open	If security type is open, returns Checked, else returns Not_Checked. Used in the security type radio button check/not checked.
__SL_G_W.E	Security Type WEP	If security type is WEP, returns Checked, else returns Not_Checked. Used in the security type radio button check/not checked.
__SL_G_W.F	Security Type WPA	If security type is WPA, returns Checked, else returns Not_Checked. Used in the security type radio button check/not checked.
__SL_G_SR1	The configured max number of connected stations.	The token representing the max number of connected stations returns Checked. Others return Not Checked.
__SL_G_SR2		
__SL_G_SR3		
__SL_G_SR4		
__SL_G_CN1	Name of the connected station (string) in the given index.	Each token returns the host name of the station in the specified index. " - " is returned if the client does not exist.
__SL_G_CN2		
__SL_G_CN3		
__SL_G_CN4		
__SL_G_CM1	MAC address (string in the format AA:BB:CC:DD:EE:FF) of the connected station in the given index.	Each token returns the MAC address of the station in the specified index. " - " is returned if no station is connected.
__SL_G_CM2		
__SL_G_CM3		
__SL_G_CM4		
__SL_G_CI1	IP address (string in the format W.X.Y.Z) of the connected station in the given index.	Each token returns the IP address of the station in the specified index. " - " is returned if no station is connected.
__SL_G_CI2		
__SL_G_CI3		
__SL_G_CI4		

Table 8-16. Network Information Tokens (continued)

Token	Name	Value and Usage
__SL_G_SM1	Access control filter is enabled	If AP access control filter is enabled, returns Checked, else returns Not_Checked.
__SL_G_SM0	Access control filter is disabled	If AP access control filter is disabled, returns Checked, else returns Not_Checked.
__SL_G_CLS	Number of filtered MAC addresses.	
__SL_G_CL1	The MAC filter of the given index (string in the format AA:BB:CC:DD:EE:FF).	Return the configured MAC address to filter. " - " is returned if no filter is configured.
__SL_G_CL2		
__SL_G_CL3		
__SL_G_CL4		
__SL_G_CL5		
__SL_G_CL6		
__SL_G_CL7		
__SL_G_CL8		
__SL_G_NW1	Tokens for retrieving the scan results. Each result is in the format Nssid; where N is the security indicator (0=open, 1=WEP, 2=WPA/WPA2).	Incrementally returns scan results (one for each get request).
__SL_G_NW0		Always returns the first scan result and resets the internal pointer.

8.5.6 Ping Results

Table 8-17 lists the ping results tokens.

Table 8-17. Ping Results Tokens

Token	Name	Value and Usage
__SL_G_T.A	IP Address	String format: xxx.yyy.zzz.ttt
__SL_G_T.B	Packet Size	
__SL_G_T.C	Number of Pings	
__SL_G_T.D	Ping Results – total sent	Number of total pings sent
__SL_G_T.E	Ping Results – successful sent	Number of successful pings sent
__SL_G_T.F	Ping Test Duration	In seconds

8.5.7 Connection Policy Status

Table 8-18 lists the connection policies status tokens.

Table 8-18. Connection Policies Status Tokens

Token	Name	Value and Usage
__SL_G_P.E	Auto Connect	If auto connect is enabled, returns Checked, else returns Not_Checked. Used in the auto connect checkbox.
__SL_G_P.F	Fast Connect	If fast connect is enabled, returns Checked, else returns Not_Checked. Used in the fast connect checkbox.
__SL_G_P.G	Any P2P	If any P2P is enabled, returns Checked, else returns Not_Checked. Used in the Any P2P checkbox.
__SL_G_P.P	Auto Smart Config	If auto smart config is enabled, returns Checked, else returns Not_Checked. Used in the Auto Smart Config checkbox.

8.5.8 Provisioning

Table 8-19 lists the provisioning tokens.

Table 8-19. Provisioning Tokens

Token	Name	Value and Usage
__SL_G_P.Q	Auto Provisioning	If auto provisioning is enabled, returns Checked, else returns Not_Checked.
__SL_G_MCH	Returns a human-readable text representing the status of the provisioning process	
__SL_G_MCR	Returns a number code of the provisioning status	
__SL_G_PST	Provisioning Active Indication	Simple text indication returning 1 if provisioning is active; 0 otherwise.
__SL_G_PIP	Provisioned IP Address	IP address obtained in the provisioning process

8.5.9 Display Profile Information

Table 8-20 lists the display profile information tokens.

Table 8-20. Display Profile Information Tokens

Token	Name	Value and Usage
__SL_G_PN1	Return profile 1 SSID	SSID string
__SL_G_PN2	Return profile 2 SSID	
__SL_G_PN3	Return profile 3 SSID	
__SL_G_PN4	Return profile 4 SSID	
__SL_G_PN5	Return profile 5 SSID	
__SL_G_PN6	Return profile 6 SSID	
__SL_G_PN7	Return profile 7 SSID	
__SL_G_PS1	Return profile 1 Security Status	Returned values: Open, WEP, WPA, WPS, ENT, P2P_PBC, P2P_PIN or " – " for empty profile.
__SL_G_PS2	Return profile 2 Security Status	
__SL_G_PS3	Return profile 3 Security Status	
__SL_G_PS4	Return profile 4 Security Status	
__SL_G_PS5	Return profile 5 Security Status	
__SL_G_PS6	Return profile 6 Security Status	
__SL_G_PS7	Return profile 7 Security Status	
__SL_G_PP1	Return profile 1 Priority	Profile priority: 0–7
__SL_G_PP2	Return profile 2 Priority	
__SL_G_PP3	Return profile 3 Priority	
__SL_G_PP4	Return profile 4 Priority	
__SL_G_PP5	Return profile 5 Priority	
__SL_G_PP6	Return profile 6 Priority	
__SL_G_PP7	Return profile 7 Priority	

8.5.10 P2P Information

Table 8-21 lists the P2P information tokens.

Table 8-21. P2P Information Tokens

Token	Name	Value and Usage
__SL_G_R.A	P2P Device Name	String
__SL_G_R.B	P2P Device Type	String
__SL_G_R.C	P2P Listen Channel	Returns string of the listen channel number
__SL_G_R.T	Listen Channel #1	If current listen channel is #1, returns selected, else returns not_selected. Used for the drop down menu of the listen channel.
__SL_G_R.U	Listen Channel #6	If current listen channel is #6, returns selected, else returns not_selected. Used for the drop down menu of the listen channel.
__SL_G_R.V	Listen Channel #11	If current listen channel is #11, returns selected, else returns not_selected. Used for the drop down menu of the listen channel.
__SL_G_R.E	P2P Operation Channel	Returns string of the operational channel number
__SL_G_R.W	Operational Channel #1	If current operational channel is #1, returns selected, else returns not_selected. Used for the drop down menu of the operational channel.
__SL_G_R.X	Operational Channel #6	If current operational channel is #6, returns selected, else returns not_selected. Used for the drop down menu of the operational channel.
__SL_G_R.Y	Operational Channel #11	If current operational channel is #11, returns selected, else returns not_selected. Used for the drop down menu of the operational channel.
__SL_G_R.L	Negotiation Intent Value	Returned values: Group Owner, Negotiate, Client
__SL_G_R.M	Role Group Owner	If intent is Group Owner, returns Checked, else returns Not_Checked. Used for negotiation intent radio button.
__SL_G_R.N	Role Negotiate	If intent is Negotiate, returns Checked, else returns Not_Checked. Used for negotiation intent radio button.
__SL_G_R.O	Role Client	If intent is Client, returns Checked, else returns Not_Checked. Used for negotiation intent radio button.
__SL_G_R.P	Negotiation Initiator Policy	Returned Values: Active, Passive, Random Backoff
__SL_G_R.Q	Neg Initiator Active	If negotiation initiator policy is Active, returns Checked, else returns Not_Checked. Used for negotiation initiator policy radio button.
__SL_G_R.R	Neg Initiator Passive	If negotiation initiator policy is Passive, returns Checked, else returns Not_Checked. Used for negotiation initiator policy radio button.
__SL_G_R.S	Neg Initiator Rand Backoff	If negotiation initiator policy is Random Backoff, returns Checked, else returns Not_Checked. Used for negotiation initiator policy radio button.

8.5.11 Host Tokens

All tokens not defined in the previous sections are transferred to the host for conversion.

8.6 Resource Search Order

This section describes the way in which the HTTP server handles each HTTP request according to its type and resource name.

8.6.1 GET Request Search Order

A GET request is processed according to the flow in Figure 8-8.

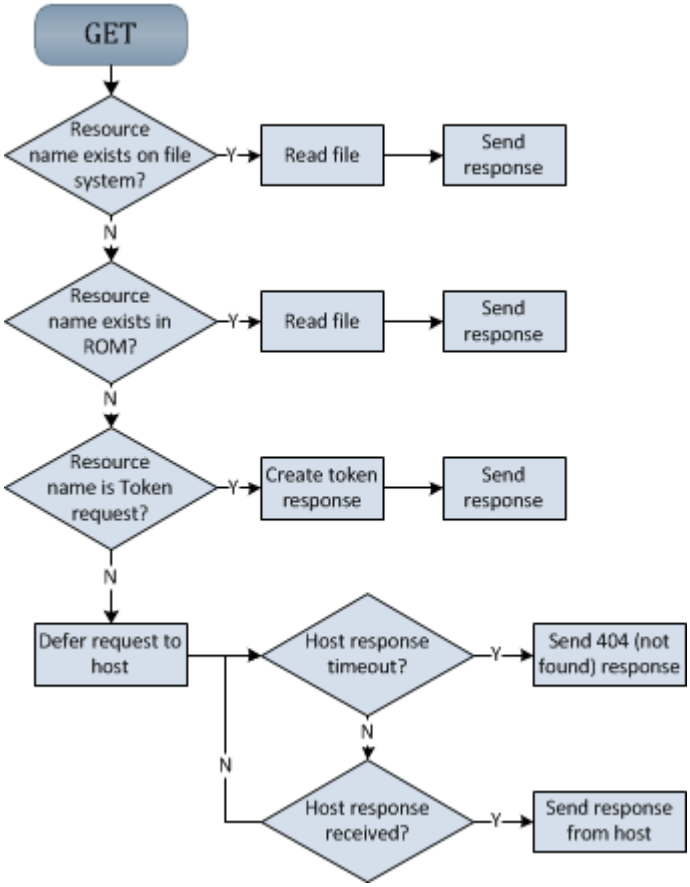

Figure 8-8. GET Request Flow

If a file is not found in the file system, it is searched in the device ROM, where the following files always exist:

- index.html
- netlist.txt
- param_product_version.txt
- param_device_name.txt
- param_ip_address.txt
- param_cfg_result.txt

Any request other than GET is not associated with these resources, and is transferred directly to the host. Additionally, to use the built-in configuration page, do not override any of the built-in pages, because this breaks functionality.

8.6.2 POST Request Search Order

A POST request is processed according to the flow in Figure 8-9.

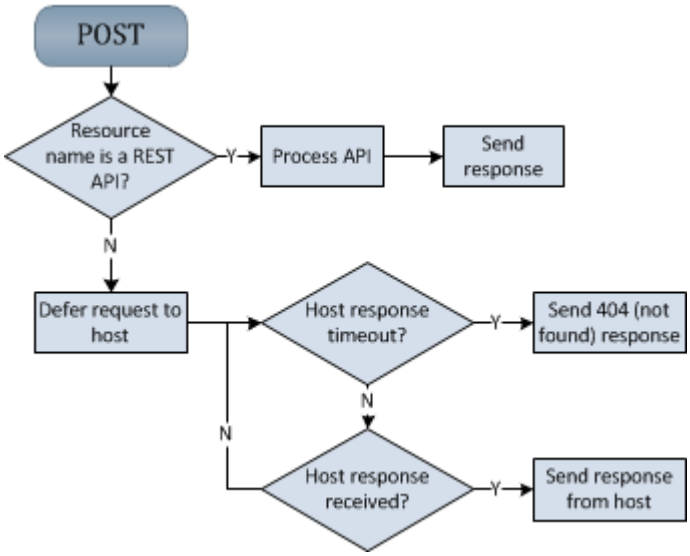

Figure 8-9. POST Request Flow

8.6.3 PUT and DELETE Request Search Order

PUT and DELETE requests are always deferred to host regardless of resource name, as shown in Figure 8-10.

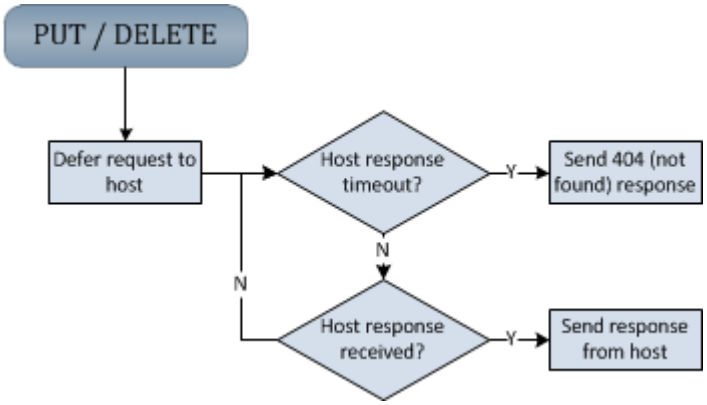

Figure 8-10. PUT and DELETE Request Flow

8.7 Host HTTP Requests Processing

All HTTP requests transferred to the host are processed through the macro slcb_NetAppRequestHdlr, which should be mapped to a user function by user.h. The function receives two parameters (and returns void): A pointer to the request structure containing the parameters and type of the HTTP request, and a pointer to the response structure which should be populated with the desired HTTP response.

The HTTP headers are transferred to the host as TLVs (type length value) in the metadata section of the request. The HTTP message (if present) transferred as is, and should be parsed and processed by the user function. The HTTP user handler is invoked from the SimpleLink driver context, and must therefore return quickly and without calling any other SimpleLink APIs. If the response cannot be determined immediately by the handler, it should set the response status to pending and return. The application must then generate and send a response from its own context.

8.7.1 Metadata (TLVs) Description

Each HTTP request consists of two parts: the HTTP headers, and HTTP body (which is optional). The headers are standard fields defined by the HTTP RFCs and set various parameters of the HTTP transaction. To allow easy parsing of the headers, they are converted to TLV representation. Each TLV has the structure listed in Table 8-22.

Table 8-22. TLV Structure

Size	1 Byte	2 Bytes	n Bytes
Name	Metadata Type	Length	Value
Description	A unique number identifying the HTTP header, see Table 8-23.	Size in bytes of the entire TLV including the Length and Type fields.	Raw value of the HTTP header copied directly from the HTTP request without line termination (\r or \n characters).(1)

(1) The only exception is the HTTP Content Length field, which is automatically converted to an integer.

Table 8-23 lists the metadata types.

Table 8-23. HTTP Metadata Types

Metadata Type	HTTP Header Name
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_CONTENT_TYPE	Content-Type
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_CONTENT_LEN	Content-Length
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_LOCATION	Location
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_SERVER	Server
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_USER_AGENT	User-Agent
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_COOKIE	Cookie
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_SET_COOKIE	Set-Cookie
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_UPGRADE	Upgrade
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_REFERER	Referer
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_ACCEPT	Accept
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_CONTENT_ENCODING	Content-Encoding
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_CONTENT_DISPOSITION	Content-Disposition
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_CONNECTION	Connection
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_ETAG	Etag
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_DATE	Date
SL_NETAPP_REQUEST_METADATA_TYPE_HEADER_HOST	Host
SL_NETAPP_REQUEST_METADATA_TYPE_ACCEPT_ENCODING	Accept-Encoding
SL_NETAPP_REQUEST_METADATA_TYPE_ACCEPT_LANGUAGE	Accept-Language
SL_NETAPP_REQUEST_METADATA_TYPE_CONTENT_LANGUAGE	Content-Language
SL_NETAPP_REQUEST_METADATA_TYPE_ORIGIN	Origin
SL_NETAPP_REQUEST_METADATA_TYPE_ORIGIN_CONTROL_ACCESS	Access-Control-Allow-Origin

All HTTP headers not present in Table 8-23 are skipped. Additionally, the metadata types listed in Table 8-24 are generated internally by the HTTP server to provide more information on the HTTP request.

Table 8-24. Internal Metadata Types

Metadata Type	Description
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_VERSION	Version field of the HTTP request
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_REQUEST_URI	URI string of the HTTP request
SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_QUERY_STRING	Query string of the HTTP request

The TLVs are packed continuously in the metadata section of the request. The user's code should begin parsing from byte 0, which is always the type field of the first TLV, and finish when metadata-length bytes are processed (which should point to the last byte of the value field of the last TLV). The TLVs are packed in no particular order. Table 8-25 is an example metadata breakout containing two TLVs

Table 8-25. Metadata Breakout Examples

Metadata Offset / Content	0 (TLV1 Type)	1–2 (TLV1 Length)	3–10 (TLV1 Value)	11 (TLV2 Type)	12–13 (TLV2 Length)	14–24 (TLV2 Value)
Data	1 (HTTP Version)	11	"HTTP/1.0"	19 (Header Host)	14	10.123.45.1

An example of how to find and extract the content of a specific TLV from the metadata buffer follows:

```
_i32 ExtractLengthFromMetaData(_u8 *pMetaDataStart, _u16 MetaDataLen)
{
    _u8 *pTlv;
    _u8 *pEnd;
    _u8 Type;
    _u16 TlvLen;

    pTlv = pMetaDataStart;
    pEnd = pMetaDataStart + MetaDataLen;

    while (pTlv < pEnd)
    {
        Type = *pTlv; /* Type is one byte */
        pTlv++;
        TlvLen = *(_u16 *)pTlv; /* Length is two bytes */
        pTlv+=2;

        if (Type == SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_CONTENT_LEN)
        {
            _i32 LengthFieldValue=0;

            /* Found the right type, extract its value and return. */
            memcpy(&LengthFieldValue, pTlv, TlvLen);
            return LengthFieldValue;
        }
        else
        {
            /* Not the type we are looking for. Skip over the
            value field to the next type. */
            pTlv += TlvLen;
        }
    }

    return -1;
}

/* NetApp request handler*/
void NetAppRequestHandler(  SlNetAppRequest_t *pNetAppRequest,
                            SlNetAppResponse_t *pNetAppResponse)
{
    _u32 HttpContentLength;

    if (pNetAppRequest->requestData.MetadataLen > 0)
    {
        HttpContentLength = ExtractLengthFromMetaData(
            pNetAppRequest->requestData.pMetadata,
            pNetAppRequest->requestData.MetadataLen);
    }
}
```

8.7.2 GET Processing

When the HTTP server receives an HTTP GET request for a resource which is not a ROM or user page, the HTTP handler (as shown in the preceeding example) is invoked with SL_NETAPP_REQUEST_HTTP_GET as the request type. The handler function must parse the HTTP metadata, extract the resource name and any other fields of interest, and generate a response. The host may choose to respond immediately by filling all response fields in the handler function. Alternatively, the host can fill the status field to “pending”, and return, which means another part of the user application must complete the response using the sl_NetAppSend API (as shown in the examples that follow).

8.7.2.1 Fragmentation

The host may choose to send the resource as a single chunk as part of the response from the handler (the payload fields in the ResponseData structure), or split it across multiple fragments. Fragmentation must be used to transfer resources larger than 1500 bytes (this is also the maximal size of a single fragment). Without fragmentation, the entire resource data is sent as part of the response from the handler. With fragmentation, the handler does not return anything but the pending status, while the fragments of the response are sent using the sl_NetAppSend API. Each fragment may be a different size (but smaller than 1500 bytes). While there are more fragments to send, the SL_NETAPP_REQUEST_RESPONSE_FLAGS_CONTINUATION bit must be set in the flags parameter of the API. On the last fragment, this bit must be zero. The first call to sl_NetAppSend API must carry the metadata (HTTP headers) of the response. For that, the SL_NETAPP_REQUEST_RESPONSE_FLAGS_METADATA bit must be set in the flags parameter of the API. Figure 8-11 demonstrates the handling of a GET request without (1) and with (2) fragmentation.

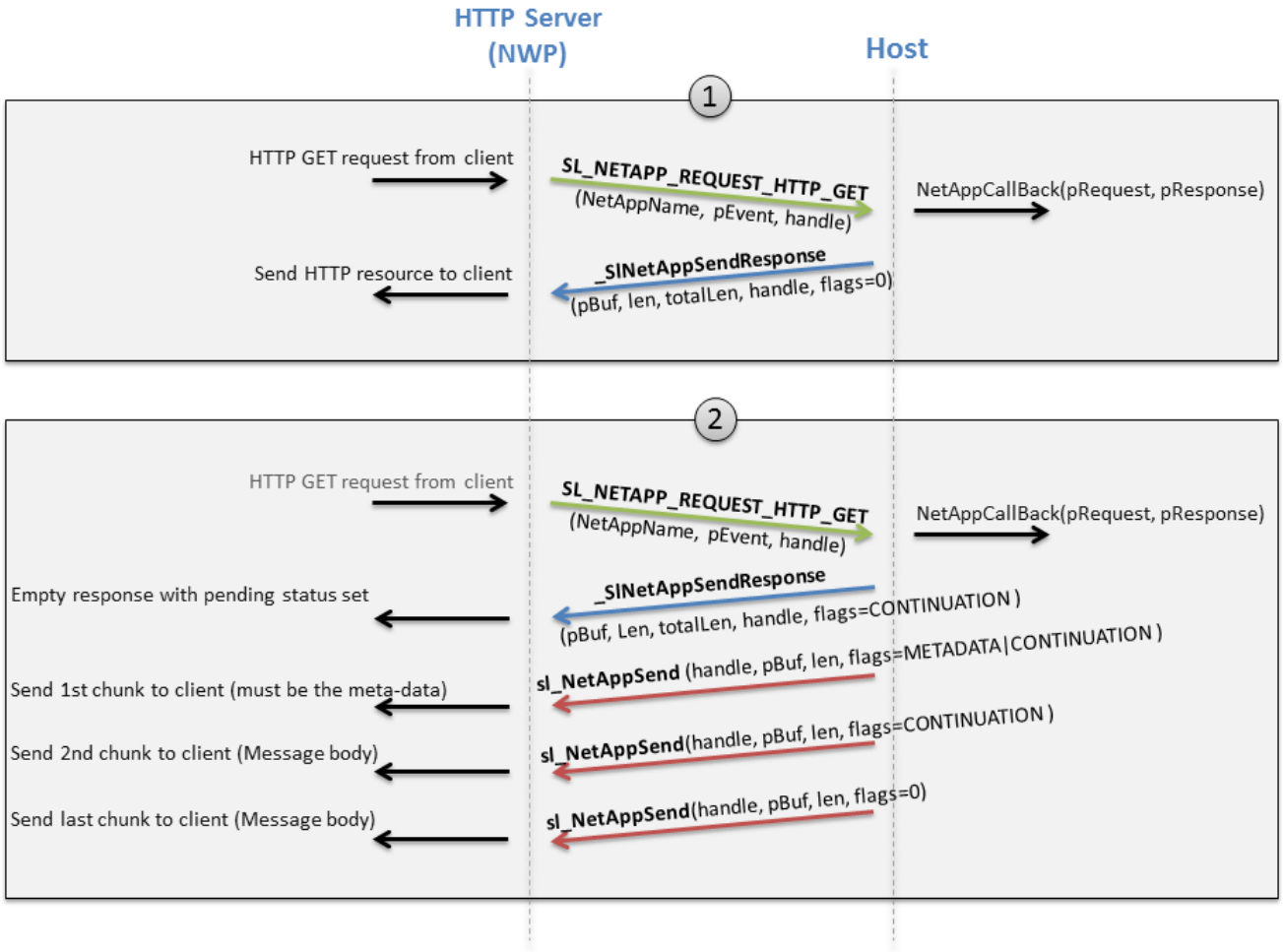

Figure 8-11. GET Request With and Without Fragmentation

The following code demonstrates how to implement an HTTP GET handler that sends a response (a short text string) immediately. The code assumes that the macro slcb_NetAppRequestHdlr is mapped to NetAppRequestHandler in file user.h.

```
#define RESPONSE_TEXT "Example text to be displayed in browser"

void NetAppRequestHandler(  SlNetAppRequest_t *pNetAppRequest,
                            SlNetAppResponse_t *pNetAppResponse)
{
    char *contentType = "text/html";
    unsigned char *pMetadata;
    unsigned char *pResponseText;

    pMetadata = (unsigned char*)malloc(128);
    pResponseText = (unsigned char*)malloc(sizeof(RESPONSE_TEXT));
    if ((NULL == pMetadata) || (NULL == pResponseText))
    {
        /* Allocation error */
    }
    memcpy(pResponseText, RESPONSE_TEXT, sizeof(RESPONSE_TEXT));

    switch(pNetAppRequest->Type)
    {
        case SL_NETAPP_REQUEST_HTTP_GET:
        {
            pNetAppResponse->Status = SL_NETAPP_HTTP_RESPONSE_200_OK;
            /* Write the content type TLV to buffer */
            pNetAppResponse->ResponseData.pMetadata = pMetadata;
            *pMetadata =
                (_u8) SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_CONTENT_TYPE;
            pMetadata++;
            *(_u16 *)pMetadata = (_u16) strlen (contentType);
            pMetadata+=2;
            memcpy (pMetadata, contentType, strlen(contentType));
            pMetadata+=strlen(contentType);
            /* Write the content length TLV to buffer */
            *pMetadata = SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_CONTENT_LEN;
            pMetadata++;
            *(_u16 *)pMetadata = 2;
            pMetadata+=2;
            *(_u16 *) pMetadata = (_u16) sizeof(RESPONSE_TEXT);
            pMetadata+=2;
            /* Calculate and write the total length of meta data */
            pNetAppResponse->ResponseData.MetadataLen =
                pMetadata - pNetAppResponse->ResponseData.pMetadata;
            /* Write the text of the response */
            pNetAppResponse->ResponseData.PayloadLen = sizeof(RESPONSE_TEXT);
            pNetAppResponse->ResponseData.pPayload = pResponseText;
            pNetAppResponse->ResponseData.Flags = 0;
        }
        break;
        default:
            /* POST/PUT/DELETE requests will reach here. */
        break;
    }
}
```

The following code demonstrates how to implement HTTP GET handler that delegates the request to some other application. The user must extract any relevant information from the request and save it as the data buffers are freed when the handler returns.

```
void NetAppRequestHandler(  SlNetAppRequest_t *pNetAppRequest,
                            SlNetAppResponse_t *pNetAppResponse)
{
    switch(pNetAppRequest->Type)
    {
        case SL_NETAPP_REQUEST_HTTP_GET:
```

```
            {
                /* Prepare pending response */
                pNetAppResponse->Status = SL_NETAPP_RESPONSE_PENDING;
                pNetAppResponse->ResponseData.pMetadata = NULL;
                pNetAppResponse->ResponseData.MetadataLen = 0;
                pNetAppResponse->ResponseData.pPayload = NULL;
                pNetAppResponse->ResponseData.PayloadLen = 0;
                pNetAppResponse->ResponseData.Flags = 0;

                /* Copy to some global buffer any relevant info from  pNetAppRequest (the handle
                In particular) and signal the user application that a new HTTP request has arrived. */
            }
            break;

            default:
                /* POST/PUT/DELETE requests will reach here. */
            break;
        }
    }
```

When signaled, the user application can then send this suggested response:

```
#define RESPONSE_TEXT "Example text part 1 --- "
#define RESPONSE_TEXT2 "Example text part 2"

_u8 *metadataBuff;
_u8 *pResponseText;
_u8 *pMetadata;
_u16 MetadataLen = 0;
const _u8 *contentType =  "text/html";
_u8 Flags = 0;
_u16 TextLength;

metadataBuff = (_u8 *) malloc (128);
pMetadata = metadataBuff;

/* HTTP status is sent as part of the meta-data*/
*pMetadata = (_u8) SL_NETAPP_REQUEST_METADATA_TYPE_STATUS;
pMetadata++;
*(_u16 *)pMetadata = (_u16) 2;
pMetadata+=2;
*(_u16 *)pMetadata = (_u16) SL_NETAPP_HTTP_RESPONSE_200_OK;
pMetadata+=2;

/* Write the content type TLV to buffer */
*pMetadata = (_u8) SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_CONTENT_TYPE;
pMetadata++;
*(_u16 *)pMetadata = (_u16) strlen((char*)contentType);
pMetadata+=2;
memcpy (pMetadata, contentType, strlen((char*)contentType));
pMetadata+=strlen((char*)contentType);

/* Write the content length TLV to buffer */
*pMetadata = SL_NETAPP_REQUEST_METADATA_TYPE_HTTP_CONTENT_LEN;
pMetadata++;
*(_u16 *)pMetadata = 2;
pMetadata+=2;
TextLength = sizeof(RESPONSE_TEXT) + sizeof(RESPONSE_TEXT2);
*(_u16 *) pMetadata = TextLength;
pMetadata+=2;

MetadataLen = pMetadata - metadataBuff;

/* First send the meta-data (note the METADATA flag).
Continuation flag indicates there are more fragments to follow.
gHandle is assumed to be populated by the handler.  */
Flags |= SL_NETAPP_REQUEST_RESPONSE_FLAGS_CONTINUATION;
```

```
Flags |= SL_NETAPP_REQUEST_RESPONSE_FLAGS_METADATA;
sl_NetAppSend (gHandle, MetadataLen, metadataBuff, Flags);

/* Send first data fragment. Continuation flag still
indicates there are more fragments to follow, */
Flags = SL_NETAPP_REQUEST_RESPONSE_FLAGS_CONTINUATION;
pResponseText = (_u8 *) malloc (sizeof(RESPONSE_TEXT));
memcpy(pResponseText, RESPONSE_TEXT, sizeof(RESPONSE_TEXT));
sl_NetAppSend (gHandle, sizeof(RESPONSE_TEXT), pResponseText, Flags);

/* Last data fragment - continuation flag is cleared. */
Flags = 0;
pResponseText = (_u8 *) malloc (sizeof(RESPONSE_TEXT2));
memcpy(pResponseText, RESPONSE_TEXT2, sizeof(RESPONSE_TEXT2));
sl_NetAppSend (gHandle, sizeof(RESPONSE_TEXT2), pResponseText, Flags);
```

8.7.3 POST Processing

POST requests that were not recognized as RESTFul APIs are transferred to the host with SL_NETAPP_REQUEST_HTTP_POST as the request type. The user handler must parse the HTTP metadata, extract the resource name and any other fields of interest, and generate a response. The host may choose to respond immediately by filling all response fields in the handler. Alternatively, the host can fill the status field to pending and return, which means another part of the user application must complete the reception of the request using the sl_NetAppRecv API. Then it must use the sl_NetAppSend API to send a response. Figure 8-12 shows the data flow when the response is sent immediately.

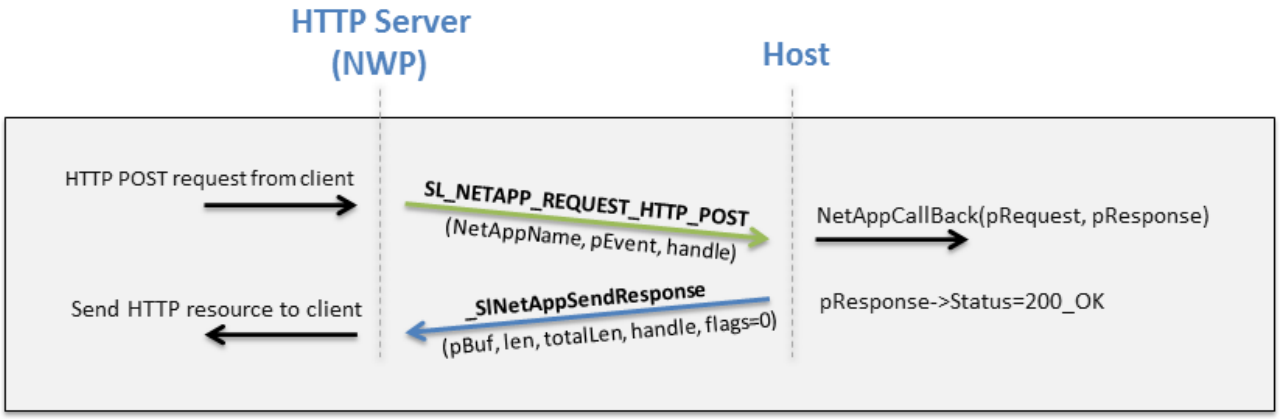

Figure 8-12. POST Processing Flow

Similarly to Figure 8-12, the following code receives acknowledgments for every POST request to the host with an HTTP 200 OK response:

```
void NetAppRequestHandler(  SlNetAppRequest_t *pNetAppRequest,
                            SlNetAppResponse_t *pNetAppResponse)
{
    extern _u16 gHandle;

    switch(pNetAppRequest->Type)
    {
        case SL_NETAPP_REQUEST_HTTP_POST:
        {
            pNetAppResponse->Status = SL_NETAPP_HTTP_RESPONSE_200_OK;
            pNetAppResponse->ResponseData.pMetadata = NULL;
            pNetAppResponse->ResponseData.MetadataLen = 0;
            pNetAppResponse->ResponseData.pPayload = NULL;
            pNetAppResponse->ResponseData.PayloadLen = 0;
            pNetAppResponse->ResponseData.Flags = 0;
```

```
            }
            break;

            default:
                /* GET/PUT/DELETE requests will reach here. */
            break;
        }
    }
```

8.7.3.1 Long Requests and Delayed Responses

Only the first 1364 bytes of the request are passed to the handler (this includes the meta-data). The reset (if present) should be requested using the sl_NetAppRecv API outside the handler. The user may choose at what fragment size to pull the remaining payload from the device. The last fragment indicates when the flags returned by the sl_NetAppRecv API no longer contain the continuation flag. The same flow can be used if the response cannot be determined by the NetApp handler and must be delegated to another process. In this case, the handler must fill the response filed as pending and return. The process must then be invoked to retrieve the reset of the request (if present) and actually send the response. Figure 8-13 demonstrates the data flow with delayed response.

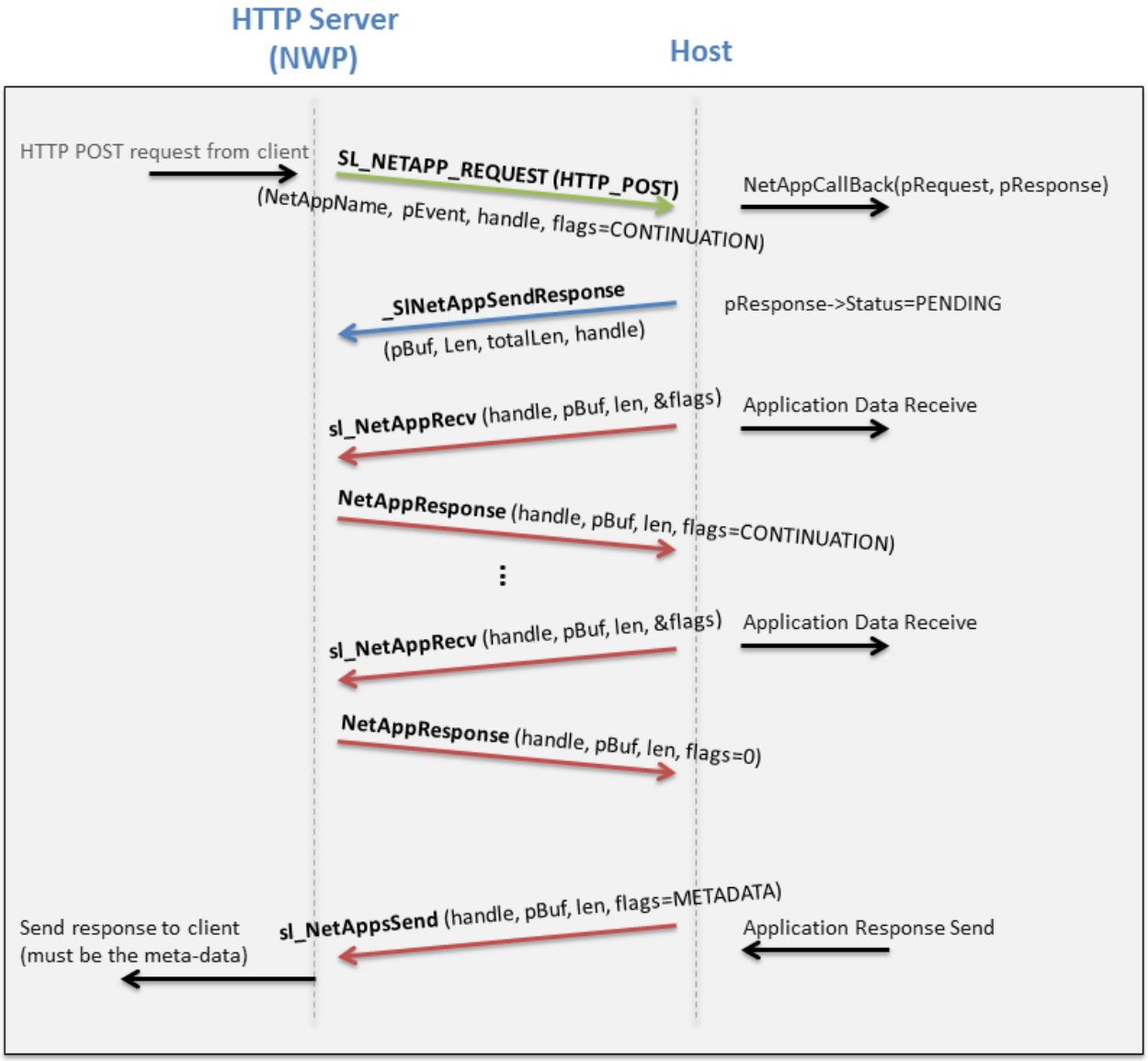

Figure 8-13. Delayed Response

The following code implements a handler for a POST request. It sends an HTTP 200 OK response immediately if the entire request was received, or sets the pending status and signals the user application to handle the remaining of the request. It also parses the metadata in search for the content length field, which represents the expected size of the payload, and extracts it. This field (similar to all other metadata) is not generated by the SimpleLink device, but transferred as is, and must be validated by the user.

```
void NetAppRequestHandler(  SlNetAppRequest_t *pNetAppRequest,
                            SlNetAppResponse_t *pNetAppResponse)
{
    extern _u16 gHandle;

    switch(pNetAppRequest->Type)
    {
        case SL_NETAPP_REQUEST_HTTP_POST:
        {
            _u32 RequestFlags;
            _u32 ContentLength;
```

```
                RequestFlags = pNetAppRequest->requestData.Flags;

                /* Prepare pending response */
                pNetAppResponse->ResponseData.pMetadata = NULL;
                pNetAppResponse->ResponseData.MetadataLen = 0;
                pNetAppResponse->ResponseData.pPayload = NULL;
                pNetAppResponse->ResponseData.PayloadLen = 0;
                pNetAppResponse->ResponseData.Flags = 0;

                if (pNetAppRequest->requestData.MetadataLen > 0)
                {
                    /* Process the meta data in
                    pNetAppRequest->requestData.pMetadata */
                    ContentLength = ExtractLengthFromMetaData(
                        pNetAppRequest->requestData.pMetadata,
                        pNetAppRequest->requestData.MetadataLen);
                    /* Allocate buffer to receive the entire content if needed */
                }

               if (pNetAppRequest->requestData.PayloadLen > 0)
               {
                /* First fragment of the payload is @
                pNetAppRequest->requestData.pPayload */
               }

                if (RequestFlags & SL_NETAPP_REQUEST_RESPONSE_FLAGS_CONTINUATION)
                {
                    /* More fragments to follow. */
                    pNetAppResponse->Status = SL_NETAPP_RESPONSE_PENDING;
                    /* Signal the user application to receive the rest.*/
                    SetEvent(g_netAppRequestSyncObj);
                    /* The handle will be used to receive the rest + send response*/
                    gHandle = pNetAppRequest->Handle;
                }
                else
                {
                    pNetAppResponse->Status = SL_NETAPP_HTTP_RESPONSE_200_OK;
                }

                break;
            }

            default:
            /* GET/PUT/DELETE requests will reach here. */
            break;
        }
    }
```

The following code can be placed in the user application, to retrieve the remaining fragments and send a response in the end when signaled from the preceeding handler.

```
_u8 *MetadataBuff;
_u8 *pMetadata;
_u16 MetadataLen = 0;
_u8 Fragment[100]; /* Fragment buffer of arbitrary size */
_u16 FragmentLen;
_SlReturnVal_t RetVal;
_u32 Flags;

do
{
    FragmentLen = sizeof(Fragment); /* Indicates max buffer size */
    RetVal = sl_NetAppRecv(gHandle, &FragmentLen, Fragment, &Flags);
    if ((RetVal < 0) | (Flags & SL_NETAPP_REQUEST_RESPONSE_FLAGS_ERROR))
    {
        // API error, abort. Error code can be extracted as follows:
```

```
            // ErrorCode = (short)(0x0000ffff & Flags);
        }

        /* Process the received fragment here.
         FragmentLen contains the actual fragment size. */

    } while (Flags & NETAPP_REQUEST_RESPONSE_FLAGS_CONTINUATION);

    /* Send response OK */
    MetadataBuff = (_u8*) malloc(128);
    pMetadata = MetadataBuff;
    *pMetadata = (_u8) SL_NETAPP_REQUEST_METADATA_TYPE_STATUS;
    pMetadata++;
    *(_u16 *)pMetadata = (_u16) 2;
    pMetadata+=2;
    *(_u16 *)pMetadata = (_u16) SL_NETAPP_HTTP_RESPONSE_200_OK;
    pMetadata+=2;
    MetadataLen = 5;

    Flags = SL_NETAPP_REQUEST_RESPONSE_FLAGS_METADATA;
    sl_NetAppSend (gHandle, MetadataLen, MetadataBuff, Flags);
```

There is no way to return payload data as part of the response, only HTTP headers as part of the metadata.

8.7.4 PUT Processing

PUT requests are similar to POST requests. The only difference (aside from the command type received) is that there is no processing of RESTful APIs; all requests are transferred directly to the host. As in POST, the response cannot contain any payload, only HTTP headers.

8.7.5 DELETE Processing

DELETE requests are similar to POST requests. The only difference (aside from the command type received) is that there is no processing of RESTful APIs; all requests are transferred directly to the host. As in POST, the response cannot contain any payload, only HTTP headers.

8.8 Security

8.8.1 Authentication

When authentication is enabled (see *Section 8.3*), the client must provide a username and password before the HTTP server processes any requests. Both user name and password are limited to 20 characters, and both are case sensitive.

8.8.1.1 HTTP Realm

A realm in HTTP context is a group of resources protected by the same username and password. Therefore, it is relevant only when authentication is enabled. All resources served by the SimpleLink HTTP server (including those residing in the host) belong to one realm. The name of this realm can be set as described in *Section 8.3*. The realm name is presented in the client browser when it prompts for username and password.

8.8.2 Secure Connection

The HTTP server can accept connections over a secure socket (SSL). When enabled, the primary server port accepts only secure connections, and unsecure connection requests are rejected. The secondary port can be enabled to redirect nonsecure connection attempts to the primary (secure) port. This scheme is commonly used to redirect browsers, which by default initiate a nonsecure connection on port 80. When the secure connection is enabled, a server certificate and a private key must be placed on the file system in PEM or DER format, and their names must be configured in the HTTP server. The following example shows how to enable the secure socket and use the secondary socket for redirection.

```
unsigned char ServerCertificateFileName[] = "server-cert.der";
unsigned char ServerKeyFileName[] = "server-key.der";
unsigned char SecurityMode[] = {0x1};
unsigned char HttpsPort[] = {0xBB, 0x01}; // 0x1BB = 443
unsigned char SecondaryPort[] = {0x50, 0x00}; // 0x050 = 80
unsigned char SecondaryPortEnable[] = {0x1};

// Set the file names used for SSL key exchange.
sl_NetAppSet(SL_NETAPP_HTTP_SERVER_ID,
                SL_NETAPP_HTTP_DEVICE_CERTIFICATE_FILENAME,
                sizeof(ServerCertificateFileName),
                ServerCertificateFileName);

sl_NetAppSet(SL_NETAPP_HTTP_SERVER_ID,
                SL_NETAPP_HTTP_PRIVATE_KEY_FILENAME,
                sizeof(ServerKeyFileName),
                ServerKeyFileName);

// Activate SSL security on primary HTTP port and change it to
// 443 (standard HTTPS port)
sl_NetAppSet(SL_NETAPP_HTTP_SERVER_ID,
                SL_NETAPP_HTTP_PRIMARY_PORT_SECURITY_MODE,
                sizeof(SecurityMode),
                SecurityMode);

sl_NetAppSet(SL_NETAPP_HTTP_SERVER_ID,
                SL_NETAPP_HTTP_PRIMARY_PORT_NUMBER,
                sizeof(HttpsPort),
                HttpsPort);

// Enable secondary HTTP port (can only be used for redirecting
// connections to the secure primary port).
sl_NetAppSet(SL_NETAPP_HTTP_SERVER_ID,
                SL_NETAPP_HTTP_SECONDARY_PORT_NUMBER,
                sizeof(SecondaryPort),
                SecondaryPort);

sl_NetAppSet(SL_NETAPP_HTTP_SERVER_ID,
                SL_NETAPP_HTTP_SECONDARY_PORT_ENABLE,
                sizeof(SecondaryPortEnable),
                SecondaryPortEnable);

// Restart HTTP server for new configuration to take effect.
sl_NetAppStop(SL_NETAPP_HTTP_SERVER_ID);
sl_NetAppStart(SL_NETAPP_HTTP_SERVER_ID);
```

It is also possible to require client authentication by providing a Root CA file using the SL_NETAPP_HTTP_CA_CERTIFICATE_FILE_NAME option. If provided, all client connections are verified, and those failing the test are not accepted. SSL client verification is described in more detail in *Section 4.3*.

NOTE: Currently internal HTTPs server supports only RSA cipher suite due to performance optimization.

8.9 Other

8.9.1 Processing of Parallel Requests

Each HTTP request is handled over a single TCP connection. The client initiates a connection, and sends the request. The server processes the request and sends the response over the same connection, closing it once sent. The server then waits to accept a new connection. Even though only one request can be processed at any given time, many clients can initiate TCP connections to the server simultaneously, and each request is handled in order.

mDNS

Topic Page

9.1 Introduction

The mDNS/DNS-SD protocol enables the automatic discovery of computers, devices, and services by resolving IP addresses and ports on the local IP network. mDNS is based on the DNS protocol. In contrast to DNS, which uses a DNS server, mDNS protocol is distributed, where each device can advertise and discover services. Each mDNS device on the local IP network can join an mDNS IP multicast group, and advertises its services. mDNS protocol supports IPv4 and IPv6 local networks. IPv4 multicast address 224.0.0.251, IPv6 multicast address FF02::FB, and UDP port 5353 are all reserved to mDNS messages.

The SimpleLink host application can register up to five services. The services can be advertised and discovered on IPv4 network, IPv6 network, or both networks, depending on service interface registration and interface status.

The mDNS service must be enabled to allow query and advertisement operations. By default, the mDNS service is enabled and the internal HTTP server and host name are advertised on the enabled interfaces, IPv4, IPv6, or both. The mDNS service can be disabled.

The host application can trigger one-shot or continuous discovery. The results are cached by the SimpleLink Wi-Fi device, and the application can retrieve the list of discovered devices and services.

The mDNS service is not power-wise-optimized; therefore, TI recommends turning this service off in power-constrained systems. This service is turned off automatically if the configured power mode is LSI with a sleep time greater than 2000 ms.

9.2 Key Features

Table 9-1 lists the key features of the mDNS.

Table 9-1. Key Features

Key Features	Description
Advertise IPv4/IPv6 services	Advertise up to five registered services IPv4, IPv6, or both. If internal HTTP server is disabled, six services can be registered.
Discover IPv4/IPv6 services	Discover services IPv4, IPv6, or both
One-shot discovery	Support IPv4\IPv6 single query
Continuous discovery	Support IPv4\IPv6 continuous query
Mask services	Support masking specific services types in the discovery process
Set advertisement timing	Set advertisement timing parameters
Update Service text	Update existing services text field

9.3 Configurations and Settings

Starting or stopping mDNS service: mDNS service is enabled by default. mDNS can be stopped and started by the host application by using the host APIs sl_NetAppStart and sl_NetAppStop. This action takes effect immediately and reset is not required. This configuration is persistent according to system-persistent configuration.

Example:

```
_i16 Status;

/* Start mDNS */
Status = sl_NetAppStart(SL_NET_APP_MDNS_ID);
if( Status )
{
    /* Error */
}
```

9.4 Query

The SimpleLink Wi-Fi device can discover remote services on the local network. The discovery is performed by sending one-shot or continuous queries. The queries are transmitted on IPv4 or IPv6 interfaces, according to the host request and configuration.

9.4.1 One Shot Query

The SimpleLink Wi-Fi device can issue one-shot queries in which the device triggers only a single mDNS query to the network by calling the API sl_NetAppDnsGetHostByService. The query can be set as IPv4 or IPv6 (If enabled) format, or both. A discovery result returns the first response received with information regarding the remote service: IP address, port, and service text description.

Example:

```
_i16 Status;
_i8 query[] = "_http._tcp.local";
_u32 addr;
_u32 Port  = 0;
_u16 TextLen = 800;
_i8  pText[800];

Status =  sl_NetAppDnsGetHostByService(query, (unsigned char)strlen(&query[0]), SL_AF_INET,
&addr, &Port, &TextLen,pText);
if( Status )
{
    /* Error */
}
```

9.4.2 Continuous Query

In a continuous mDNS query mode, the device keeps sending queries to the network according to a specific service name. The queries are sent in IPv4 and IPv6 (if enabled) formats or both. To see the complete list of responding services, sl_NetAppGetServiceList must be called. To stop the continuous query, call the same API with length 0.

Continuous query configuration is persistent by default, and can be set according to a system-persistent configuration.

Example:

```
_i16 Status;
_i8 query[] = "_http._tcp.local";

/* Start continues query */
Status = sl_NetAppSet(SL_NETAPP_MDNS_ID, SL_NETAPP_MDNS_CONT_QUERY_OPT, (unsigned
char)strlen(&query[0]), query);
if( Status )
{
    /* Error */
}
/* Stop continues query */
Status = sl_NetAppSet(SL_NETAPP_MDNS_ID, SL_NETAPP_MDNS_CONT_QUERY_OPT,0  , 0); /* Set length to
zero to stop continuous query */
if( Status )
{
    /* Error */
}
```

9.4.3 Mask Services

The SimpleLink Wi-Fi device offers the ability to predefine specific services types to monitor. If the host application decides not to get responses from certain types of services (not stored in the cache), the adapt bit can be set in the event mask related to according to the following list:

- _ipp – bit 0
- _divice-info – bit 1
- _http – bit 2
- _https – bit 3
- _workstation – bit 4
- _guid – bit 5
- _h323 – bit 6
- _ntp – bit 7
- _objective – bit 8
- _rdp – bit 9
- _remote – bit 10
- _rtsp – bit 11
- _sip – bit 12
- _smb – bit 13
- _soap – bit 14
- _ssh – bit 15
- _telnet – bit 16
- _tftp – bit 17
- _xmpp-client – bit 18
- _raop – bit 19

Example:

```
_i16 Status;
_u32 EventMask;

EventMask = BIT0 | BIT1 | BIT18;
Status  = sl_NetAppSet(SL_NETAPP_MDNS_ID,
SL_NETAPP_MDNS_QEVETN_MASK_OPT,sizeof(EventMask),&EventMask);
if( Status )
{
    /* Error */
}
```

9.5 Get Service List

The SimpleLink device can return a list of peer services, which are stored in the device, without issuing any queries (relying on previously collected data stored in the cache). The list is in a form of a service structure, which can include full-service parameters with text, full-service parameters, or short-service parameters (port and IP only), dedicated for hosts with memory limitations (for different size of buffers). The list size can store up to eight services, and when a new service is discovered, the oldest service entry is replaced. The list is cleared when mDNS service is disabled or if Wi-Fi disconnects.

The host can retrieve or return different levels of details to support memory reduction in the host application:

- IPv4/IPv6 full-service parameters – IP address, port, service name, service host, and service text
- IPv4/IPv6 partial-service parameters – IP address, port, service name, and service host
- IPv4/IPv6 minimal-service parameters – IP address and port only

Example:

```
_i16 Status;
SlNetAppGetShortServiceIpv4List_t  listMdns[6];

 /* Get a list of discovered services */
Status = sl_NetAppGetServiceList(0, 6, /* Maximum number of services to receive */
```

```
        SL_NETAPP_FULL_SERVICE_WITH_TEXT_IPV6_TYPE, /* receive full ipv6 services with text */
        (_i8*) &listMdns6[0], sizeof(listMdns));
if( Status )
{
    /* Error */
}
```

9.6 Advertisement

9.6.1 Registering mDNS Services

Registration of a new service should be performed only if the mDNS service is enabled (it is enabled by default). Services can be registered as IPv4 or IPv6 services, or both. All registered services are advertised at once. Each service includes a name, text description, port number, and TTL (time to live) value. The registered service is persistent by default, unless set otherwise by using the flag SL_NETAPP_MDNS_OPTIONS_IS_NOT_PERSISTENT.

The maximum number of registered services is five (or six if the internal web server is not running). The following flags can be set when registering the service:

- SL_NETAPP_MDNS_OPTIONS_IS_UNIQUE_BIT – Set service as unique.
- SL_NETAPP_MDNS_IPV6_IPV4_SERVICE – Service is set for IPV4 and IPV6 interfaces (IPV6 should be enabled).
- SL_NETAPP_MDNS_IPV4_ONLY_SERVICE – Service is set for IPV4 interface only (default mode).
- SL_NETAPP_MDNS_IPV6_ONLY_SERVICE – Service is set for IPV6 interface only (IPV6 should be enabled).
- SL_NETAPP_MDNS_OPTION_UPDATE_TEXT – Update text fields (without reregistering the service).
- SL_NETAPP_MDNS_OPTIONS_IS_NOT_PERSISTENT – Set a nonpersistent service.

Example:

```
_i16 Status;
_u32 Options;
const signed char AddService[40] = "printer._ipp._tcp.local";

Options = SL_NETAPP_MDNS_OPTIONS_IS_NOT_PERSISTENT | SL_NETAPP_MDNS_IPV4_ONLY_SERVICE;
Status  = sl_NetAppMDNSRegisterService(AddService, strlen(AddService),
"Service 5;payper=A4;size=10",strlen("Service 5;payper=A4;size=10"),4578,120,Options);
if( Status )
{
    /* Error */
}
```

9.6.2 Unregistering mDNS Services

A registered service can be unregistered according to its registered name. Setting the length variable to zero deletes all services at once.

If the service was originally created as persistent, it is optional to unregister it as persistent or as nonpersistent:

- Unregister the service with the nonpersistent flag to cause the service to be currently deleted (send advertisement with TTL set to 0) until the device resets, which then returns to advertise the service with the original configured TTL.
- Unregister the service as persistent, to cause the service to be permanently deleted (send advertisement with TTL set to 0); also after reset.

If the service was originally created as nonpersistent, unregister should apply with the nonpersistent flag accordingly, otherwise an error returns.

Example:

```
_i16 Status;
_u32 Options;
```

```
const signed char AddService[40] = "printer._ipp._tcp.local";

Options = SL_NETAPP_MDNS_OPTIONS_IS_NOT_PERSISTENT;
Status = sl_NetAppMDNSUnRegisterService(AddService,strlen(AddService),Options);
if( Status )
{
    /* Error */
}
```

9.6.3 Advertisement Settings

9.6.3.1 Timing

This option allows the control and reconfiguration of the timing parameters for all services advertisements. The API includes a unique structure for this specific configuration, with the following parameters:

- T – Number of ticks for the initial period. Default is 100 ticks for 1 second.
- P – Number of repetitions. Default value is 1.
- K – Increasing interval factor. Default value is 2.
- Retransmission interval – Number of ticks to wait before sending out repeated announcement message. Default value is 0.
- Max interval – Period interval. Number of ticks between two announcement periods. Default value is 0xFFFFFFFF.
- Max time – Maximum time of an announcing period, default value is 3 seconds.

For example, if period is set to T, repetitions are set to P, – increasing interval factor is K = 2, the transmission shall be: advertise P times, wait T, advertise P times, wait 4 × T, advertise P time, wait 16 × T ... (until max time reached / configuration changed /query issued).

Example:

```
_i16 Status;
SlNetAppServiceAdvertiseTimingParameters_t Timing;

Timing.t = 200; /* 2 seconds */
Timing.p = 2;   /* 2 repetitions */
Timing.k = 2;   /* Telescopic factor 2 */
Timing.RetransInterval = 0;
Timing.Maxinterval = 0xFFFFFFFF;
Timing.max_time = 5;

Status = sl_NetAppSet(SL_NETAPP_MDNS_ID, SL_NETAPP_MDNS_TIMING_PARAMS_OPT,sizeof(Timing),&Timing);
if( Status )
{
    /* Error */
}
```

9.6.3.2 Update Text

The SimpleLink device offers the ability to update the text field for registered services. The update can be performed for the text field only. The API must be applied with the previous registered service name. If the service was originally created as persistent, it is optional to update the text field as persistent or as nonpersistent:

- Updating the text with the nonpersistent flag causes the service to hold the updated text until the device resets, which then returns to the original text.
- Updating the text as persistent causes the service to store the updated text, even after a device reset.

If the service was originally created as nonpersistent, updating the text should apply with the nonpersistent flag accordingly, otherwise an error returns.

Example:

```
_i16 Status;
```

```
_u32  Options;
const signed char AddService[40] = "printer._ipp._tcp.local";

/* Update Service text (as persistent)*/
Options = SL_NETAPP_MDNS_OPTIONS_IS_UNIQUE_BIT | SL_NETAPP_MDNS_IPV4_ONLY_SERVICE |
SL_NETAPP_MDNS_OPTION_UPDATE_TEXT;
Status = sl_NetAppMDNSRegisterService(AddService, sizeof(AddService),"Printer=2;Size=A3;size=8",
strlen("Printer=2;Size=A3;size=8"),4578,120,Options);
if( Status )
{
    /* Error */
}
```

9.7 Limitations

- The maximum number of registered services is five (or six, if the internal web server is not running).
- The size of the service total length should be smaller than 255 bytes.
- The size of the discovered services text length should be smaller than 120 bytes.
- This discovered service list is limited to eight services.

Rx Filters

Topic Page

10.1 Introduction

The Rx filter is a powerful feature which enables the host to save power consumption and reduce application code.

The host can define reception filters that have been processed by the device. Each frame is tested against the filters; if there is a match, the filter actions are executed. Filter actions can be set to drop the frame or send an event to the host.

The Rx filters can be used for implementing features such as Wake on LAN, in which the host can enter deep sleep until a specific frame is detected by the device, then wake up the host by sending the programmed event.

The Rx filter feature can filter frames by standard protocol fields of a frame (MAC, frame type, IP, and so forth), or by pattern on the frame payload.

The Rx filters are rule-based systems embedded in the SimpleLink Wi-Fi device. They let the user simply define a set of filters that determine which of the received frames will be dropped by the SimpleLink Wi-Fi device. They also let the user configure filters that trigger asynchronous events to the host.

Operating the Rx filter with the event mechanism can reduce the power consumption and code size of the host MCU. Using filters can also reduce the processing efforts of the SimpleLink Wi-Fi device itself, because frames can be dropped before their processing is finished.

The filters are organized in a tree structure, and the tree traversal is from the root to the leaf. The maximum number of supported filters is 64; 15 filters are used by the SimpleLink Wi-Fi device and have no access from the host, and 49 filters are available for the host to use.

The host interface includes the following operations:

- Create filter
- Update filter arguments
- Enable filters; the function enables several filters at once
- Disable filters; the function disables several filters at once
- Remove filters; the function deletes several filters at once
- Store filters; the function stores all the filters on the FS

Figure 10-1 describes the processing of the Rx filters in high level.

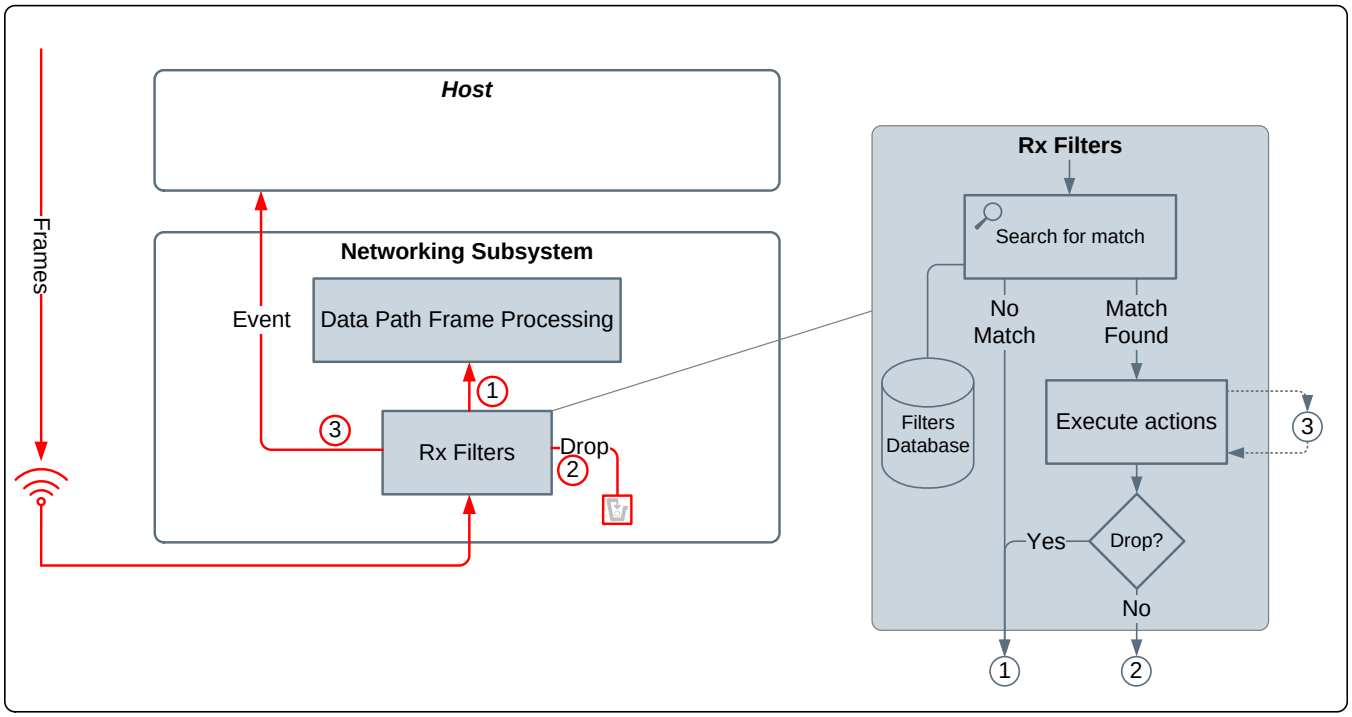

Figure 10-1. Rx Filters

The Rx filter module includes a database of filters, an interface for defining the filters from the host, a match process, and actions execution.

- Database – The filter database is created by the host application and contains the filters definitions and the relation between the filters. By default, the database is not persistent; when the device resets, the host application must redefine the filters. There is an option to store the filters database to the FS. If a database file exists, it automatically loads when the device is powered up.
- Host interface – The host has a simple interface which lets the user define, delete, enable, disable, or store existing filters. This interface is part of the WLAN silo.
- Matching process – The process verifies if a match exists between the received frame and a set of filters. For each filter, if a match is found, the filter actions are executed. If the actions do not include dropping the frame, the processing of the frame continues normally. If there is no match between the frame and any filter, the processing of the frame continues normally.
- Action execution – If a match is found, all the actions of the matched filter are executed. These actions are defined as part of the filter definition.

10.2 Matching Process

The filters database is organized as a series of decision trees, according to the network stack layers. During a reception of a frame, the networking subsystem passes through the filters and checks for matching between the filters and the received frame. The filters tree traversal is the process of passing through the filters, and is done such that any filter is visited a maximum of once per frame, and only the relevant filters are visited. The traversing is done layer by layer among all the trees, and the process stops when the frame reaches a drop action in one of the trees.

10.2.1 Filter Matching

The basic Rx filter contains three major attributes:

- **Trigger** is the precondition which should be fulfilled before the rule is checked, such as the system state. For example, the rule can be defined to be active only on promiscuous mode.

- **Rule** is the match criteria. It contains the compare filed name, the expected value, and the compare function. For example, the rule can be: source port is equal to 23.
- **Actions** are the operations that execute if the rule is matched.

The outcome of the filter matching could be: No Match, Pass, and Drop. Figure 10-2 shows the Rx filter matching flow.

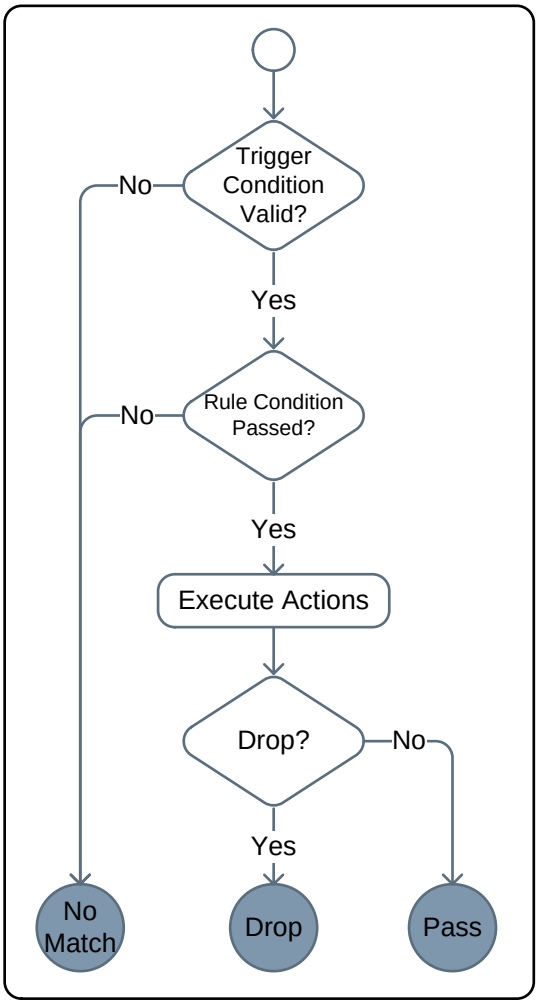

Figure 10-2. Rx Filter Match Flow

Table 10-1 lists the possible triggers of a filter.

Table 10-1. Possible Triggers

Trigger Type	Possible Values
Wi-Fi Mode	Station (Station Connected / Wi-Fi Direct client) AP (Access Point / Wi-Fi Direct GO) Promiscuous
Wi-Fi Connection State	Connected Disconnected
Counter Value	Numeric value

Table 10-2 lists the possible rules of a filter.

Table 10-2. Possible Rules

Rule Layer	Field Name
MAC	Frame type Frame subtype BSSID Source MAC address Destination MAC address Frame length Payload value
LLC	Protocol type
IP	IP version IP protocol Source IP address Destination IP address ARP operation ARP target IP address Source port number Destination port number Payload value

Table 10-3 lists all possible actions of a filter.

Table 10-3. Possible Actions

Action	Possible Values
Drop	Drop the frame and abort and processing on this frame.
Event	Send an asynchronous event to the host.
Counter	Increase or decrease counter value.

To perform a logical operation on filters such as logical OR or logical AND, create a special filter. This combined filter node has two parent nodes (unlike a regular node, which has one parent node), and is checked only if one or both (user-defined) of its parent nodes passed the match.

10.2.2 Tree Traversal

The filters are organized as a decision tree in layers. This structure enables the user to combine several filters to identify a specific frame; the division to filter layers optimizes the traversal processing. For example, three filters are required to detect a specific IP frame from a specific source MAC and a specific word in the payload:

- Filter 1: Specific source of the MAC address
- Filter 2: The packet protocol type is IP.
- Filter 3: The payload of the IP layer contains a specific word.

Filter 1 is the root, Filter 2 is a child of Filter 1, and Filter 3 is a child of Filter 2.

In this example, all of the filters are part of the same tree, but each filter is of a different layer. For every received frame the device traverses through a series of decision trees that determine how the frame is treated. The decision trees are composed of filter nodes. The tree traversal process starts with the root nodes of the trees:

- If a filter node passes the match, its actions are performed.
- For drop action, the packet is dropped and the matching process for this frame stops. For any other action, the frame matching process continues to its child nodes.
- If the filter node does not pass the match, the match does not proceed to the child nodes; however, the match process for this frame continues for other filter trees.
- In any case, packets that were not dropped during the matching process continue with the other (regular) network stack processing.

10.3 Examples of Filter Use

This section provides some basic examples of filters. The examples do not represent a real use case scenario, and their purpose is only to demonstrate and explain the structure of the Rx filters.

10.3.1 Example 1

The system has the following requirements:

- Receive only WLAN management beacon frames from all MAC addresses.

The following trees should be created:

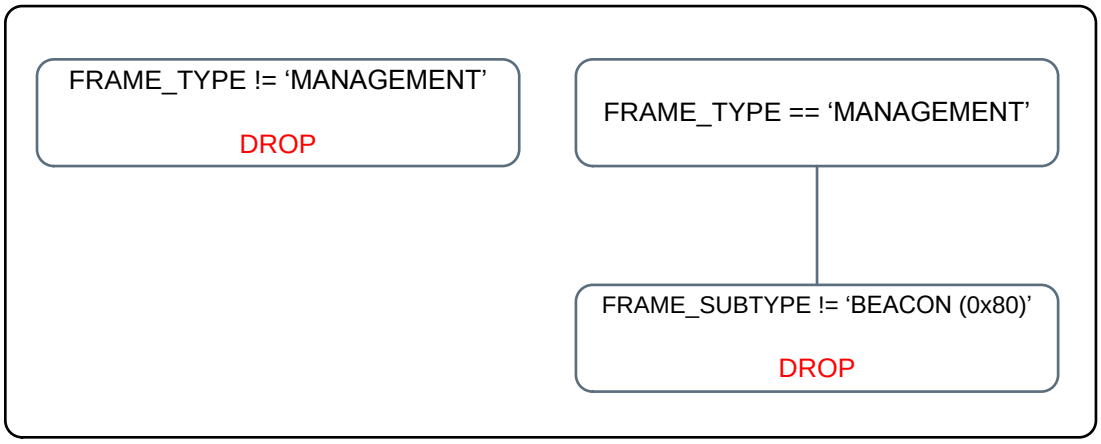

Figure 10-3. Example 1

10.3.2 Example 2

The system has the following requirements:

- Receive WLAN data broadcast frames only from two specific MAC addresses.
- Do not receive WLAN unicast frames from a certain SRC_IP address range.
- If a unicast frame is received from MAC address AA.BB.CC.DD.EE.FF, increase counter_1.
- If a unicast frame is received from MAC address CC.HH.II.JJ.KK.LL, increase counter_2.
- If a unicast UDP frame is received from MAC address AA.BB.CC.DD.EE.FF or CC.HH.II.JJ.KK.LL, pass only packets from port 5001.

The following filters should be created:

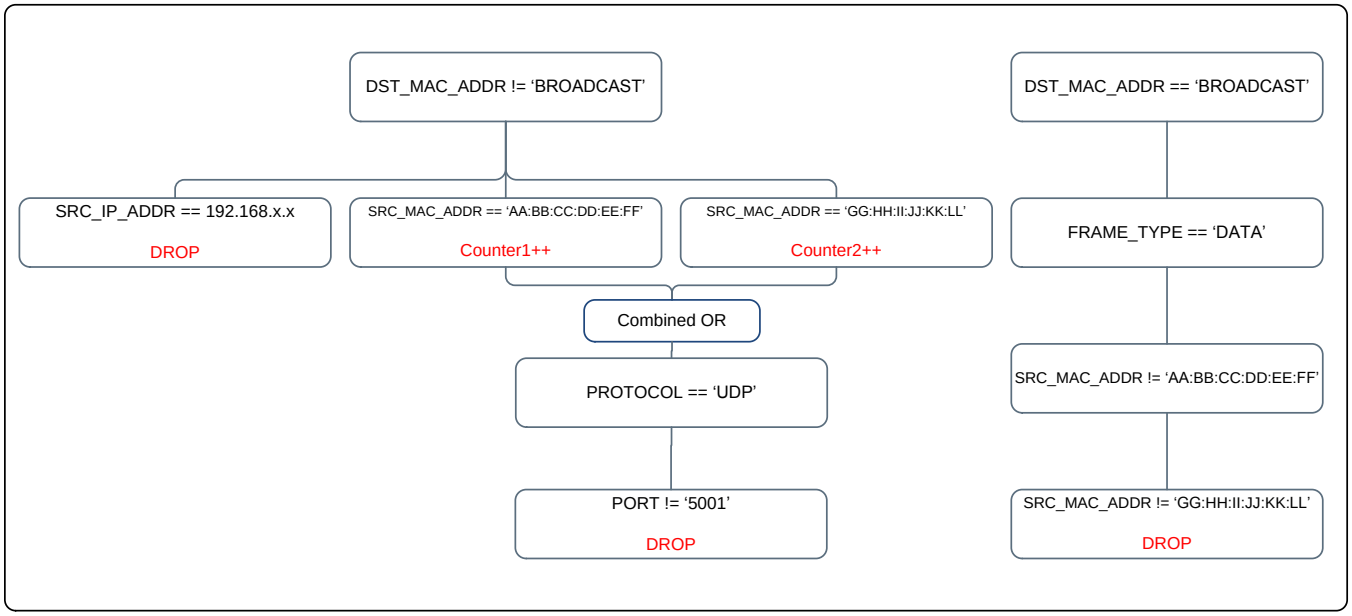

Figure 10-4. Example 2

10.4 Filter Creation

Application filters are created by the host application, and defined in a hierarchical way. The maximal number of application filters is 50. Application filters can be created, removed, enabled, disabled, and stored by the host application. During these operations, there are some transactional periods in which the filters might behave differently than the final behavior (it is possible that some of the filters are disabled and some enabled during these transactions). Therefore, TI recommends first creating each filter as disabled, and then enabling all of them at once (one enable command).

Stored filters are created once, stored on the SFLASH, and then loaded by the device as part of the device start-up.

Creating a basic Rx filter requires a definition for three attributes: trigger, rule, and actions. This subsection describes these attributes and the additional parameters that are required by the creation function (sl_WlanRxFilterAdd). The creation function requires the following parameters:

- FilterType
- Flags
- pRule
- pTrigger
- pAction
- pFilterId, return value of the function

10.4.1 Filter Type

There are two kinds of filters: the basic filter (header), and the combination filter.

- For the basic filter, the field should be set to SL_WLAN_RX_FILTER_HEADER.
- The SL_WLAN_RX_FILTER_COMBINATION filter type creates a combined filter, which defines the compare function on one or two filters.

10.4.2 Filter Flags

The filter flag dictates the filter behavior, by a bit field, and the following flags are supported:

- SL_WLAN_RX_FILTER_BINARY – For creating a basic filter, usually the binary flag is set; a nonbinary

filter is supported for few filter types, and the nonbinary filter lets the user set the rule argument as a string instead of binary values. See Section 10.4.4 for the filters that support the nonbinary filter.

- SL_WLAN_RX_FILTER_ENABLE – A filter can be enabled or disabled. A disabled filter is skipped during the matcher process; for better performance for the filter creation, first create the set of the required filters with the disable flag, and then, when all the filters have been created, enable all of them with a single function call.
- SL_WLAN_RX_FILTER_PERSISTENT – A filter set with a persistent flag is saved to the SFLASH and loads on each device reset. The act of saving the persistent filters to the flash is executed by calling the sl_WlanSet function with the store command SL_WLAN_RX_FILTER_STORE.

10.4.3 Rule Structure for Header Filters

The rule structure describes the match criteria. The rule is a combination of:

- Field
- Argument
- Compare function

During the packet processing, the value of the frame field is compared with the value of the rule arguments. For example:

- Destination IP is equal to 123.44.55.66 means:
  - Field is destination IP.
  - Argument is 123.44.55.66.
  - Compare function is equal.
- Source MAC is different from 0x34567899 means:
  - Field is source MAC.
  - Argument is 0x34567899.
  - Compare function is not equal.
- Frame length is higher than 500 and lower than 900 means:
  - Field is frame length.
  - Arguments is 500, 900.
  - Compare function is in between.

The structure of the rule is a union which serves two type of rules: the header rule type and the combination rule type. For rule of type SL_WLAN_RX_FILTER_HEADER, the structure used is SlWlanRxFilterRuleHeader_t.

10.4.3.1 Field

The field value defines the field that is checked during the processing. The list of supported fields follows:

- SL_WLAN_RX_FILTER_HFIELD_FRAME_TYPE
- SL_WLAN_RX_FILTER_HFIELD_FRAME_SUBTYPE
- SL_WLAN_RX_FILTER_HFIELD_BSSID
- SL_WLAN_RX_FILTER_HFIELD_MAC_SRC_ ADDR
- SL_WLAN_RX_FILTER_HFIELD_MAC_DST_ADDR
- SL_WLAN_RX_FILTER_HFIELD_FRAME_LENGTH
- SL_WLAN_RX_FILTER_HFIELD_ETHER_TYPE
- SL_WLAN_RX_FILTER_HFIELD_IP_VERSION
- SL_WLAN_RX_FILTER_HFIELD_IP_PROTOCOL
- SL_WLAN_RX_FILTER_HFIELD_IPV4_SRC_ADDR
- SL_WLAN_RX_FILTER_HFIELD_IPV4_DST_ADDR
- SL_WLAN_RX_FILTER_HFIELD_IPV6_SRC_ADRR

- SL_WLAN_RX_FILTER_HFIELD_IPV6_DST_ADDR
- SL_WLAN_RX_FILTER_HFIELD_PORT_SRC
- SL_WLAN_RX_FILTER_HFIELD_PORT_DST
- SL_WLAN_RX_FILTER_HFIELD_L1_PAYLOAD_PATTERN
- SL_WLAN_RX_FILTER_HFIELD_L4_PAYLOAD_PATTERN

10.4.3.2 Compare Functions

A list of the supported compare functions follows:

- Equal – SL_WLAN_RX_FILTER_CMP_FUNC_EQUAL
- Not equal – SL_WLAN_RX_FILTER_CMP_FUNC_NOT_EQUAL_TO
- In between – SL_WLAN_RX_FILTER_CMP_FUNC_IN_BETWEEN; in this case, two arguments are required.
- Not in between – SL_WLAN_RX_FILTER_CMP_FUNC_NOT_IN_BETWEEN; in this case, two arguments are required.

Table 10-4 lists the possible compare function for per filter type.

Table 10-4. Possible Compare Functions

Header Rule	Supported functions			
	==	!=	<>	!<>
SL_WLAN_RX_FILTER_HFIELD_FRAME_TYPE	+	+	–	–
SL_WLAN_RX_FILTER_HFIELD_FRAME_SUBTYPE	+	+	+	+
SL_WLAN_RX_FILTER_HFIELD_BSSID	+	+	+	+
SL_WLAN_RX_FILTER_HFIELD_MAC_SRC_ ADDR	+	+	+	+
SL_WLAN_RX_FILTER_HFIELD_MAC_DST_ADDR	+	+	+	+
SL_WLAN_RX_FILTER_HFIELD_FRAME_LENGTH	+	+	+	+
SL_WLAN_RX_FILTER_HFIELD_ETHER_TYPE	+	+	+	+
SL_WLAN_RX_FILTER_HFIELD_IP_VERSION	+	+	+	+
SL_WLAN_RX_FILTER_HFIELD_IP_PROTOCOL	+	+	+	+
SL_WLAN_RX_FILTER_HFIELD_IPV4_SRC_ADDR	+	+	+	+
SL_WLAN_RX_FILTER_HFIELD_IPV4_DST_ADDR	+	+	+	+
SL_WLAN_RX_FILTER_HFIELD_IPV6_SRC_ADRR	+	+	+	+
SL_WLAN_RX_FILTER_HFIELD_IPV6_DST_ADDR	+	+	+	+
SL_WLAN_RX_FILTER_HFIELD_PORT_SRC	+	+	+	+
SL_WLAN_RX_FILTER_HFIELD_PORT_DST	+	+	+	+
SL_WLAN_RX_FILTER_HFIELD_L1_PAYLOAD_PATTERN	+	–	–	–
SL_WLAN_RX_FILTER_HFIELD_L4_PAYLOAD_PATTERN	+	–	–	–

10.4.3.3 Rule Fields

A list of the header rule fields follows. Each field is described with its possible values, and in which system state it is available (for system states, see Section 10.4.5).

- In Table 10-5, whenever ASCII parameters are used, the host code must set the filter flags as follows: FilterFlags |= ~SL_WLAN_RX_FILTER_BINARY
- In Table 10-5, whenever byte stream parameters are used, the host code must set the filter flags as follows: FilterFlags |= RX_FILTER_BINARY

Table 10-5 describes the rules types and their possible values.

Table 10-5. Rule Types

Field	Argument	Size	Values
SL_WLAN_RX_FILTER_HFIELD_FRAME_TYPE	Rule.Args.Value.Frametype	1	0 for mgmt 1 for ctrl 2 for data 3 for reserved
	Rule.Args.Value.FrametypeAscii	4	“MGMT” “CTRL” “DATA”
SL_WLAN_RX_FILTER_HFIELD_FRAME_SUBTYPE	Rule.Args.Value.FrameSubtype	1	0x00 ASSOCIATION REQ 0x10 ASSOCIATION RESPONSE 0x20 REASSOCIATION REQ 0x30 REASSOCIATION RESPONSE 0x40 PROBE REQ 0x50 PROBE RESPONSE 0x80 BEACON 0x90 ATIM 0xA0 DISASSOCIATION 0xB0 AUTHENTICATION 0xC0 DEAUTHENTICATION 0xD0 ACTION CTRL FRAMES 0x74 CONTROL WRAPPER 0x84 BLOCK ACK REQ 0x94 BLOCK ACK 0xA4 PS POLL 0xB4 RTS 0xC4 CTS 0xD4 ACK 0xE4 CF END 0xF4 CF END ACK DATA FRAMES 0x08 DATA 0x18 DATA CF ACK 0x28 DATA CF POLL 0x38 DATA CF ACK POLL 0x48 NO DATA FRAME 0x58 CF ACK 0x68 CF POLL 0x78 CF ACK POLL 0x88 QOS DATA 0x98 QOS DATA CF ACK 0xA8 QOS DATA CF POLL 0xB8 QOS DATA CF ACK POLL 0xC8 QOS NO DATA FRAME 0xD8 QOS CF ACK 0xE8 QOS CF POLL 0xF8 QOS CF ACK POLL
SL_WLAN_RX_FILTER_HFIELD_BSSID	Rule.Args.Value.Bssid	6	
SL_WLAN_RX_FILTER_HFIELD_MAC_SRC_ ADDR	Rule.Args.Value.Mac	6	
SL_WLAN_RX_FILTER_HFIELD_MAC_DST_ADDR	Rule.Args.Value.Mac	6	
SL_WLAN_RX_FILTER_HFIELD_FRAME_LENGTH	Rule.Args.Value.FrameLength	4	

Table 10-5. Rule Types (continued)

Field	Argument	Size	Values
SL_WLAN_RX_FILTER_HFIELD_ETHER_TYPE	Rule.Args.Value.EtherType	4	
SL_WLAN_RX_FILTER_HFIELD_IP_VERSION	Rule.Args.Value.IpVersion	1	
	Rule.Args.Value.IpVersionAscii	4	“IPV4” “IPV6”
SL_WLAN_RX_FILTER_HFIELD_IP_PROTOCOL	Rule.Args.Value.IpProtocol	1	1 – ICMP (IPV4 Only) 2 - IGMP (IPV4 only) 6 – TCP 17 – UDP 58 – ICMPV6
	Rule.Args.Value.IpProtocolAscii	5	“ICMP” “ICMP6” “IGMP” “TCP” “UDP
SL_WLAN_RX_FILTER_HFIELD_IPV4_SRC_ADDR	Rule.Args.Value.Ipv4	4	
SL_WLAN_RX_FILTER_HFIELD_IPV4_DST_ADDR	Rule.Args.Value.Ipv4	4	
SL_WLAN_RX_FILTER_HFIELD_IPV6_SRC_ADRR	Rule.Args.Value.Ipv6	16	
SL_WLAN_RX_FILTER_HFIELD_IPV6_DST_ADDR	Rule.Args.Value.Ipv6	16	
SL_WLAN_RX_FILTER_HFIELD_PORT_SRC	Rule.Args.Value.Port	4	1–65535
SL_WLAN_RX_FILTER_HFIELD_PORT_DST	Rule.Args.Value.Port	4	1–65535
SL_WLAN_RX_FILTER_HFIELD_L1_PAYLOAD_PATTERN	Rule.Args.Value.Pattern.Offset	2	
	Rule.Args.Value.Pattern.Length	1	
	Rule.Args.Value.Pattern.Value	16	
SL_WLAN_RX_FILTER_HFIELD_L4_PAYLOAD_PATTERN	Rule.Args.Value.Pattern.Offset	2	
	Rule.Args.Value.Pattern.Length	1	
	Rule.Args.Value.Pattern.Value	16	

10.4.3.4 Pattern-Matching Rule Fields

Pattern matching can be used to look for a specific payload on the frame. The SimpleLink Wi-Fi device currently supports two types of pattern matching:

- L1 payload matching (L1_PAYLOAD_EXACT_PATTERN_FIELD). The offset is counted from the beginning of the 802.11 MAC headers (that is, the frame control field). This is useful in transceiver mode, but can also be used while connected.
- L4 payload matching (L4_PAYLOAD_EXACT_PATTERN_FIELD). The offset is counted from the beginning of the TCP or UDP payload.

The inputs to this field header rule are as follows:

- Offset, or where to start checking for the requested pattern (offset can be set between 0x0 to 0x5ff)
- Length, or how many bytes: can be 1 to 16
- Pattern to compare with: can be up to 16 bytes
- Masking: bit masking on the pattern

Usage notes on pattern-matching filters:

- L4_PAYLOAD_EXACT_PATTERN_FIELD type filter only applies to STA or AP (not to transceiver mode)
- L1_PAYLOAD_EXACT_PATTERN_FIELD type filter can be in any mode (such as to a STA in transceiver mode or to a connected STA or to an AP)

Usage note for pattern matching while the device connected to TCP transmitter:

To ensure that an application frame arrives with high probability as sent by the transmitter host application, use a long interval (and short time-out) between the TCP sends, because TCP by nature is a streaming protocol. The TCP stack may aggregate or fragment frames into bytes, and send them in accordance with the current network or receiver congestion conditions.

Therefore, when a stream of bytes representing an application frame is sent over a TCP socket of the SimpleLink Wi-Fi device, there is no guarantee that this application frame will arrive in a single WLAN frame, as when it was sent by the transmitter host to the SimpleLink device; in these cases, the filter may not be relevant.

The following example demonstrates a definition of a rule that finds a frame from a specific MAC address. In this example, the rule searches for the MAC address: 0x08, 0x09, 0x76, 0x54, 0x32, 0x45:

```
_u8 MacMask[6]            = {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};
_u8 MacAddress[6]         = {0x08,0x09,0x76,0x54,0x32,0x45};

Rule.CompareFunc = SL_WLAN_RX_FILTER_CMP_FUNC_EQUAL_TO;
Rule.Field = SL_WLAN_RX_FILTER_HFIELD_MAC_SRC_ADDR;
memcpy( Rule.Args.Value.Mac[0], MacAddress, 6 );
memcpy( Rule.Args.Mask, MacMask , 6 );
```

The following example demonstrates a definition of a rule that finds a frame from a specific group of MAC addresses by address mask. In this example, the rule searches for MAC addresses that end with 0x45:

```
_u8 MacMask[6]            = {0x0 ,0x0 ,0x0 ,0x0 ,0x0 ,0xFF};
_u8 MacAddress[6]         = {0x08,0x09,0x76,0x54,0x32,0x45};

Rule.CompareFunc = SL_WLAN_RX_FILTER_CMP_FUNC_EQUAL_TO;
Rule.Field = SL_WLAN_RX_FILTER_HFIELD_MAC_SRC_ADDR;
memcpy(Rule.Args.Value.Mac[0], MacAddress, 6);
memcpy(Rule.Args.Mask, MacMask , 6);
```

10.4.4 Rule Structure for Combined Filters

The rule for combined filters is built from the following parameters:

- Compare function: not, and, or, defines the compare method of the parent filters.
- Parent filters, the filters which are compared when the compare function is not only filter supplied.

The following example demonstrates a combined filter (the parent filters are already created):

```
SlWlanRxFilterRule_u RuleCombination;
RuleType = SL_WLAN_RX_FILTER_COMBINATION;
RuleCombination.Combination.CombinationFilterId[0] = ParentFilter1;
RuleCombination.Combination.CombinationFilterId[1] = ParentFilter2;
RuleCombination.Combination.Operator = SL_WLAN_RX_FILTER_COMBINED_FUNC_OR;

RetVal = sl_WlanRxFilterAdd(     RuleType,
                  FilterFlags,
                  ( const SlWlanRxFilterRule_u* const )&RuleCombination,
                  ( const SlWlanRxFilterTrigger_t* const) &Trigger,
                  ( const SlWlanRxFilterAction_t* const )&Action,
                  &FilterId);
```

10.4.5 Filter Trigger

The trigger is the environment conditions verified before the matcher tests the rule. If the environment is not fulfilling, the rule is not tested and the matching result is FALSE.

A list of parameters that define a trigger follows:

- Parent filter ID
- Connection state
- Role

The advance features of defining counters is related to the following fields:

- Counter
- Counter val
- Compare function

The following subsection describes the parameters required to create the filter trigger.

10.4.5.1 Parent Filter ID

The filter ID contains the ID of the parent filter; filters can be organized in a tree hierarchy.

- Setting ParenFilterId to 0 creates a root filter. All filters can be assigned as root.
- The parent filter ID is the ID of the filter which is the parent of the current filter.
- The filter rule is tested only if its parent match result is TRUE.

The parent filter can be from the same layer or a lower layer than the child filter.

The following defines the filter layer of each rule type.

Filter layers: The header rules can be specified in a tree form, but the rules must also preserve a layered approach.

Therefore, a transport layer field (such as TCP or UDP source or destination ports) cannot be a parent of a MAC header field (such as frame type).

Table 10-6 presents which groups of header rule types can be parents of other header rule types. The general guideline is that the lower the communication layer to which the header rule filter applies, the more filters can depend on this filter.

NOTE: When a filter contains a drop action, it cannot be a parent of any other filter, because if a packet is dropped the tree traversal is stopped.

Table 10-6. Rule Types Layers

Group	Rule Types	Can be Parent of Rules from Group
A	FRAME_TYPE FRAME_SUBTYPE BSSID_ADDRESS MAC_SRC_ADDRESS MAC_DST_ADDRESS FRAME_LENGTH	A, B, C, D
B	ETHER_TYPE IP_VERSION Multicast destination IPs (V4 and V6) L1_PAYLOAD	B, C, D
C	Source IP (V4 and V6) Unicast destination IP (V4 and V6) IP PROTOCOL field (UDP, TCP, ICMP, IGMP, and so forth)	C, D
D	Source port (UDP/TCP) Destination port (UDP/TCP) L4_PAYLOAD	D

10.4.5.2 Connection State and Role

The filter can be set to be tested only in specific connection state; for example, only in STA mode or only in AP mode. The state in which the filter is considered is a combination of role and connection state.

Supported roles:

- SL_WLAN_RX_FILTER_ROLE_AP
- SL_WLAN_RX_FILTER_ROLE_STA
- SL_WLAN_RX_FILTER_ROLE_PROMISCUOUS (transceiver)
- SL_WLAN_RX_FILTER_ROLE_NULL

Supported connection states:

- SL_WLAN_RX_FILTER_STATE_STA_CONNECTED
- SL_WLAN_RX_FILTER_STATE_STA_NOT_CONNECTED
- SL_WLAN_RX_FILTER_STATE_STA_HAS_IP
- SL_WLAN_RX_FILTER_STATE_STA_HAS_NO_IP

Example of defining a filter which only works in transceiver mode:

```
/* Parent */
Trigger.ParentFilterID = parentId;

/* No counter is used, which is the common scenario */
Trigger.Counter = SL_WLAN_RX_FILTER_NO_TRIGGER_COUNTER;

/* Role is set to Transceiver mode */
Trigger.Role = SL_WLAN_RX_FILTER_ROLE_PROMISCUOUS;

/* The connection state is ignored  since the filter works in the Transceiver mode */
Trigger.ConnectionState = SL_WLAN_RX_FILTER_STATE_STA_CONNECTED;
```

Example of defining a filter which only works in STA mode after the IP is acquired:

```
/* Parent */
Trigger.ParentFilterID = parentId;

/* No counter is used, which is the common scenario */
Trigger.Counter = SL_WLAN_RX_FILTER_NO_TRIGGER_COUNTER;

/* Connection state and role, role is STA */
Trigger.Role = SL_WLAN_RX_FILTER_ROLE_STA;

/* Works only in case ip is acquired */
Trigger.ConnectionState = SL_WLAN_RX_FILTER_STATE_STA_HAS_IP;
```

Example of defining a filter which only works in STA mode:

```
/* Parent */
Trigger.ParentFilterID = parentId;

/* No counter is used, which is the common scenario */
Trigger.Counter = SL_WLAN_RX_FILTER_NO_TRIGGER_COUNTER;

/* Connection state and role, role is STA */
Trigger.Role = SL_WLAN_RX_FILTER_ROLE_STA;

/* Work on any connection state */
Trigger.ConnectionState =
SL_WLAN_RX_FILTER_STATE_STA_CONNECTED |
SL_WLAN_RX_FILTER_STATE_STA_NOT_ CONNECTED |
SL_WLAN_RX_FILTER_STATE_STA_HAS_IP |
SL_WLAN_RX_FILTER_STATE_STA_HAS_NO_IP;
```

10.4.5.3 Filter During Transceiver Mode

In case of transceiver mode filtering, the transceiver socket receive function must be invoked for receiving frames; the function triggers the device to start the receiving of Rx frames. Once the device receives frames, the frames are processed by the Rx filter marcher. The socket receive opens a datagram in raw socket. In transceiver mode, TCP and UDP frames are carried over fragmented IPv4 or IPv6 datagrams, and therefore are not filtered on L4 ports or on payload.

10.4.6 Rx Filter Action

The actions execute if the filter trigger and the filter rule are matched. Each filter can be defined with several actions.

The following actions are supported:

- SL_WLAN_RX_FILTER_ACTION_NULL, no action
- SL_WLAN_RX_FILTER_ACTION_DROP, drop the packet
- SL_WLAN_RX_FILTER_ACTION_EVENT_TO_HOST, send event

The following actions are relevant to the counters feature:

- SL_WLAN_RX_FILTER_ACTION_ON_REG_INCREASE
- SL_WLAN_RX_FILTER_ACTION_ON_REG_DECREASE
- SL_WLAN_RX_FILTER_ACTION_ON_REG_RESET

The following subsection describes the event actions.

10.4.6.1 Send Events Action

A typical usage for the send event capability is to perform wake on WLAN (that is, to wake the host on a specific packet matching a filter).

Events can be sent from the SimpleLink Wi-Fi device to the host as a result of a matched Rx filter.

The event action arguments (Action.UserId) define the bit number set in the triggered event.

The supported ID range is from 0 to SL_RX_FILTER_MAX_USER_EVENT_ID (=63).

A single host event aggregates all the events actions which have been triggered for a single frame. The aggregation is based on the filter groups, as described in rule types. Examples follow.

10.4.6.2 Multiple Bits Set on the Same Event

Consider the following case:

- Source IP has event action with argument ID X.
- Destination IP has event action with argument ID Y.
- Both header rule fields are from the same group C.
- A received frame passes both filters.

This results in a single host event with bit X and bit Y set.

10.4.6.3 Multiple Events From the Same Rx Frame

Consider the following case:

- Src MAC (group A) has event action with arg ID X.
- Dest IP Y (group C) has event action with arg ID.
- A received frame passes both filters.

This results in two host events: event with bit X set and event with bit Y set.

10.4.6.4 Code Example

In the following code example, the event ID is set in byte 3 of the action arguments and may be set to a value between 0 and SL_RX_FILTER_MAX_USER_EVENT_ID (=63).

The code example for adding a filter with an event (the event input is highlighted in yellow, and the event output parsing is highlighted in gray):

```
void AddSomeFilters()
{
    // declerations
    SlWlanRxFilterID_t                  FilterId;
    SlWlanRxFilterRuleType_t            RuleType;
    SlWlanRxFilterFlags_u               FilterFlags;
    SlWlanRxFilterRuleHeader_t          Rule;
    SlWlanRxFilterTrigger_t             Trigger;
    SlWlanRxFilterAction_t              Action;
    ...
    // here comes the code for adding a header rule filter
    ...

       // now for events args setting.
       // request an event as one of the actions to perform
       Action.Type = SL_WLAN_RX_FILTER_ACTION_EVENT_TO_HOST;
       // The output of the event is to set bit number (in this case it is bit 2).
       Action.UserId = 2;

  ...
  // finally the code to add the event.
       RetVal = sl_WlanRxFilterAdd(
                    RuleType,
                    FilterFlags,
                    ( const SlWlanRxFilterRule_u* const )&Rule,
                    ( const SlWlanRxFilterTrigger_t* const)&Trigger,
                    ( const SlWlanRxFilterAction_t* const )&ction,
                    &FilterId);

// rx filters events handling is a specific case in the WLAN events handling
void SLWlanEventHandler(SlWlanEvent_t *pWlanEventHandler)
{
    int i = 0;

    switch(pWlanEventHandler->Id)
    {
        case SL_WLAN_EVENT_CONNECT:
            break;
        case SL_WLAN_EVENT_STA_ADDED:
            break;
        case SL_WLAN_EVENT_DISCONNECT:
            break;
        case SL_WLAN_EVENT_RXFILTER:
            {
                SlWlanEventRxFilterInfo_t *pEventData =
                            (SlWlanEventRxFilterInfo_t *)&pWlanEventHandler->Data;
                /*
                           printf("\n\nRx filter event %d, event type = %d
                                  \n",g_RxFilterEventsCounter,pEventData->Type);
                for(i = 0;i < 64;i++)
                {
                    if(SL_WLAN_ISBITSET8(pEventData->UserActionIdBitmap,i))
                    {
                        printf("User action %d filter event
                                                arrived\n",i);
                    }
                }
                          */
            }
```

```
                break;
        }
    }
```

10.4.6.5 Counter Action

Two sets of filter counters can be used. Each counter is associated with a filter type groups (see rule types layers).

The counter ID that can be used for each rule layer follows:

- RX_FILTER_COUNTER1-4: can be used with filters from groups C-D
- RX_FILTER_COUNTER5-8: can be used with filters from groups A-B

10.5 Managing Filters

Manage the filters performed by calling the regular sl_WlanSet and sl_WlanGet APIs. The available operations contain:

- Enable and disable
- Remove
- Save
- Update

To manage several filters simultaneously, the SimpleLink Wi-Fi device receives a bit field of the filters that the operation should take on. This bit field contains up to 128 bits. The following macro can be used to define a correct bit for a filter ID:

```
SL_WLAN_SETBIT8 (BitField.FilterIdMask, FilterId);
```

For example, to set the operation on filter 1 and filter 35, the macro should be called twice:

```
SL_WLAN_SETBIT8 (FilterIdMask, 1);
SL_WLAN_SETBIT8 (FilterIdMask, 35);
```

If the filter is not defined or created, the SimpleLink Wi-Fi device ignores its bit on the bit field mask. Therefore, operations can be performed with a bit field of all 1s.

NOTE: TI highly recommends updating the Rx filters while the sockets are closed.

10.5.1 Enable and Disable Filters

TI recommends creating a filter as a disable, and then enable all the relevant filters at once. A filter with its corresponding bit is set to 1 is enabled, and a filter in which its corresponding bit is set to 0 is disabled; filters which are not defined are ignored.

To enable or disable filters, call the sl_WlanSet API with the following arguments:

- ConfigId: SL_WLAN_RX_FILTERS_ID
- ConfigOpt: SL_WLAN_RX_FILTER_STATE

Example:

```
_u16    Size = sizeof(SlWlanRxFilterRetrieveStateBuff_t);
_u16 opt = SL_WLAN_RX_FILTER_STATE;
RetVal = sl_WlanGet( SL_WLAN_RX_FILTERS_ID, &opt , &Size ,
                     (unsigned char *)&RxFilterIdBitField);
SL_WLAN_CLEARBIT8(OutputBuff.FilterIdMask,selectedfilter);
RetVal = sl_WlanSet( SL_WLAN_RX_FILTERS_ID, SL_WLAN_RX_FILTER_STATE,
                     sizeof(SlWlanRxFilterOperationCommandBuff_t),
                     (unsigned char*)&RxFilterIdBitField);
```

10.5.2 Get Filter Status

To get the enable status of filters, call the sl_WlanGet API with the following arguments:

- ConfigId: SL_WLAN_RX_FILTERS_ID
- ConfigOpt: SL_WLAN_RX_FILTER_STATE

Example:

```
_u16 Size  = sizeof(SlWlanRxFilterRetrieveStateBuff_t);
_u16 opt = SL_WLAN_RX_FILTER_STATE;

RetVal = sl_WlanGet( SL_WLAN_RX_FILTERS_ID, &opt ,(_u16*)&Size , (_u8*)&RxFilterIdBitField );
```

10.5.3 Removing a Filter

Removing a filter is started by removing the filters from the active filters list. If the filter is persistent, removing it alone is not enough, and the STORE operation must also be called.

In this command, filters with bits set to 1 are removed, and filters with bits set to 0 or filters which are not defined are ignored.

To remove filters, call the sl_WlanSet API with the following arguments:

- ConfigId: SL_WLAN_RX_FILTERS_ID
- ConfigOpt: SL_WLAN_RX_FILTER_REMOVE

10.5.4 Storing Filters into the SFLASH

The filters are not stored on the storage automatically. This operation must be initiated by the host. In this command, filters where a persistent bit is set are stored. The stored filters are loaded each time the device is started.

To store the filters, call the sl_WlanSet command with the following arguments:

- ConfigId: SL_WLAN_RX_FILTERS_ID
- ConfigOpt: SL_WLAN_RX_FILTER_STORE

Example:

```
retVal = sl_WlanSet(SL_WLAN_RX_FILTERS_ID, SL_WLAN_RX_FILTER_STORE, 0, NULL);
```

10.5.5 Update Filter Arguments

To update the rule attributes of an existing filter, call the sl_WlanSet command with the following arguments:

- ConfigId: SL_WLAN_RX_FILTERS_ID
- ConfigOpt: SL_WLAN_RX_FILTER_UPDATE_ARGS

Example:

```
memcpy(updateFilterBuff.Args.Value.Bssid[0], filterData, 6);
memcpy(updateFilterBuff.Args.Mask, MacMAsk, 6);

updateFilterBuff.FilterId = FilterId;
updateFilterBuff.BinaryOrAscii = 1;
retVal = sl_WlanSet(SL_WLAN_RX_FILTERS_ID, SL_WLAN_RX_FILTER_UPDATE_ARGS,
                    sizeof(SlWlanRxFilterUpdateArgsCommandBuff_t),
                    (unsigned char *) &updateFilterBuff);
```

Ping

Topic Page

11.1 General Description

Ping is a network utility, part of the device internal network utilities, which verifies if a particular IP address exists. This utility is based on the ICMP (control message protocol), and sends an echo request to a specified entity in the network and waits for a reply. Ping supports IPv4 and IPv6 standards. This utility can be used to test connectivity and determine the round trip time.

11.2 Start and Stop Ping

The same API starts and stops the ping process. To stop the ping process, apply value 0 in the IP field. The following parameters can be configured in the ping start command:

- Ping parameters – holds configuration regarding the ping command:
  - Ping interval time: interval between ping packets, in ms
  - Ping size: ping packet size
  - Ping request time-out: time-out time for every ping, in ms
  - Total number of attempts: number of ping requests. 0 indicates infinite.
  - Flags: flag options are as follows:
    - 0 – send ping report only when finishing transmitting all the requests
    - 1 – send ping report for every ping request
    - 2 – stop ping after one successful ping request (received reply)
    - 4 – Do not fragment the ping packet. This flag can be set with other flags.
  - IP: destination IPv4\IPv6 address. In case of IPv4, use this field only.
  - Ip1OrPadding: destination IPv6 address
  - Ip2OrPadding: destination IPv6 address
  - Ip3OrPadding: destination IPv6 address
- Family – specifies the protocol family IPv4 or IPv6
- Report – Return value. If callback is not set, the API is blocked until the ping report is received. Hold information regarding the results of the ping request and include the following parameters:
  - Packets sent – number of sent ping requests
  - Packets received – number of received ping replied
  - Min round time – shortest round time, in ms
  - Max round time – longest round time, in ms
  - Average round time – average round time, in ms
  - Test time – total time the test took, in ms
- Ping callback – optional parameter. If the callback is provided, the API does not block, and immediately returns. When results are available, the callback is called. If it is not implemented, NULL should be placed and API blocks, until the results are ready.

Example of sending an IPV4 ping request with a report for every successful ping:

```
_i16 Status;
SlNetAppPingReport_t report;
SlNetAppPingCommand_t pingCommand;

pingCommand.Ip = SL_IPV4_VAL(10,1,1,200);     /* destination IP address is 10.1.1.200 */
pingCommand.PingSize = 150;                   /* size of ping, in bytes */
pingCommand.PingIntervalTime = 100;           /* delay between pings, in milliseconds */
pingCommand.PingRequestTimeout = 1000;        /* timeout for every ping in milliseconds */
pingCommand.TotalNumberOfAttempts = 20;       /* number of ping requests */
pingCommand.Flags = 0;                        /* report only when finished */

Status = sl_NetAppPing( &pingCommand, SL_AF_INET, &report, NULL );
if (Status)
{
    /* error */
}
```

Example of stopping the ping request:

```
_i16 Status;
SlNetAppPingCommand_t pingCommand;

pingCommand.Ip = 0;
Status = sl_NetAppPing( &pingCommand, SL_AF_INET, &report, NULL ) ;
if (Status)
{
    /* error */
}
```

Example of sending an IPV6 infinite ping request:

```
_i16 Status;
SlNetAppPingReport_t report;
SlNetAppPingCommand_t pingCommand;

pingCommand.Ip = 0xFF020000;                /* IPV6 Address */
pingCommand.Ip1OrPadding = 0;               /* IPV6 Address */
pingCommand.Ip2OrPadding = 0;                /* IPV6 Address */
pingCommand.Ip3OrPadding = 0xFB;            /* IPV6 Address */
pingCommand.PingSize = 150;                 /* size of ping, in bytes */
pingCommand.PingIntervalTime = 100;         /* delay between pings, in milliseconds */
pingCommand.PingRequestTimeout = 1000;      /* timeout for every ping in milliseconds */
pingCommand.TotalNumberOfAttempts = 0;      /* max number of ping requests. 0 - forever */
pingCommand.Flags = 0;                      /* report only when finished */


Status = sl_NetAppPing( &pingCommand, SL_AF_INET6, &report, NULL ) ;
if (Status)
{
    /* error */
}
```

11.3 Limitations

An infinite number of ping requests can be implemented only with a callback.

Transceiver

Topic Page

12.1 Introduction

The transceiver mode is a powerful tool that gives the ability to send and receive any raw data in Layer 2. The user can use the entire frame, including the 802.11 header (excluding duration field), to receive and transmit its own data. Transceiver mode is only enabled when the SimpleLink Wi-Fi device is not connected to an AP. Receiving packets in transceiver mode is enabled only after the first call to the sl_Recv API. Before this call, no packets can be received. By default, there is no frame acknowledgements or retries; therefore, there are no promises that the frames reach their destination (when working in L1 mode, it is also not ensured that there will be no collision with other frames or with other interference).

One common use case for transceiver mode applications is for transmitting the same packet in continues. This is used mostly for tagging and for measuring loss, using the RX statistics feature. Another use case can be promiscuous mode, such as with as a sniffer.

12.2 Key Features

Table 12-1 lists the key features of the transceiver.

Table 12-1. Key Features

Key Features	Description
TX\RX Layer 1 raw data	Send and receive any L1 raw data
TX\RX Layer 2 raw data	Send and receive any L2 raw data
Internal Packer Generator	The device can auto-generate packet internally with infinite transmission.
CW	Carrier-wave signal transmission

12.3 Configurations and Setting

Host driver commands are used to start and operate the transceiver mode.

NOTE: To use transceiver mode, the device must be set in STA role, be disconnected, and have disabled previous connection policies that might try to automatically connect to an AP.

Example:

```
_i16 Status;
Status = sl_WlanPolicySet(SL_POLICY_CONNECTION,SL_CONNECTION_POLICY(0,0,0,0,0),NULL,0);
if( Status )
{
    /* error */
}
Status = sl_WlanDisconnect();
if( Status )
{
    /* error */
}
```

12.3.1 Open Transceiver Socket

Only a single transceiver socket is supported. To start the transceiver mode, use the sl_Socket API with the following arguments:

- Domain – Set to SL_AF_RF; indicates transceiver mode socket. Configure this value as the family parameter.
- Type – Set to one of the following options:
  - SL_SOCK_RAW – Indicates an L1 mode raw socket (no respect for 802.11 medium access policy - CCA)
  - SL_SOCK_DGRAM – Indicates an L2 mode raw socket (respecting 802.11 medium access policies)

- Channel – Used for configuring the operational channel from which the device should start receiving or transmitting traffic. If the channel is set to 0, the channel is set as the last transceiver channel used. If this is the first time the transceiver socket is open, a channel should be applied by the sl_SetSockOpt operation, or by the flags parameter in the sl_Send operation.

This command must be called only when the device is in STA role and disconnected. The command returns the socket ID, which is used from now on to reference the socket. If there is a problem with the socket, the command returns a negative error code.

Example:

```
_i16 sd;
_i16 channel = 6;

sd = sl_Socket(SL_AF_RF ,SL_SOCK_RAW/SL_SOCK_DGRAM, channel);
```

12.3.2 Close Transceiver Socket

The sl_Close API is used to close the transceiver mode.

Example:

```
_i16 Status, sd;

Status =  sl_Close(sd);
if( Status )
{
    /* error */
}
```

12.3.3 Send Data

Transmitting raw socket data is done by calling sl_Send after successfully opening the transceiver socket. The API return value is the number of bytes sent, or negative value in case of an error.

The SimpleLink Wi-Fi device allows the option to set the following parameters as part of the send operation as part of the flags parameter:

- Channel
- Rate
- Tx Power
- 802.11b preamble

The flags parameters given as part of the sl_Send API are valid only for this specific send operation, and are not kept for any further operation. If the flag parameter is set to 0, the default values remain. These parameters can also be set through the sl_SetSockOpt API, as specified in the example that follows.

NOTE: These parameters have no default values, and therefore must be set through the sl_Send API or sl_SetSockOpt, as specified below.

Example: transmit a frame on channel 1, with 1-MBps data rate, maximum TX power and long preamble:

```
void sendPacket(char * data)
{
  /* Base frame: */
  #define FRAME_TYPE 0x88
  #define FRAME_CONTROL 0x00
  #define DURATION 0xc0,0x00
  #define RECEIVE_ADDR 0x08, 0x00, 0x28, 0x5A, 0x72, 0x3C
  #define TRANSMITTER_ADDR 0x08, 0x00, 0x28, 0x5a, 0x78, 0x1e
  #define BSSID_ADDR 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
  #define FRAME_NUMBER   0x00, 0x00
  #define QOS_CTRL 0x00, 0x00

  _i32 NumOfBytes =0;
  _i32 sock=0;
```

```
  /* MAC header */
  char buff[1536];
  char FrameBaseData[] = {
  FRAME_TYPE,                          /* version, type sub type */
  FRAME_CONTROL,                      /* Frame control flag */
  DURATION,                           /* duration */
  RECEIVE_ADDR,                      /* Receiver ADDr */
  TRANSMITTER_ADDR,                 /* Transmitter Address */
  BSSID_ADDR,                       /* destination */
  FRAME_NUMBER,                    /* Frame number */
  QOS_CTRL};                      /* Transmitter */

  memcpy(buff,FrameBaseData,sizeof(FrameBaseData));
  memcpy (buff + sizeof(FrameBaseData), data, sizeof(buff ) -
 sizeof(FrameBaseData));/* Example data */
  sock = sl_Socket(SL_AF_RF, SL_SOCK_RAW, 1);
  NumOfBytes = sl_Send(sock,buff,sizeof(buff),SL_WLAN_RAW_RF_TX_PARAMS(CHANNEL_1,
SL_WLAN_RATE_1M,0,SL_WLAN_LONG_PREAMBLE));
}
```

12.3.4 Receive Data

Receiving raw socket data is done by calling sl_Recv after successfully opening the transceiver socket. The API return value is the number of bytes received, or a negative value in case of an error. Each receive packet has an 8-byte proprietary header which includes the following parameters:

- Rate – packet received rate
- Channel – packet received channel
- RSSI – computed RSSI value in dBm of current frame
- Time Stamp – frame timestamp in µs

If the packet is longer than the receive buffer, the remainder of the packet is discarded. The maximum packet size which can be received is 1544 (1536 bytes of data and 8 bytes of proprietary header).

Example:

```
_i16 NumOfByets;
signed char buf [1000];
_i16 Soc;
_i16 channel = 6;
_i16 len = 1000;

Soc = sl_Socket(SL_AF_RF ,SL_SOCK_RAW, channel);
NumOfBytes = sl_Recv(Soc, buf,500,0);
```

12.4 Internal Packet Generator

The SimpleLink Wi-Fi device can internally generate packets in transceiver mode. The device is capable of repeating a user-predefined pattern of data.

Before calling sl_Send, you must set the number of frames using the sl_SetSockOpt API to the number of frames desired to be transmitted (0 means infinite number of frames).

A single call to the sl_Send API triggers the frames transmission. The SimpleLink Wi-Fi device keeps transmitting until it has sent all the requested frames, or until the socket is closed or another socket property changes (through sl_SetSockOpt). Receiving packets operation is available during the send operation. Setting the number of frames to transmit to 1 returns the socket to the regular transceiver socket state.

Example of transmitting multiple data packets:

```
void sendPacket(char * data)
{
  /* Base frame: */
  #define FRAME_TYPE 0x88
  #define FRAME_CONTROL 0x00
  #define DURATION 0xc0,0x00
```

```
  #define RECEIVE_ADDR 0x08, 0x00, 0x28, 0x5A, 0x72, 0x3C
  #define TRANSMITTER_ADDR 0x08, 0x00, 0x28, 0x5a, 0x78, 0x1e
  #define BSSID_ADDR 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
  #define FRAME_NUMBER   0x00, 0x00
  #define QOS_CTRL 0x00, 0x00

  _i32 NumOfBytes =0;
  _i32 Soc=0;
  _i16 Status =0;
  _u32 numFrames=20;


  /* Mac header */
  char buff[1536];
  char FrameBaseData[] = {
  FRAME_TYPE,                          /* version, type sub type */
  FRAME_CONTROL,                      /* Frame control flag */
  DURATION,                           /* duration */
  RECEIVE_ADDR,                      /* Receiver ADDr */
  TRANSMITTER_ADDR,                 /* Transmitter Address */
  BSSID_ADDR,                       /* destination */
  FRAME_NUMBER,                    /* Frame number */
  QOS_CTRL};                      /* Transmitter */

  memcpy(buff,FrameBaseData,sizeof(FrameBaseData));
 /* Example data */
  memcpy (buff + sizeof(FrameBaseData), data, sizeof(buff ) - sizeof(FrameBaseData));
  Soc = sl_Socket(SL_AF_RF, SL_SOCK_RAW, 1);
  /* Set 20 frames to transmit */
  Status = sl_SetSockOpt(Soc, SL_SOL_PHY_OPT,SL_SO_PHY_NUM_FRAMES_TO_TX,
&numFrames,sizeof(numFrames));
  if (Status)
  {
      /* Error */
  }
  /* Send 20 packet with the same buffer */
  NumOfBytes = sl_Send(Soc,buff,sizeof(buff),SL_WLAN_RAW_RF_TX_PARAMS(CHANNEL_1,
SL_WLAN_RATE_1M,1,   SL_WLAN_LONG_PREAMBLE));
}
```

12.5 CW

The SimpleLink Wi-Fi device can transmit infinite carrier-wave signals using the sl_Send API, with NULL buffer and 0 (zero) length.

The flags parameter in the sl_Send API is used to signal the tone offset (–25 to 25).

The CW is continuously transmitted until stopped. Stopping CW transmission is done by triggering another sl_Send API with flags= –128 (decimal).

12.6 Changing Socket Properties

The SimpleLink Wi-Fi device offers multiple transceiver socket configurations by using the sl_SetSockOpt API. All configurations must be set after successfully opening the socket. The configurations are not persistent, and are deleted after the socket is closed.

12.6.1 Change Operating Channel

Change the transceiver operational channel if changing the channel during packet transmission results in changing the channel, only after all packet transmission completes.

Example:

```
_i16 Status;
_i16 channel = 9;
```

```
Status = sl_SetSockOpt(soc, SL_SOL_SOCKET,  SL_SO_CHANGE_CHANNEL, &channel,sizeof(channel));
if (Status)
{
    /* Error */
}
```

NOTE: The channel parameter has no default value, and therefore must be set when opening the socket, through the sl_Send operation, or through sl_SetSockOpt, as specified in this section.

12.6.2 Change Default PHY Data Rate

Change the WLAN transmit rate. The values can be one of the following:

- SL_WLAN_RATE_1M = 1
- SL_WLAN_RATE_2M = 2
- SL_WLAN_RATE_5_5M = 3
- SL_WLAN_RATE_11M = 4
- SL_WLAN_RATE_6M = 6
- SL_WLAN_RATE_9M = 7
- SL_WLAN_RATE_12M = 8
- SL_WLAN_RATE_18M = 9
- SL_WLAN_RATE_24M = 10
- SL_WLAN_RATE_36M = 11
- SL_WLAN_RATE_48M = 12
- SL_WLAN_RATE_54M = 13
- SL_WLAN_RATE_MCS_0 = 14
- SL_WLAN_RATE_MCS_1 = 15
- SL_WLAN_RATE_MCS_2 = 16
- SL_WLAN_RATE_MCS_3 = 17
- SL_WLAN_RATE_MCS_4 = 18
- SL_WLAN_RATE_MCS_5 = 19
- SL_WLAN_RATE_MCS_6 = 20
- SL_WLAN_RATE_MCS_7 = 21

Example:

```
_i16 Status;
_i16 rate = SL_WLAN_RATE_1M;

Status = sl_SetSockOpt(soc, SL_SOL_PHY_OPT,  SL_SO_PHY_RATE, &rate, sizeof(rate));
if (Status)
{
    /* Error */
}
```

NOTE: The PHY data rate parameter has no default value, and therefore must be through the sl_Send operation or through sl_SetSockOpt, as specified in this section.

12.6.3 Change Tx Power

Setting the Tx power lets the user change the transmission power relative to the maximum Tx power. The values represent steps 0 to 15, which reflect as dBm offset from maximum power (0 means MAX power). For more information, see Chapter 3.

Example:

```
_i16 Status;
_u32 TxPower = 1; /* valid range is 1-15 */

Status = sl_SetSockOpt(soc, SL_SOL_PHY_OPT,  SL_SO_PHY_TX_POWER, &TxPower, sizeof(TxPower));
if (Status)
{
    /* Error */
}
```

NOTE: The Tx power parameter has no default value, and therefore must be through the sl_Send operation or sl_SetSockOpt, as specified in this section.

12.6.4 Change Number of Frames to Transmit (Internal Packet Generator)

The RAW socket packet generator sets the number of frames to transmit in the internal packet generator.

Example:

```
_i16 Status;
_u32 NumFrames = 10;

Status = sl_SetSockOpt(soc, SL_SOL_PHY_OPT, SL_SO_PHY_NUM_FRAMES_TO_TX, &NumFrames,
sizeof(NumFrames));
if (Status)
{
    /* Error */
}
```

12.6.5 Change 802.11b Preamble

Set Long or Short WLAN PHY preamble for 802.11b rates only. Set 1 for short preamble or 0 for long.

Example:

```
_u32 preamble = 1; /* set short preamble */
_i16 Status;

Status = sl_SetSockOpt(soc, SL_SOL_PHY_OPT, SL_SO_PHY_PREAMBLE, &preamble, sizeof(preamble));
if (Status)
{
    /* Error */
}
```

NOTE: The 802.11b preamble parameter has no default value, and therefore must be through the sl_Send operation or sl_SetSockOpt, as specified in this section.

12.6.6 Set CCA Threshold

The CCA threshold can be configured to set the specific threshold when the channel is considered as occupied. The following values can be set:

- SL_TX_INHIBIT_THRESHOLD_MIN (–88 dBm)
- SL_TX_INHIBIT_THRESHOLD_LOW (–78 dBm)
- SL_TX_INHIBIT_THRESHOLD_DEFAULT (–68 dBm)
- SL_TX_INHIBIT_THRESHOLD_MED (–58 dBm)

- SL_TX_INHIBIT_THRESHOLD_HIGH (–48 dBm)
- SL_TX_INHIBIT_THRESHOLD_MAX (–38 dBm)

Example:

```
_i16 Status;
_u32 thrshld = SL_TX_INHIBIT_THRESHOLD_MED;

Status = sl_SetSockOpt(soc, SL_SOL_PHY_OPT, SL_SO_PHY_TX_INHIBIT_THRESHOLD,
&thrshld,sizeof(thrshld));
if (Status)
{
    /* Error */
}
```

12.6.7 Set Tx Frames Time-out

Tx time-out for transceiver frames (lifetime) can be set. The value is given in ms (maximum value is 100 ms).

Example:

```
_i16 Status;
 _u32 TimeOut = 50;

Status = sl_SetSockOpt(soc, SL_SOL_PHY_OPT,  SL_SO_PHY_TX_TIMEOUT,&TimeOut,sizeof(TimeOut));
if (Status)
{
    /* Error */
}
```

12.6.8 Enable or Disable Sending ACKs

Enable or disable sending ACKs in transceiver mode (enable = 1, disable = 0). This option is disabled by default.

Example:

```
_i16 Status;
_u32 Acks = 1; /* 0 = disabled / 1 = enabled */

Status = sl_SetSockOpt(soc, SL_SOL_PHY_OPT,  SL_SO_PHY_ALLOW_ACKS ,&Acks, sizeof(Acks));
if (Status)
{
    /* Error */
}
```

12.7 Limitations

- Only one transceiver socket is supported in the system.
- Transceiver mode is available in STA mode only.
- Length of a received packet is trimmed if it exceeds 1536 bytes of data. Each packet includes the 8 bytes of proprietary header. Therefore, the receive buffer should be set to a maximum of 1544 bytes.
- Cannot transmit a frame over 1536 total bytes (including any header) and below 14 bytes (shortest MAC header).
- Transceiver mode is not available in connected mode. Auto-connection mode is also considered as connected mode, even if not connected.
- sl_SendTo and sl_RecvFrom are not available in transceiver mode.

Power Managment

Topic **Page**

13.1 Introduction

13.1.1 LPDS

Whenever possible, the SimpleLink device strives to enter and remain in its low-power state (LPDS). In this state, most of its clocks and logic are powered down, and only the memory and basic supervision circuitry are enabled, which leads to low power consumption. The host interface is designed such that entering and exiting LPDS is completely transparent to the host. The host may initiate a new command at any time, regardless of the power state of the device.

13.1.2 802.11 Power Save

When the device is in station mode and connected to an access point, it automatically tries to use the power-save mechanism defined in the 802.11 standard. This mechanism allows entering into low-power mode while maintaining a connection to the access point.

13.1.2.1 LSI (Long Sleep Interval)

The 802.11 power-save feature lets the device remain in low-power mode without the risk of missing data destined to it (including network broadcast data). It is achieved by sending all broadcast data after a DTIM beacon. The transmission time of this beacon is known in advance to the SimpleLink device, which wakes it up in time for the traffic. When the device is in LSI mode, the device only wakes up for a single broadcast period within the time interval specified by the user. This process allows further power reduction, but may cause the device to miss broadcast data on the network.

13.1.3 Low Power versus Latency

Both the LPDS and 802.11 power-save features have an overhead, which causes an increased latency in data transmission and reception. The user may optimize the device for low latency instead of low power by changing the power policy of the system, as described in Section 13.3.1. Table 13-1 summarizes the available policies and their effect on power and latency.

Table 13-1. Power and Latency

Policy	LPDS	802.11 Power Save	Device Power Saving	Device Latency
Always On	Disabled	Disabled	None	Minimal
Low Latency	Enabled	High entry threshold	Low	Low
Normal	Enabled	Normal entry threshold	Medium	Medium
Low Power	Enabled	Low entry threshold	High	High
LSI	Enabled	Entry threshold set by user	Highest	Highest

13.1.4 Power Modes versus Device Modes

The low-power policies of the SimpleLink device are only available when it is in STA and P2P client mode. Once the device is put into AP or P2P group owner mode, the power-management profile is forced to always on.

13.2 Key Features

Table 13-2 lists the key power management features.

Table 13-2. Key Features

Key Features	Description
Auto-power management	The SimpleLink device has advanced internal power-management logic that puts it in LPDS (low-power deep sleep) in a manner transparent to the host.
802.11 Power save	Use of the power-save feature of the 802.11 standard allows the device to consume very little power while maintaining a connection to an access point.

Table 13-2. Key Features (continued)

Key Features	Description
Power-optimized out of the box	Device is power-optimized by default. No configuration by the host is necessary to activate these features.
Fast connect	Can connect to the last known network without performing a WLAN scan, which dramatically decreases connection time and saves power.

13.3 Configurations and Settings

13.3.1 Changing Power Policy

Power policy is changed through the generic sl_WlanPolicySet API, as shown in Table 13-3.

Table 13-3. Power Policy

Desired Policy	API Parameters			
	Type	Policy	pVal	ValLen
Normal (default)	SL_POLICY_PM	SL_NORMAL_POLICY	NULL	0
Low Latency	SL_POLICY_PM	SL_LOW_LATENCY_POLICY	NULL	0
Low Power	SL_POLICY_PM	SL_LOW_POWER_POLICY	NULL	0
Always On	SL_POLICY_PM	SL_ALWAYS_ON_POLICY	NULL	0
Long Sleep	SL_POLICY_PM	SL_LONG_SLEEP_INTERVAL_POLICY	SlWlanPmPolicyParams_t*	sizeof(SlWlanPmPolicyParams_t)

All settings of the sl_WlanPolicySet API are effective immediately after the call, and persistent between device resets.

13.3.2 Enabling Fast Connect

Fast connect is controlled by the WLAN API (for details see Chapter 3). When enabled, the scan process is skipped if the connection attempt is to the last connected network. Skipping the scan can save hundreds of milliseconds in the connection time, thereby reducing the power consumption.

13.4 Network Applications and Power Consumption

13.4.1 mDNS

The device mDNS service (enabled by default) is based on sending and receiving broadcast and multicast data frames on the connected network, without any use interaction. Because the effects of this behavior on power consumption cannot be determined in advance, TI recommends turning this service off in power-constrained systems (assuming it is not necessary for the application). This service will be turned off automatically if LSI mode with a sleep time greater than 2000 ms is specified.

13.4.2 HTTP Server

The device HTTP server is automatically turned off if LSI is set to 2000 ms. This occurs because the chances of the server to successfully accept a client connection in these conditions are extremely low.

13.5 Design Guidelines

13.5.1 LSI and Packet Loss

When setting the LSI sleep time as greater than the DTIM period of the network (the period of the beacon after which all broadcast messages are sent), the device will most likely miss some of the network broadcast traffic. The effect of this is application-specific: if the application always initiates traffic and relies on unicast rather than broadcast response, no behavioral impact is expected other than higher latency. If the application is expected to respond to unsolicited traffic (run a UDP/TCP server, respond to pings or mDNS) the effect might be more significant, and may result in clients failing to connect to the device or sense its presence on the network.

13.5.2 PHY Calibration Mode

The PHY calibration mode directly affects system power consumption, because it prolongs the initialization phase of the device. In normal mode, PHY is calibrated every time the networking subsystem is started and the device was either reset (using nShut pin), or 24 hours have passed since the last calibration. This mode is set by default, and provides maximum Tx power flexibility at the expense of occasionally prolonged initialization time. The triggered calibration mode provides more power saving, by performing calibrations after reset only if the Tx power was changed. The one-time calibration mode provides further power savings, by performing calibrations on the first system power up only (this also prevents changing the Tx power). For more information, see Section 3.8.

Provisioning

Topic **Page**

14.1 Introduction

Wi-Fi provisioning is the process of providing an IoT (Internet of Things) device the information needed to connect to a wireless network for the first time (network name, password, and so forth). Providing this information may be challenging, because not all IoT devices are equipped with conventional input peripherals such as keyboards or touchscreens.

The SimpleLink Wi-Fi CC3120, CC3220 Internet-on-a chip solution offers smart and fast built-in Wi-Fi provisioning capabilities, which lets end-users configure their IoT devices wirelessly using a smartphone or a tablet running a dedicated provisioning app. The provisioning capabilities can be easily embedded by developers on their own wireless applications.

This document describes the various provisioning methods supported by SimpleLink Wi-Fi family, and provides a detailed overview about the provisioning process flow.

14.2 Key Features

Table 14-1 lists the key provisioning features of the device.

Table 14-1. Key Features

Key Features	Description
Access-Point Provisioning	The SimpleLink Wi-Fi device creates a wireless network of its own with a predefined network name, letting the user connect it with an external device (such as smartphone, tablet, or PC) and add a profile through the internal HTTP web server.
SmartConfig Provisioning	TI proprietary provisioning method that uses a smartphone or a tablet to broadcast network credentials to an unprovisioned device. The user can add a profile using any SmartConfig-capable smartphone or tablet app.

14.3 Provisioning Process Overview

14.3.1 Configuring a Profile

When a provisioning process is started, the SimpleLink Wi-Fi device waits for the end-user to provide it (using an external tablet or smartphone app) the information needed to connect to a wireless network:

- Network name (SSID)
- Password
- Device name (optional)
- UUID (Universally Unique Identifier; optional)

The provided information is saved into the device serial flash memory as a new profile.

14.3.2 Confirming a Profile

Once a profile is configured, the device tries to connect to it to confirm that it was properly provided (the user might type the wrong SSID or password), and that the wireless network is valid. If the connection attempt was successful (such as a WLAN connection was successfully established and an IP address was successfully acquired), the device tries to provide the end-user who configured the profile a feedback (through a proper message on their tablet or smartphone app) about the successful connection.

If the feedback about a successful connection was successfully delivered to the user, the profile confirmation is successful and the provisioning process successfully ends.

If the connection was not successful, or if the connection was successful but the feedback was not successfully delivered to the user, the confirmation fails, and the device waits for the user to try to configure another profile, as shown in Figure 14-1.

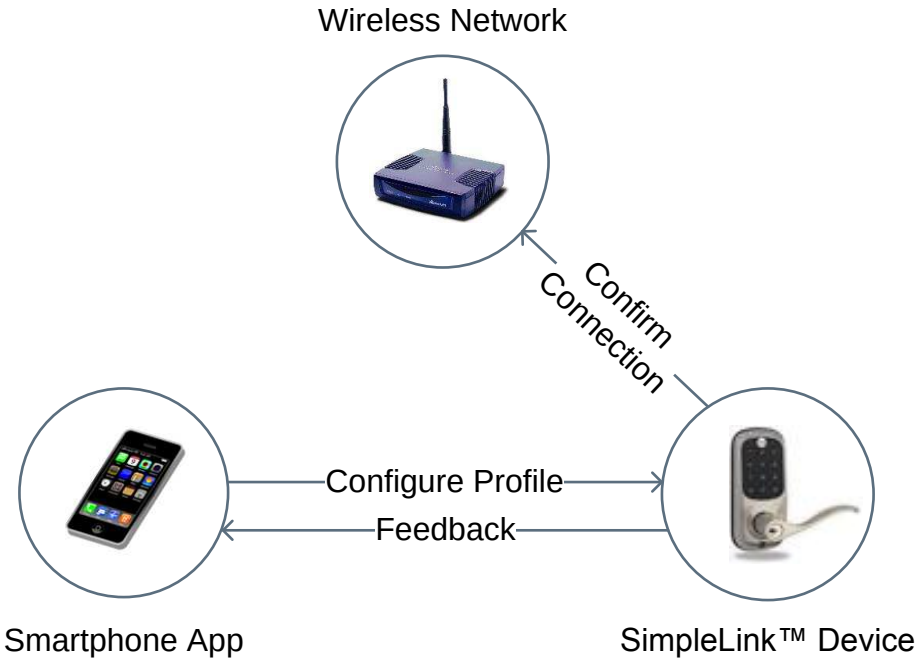

Figure 14-1. The Provisioning Environment

14.4 Host Provisioning Application Flow

The entire provisioning process (adding profiles, confirming profiles, delivering confirmation results to the user, and so forth) is executed internally by the networking subsystem. The host application is responsible only for initiating the process. Once the process is started, no further actions are needed.

Figure 14-2 depicts the host application during a provisioning process.

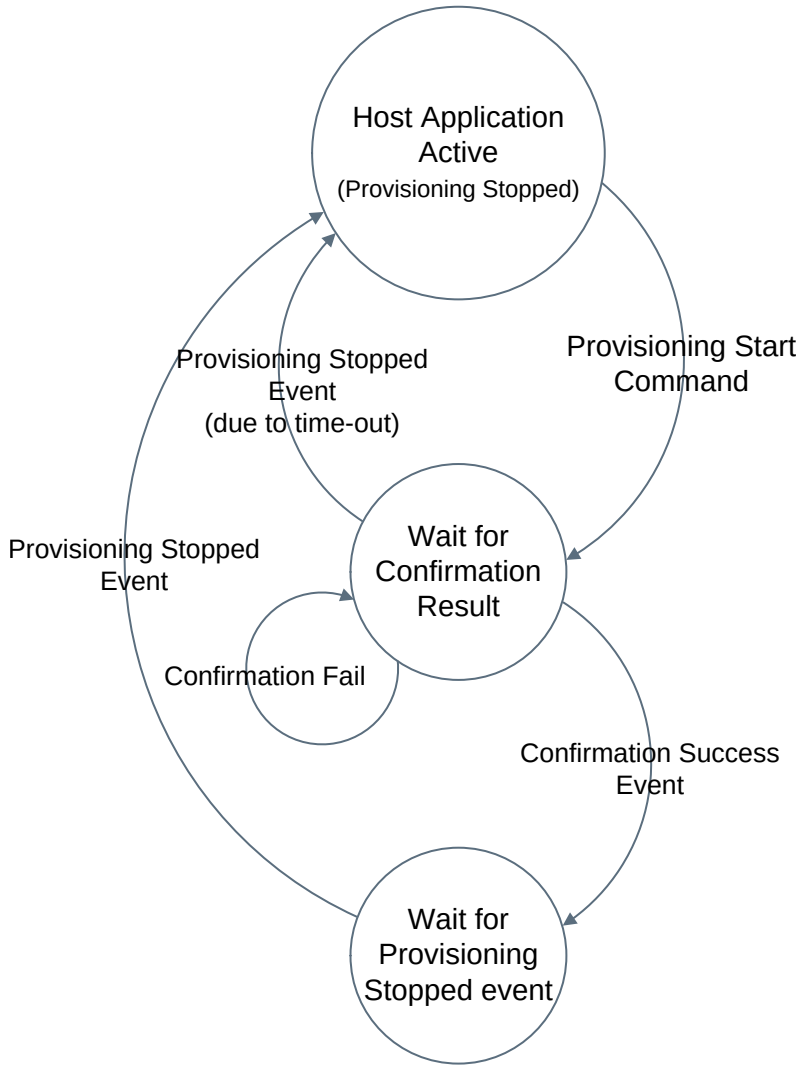

Figure 14-2. The Provisioning Process

After a provisioning process is started, the host should wait for the networking subsystem to send it the profile confirmation result. During this time, the host application should not perform any networking activity that may interrupt the ongoing provisioning process. The confirmation result is sent after the end-user has configured a profile and the networking subsystem has finished confirming it.

Possible confirmation result values:

- Confirmation failed, SSID was not found during scan
- Confirmation failed, SSID was found, but WLAN connection failed
- Confirmation failed, WLAN connection was successful, but IP address was not acquired
- Confirmation failed, IP address was successfully acquired but feedback to the user's smartphone app was not delivered
- Confirmation succeeded

If the received confirmation result is successful (that is, a profile was configured, connection was successful, and feedback was successfully delivered to the user), the provisioning process automatically stops, and the host should wait for the provisioning-stopped event before it may continue with its activities. If the profile confirmation failed, the provisioning process continues, to let the user configure another profile. If no profile was configured by the user for some time (inactivity time-out), the provisioning process automatically stops, and the provisioning-stopped event is sent to the host.

Updates regarding the progress of the provisioning process are constantly sent by the networking subsystem to the host.

14.5 Configuration Modes

The provisioning process can be started in four different configuration modes: AP provisioning, SC provisioning, AP+SC provisioning, and AP+SC+External configuration mode.

14.5.1 AP Provisioning

In this configuration mode, The SimpleLink Wi-Fi device is in AP role, creating a wireless network of its own with a predefined network name. Users can connect with an external device (such as a tablet or smartphone running a dedicated provisioning app) to the SimpleLink AP and can configure a profile through the SimpleLink HTTP server.

14.5.2 SC Provisioning

SmartConfig is a TI proprietary provisioning method that uses a smartphone or a tablet to broadcast network credentials to an unprovisioned device. In this mode, the SimpleLink Wi-Fi device is in STA role, scanning for SmartConfig data broadcasts. Users can configure a profile using any SmartConfig-capable tablet or smartphone app.

14.5.3 AP and SC Provisioning

In this mode, the SimpleLink Wi-Fi device is in AP role, simultaneously scanning for SmartConfig broadcasts. Users can either connect with an external device (such as a tablet or a smartphone running a dedicated provisioning app) to the SimpleLink AP and configure a profile through the SimpleLink HTTP server, or configure a profile using SmartConfig.

14.5.4 AP and SC and External Configuration Provisioning

In this mode, the SimpleLink Wi-Fi device is in AP role, enabling users to use AP provisioning or SmartConfig provisioning (same as AP+SC mode), or in addition, to use an external configuration method executed by the host application (for example, WAC provisioning).

14.6 Starting and Stopping the Provisioning Process

The provisioning process can be started after receiving an explicit request from the host application. When the host initiates the provisioning process, it should provide the desired configuration mode, the role (AP/STA) to which the device should switch in case of a successful provisioning, and an inactivity time-out value which defines the period of time (in seconds) the system waits before it automatically stops the provisioning process when no user activity is detected. During provisioning, the device may have higher power consumption than usual, so TI does not recommend using long inactivity time-out values (of more than few minutes).

An example of starting provisioning from the host application (in AP+SC configuration mode):

```
_i32 status;

status = sl_WlanProvisioning(SL_WLAN_PROVISIONING_CMD_START_MODE_APSC,
                             ROLE_STA,
                             PROVISIONING_INACTIVITY_TIMEOUT,
                             NULL, 0x0);
If (0 > status)
{
    /* handle error */
}
```

Once the provisioning process has started, it continues running until one of the following occurs:

- A configured profile is successfully confirmed.
- The host requested to stop the provisioning process by issuing a provisioning stop command.
- There was no user activity for some time (defined by the inactivity time-out parameter).

- The device is reset.

When the provisioning process is stopped due to host request or after inactivity time-out expires, the device switches back to the role (AP/STA) that was active before the provisioning process was started. If the process is stopped because a profile was successfully confirmed, the device switches to the role defined by the host during the provisioning start command. After a provisioning process is successfully stopped, a PROVISIONING_STOPPED event is sent to the host. The event is sent after the switching to the desired role is done. The host application should wait for the PROVISIONING_STOPPED event before issuing any additional commands. If the host application tries to issue a command during an active provisioning process, the command is not served and an error is returned.

An example of stopping provisioning from the host application follows:

```
_i32 status;

status = sl_WlanProvisioning(SL_WLAN_PROVISIONING_CMD_STOP,
                             0,
                             0,
                             NULL, 0x0);
If (0 > status)
{
    /* handle error */
}
```

14.7 Auto-Provisioning

When the auto-provisioning connection policy is enabled, the networking subsystem automatically starts provisioning process in the following cases:

- The device was started without any saved profiles, and 2 seconds have passed without receiving any command from host.
- The device is in STA role, auto-start connection policy is enabled, the profile list is not empty, and the device is disconnected from WLAN network for more than 2 minutes.

If the provisioning process is auto-started while in STA role, SC-only configuration mode is used. If it was auto-started while in AP role, AP+SC configuration mode is used. Whenever a provisioning process is auto-started by the networking subsystem, the SL_WLAN_PROVISIONING_AUTO_STARTED provisioning status event is sent to the host. The auto-provisioning connection policy is enabled by default.

14.8 Delivering Feedback to the User

After the SimpleLink device has finished confirming a profile, it should send the confirmation result to the provisioning smartphone app to report the user who configured the profile, whether the provisioning process was successful or not.

When the provisioned SimpleLink device is able to connect to the configured wireless network and acquire an IP address, it advertises itself using broadcast and multicast packets, and waits for the smartphone app to contact it. The smartphone app should connect to the same wireless network that was configured to the provisioned device, discover the device IP address by listening to its broadcasts, and send the device internal HTTP server a GET request for the confirmation result.

If the smartphone app can get the confirmation result from the device, it notifies the user about the successful provisioning, and on the side of the device, a successful confirmation result event is sent from the networking subsystem to the host.

If the provisioned device cannot successfully connect to the configured wireless network, or if it is able to connect to the configured wireless network, but the smartphone app did not receive the confirmation result, the profile confirmation fails. When a profile confirmation fails, the confirmation fail reason is sent by the networking subsystem to the host through an event, and the device waits for another profile configuration attempt. At this point, the smartphone app still does not have the confirmation result, because it was not able to find the provisioned device on the wireless network. To get the confirmation result, the smartphone app may disconnect from the configured wireless network and try to directly connect the SimpleLink device AP (possible only if AP-provisioning or AP+SC-provisioning configuration modes are used). If the smartphone app was able to connect the SimpleLink AP, it sends the device internal HTTP server a GET request for the confirmation result. If the profile confirmation failed because

the device was not able to connect to the wireless network (SSID was not found, WLAN connection failed, or an IP address was not acquired), the smartphone app reports it the user. If the profile confirmation failed because the confirmation result was not successfully delivered to the smartphone (feedback failed), the smartphone app reports to the user that the confirmation was successful, and the networking subsystem sends the host a successful confirmation result event (because the feedback was eventually successfully provided to the user's smartphone app).

14.8.1 External Confirmation

Feedback to the user's smartphone app can also be delivered through an external cloud-based server. When the SimpleLink device is able to connect to the configured network and acquire an IP address, it tries to contact a cloud-based server over the internet. The user's smartphone app, instead of connecting the same wireless network that was configured to the provisioned device, also connects the cloud-based server over the internet, and asks it if the SimpleLink device is able to connect to the cloud. In this mode, the smartphone provisioning app does not need to discover the IP address acquired by the device.

Connecting the cloud-based server is not done internally by the networking subsystem, but by the host application. When the device is able to successfully connect and acquire an IP address, it notifies the host through an event that an IP address was acquired and that it may start sending socket commands to the networking subsystem to connect the cloud server. If the device was able to successfully deliver the feedback through the cloud server to the smartphone app, the host application should manually stop the provisioning process by issuing a PROVISIONING_STOP command and order the networking subsystem to stay in STA role (instead of restoring the previous role). The networking subsystem cannot automatically stop the provisioning process, because it is not aware of the results coming from the cloud and is unaware of the successful feedback delivery. If the device can acquire an IP address, but cannot contact the smartphone app through the cloud server, the host application should notify the networking subsystem about the failure by issuing an ABORT_EXTERNAL_CONFIRMATION command, and the networking subsystem should prepare for another profile configuration attempt.

To use a cloud-based feedback, the external confirmation bit should be set in the provisioning host command flags parameter.

14.9 External Configuration

When the provisioning process is started in APSC + external configuration mode, the device is ready to serve stations trying to connect to it (for AP provisioning), ready to handle SmartConfig transmissions (SC provisioning), and can allow the host to execute an additional external provisioning method that is not implemented inside the networking subsystem (for example: WAC).

In this mode, the host is allowed to send commands and receive events from the networking subsystem while provisioning is running. After the networking subsystem has successfully started the provisioning process, it sends the EXTERNAL_CONFIGURATION_READY event to the host, which indicates that the host may start executing its external provisioning method (for example: start listening on socket). At this point, the end-user may choose which method to use: AP provisioning, SC provisioning, or the external method implemented by the host application.

If the host application identifies that the end-user chose to use the external configuration method, it should stop the internal running provisioning process by issuing a PROVISIONING_STOP command (the host should also order the networking subsystem to stay in its current role after stopping the provisioning), and continue carrying out the external provisioning process.

If the end-user has configured a profile using one of the internal provisioning methods (AP or SC provisioning), the device must be restarted before it can continue the internal provisioning process. The networking subsystem sends a RESET_REQUEST event to the host, and the host should stop its external provisioning process (close all opened sockets, and so forth), restart the SimpleLink Wi-Fi device (by sending sl_stop and sl_start commands), and wait for the internal provisioning process to end.

14.10 Common Events and Errors

14.10.1 Provisioning Status Event

The networking subsystem constantly updates the host application regarding the progress of the provisioning process, through the provisioning status event.

The provisioning status event has the following parameters:

- Status
- Role
- WlanStatus
- SsidLen
- Ssid

Table 14-2 lists the status parameter values.

Table 14-2. Provisioning Status

SL_WLAN_PROVISIONING_GENERAL_ERROR	0	The provisioning process has encountered an unknown error. TI recommends stopping and starting the process again.
SL_WLAN_PROVISIONING_CONFIRMATION_STATUS_FAIL_NETWORK_NOT_FOUND	1	Profile confirmation failed because the SSID was not found during scan.
SL_WLAN_PROVISIONING_CONFIRMATION_STATUS_FAIL_CONNECTION_FAILED	2	Profile confirmation failed; the SSID was found during scan, but the WLAN connection was not successful (possibly due to the wrong password).
SL_WLAN_PROVISIONING_CONFIRMATION_STATUS_FAIL_CONNECTION_SUCCESS_IP_NOT_ACQUIRED	3	Profile confirmation failed; the SSID was found during scan, the WLAN connection was successful, but an IP address was not successfully acquired.
SL_WLAN_PROVISIONING_CONFIRMATION_STATUS_SUCCESS_FEEDBACK_FAILED	4	Profile confirmation failed; the SSID was found during scan, the WLAN connection was successful, IP address was successfully acquired, but the feedback to user about the successful connection was not successfully delivered. This event might be followed by a profile confirmation succeeded event, if feedback is eventually delivered.
SL_WLAN_PROVISIONING_CONFIRMATION_STATUS_SUCCESS	5	Profile confirmation succeeded; the SSID was found during scan, the WLAN connection was successful, IP address was successfully acquired, and the feedback to user about the successful connection was successfully delivered.
SL_WLAN_PROVISIONING_ERROR_ABORT	6	The provisioning process was not started due to an unknown error.
SL_WLAN_PROVISIONING_ERROR_ABORT_INVALID_PARAM	7	Auto-provisioning process was not started due to an invalid parameter.
SL_WLAN_PROVISIONING_ERROR_ABORT_HTTP_SERVER_DISABLED	8	Auto-provisioning process was not started because the HTTP server is disabled.
SL_WLAN_PROVISIONING_ERROR_ABORT_PROFILE_LIST_FULL	9	Auto-provisioning process was not started because the profile list is full.
SL_WLAN_PROVISIONING_ERROR_ABORT_PROVISIONING_ALREADY_STARTED	10	Auto-provisioning process was not started because it is already running.
SL_WLAN_PROVISIONING_AUTO_STARTED	11	The provisioning process was automatically started by the networking subsystem.
SL_WLAN_PROVISIONING_STOPPED	12	The provisioning process has ended.
SL_WLAN_PROVISIONING_SMART_CONFIG_SYNCED	13	SmartConfig configuration data transmission was discovered by the device. The device starts listening and collecting the profile data.
SL_WLAN_PROVISIONING_SMART_CONFIG_SYNC_TIMEOUT	14	SmartConfig configuration data transmission was discovered by the device, but the device was not able to extract the profile data from it.
SL_WLAN_PROVISIONING_CONFIRMATION_WLAN_CONNECT	15	A WLAN connection was established during profile confirmation attempt.
SL_WLAN_PROVISIONING_CONFIRMATION_IP_ACQUIRED	16	IP address was acquired during profile confirmation attempt.
SL_WLAN_PROVISIONING_EXTERNAL_CONFIGURATION_READY	17	The host application may start running an external provisioning method (relevant only when APSC + External Configuration mode is used).

During provisioning, the device might switch between different roles and connection statuses without notifying the host application; thus, when the process is stopped, a report about the current status of the device is sent to the host. When the value of the status parameter is SL_WLAN_PROVISIONING_STOPPED (12), additional information is provided through the following parameters:

- Role: The active role (AP/STA) after the provisioning process ended.
- WlanStatus: If the active role is STA, this parameter also shows the device WLAN connection status (0-Disconnected, 1-Scanning, 2-Connecting, 3-Connected) after the provisioning process ended.
- Ssid, SsidLen: If WlanStatus is connected, these parameters provide the SSID to the connected device.

These parameters are not relevant in other provisioning status values.

14.10.2 Provisioning Profile-Added Event

When a profile is configured to the device during provisioning, the SL_WLAN_EVENT_PROVISIONING_PROFILE_ADDED event is sent to the host.

14.10.3 Reset Request Event

During the provisioning process, the SimpleLink device might automatically restart itself as part of process. If a restart is required while the host application is busy (for example, when the host has opened sockets during external configuration provisioning), instead of performing the restart automatically, the networking subsystem asks the host application to do it. When this event arrives, the host should stop its activities (for example, close all opened sockets), and restart the device by issuing sl_stop and sl_Stop commands.

14.10.4 Errors

Table 14-3 shows the following values that may be returned when a provisioning command is issued.

Table 14-3. Errors

STATUS_OK	0	Command was successfully executed.
SL_ERROR_WLAN_PROVISIONING_ABORT_PROVISIONING_ALREADY_STARTED	-2169	Start provisioning command failed because provisioning process is already running.
SL_ERROR_WLAN_PROVISIONING_ABORT_HTTP_SERVER_DISABLED	-2170	Start provisioning command failed because the HTTP server is disabled.
SL_ERROR_WLAN_PROVISIONING_ABORT_PROFILE_LIST_FULL	-2171	Start provisioning command failed because the profile list is full.
SL_ERROR_WLAN_PROVISIONING_ABORT_INVALID_PARAM	-2172	Start provisioning command failed because one of the parameters was invalid.
SL_ERROR_WLAN_PROVISIONING_ABORT_GENERAL_ABORT	-2173	Start provisioning command failed because of an unknown reason.
SL_ERROR_WLAN_PROVISIONING_CMD_NOT_EXPECTED	-2177	Provisioning command failed because it was not expected.

14.10.5 Host Commands During Provisioning

During the provisioning process, the device switches between different roles, connects to different APs, and changes its IP addressthus, the host commands may not be properly served. As a result, when a command is issued by the host application during an active provisioning process, the SL_RET_CODE_PROVISIONING_IN_PROGRESS (–2014) error is returned. The only allowed commands are sl_WlanProvisioning and sl_stop. If host is interested to execute a different command, it must either wait for the provisioning process to end, or to manually stop it (using the SL_WLAN_PROVISIONING_CMD_STOP command). In addition, events that may be sent to the host during the provisioning connection attempts (such as NETAPP_IPACQUIRED) are blocked, and will not reach the host application (except for dedicated provisioning events, such as the provisioning status event).

In some cases, after provisioning starts, the host is allowed to send commands and receive all events to perform some actions necessary for completing the provisioning process:

- External confirmation: When feedback to the user's smartphone app is done using an external cloud-based server, the host application must be able to access the internet. Therefore, commands are allowed right after the PROVISIONING_CONFIRMATION_IP_ACQUIRED status event is sent to the host.
- External configuration: When APSC + external configuration mode is used, the host application might need to issue a socket command as part of its external provisioning process. To enable this, commands are allowed right after the PROVISIONING_EXTERNAL_CONFIGURATION_READY status event is sent to the host.
- Auto-provisioning: When provisioning is auto-started, commands are still allowed (unlike host-initiated provisioning, where the commands are blocked right after the provisioning process was started). They are blocked only after user activity was detected (such as when a profile is being configured).

14.11 Usage Examples

14.11.1 Successful SmartConfig Provisioning

In Figure 14-3, a profile is configured using SmartConfig. The provisioned device connects to the wireless network from the configured profile and waits for the smartphone app to contact its HTTP web server. When the confirmation result is delivered to the smartphone app, the device sends the successful result to the host, and stops the process.

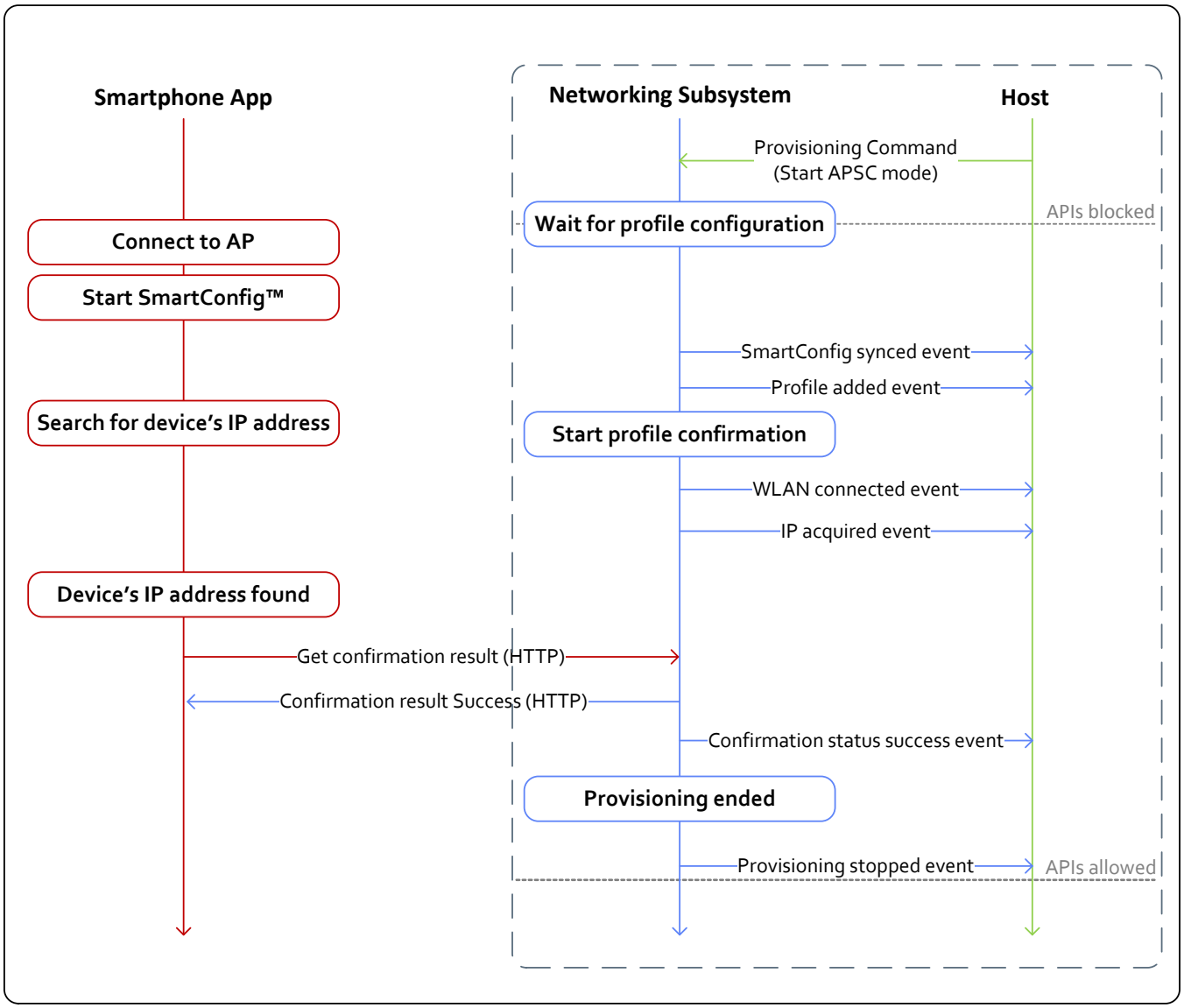

Figure 14-3. Successful SmartConfig Provisioning

The APIs between the host driver and the networking subsystem (commands and events) are blocked during the entire provisioning process. The host is allowed to send commands only after the provisioning_stopped event arrives.

14.11.2 Unsuccessful SmartConfig Provisioning

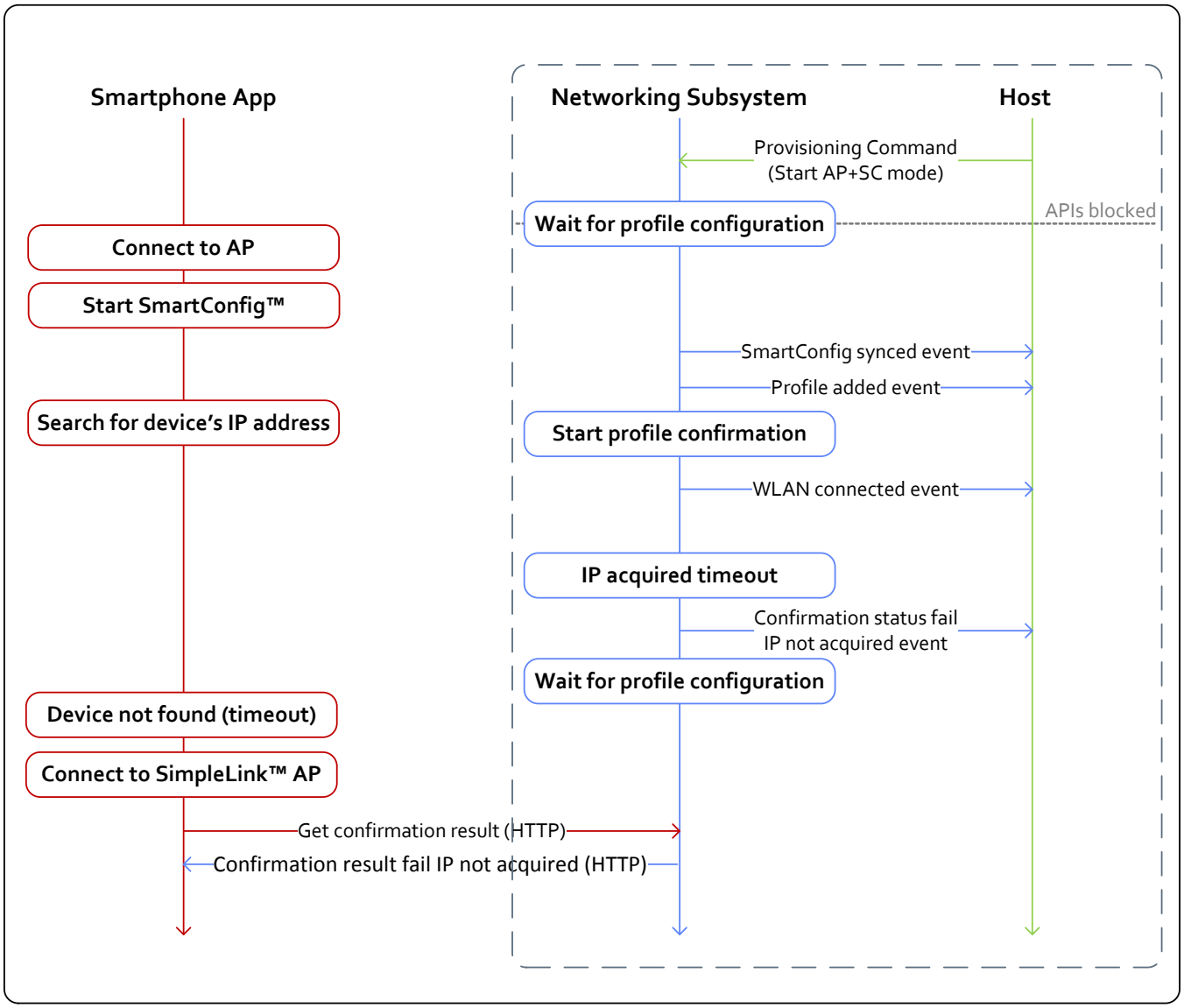

Figure 14-4. Unsuccessful SmartConfig Provisioning

In AP+SC mode, the device is waiting for a profile configuration while in AP role.

After profile confirmation fails (because an IP address was not acquired in the configured wireless network), the device is ready for another profile configuration (back in AP role).

After the smartphone app fails to find the device and collect the confirmation result on the local wireless network, it tries to get it by connecting directly to the SimpleLink device AP.

14.11.3 Successful SmartConfig Provisioning With AP Fallback

In APSC mode, the device is waiting for a profile configuration while in AP role.

In Figure 14-5, the provisioned device connects to the wireless network, but the smartphone app fails to find the device and collect the confirmation result.

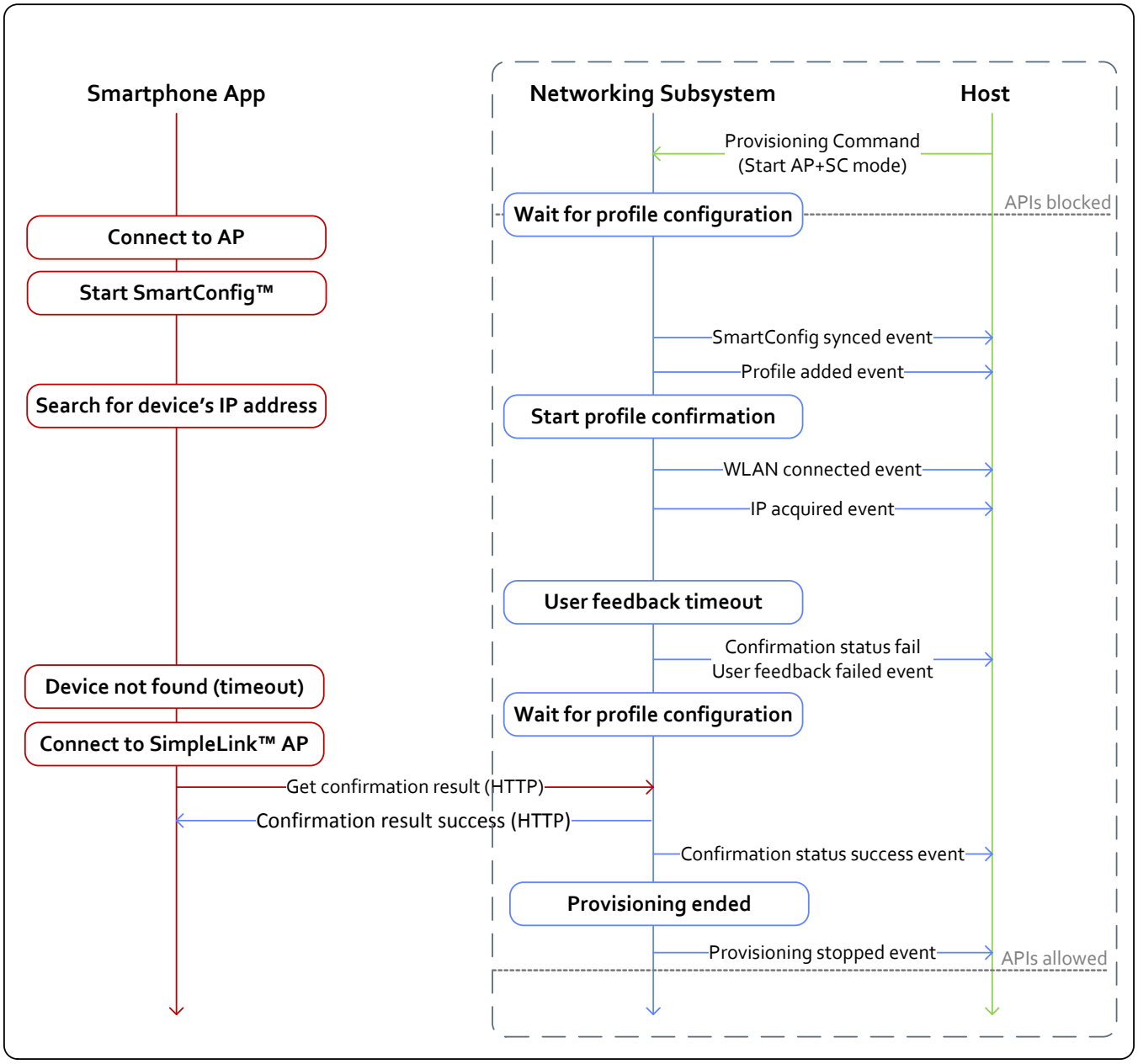

Figure 14-5. Successful SmartConfig Provisioning With AP Fallback

After profile confirmation fails, the device waits for another configuration attempt.

After the smartphone app fails to find the device and collect the confirmation result on the local wireless network, it tries to get it by connecting directly to the SimpleLink device AP.

14.11.4 Successful AP Provisioning

In Figure 14-6, a profile is configured to the SimpleLink device through its internal HTTP server while the device is in AP mode.

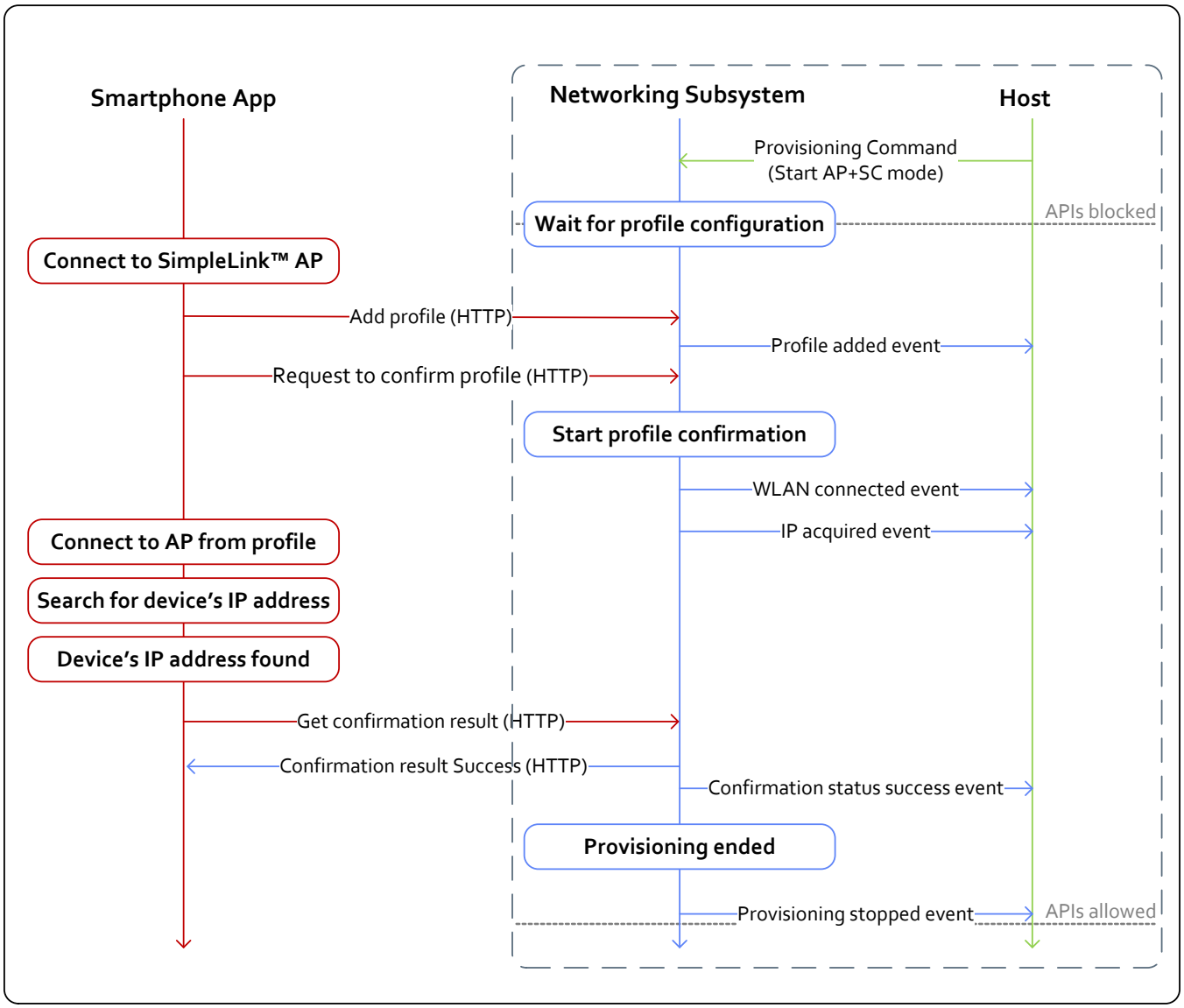

Figure 14-6. Successful AP Provisioning

14.11.5 Successful AP Provisioning With Cloud Confirmation

Figure 14-7 depicts a successful AP provisioning with cloud confirmation.

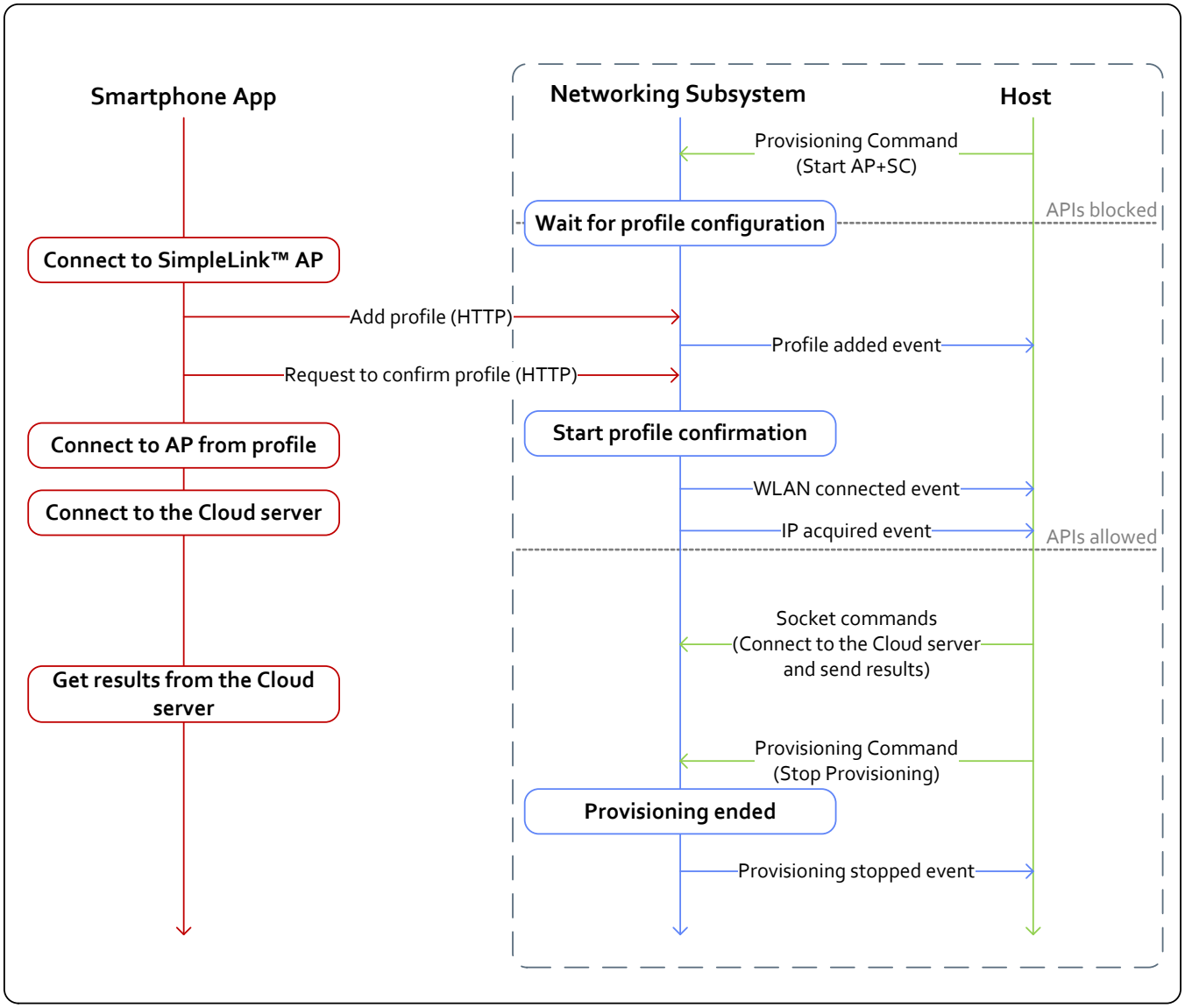

Figure 14-7. Successful AP Provisioning With Cloud Confirmation

When feedback is provided through a cloud server (external confirmation), the host can send commands to the networking subsystem to connect the cloud-based server only after the IP acquired event is received.

Because the networking subsystem is unaware of the results coming from the cloud server, the host is responsible for stopping the provisioning process (and for ordering the networking subsystem to stay in its active role – STA), in case confirmation is successful. For the same reason, the host must order the networking subsystem to stop the profile confirmation attempt (by sending the ABORT_EXTERNAL_CONFIRMATION command) in case confirmation failed.

14.11.6 Using External Configuration Method: WAC

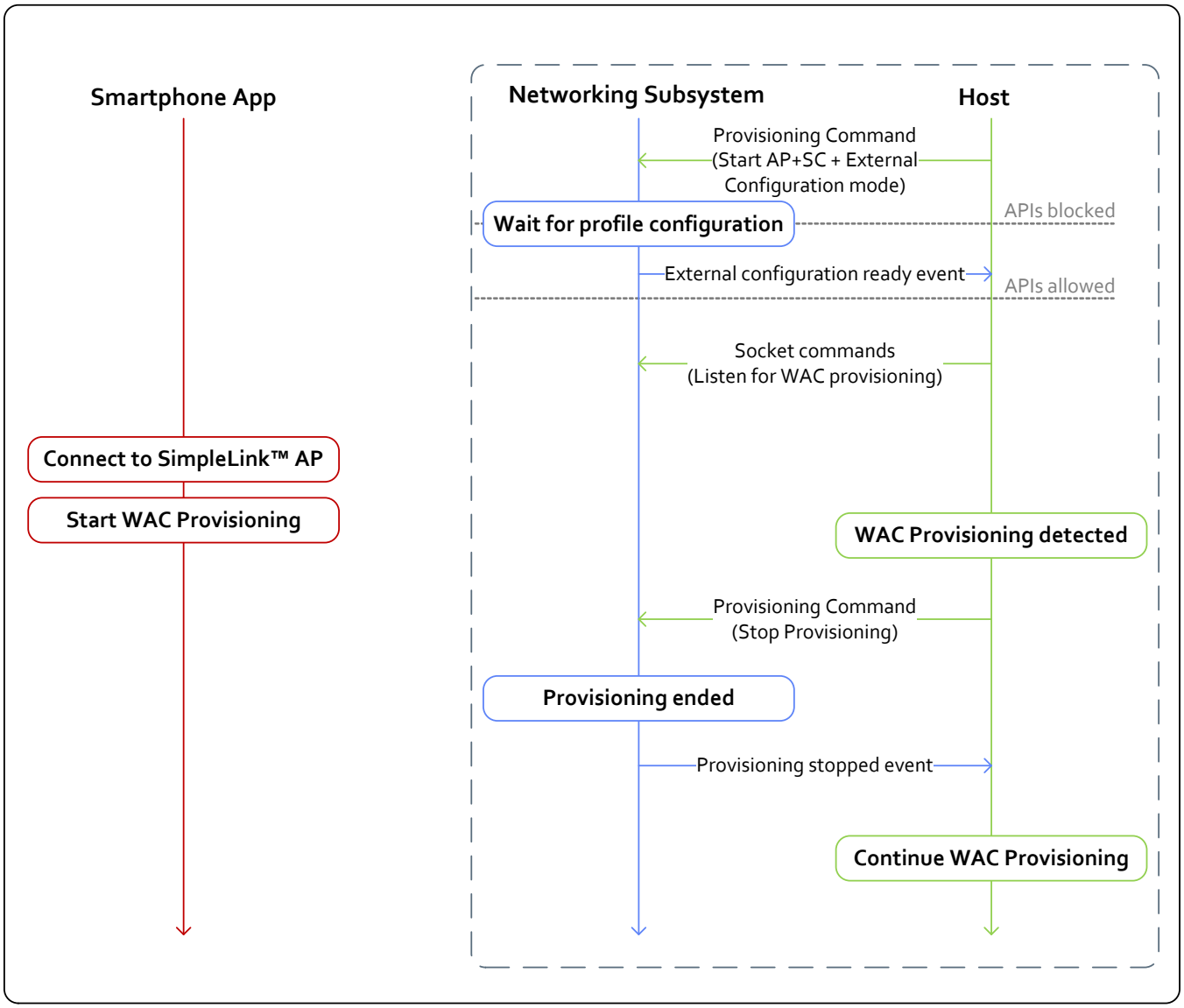

Figure 14-8. External Configuration Method: WAC

When provisioning is started in AP + SC + external configuration mode, the host can start sending commands to the networking subsystem only after the external configuration ready event is received. When the host identifies that a user has started a provisioning process using the external configuration method, it should order the networking subsystem to stop the internal provisioning process. When the internal provisioning process is stopped, the host can continue with its external provisioning process.

14.11.7 Successful SmartConfig Provisioning While External Configuration Enabled

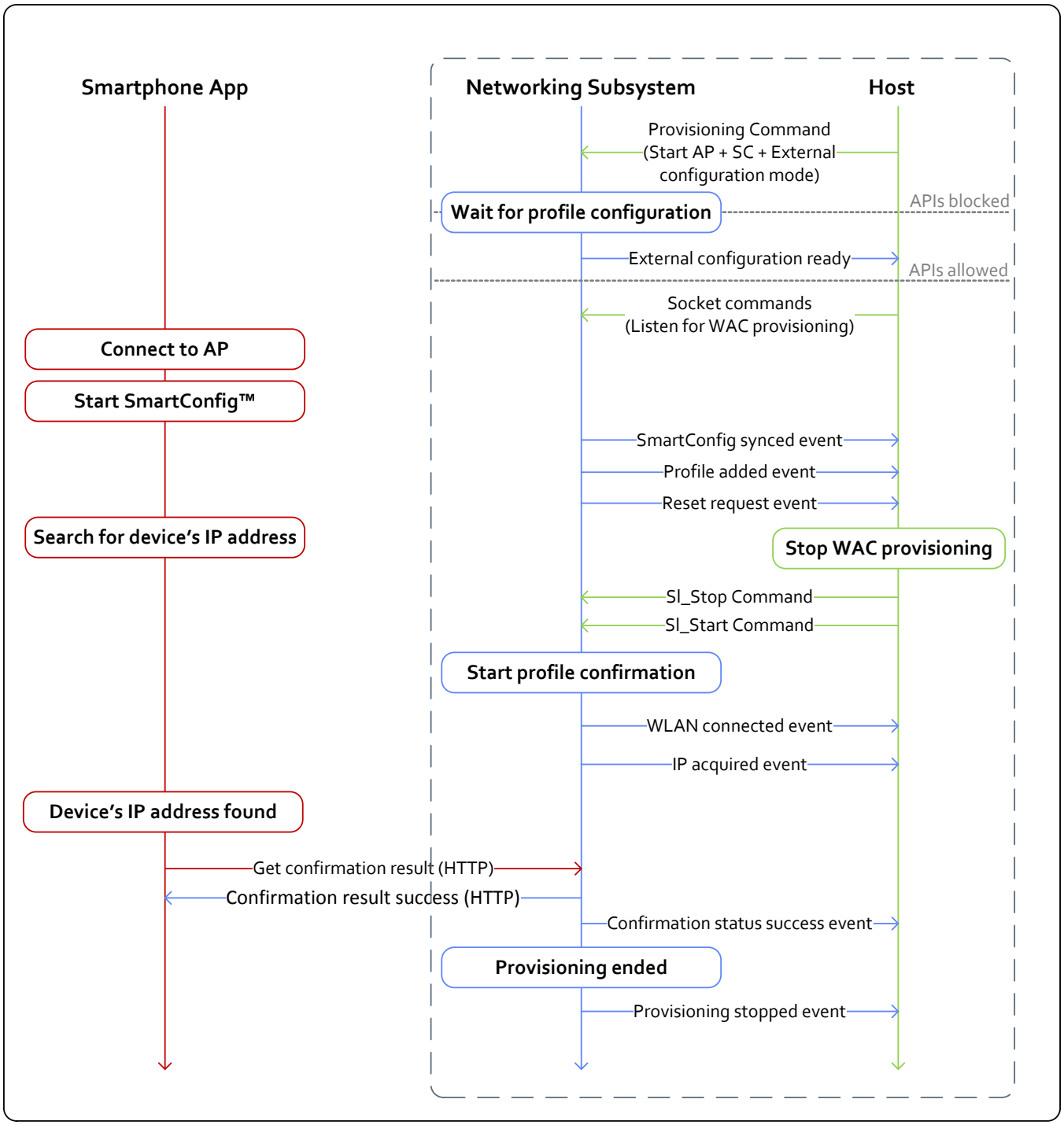

Figure 14-9. Successful SmartConfig Provisioning While External Configuration Enabled

When provisioning is started in AP + SC + external configuration mode, and the user is using one of the internal provisioning methods (AP or SC), the device sends a reset request event to the host. The host should stop all of its external provisioning activities, and restart the device. Once the device is restarted, it starts the profile confirmation, and the internal provisioning process continues as usual.

Crypto Utilities

Topic **Page**

15.1 Introduction

The SimpleLink device supports on-chip asymmetric key-pair storage (secure key storage) with built-in crypto acceleration and crypto services to assist in some common cryptographic-related operations.

These crypto services provide a mechanism to manage up to eight ECC key-pairs, and use them to sign or verify data buffers. This capability could be used for authenticating the device identity, among other usage options.

There are three types of supported key-pairs:

- Device-unique key-pair: A single 256-bit unique key of the device, embedded in hardware
- Temporary key-pair: Created upon request using the internal TRNG engine
- Installed key-pair: Installed and maintained by the vendor

The system supports a single constant key-pair. Entry 0 is reserved for this key-pair. Entries 1 to 7 could be used for temporary or installed key-pairs, according to the application needs. All keys are ECC keys using the SECP256R1 curve. This applies to all entries:

- Constant and temporary key types – The SimpleLink Wi-Fi device is responsible for using the correct type and curve.
- Installed key type – The vendor is responsible only for installing keys of this type and curves.

For all key pairs, the private key is never exposed, and can only be accessed indirectly when using it to sign buffers. The public key may be retrieved by the host application (see Section 15.1.1). Table 15-1 lists the key crypto utility features.

Table 15-1. Key Features

Main Features	Description
Manage temporary key-pair	Create or remove temporary keys at a provided index. Temporary keys are not persistent over the power cycle. Creating a temporary key in an index occupied by another temporary key, and overrides the occupied key. An installed key on that index cannot be overridden.
Installed key-pairs	Install or uninstall key pairs provided by the host application. The keys must be preprogrammed in the file system. The install action adds them to the data base, and allows using them to sign and verify buffers. This operation is persistent over the power cycle without consideration of system-persistent configuration. Cannot install a key in an occupied index.
Constant key-pair	Unique key for the given device, embedded in the hardware. Always available and constant.
Retrieve public key	For any key-pair type, the host may request to retrieve the public key of the pair in x9.63 raw format. The host application can also retrieve the metadata of this key (type, curve, length, filename, and so forth).
Verify buffer	Given a buffer and a signature, the host can request to use any key-pair to verify the ECDSA signature.
Sign buffer	Given a buffer, the host can request to use any key-pair to create a signature using ECDSA.
Secured content delivery	Transfer secure content by sending the public key to the application server which encrypts the file, and decrypts in the device internally using the private key only
True random number	Retrieve buffer with true random numbers from the networking subsystem. Maximum buffer length 172 bytes for each retrieval.

15.1.1 API and Usage

15.1.1.1 Install and Uninstall Key-Pairs and Certificates

This command is used to install and uninstall a key-pair in one of the crypto utilities key-pair management mechanism. The key must be an ECC key-pair using the SECP256R1 curve, and already programmed to the file system in a DER format file. The install and uninstall operations are done using the sl_NetUtilsCmd API. The key installation is persistent without consideration of system-persistent configuration, and is not erased over the power cycle. If the wanted index is already occupied by a key-pair, the install operation fails. Installation of the certificate without a key pair is used to verify buffers.

An example of installing a key and then uninstalling it follows:

```
SlNetUtilCryptoCmdKeyMgnt_t keyAttrib;
SlNetUtilCryptoPubKeyInfo_t *pInfoKey;
_i16 Status;
_u8 buf[256];
_u16 resultLen;


keyAttrib.ObjId = 5;  /* key index is 5 */
keyAttrib.SubCmd = SL_NETUTIL_CRYPTO_INSTALL_SUB_CMD;
pInfoKey->KeyAlgo = SL_NETUTIL_CRYPTO_PUB_KEY_ALGO_EC;
pInfoKey->KeyParams.EcParams.CurveType = SL_NETUTIL_CRYPTO_EC_CURVE_TYPE_NAMED;
pInfoKey-
>KeyParams.EcParams.CurveParams.NamedCurveParams = SL_NETUTIL_CRYPTO_EC_NAMED_CURVE_SECP256R1;

pInfoKey = (SlNetUtilCryptoPubKeyInfo_t *)buf;
name = ((_u8 *)pInfoKey) + sizeof(SlNetUtilCryptoPubKeyInfo_t);
pInfoKey->KeyAlgo = SL_NETUTIL_CRYPTO_PUB_KEY_ALGO_EC;
pInfoKey->KeyParams.EcParams.CurveType = SL_NETUTIL_CRYPTO_EC_CURVE_TYPE_NAMED;
pInfoKey-
>KeyParams.EcParams.CurveParams.NamedCurveParams = SL_NETUTIL_CRYPTO_EC_NAMED_CURVE_SECP256R1;

pInfoKey->CertFileNameLen = 0;        /* unused */
name += pInfoKey->CertFileNameLen;
strcpy((char *)name, "extkey.der");   /* private key in the file system */
pInfoKey->KeyFileNameLen = strlen("extkey.der")+1;

Status = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_INSTALL_OP,
                       (_u8 *)&keyAttrib, sizeof(SlNetUtilCryptoCmdKeyMgnt_t),
                       (_u8 *)pInfo,
                       sizeof(SlNetUtilCryptoPubKeyInfo_t) + pInfoKey->KeyFileNameLen,
                       NULL, &resultLen);
if(Status < 0)
{
    /* error */
}
resultLen = 0;
keyAttrib.ObjId = 5;
keyAttrib.SubCmd = SL_NETUTIL_CRYPTO_UNINSTALL_SUB_CMD;
/* Uninstall the key */
Status = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_INSTALL_OP, (_u8 *)&keyAttrib,
                       sizeof(SlNetUtilCryptoCmdKeyMgnt_t), NULL, 0 , NULL, &resultLen);
if(Status < 0)
{
    /* error */
}
```

15.1.1.2 Create or Remove Temporary Key

This command is used to create or remove a temporary ECC key-pair with the SECP256R1 curve on a given index. Create and remove operations are done using the sl_NetUtilsCmd API. The key is generated internally by the SimpleLink Wi-Fi device. The key is not persistent over the power cycle, and is overridden if using create temporary key again on that index. The operation fails if the desired index is already occupied by an installed key-pair (not a temporary one).

An example of creating a temporary key pair follows:

```
SlNetUtilCryptoCmdKeyMgnt_t keyAttrib;
_i16 Status;
_u16 resultLen;

keyAttrib.ObjId = 1;  /* key index is 1 */
keyAttrib.SubCmd = SL_NETUTIL_CRYPTO_TEMP_KEYS_CREATE;

Status = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_TEMP_KEYS,
                       (_u8 *)&keyAttrib, sizeof(SlNetUtilCryptoCmdKeyMgnt_t),
```

```
                        NULL,
                        0,
                        NULL, &resultLen);
if(Status < 0)
{
    /* error */
}
```

15.1.1.3 Get Public Key

This command is used to retrieve the public key of the key-pair installed or temporarily created in a certain index.

The key is in x9.63 raw format. The operation is done using the sl_NetUtilGet API.

```
_i16 Status;
_u8 configOpt = 0;
_u32 objId = 0;
_u16 configLen = 0;

configOpt = SL_NETUTIL_CRYPTO_PUBLIC_KEY;
objId = 1;
configLen = 255;

/* get the Public key */
Status = sl_NetUtilGet(configOpt, objId, buf, &configLen);
if(Status < 0)
{
    /* error */
}
```

15.1.1.4 Sign Buffer

This command is used to create a digital signature using the ECDSA algorithm and a key-pair from the crypto-utilities key management mechanism. This operation is done using the sl_NetUtilCmd. Signing a buffer is only allowed with ECDSAwithSHA.

NOTE: The input buffer for signing must not exceed 1.5KB.

15.1.1.5 Verify Buffer

This command is used to verify a digital signature using the ECDSA algorithm. The signature must be created with one of the key-pairs from the crypto-utilities key management mechanism. Verification of a buffer can be done by ECDSAwithSHA or ECDSAwithSHA256, where the buffer to digest is given in the API. If a predigested message is used, verification occurs when the digest is passed in the verify command, instead of the buffer.

NOTE: The input buffer for signing must not exceed 1.5KB. If a larger buffer must be verified, predigest the buffer and pass it as the verify buffer with SL_NETUTIL_CRYPTO_SIG_DIGESTwECDSA sigType.

An example of sign and verify buffer:

```
_u16 configLen = 0;
_u8 buf[256];
_u8 verifyBuf[2048];
SlNetUtilCryptoCmdSignAttrib_t signAttrib;
SlNetUtilCryptoCmdVerifyAttrib_t verAttrib;
_i32 verifyResult;
_u16 resultLen;
_u8 messageBuf[1500];
_i16 Status;
```

```
signAttrib.Flags = 0;
signAttrib.ObjId = 3;
signAttrib.SigType = SL_NETUTIL_CRYPTO_SIG_SHAwECDSA; /* this is the only type supported */
configLen = 255;

Status = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_SIGN_MSG, (_u8 *)&signAttrib,
                       sizeof(SlNetUtilCryptoCmdSignAttrib_t),
                       messageBuf,  sizeof(messageBuf), buf, &configLen);
if(0 > Status)
{
    /* error */
}

/* now verify the buffer */
memcpy(verifyBuf, messageBuf, sizeof(messageBuf));
memcpy(verifyBuf + sizeof(messageBuf), buf, configLen);

verAttrib.Flags = 0;
verAttrib.ObjId = 3;
verAttrib.SigType = SL_NETUTIL_CRYPTO_SIG_SHAwECDSA; /* this is the only type supported */
verAttrib.MsgLen = sizeof(messageBuf);
verAttrib.SigLen = configLen;

/* use the created keys to verify the signature from the previous step */
resultLen = 4;
Status = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_VERIFY_MSG, (_u8 *)&verAttrib,
                       sizeof(SlNetUtilCryptoCmdVerifyAttrib_t),
                       verifyBuf, sizeof(messageBuf) + configLen,
                       (_u8 *)&verifyResult , &resultLen);
if(0 > Status)
{
    /* error */
}
```

15.1.1.6 True Random Number

Retrieve a buffer of true random numbers from the networking subsystem. Maximum buffer length is 172 bytes for each retrieval. if the requested length exceeds 172 bytes, it is trimmed to 172 bytes.

```
Status = sl_NetUtilGet(SL_NETUTIL_TRUE_RANDOM,0,buffer,&len);
```

15.1.2 Limitations and Constraints

- Mechanism supports a total of eight keys, where index 0 is reserved for the constant key-pair.
- Only ECC keys using the SECP256R1 curve are supported.
- Index management is a host application responsibility; find free index or retrieve index list are not provided.
- For signing and verifying operations, the buffer size is limited to 1.5KB.

15.1.3 Errors

Table 15-2 lists the common errors.

Table 15-2. Common Errors

Error Code	Value	Comments
SL_ERROR_NETUTIL_CRYPTO_GENERAL	-12289	An unspecified general error has occurred.
SL_ERROR_NETUTIL_CRYPTO_INVALID_INDEX	-12290	The provided index is out of the valid range.
SL_ERROR_NETUTIL_CRYPTO_INVALID_PARAM	-12291	One of the provided parameters is invalid or illegal.
SL_ERROR_NETUTIL_CRYPTO_MEM_ALLOC	-12292	A memory-allocation failure has occurred.

Table 15-2. Common Errors (continued)

Error Code	Value	Comments
SL_ERROR_NETUTIL_CRYPTO_INVALID_DB_VER	-12293	Not in use
SL_ERROR_NETUTIL_CRYPTO_UNSUPPORTED_OPTION	-12294	One of the provided parameters requires an unsupported capability or option.
SL_ERROR_NETUTIL_CRYPTO_BUFFER_TOO_SMALL	-12295	The buffer provided by the host-application is not large enough to contain the returned output.
SL_ERROR_NETUTIL_CRYPTO_EMPTY_DB_ENTRY	-12296	The provided index points to an empty database entry.
SL_ERROR_NETUTIL_CRYPTO_NON_TEMPORARY_KEY	-12297	The host application is trying to perform an operation related to temporary keys, but the provided index does not contain a temporary key.
SL_ERROR_NETUTIL_CRYPTO_DB_ENTRY_NOT_FREE	-12298	The provided index points to a nonempty database entry (while the requested operation requires the entry to be empty).
SL_ERROR_NETUTIL_CRYPTO_CORRUPTED_DB_FILE	-12299	The file that stores the database on the filesystem (for persistency) has been corrupted.

15.2 Secured Content Delivery

The secure content delivery feature lets the user program a secured file, which is encrypted by a remote device and decrypted inside the NWP. The private key used for the process remains inside the SimpleLink Wi-Fi networking subsystem alone with no access from the host. This ability lets the user transfer a file to the system on any unsecured tunnel.

NOTE: Secured content delivery is designed to work with a temporary key generated on secure key index 1.

15.2.1 Process Flow

1. Retrieve a temporary, nonpersistent ECC public key using the NetUtils APIs described in *Appendix A*.
2. Send the public key to the application remote server.
3. Receive the encrypted file.
4. Open a file with a special flag, indicating secure content delivery is about to be written:

```
secAccessFlags = SL_FS_FILE_MODE_OPEN_CREATE(fpInSize,SL_FS_FILE_DOWNLOAD_SECURED_CONTENT);
fileHande = sl_FsOpen("sec_cont1.txt",secAccessFlags,NULL);
if(0 > fileHande)
{
    /* error */
}
```

5. Write the file sequentially (all bytes in order with no random access) – the offset attribute in the sl_FsWrite has no meaning regarding a secured content delivery write, and is ignored.
6. Close the file using the sl_FsClose API.

At the end of this process, the file is saved on the SFLASH, and encrypted as a normal secured file on the file system. The file system uses a different key and method than is used to encrypt the file for the secure content delivery process.

Figure 15-1 depicts this process.

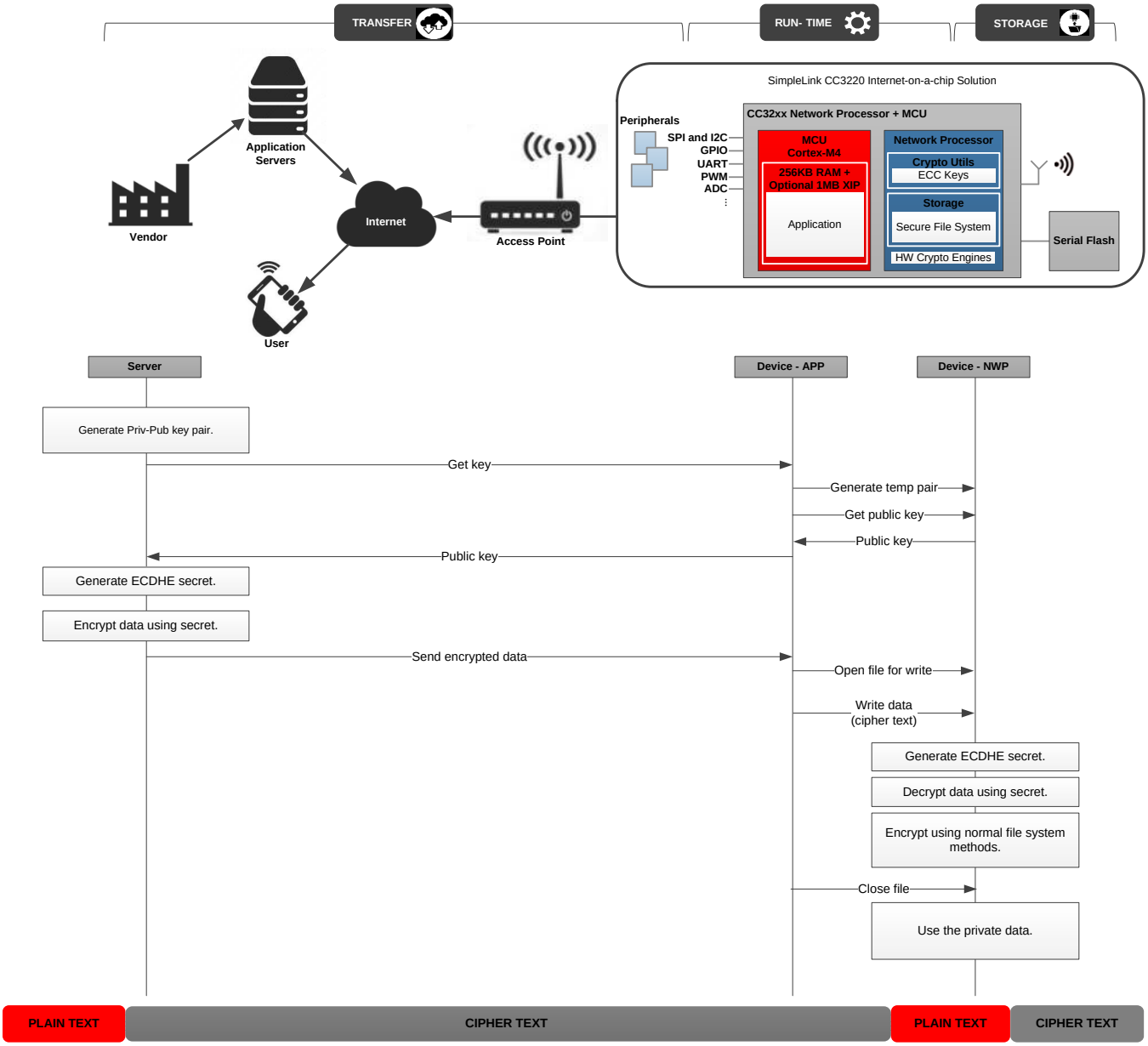

Figure 15-1. Secure Content Delivery

15.2.2 Encrypted File Format

Multiple steps are involved in building the secure content file in the format expected by the network processor. Before building the file, the server must first derive the ECDHE secret from the public key sent by the SimpleLink device and the private key of the server. The AES key and initialization vector (IV) used to encrypt the data are formed as follows:

- AES IV: Upper 128 bits of ECDHE secret
- AES Key
  - Upper 128 bits of AES Key = Bitwise XOR of upper and lower 128 bits of ECDHE secret
  - Lower 128 bits of AES Key = Lower 128 bits of ECDHE secret

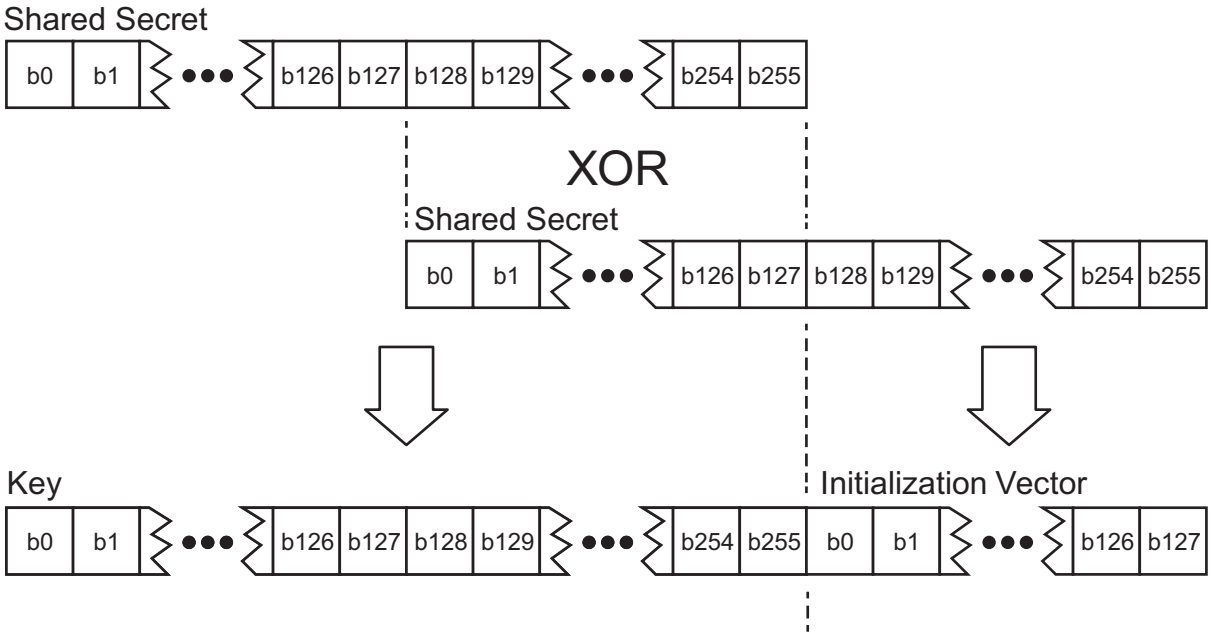

Figure 15-2. AES Key Diagram

The order of operations for building the bundle is:

1. Use SHA256 to generate a digest of the RAW data.
2. Append the RAW data to the digest in a single file.
3. Encrypt the file (digest + RAW data) with AES 256 CBC (allow the encryption function to pad the file as needed).
4. Add the bundle header that includes the RAW data size, padding, and server ECC public key.

When creating the bundle header, the RAW data size should be specified in little endian format (that is, a raw data size of 16 is specified as "10 00 00 00" at the beginning of the file).

The file delivered with this process should be in the proprietary format (see Figure 15-3).

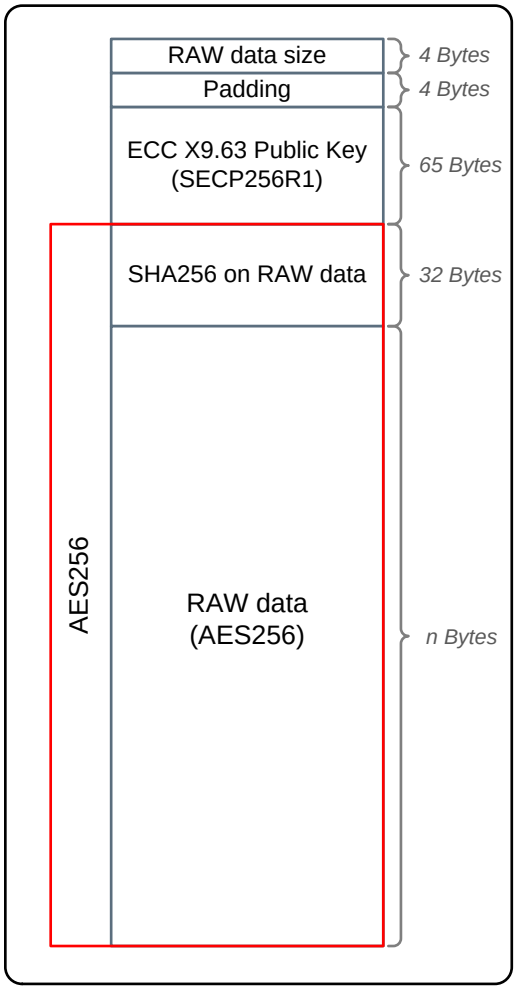

Figure 15-3. File Format

Porting the Host Driver

Topic **Page**

16.1 Introduction

The SimpleLink Wi-Fi device family consists of several device types: the CC3220S and CC3220SF, which are fully-integrated system-on-chip (SoC) solutions consisting of both applications MCU and the network processor, and the CC3120, which consists only of the network subsystem processor. The CC3120 device can be bundled with any platform (MCU, MPU, or other). While the CC3220x is already fully integrated, to work with the CC3120 device, the user must port its host driver to the new platform. The porting of the SimpleLink Wi-Fi host driver to any new platform is based on a few simple steps. This chapter provides basic step-by-step guidelines on how to port the SimpleLink host driver to new platforms. Follow the instructions carefully to avoid any problems during this process and to enable efficient and proper work with the CC3120 device.

The basic concept of the porting is that all modifications and porting adjustments of the host driver are made in one file (user.h header file). Strictly following these guidelines ensures a smooth transition to new versions of the driver. The porting process consists of a few simple steps:

1. Create the user.h file for the target platform.
2. Select capabilities set.
3. Bind the device enable/disable line.
4. Implement the interface communication driver.
5. Choose memory-management model.
6. Implement OS adaptation layer.
7. Implement timestamp services.
8. Bind asynchronous event handlers routines.

The remainder of this chapter describes these steps in more detail.

16.2 Create Platform Porting File

The first step is to create a user.h file, which is tailored to the specific requirements of the target platform. The file must be under the porting folder, as shown in Figure 16-1.

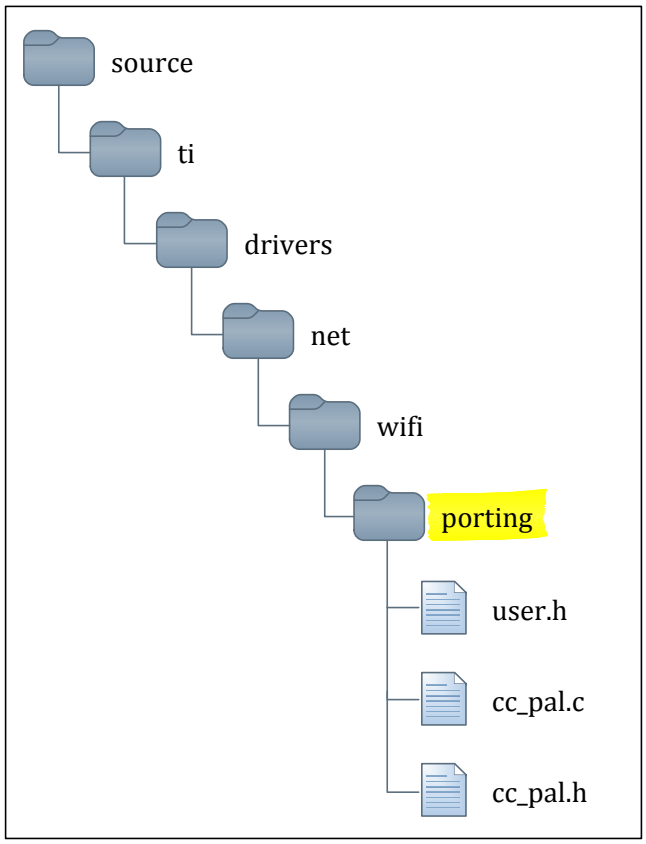

Figure 16-1. User.h Location

As a basis for this file, TI recommends using one of the porting layers provided with the SimpleLink Wi-Fi SDKs and plug-ins.

16.3 Select Capabilities Set

The SimpleLink host driver supports different predefined sets of capabilities that can fit most target platforms:

- SL_TINY – Provides limited functionality set, compatible for platforms with very limited resources.
- SL_FULL – Provides access to all SimpleLink functionality

TI recommends trying and choosing one of these predefined capabilities set before building a customized set. To choose one of these sets, the set name must be defined in user.h (only one of them). For example:

```
#define SL_TINY
```

If any of these predefined sets do not provide the required functionality, it is possible to tailor the driver in a more flexible way. This is done by enabling or disabling API groups (silos) and setting the APIs level. The levels of the APIs are divided into two categories: normal and extended. Table 16-1 describes the available groups and their relative macros.

Table 16-1. : Selecting Capabilities

GroupName and Macro	Normal APIs Level (#undef SL_INC_EXT_API)	Extended APIs Level (#define SL_INC_EXT_API)
Default	sl_Start sl_Stop sl_StatusGet sl_Task	sl_Start sl_Stop sl_StatusGet sl_Task sl_DeviceGet sl_DeviceSet sl_DeviceEventMaskGet sl_DeviceEventMaskSet
SL_INC_WLAN_PKG	sl_WlanSet sl_WlanSetMode sl_WlanProvisioning	sl_WlanSet sl_WlanSetMode sl_WlanProvisioning sl_WlanConnect sl_WlanDisconnect sl_WlanProfileAdd sl_WlanProfileGet sl_WlanProfileDel sl_WlanPolicySet sl_WlanPolicyGet sl_WlanGetNetworkList sl_WlanRxFilterAdd sl_WlanRxStatStart sl_WlanRxStatStop sl_WlanRxStatGet
SL_INC_SOCKET_PKG	sl_Socket sl_Close sl_Bind sl_Connect sl_Select sl_SetSockOpt sl_Recv sl_RecvFrom sl_Send sl_SendTo sl_Htonl sl_Htons	sl_Socket sl_Close sl_Bind sl_Connect sl_Select sl_SetSockOpt sl_Recv sl_RecvFrom sl_Send sl_SendTo sl_Htonl sl_Htons sl_Accept sl_Listen sl_GetSockOpt
SL_INC_NET_APP_PKG	sl_NetAppDnsGetHostByName sl_NetAppStart sl_NetAppStop	sl_NetAppDnsGetHostByName sl_NetAppStart sl_NetAppStop sl_NetAppSet sl_NetAppGet sl_NetAppRecv sl_NetAppSend sl_NetAppDnsGetHostByService sl_NetAppMDNSRegisterService sl_NetAppMDNSUnRegisterService sl_NetAppGetServiceList sl_NetAppPing

Table 16-1. : Selecting Capabilities (continued)

GroupName and Macro	Normal APIs Level (#undef SL_INC_EXT_API)	Extended APIs Level (#define SL_INC_EXT_API)
SL_INC_NET_CFG_PKG	sl_NetCfgGet sl_NetCfgSet	sl_NetCfgGet sl_NetCfgSet sl_MacAdrrGet sl_MacAdrrSet
SL_INC_NET_UTIL_PKG	sl_NetUtilGet sl_NetUtilSet sl_NetUtilCmd	sl_NetUtilGet sl_NetUtilSet sl_NetUtilCmd
SL_INC_NVMEM_PKG	sl_FsOpen sl_FsClose sl_FsRead sl_FsWrite sl_FsDel	sl_FsOpen sl_FsClose sl_FsRead sl_FsWrite sl_FsDel sl_FsGetInfo sl_FsCtl sl_FsProgram sl_FsGetFileList

NOTE: There is no option to enable or disable a particular function.

16.4 Bind the Device Enable/Disable Line

The CC3120 has two external hardware lines that can be used to enable or disable the device:

- nReset – Puts the device in shutdown mode
- nHib – Puts the device in hibernate mode

For more information, see Section 2.3.

NOTE: Only one of these lines or modes can be used. During sl_Start or sl_Stop, the driver calls the macros to force one of these lines to high or low in the correct sequence.

To bind one of these lines, the following macros must be defined correctly:

- sl_DeviceEnable – To force the line to high level
- sl_DeviceDisable – To force the line to low level

Example:

```
#define sl_DeviceEnable()   (P4OUT |= BIT1)

#define sl_DeviceDisable()  (P4OUT &= ~BIT1)
```

If some initializations are required before the enable or disable macros are called, the host application can also define the following optional macro:

sl_DeviceEnablePreamble

This macro is called during sl_Start before sl_DeviceEnable is called. The macro can be used as a placeholder to implement any preprocess operations before enabling networking operations.

16.5 Implement the Interface Communication Abstract Layer

The SimpleLink Wi-Fi CC3120 device supports two standard communication interfaces: SPI and UART.

The device automatically detects the active interface during initialization. From the device perspective, after the detection, the second interface is closed and cannot be used. The host driver uses a unified interface for both communication interfaces (abstract layer). The following functions should be implemented:

- sl_IfOpen – Opens the interface communication port to be used for communicating with the SimpleLink Wi-Fi device. Prototype:

```
_SlFd_t sl_IfOpen(char* pIfName , unsigned long flags);
```

- sl_IfClose – Closes an opened interface communication port. Prototype:

```
int sl_IfClose(_SlFd Fd);
```

- sl_IfRead – Reads bytes from an opened communication channel into a buffer. Prototype:

```
int sl_IfRead(_SlFd Fd , char* pBuff , int Len);
```

- sl_IfWrite – Transmits buffer of bytes on an opened communication channel. Prototype:

```
int sl_IfWrite(_SlFd Fd , char* pBuff , int Len);
```

- sl_IfRegIntHdlr – Registers an interrupt handler routine for the host IRQ line. Prototype:

```
sl_IfRegIntHdlr(InterruptHdl , pValue);
```

The way these functions are implemented has a direct impact on the performances of the SimpleLink Wi-Fi device. Consider using DMA or jitter buffer, if possible.

The function sl_IfOpen returns a file descriptor used later by sl_IfClose, sl_IfRead, and sl_IfWrite. The host application can define the type of this descriptor to any type required by defining _SlFd_t as a macro or typedef in user.h.

Example:

```
typedef _u32 _SlFd_t;
```

The sl_IfOpen function opens and configures the interface communication port using given interface name and option flags. The interface name is a parameter of the sl_Start function that passes as is to the sl_IfOpen function. The value of the option flags is set to constant value by defining the macro _SlIfOpenFlags in user.h.

The baud rate, clock polarity, clock phase, chip select, flow control, or any other specific attributes of the communication channel must be configured in this function. If the interface name and option flags are not enough for this configuration and the communication channel could not be entirely configured on this function, the host application alternatively can leave the sl_IfOpen function empty, and open and configure the communication channel externally before calling sl_Start. In this method, the host application should provide to sl_Start the file descriptor of the opened channel. The sl_IfClose function is always called on sl_Stop, even if the host application opened the communication channel externally.

For most of the platform, implementing the five macros above for the interface communication is sufficient.

By default, the host driver is running in a zero-copy method. This method is good for most cases, but essential for microcontrollers with tight availability of resources. However, it means that some commands or messages are sent in several transactions. In some platforms, it can be more efficient, in terms of performances, to copy the data to a temporary buffer and send it all at once. The driver allows this method by implementing additional two macros in user.h:

- sl_IfStartWriteSequence – Indicates that a write sequence is starting. From this point, the host application can store all the data from sl_IfWrite in a buffer. Prototype:

```
int sl_IfStartWriteSequence (_SlFd Fd);
```

- sl_IfEndWriteSequence – Indicates that a write sequence completed. At this point, the host should send the temporary buffer. Prototype:

```
int sl_IfEndWriteSequence (_SlFd Fd);
```

In some platforms, the host application might need to mask the IRQ line whenever this interrupt could be masked. The host driver provides a method to implement such schema. To allow this functionality, the user can define and implement the following macros:

- sl_IfMaskIntHdlr
- sl_IfUnMaskIntHdlr

16.6 Choose Memory-Management Model

The SimpleLink host driver supports two memory models: static (default) and dynamic.

The major difference between these memory models is that the static model requires the memory allocation of the driver's control block, even when the driver is not active, and the dynamic does not. In the dynamic model, the control block and all required resources are allocated on sl_Start and freed on sl_Stop.

To enable the dynamic model, the macro SL_MEMORY_MGMT_DYNAMIC must be defined. For example:

```
#define SL_MEMORY_MGMT_DYNAMIC
```

And a complementary malloc and free functions must also be defined:

- sl_Malloc – Allocates a buffer of at least the given size and returns a pointer to this buffer. Prototype:

```
void* sl_Malloc(int Size);
```

- sl_Free – Frees a given buffer by a pointer. Prototype:

```
void  sl_Free(void* pBuff);
```

NOTE: TI recommends using the static memory management model.

16.7 Implement OS Adaptation Layer

The SimpleLink Wi-Fi host driver can run on multithreaded environment (OS), as well as a non-OS environment. This step is not required if the host application is based on a non-OS environment.

To enable the multithreaded environment, the macro SL_PLATFORM_MULTI_THREADED must be defined. For example:

```
#define SL_PLATFORM_MULTI_THREADED
```

The OS adaptation layer consists of two major objects:

- Sync objects – To allow synchronization between threads
- Locking objects – To protect access to resources from different threads

16.7.1 Sync Objects

A sync object is an object used to synchronize between two threads, or between a thread and an interrupt handler. One thread is waiting on the object and the other thread or interrupt handler sends a signal, which then releases the waiting thread. The signal can be sent from interrupt context. This object is generally implemented by binary semaphore.

The type of the sync object is defined by the host application as needed, by defining the _SlSyncObj_t function as a typedef or a macro.

```
#define _SlSyncObj_t            HANDLE
```

The following functions should also be implemented:

- sl_SyncObjCreate – Creates a sync object. The function receives a pointer to a memory control block for the object, which is later passed to the other functions of the sync object.
- sl_SyncObjDelete – Destroys a sync object. If one of the threads already waits on the sync object while this function is called, the driver expects that the waiting thread will exit with an error when this function is called.
- sl_SyncObjSignal – Generates a synchronization signal to the sync object from a thread context, which should release the other thread context that is waiting on this sync object.
- sl_SyncObjSignalFromIRQ – The same as sl_SyncObjSignal, but called from interrupt handler routine. In most operating systems, there is no difference between these functions, but in some operating systems there is a special function for this function.
- sl_SyncObjWait – Waits for a synchronization signal of a specific sync object. The calling thread is blocked on this function until the signal is generated or time-out value elapsed. If the function is called

after the signal is already generated, the waiting thread should be released immediately.

16.7.2 Locking Objects

Locking objects are used to protect resources from mutual accesses of two or more threads. A locking object should support reentrant locks by a single thread. This object is generally implemented by a mutex semaphore.

The type of the locking object could be defined by the host application as needed, by defining the _SlLockObj_t function as a typedef or a macro. For example:

```
#define _SlLockObj_t            HANDLE
```

The following functions should also be implemented:

- sl_LockObjCreate – Creates a locking object. The function receives a pointer to a memory control block for the object, which is later passed to the other functions of the locking object.
- sl_LockObjDelete – Destroys a locking object.
- sl_LockObjLock – Locks a locking object. Other threads that try to lock the same object must be suspended until the locking thread unlocks this locking object.
- sl_LockObjUnlock – Unlocks a locking object to be used by other threads.

16.8 Implement Timestamp Services

The SimpleLink host driver supports a time-out mechanism for busy loops that the operating systems object do not support (for example, while waiting for a response from the device between a small SPI transactions). These time-outs require an implementation of timestamp mechanism.

NOTE: TI recommends implementing this mechanism.

To implement this mechanism, the host application must provide a function that retrieves the current timestamp:

- slcb_GetTimestamp – Gets counter value in ticks units

In addition, the host application must declare the time resolution of the timestamp on the platform by using the following macros:

- SL_TIMESTAMP_TICKS_IN_10_MILLISECONDS
- SL_TIMESTAMP_MAX_VALUE

The default time-out values are set to meet the common values of an average system. If the host application needs to, it can set a different time-out value by defining the following macros:

- SL_DRIVER_TIMEOUT_SHORT – In ms. By default, set to 30 seconds if this macro is not defined.
- SL_DRIVER_TIMEOUT_LONG – In ms. By default, set to 60 seconds if this macro is not defined.
- SYNC_PATTERN_TIMEOUT_IN_MSEC – In ms. By default, set to 60 seconds if this macro is not defined.

16.9 Set Asynchronous Event Handler Routines

The host application can register asynchronous event handler routines for the different API silos. TI recommends registering to all of these routines and handling the different events. Registering to these routines is optional, and might be changed from one host application implementation to another.

The following asynchronous event handlers can be registered:

- slcb_DeviceFatalErrorEvtHdlr – Handles fatal errors from the device or the host driver. After this routine is called, the host application must restart the driver and the device (call to sl_Stop and sl_Start) to continue using the device.
- slcb_DeviceGeneralEvtHdlr – Handles general errors from the device.
- slcb_WlanEvtHdlr – Handles events and errors of the WLAN silo
- slcb_SockEvtHdlr – Handles events and errors of the Socket silo.

- slcb_NetAppEvtHdlr – Handles events and errors of the NetApp silo.
- slcb_NetAppHttpServerHdlr – Handles events of the HTTP server.
- slcb_NetAppRequestHdlr – Handles NetApp requests.
- slcb_NetAppRequestMemFree – Frees a buffer used in a NetApp request. Allows the use of a dynamic memory buffer in these requests.

A.1 Host APIs

Table A-1 provides a brief description of the different host APIs.

Table A-1. Host APIs

API	Silo	Description
sl_Start	Device	Start the SimpleLink device by initializing the communication interface, setting the enable pin, allocating resources, and calling to the init complete callback if provided.
sl_Stop	Device	Stop the SimpleLink device by clearing the enable pin of the device, closing the communication, and releasing all resources allocated by the driver.
sl_Task	Device	The SimpleLink task entry function. This function must be called from the main loop in non-OS platform or otherwise from dedicated thread if the internal spawn is used.
sl_DeviceGet	Device	Retrieves device configurations and statuses.
sl_DeviceSet	Device	Sets device configurations and statuses.
sl_DeviceEventMaskGet	Device	Retrieves the current asynchronous events bit mask of the device.
sl_DeviceEventMaskSet	Device	Sets the asynchronous event bit mask of the device. Masked events do not generate asynchronous messages to the host. By default all events are active.
sl_DeviceUartSetMode	Device	Relevant for UART host interface only. Used to change the baud rate of the UART after the device was started.
sl_RegisterEventHandler	Device	This API enables registration of the SimpleLink host driver in runtime
sl_WlanConnect	Wlan	Initiates a connection to Wi-Fi network.
sl_WlanDisconnect	Wlan	Initiates a disconnection from the current connected Wi-Fi network. If the Auto connection policy is active, a new connection is initiated immediately.
sl_WlanProfileAdd	Wlan	Adds a preferred network profile.
sl_WlanProfileGet	Wlan	Retrieves the nonconfidential data of existing preferred network profile.
sl_WlanProfileDel	Wlan	Deletes a preferred network profile.
sl_WlanSet	Wlan	Sets Wlan configurations.
sl_WlanGet	Wlan	Retrieves Wlan configurations
sl_WlanPolicySet	Wlan	Sets Wlan policy configurations
sl_WlanPolicyGet	Wlan	Retrieves Wlan policy configurations
sl_WlanGetNetworkList	Wlan	Gets the last Wlan scan results
sl_WlanRxStatStart	Wlan	Starts collecting wlan RX statistics
sl_WlanRxStatStop	Wlan	Stops collecting wlan RX statistics
sl_WlanRxStatGet	Wlan	Retrieves Wlan RX statistics. Upon calling this function, the statistics are cleared and collected from beginning.
sl_WlanSetMode	Wlan	Sets the Wlan mode
sl_WlanProvisioning	Wlan	Starts the provisioning process
sl_WlanRxFilterAdd	Wlan	Adds a new receive filter rule to the system
sl_Socket	Socket	Creates an endpoint for communication
sl_Listen	Socket	Listens for connections on a socket

Table A-1. Host APIs (continued)

API	Silo	Description
sl_Accept	Socket	Accepts a connection on a socket
sl_Bind	Socket	Assigns an address to a socket
sl_Close	Socket	Closes an endpoint socket. If the socket is connected, it gracefully closes the socket.
sl_Connect	Socket	Initiates a connection on a socket
sl_Select	Socket	Monitors set of sockets activities
sl_Send	Socket	Writes a data buffer to a socket. Used especially in stream sockets.
sl_SendTo	Socket	Writes a data buffer to a socket. Used especially in datagram sockets.
sl_Recv	Socket	Reads a data buffer from a socket. Used especially in stream sockets.
sl_RecvFrom	Socket	Reads a data buffer from a socket. Used especially in datagram sockets.
sl_GetSockOpt	Socket	Retrieves a socket options
sl_SetSockOpt	Socket	Sets a socket options
sl_NetAppStart	NetApp	Starts network applications (bitmask)
sl_NetAppStop	NetApp	Stops network applications (bitmask)
sl_NetAppDnsGetHostByName	NetApp	Retrieves the IP address of a host on the network
sl_NetAppDnsGetHostByService	NetApp	Retrieves service attributes like IP address, port and text according to service name
sl_NetAppGetServiceList	NetApp	Retrieves the cached services of the peer
sl_NetAppMDNSUnRegisterService	NetApp	Unregisters mDNS service
sl_NetAppMDNSRegisterService	NetApp	Registers a new mDNS service
sl_NetAppPingStart	NetApp	Sends ICMP ECHO_REQUEST to a host on the network
sl_NetAppSet	NetApp	Sets configuration for a network application
sl_NetAppGet	NetApp	Retrieves configuration for a network application
sl_NetCfgSet	NetCfg	Sets the network configuration of the device
sl_NetCfgGet	NetCfg	Retrieves the network configuration of the device
sl_NetUtilSet	NetUtil	Sets configurations of a network utility
sl_NetUtilGet	NetUtil	Retrieves configurations of a network utility
sl_NetUtilCmd	NetUtil	Activates a network utility-related command
sl_FsOpen	FS	Opens a file for read or write
sl_FsClose	FS	Closes a file
sl_FsRead	FS	Reads a block of data from a file
sl_FsWrite	FS	Writes a block of data to a file
sl_FsGetInfo	FS	Retrieves information of a file
sl_FsDel	FS	Deletes specific file from the file system
sl_FsCtl	FS	Controls various file system operations
sl_FsProgram	FS	Enables to format and configure the device with preprepared configuration
sl_FsGetFileList	FS	Retrieves the list of stored files and their basic attributes

B.1 Persistency

The SimpleLink device supports a few different persistency types for settings and configurations:

- Nonpersistent: Effective immediately, but returned to default after reset
- System-persistent: Effective immediately, and kept after reset according to system-persistent mode
- Persistent: Effective immediately, and kept after reset, regardless the system-persistent mode
- Optionally persistent: Effective immediately, and kept after reset, according to a parameter in the API call
- Reset: Persistent, but effective only after reset

Table B-1 lists the different configurations and settings of the device, and their persistency type.

Table B-1. Persistency Settings

Functionality	API	Type	Comments
Set time and date	sl_DeviceSet	Nonpersistent*	Kept during hibernate. Setting operation include write to the file system.
Set system-persistent configuration	sl_DeviceSet	Persistent	
Set Events mask	sl_EventMaskSet	System-persistent	
Set UART baud rate	sl_UartSetMode	Nonpersistent	
Start NetApp Applications	sl_NetAppStart	System-persistent	Setting effective to current Wi-Fi mode
Stop NetApp Applications	sl_NetAppStop	System-persistent	Setting effective to current Wi-Fi mode
Set Http port number	sl_NetAppSet	System-persistent	
Enable/Disable Http authentication check	sl_NetAppSet	System-persistent	
Set Http authentication name	sl_NetAppSet	System-persistent	
Set Http authentication password	sl_NetAppSet	System-persistent	
Set Http authentication realm	sl_NetAppSet	System-persistent	
Enableor Disable Http ROM pages access	sl_NetAppSet	System-persistent	
Set secondary port number	sl_NetAppSet	System-persistent	
Enable or Disable of secondary port	sl_NetAppSet	System-persistent	
Enableor Disable security on the primary port	sl_NetAppSet	System-persistent	
Set private key file name	sl_NetAppSet	System-persistent	
Set device certificate file name	sl_NetAppSet	System-persistent	
Set CA certificate file name	sl_NetAppSet	System-persistent	
Register mDNS service	sl_NetAppMDNSRegisterService	Optionally persistent	
Unregister mDNS service	sl_NetAppMDNSUnRegisterService	Optionally persistent	
Set http temporary mDNS service name	sl_NetAppSet	Nonpersistent	

Table B-1. Persistency Settings (continued)

Functionality	API	Type	Comments
Unset http temporary mDNS service name	sl_NetAppSet	Nonpersistent	
Set DHCP server parameters	sl_NetAppSet	Reset	
Set mDNS continues query	sl_NetAppSet	System-persistent	
Set mDNS event mask	sl_NetAppSet	System-persistent	
Set mDNS timing parameters	sl_NetAppSet	System-persistent	
Set Device URN	sl_NetAppSet	System-persistent	MDNS restarts internally
Set Domain Name and SNI	sl_NetAppSet	Reset	
Enable IPv4 STA\Wi-Fi Direct client DHCP	sl_NetCfgSet	Reset	
Enable and set IPv4 STA\Wi-Fi Direct client static	sl_NetCfgSet	Reset	
Enable IPv4 STA\Wi-Fi Direct client DHCP release address before disconnect	sl_NetCfgSet	System-persistent	
Enable and set IPv4 AP\Wi-Fi Direct GO static	sl_NetCfgSet	Reset	
Enable or Disable IPv6 interface (local\local+global)	sl_NetCfgSet	System-persistent	
Set IPv6 local or global state	sl_NetCfgSet	System-persistent	
Set IPv6 static\stateless\statefull	sl_NetCfgSet	System-persistent	
Enable and set IPv4 STA\W-iFi Direct client static	sl_NetCfgSet	Reset	
Set MAC address	sl_NetCfgSet	Reset	
Disconnet AP station by MAC address	sl_NetCfgSet	Nonpersistent	
Enable or Disable periodic keep-alive	sl_SetSockOpt	Nonpersistent	
Set receive time-out value	sl_SetSockOpt	Nonpersistent	
Sets tcp max recv window size	sl_SetSockOpt	Nonpersistent	
Sets socket to nonblocking operation	sl_SetSockOpt	Nonpersistent	
Sets method to tcp secured socket	sl_SetSockOpt	Nonpersistent	
Sets specific cipher to tcp secured socket	sl_SetSockOpt	Nonpersistent	
Map secured socket to CA file by name	sl_SetSockOpt	Nonpersistent	
Map secured socket to private key by name	sl_SetSockOpt	Nonpersistent	
Map secured socket to certificate file by name	sl_SetSockOpt	Nonpersistent	
Map secured socket to Diffie Hellman file by name	sl_SetSockOpt	Nonpersistent	
Sets channel in transceiver mode	sl_SetSockOpt	Nonpersistent	
Set socket TTL value of outgoing multicast packets	sl_SetSockOpt	Nonpersistent	
UDP socket; join IPv4 multicast group	sl_SetSockOpt	Nonpersistent	
UDP socket; leave IPv4 multicast group	sl_SetSockOpt	Nonpersistent	
RAW socket; remove IP header from received data	sl_SetSockOpt	Nonpersistent	

Table B-1. Persistency Settings (continued)

Functionality	API	Type	Comments
RAW socket; packet include the IP header	sl_SetSockOpt	Nonpersistent	
RAW socket; packet include the ipv6 header	sl_SetSockOpt	Nonpersistent	
RAW socket; set WLAN PHY transmit rate	sl_SetSockOpt	Nonpersistent	
RAW socket; set WLAN PHY TX power	sl_SetSockOpt	Nonpersistent	
RAW socket; set number of frames to transmit in transceiver mode	sl_SetSockOpt	Nonpersistent	
RAW socket; set WLAN PHY preamble for Long/Short	sl_SetSockOpt	Nonpersistent	
Set CCA threshold	sl_SetSockOpt	Nonpersistent	
Set TX frame time-out	sl_SetSockOpt	Nonpersistent	
Enable ACK in transceiver mode	sl_SetSockOpt	Nonpersistent	
Start secured socket	sl_SetSockOpt	Nonpersistent	
Set keepalive time	sl_SetSockOpt	Nonpersistent	
Set IPV6 Hops time-out	sl_SetSockOpt	Nonpersistent	
Join ipv6 multicast group	sl_SetSockOpt	Nonpersistent	
Leave ipv6 multicast group	sl_SetSockOpt	Nonpersistent	
Add profile	sl_WlanProfileAdd	Persistent	
Delete profile	sl_WlanProfileDel	Persistent	
Set connection policy	sl_WlanPolicySet	System-persistent	
Set system scan time interval and start scan	sl_WlanPolicySet	Nonpersistent	
Set PM policy for STA mode only	sl_WlanPolicySet	System-persistent	
Set negotiation policy parameters for P2P role	sl_WlanPolicySet	System-persistent	
Set WLAN mode	sl_WlanSetMode	Reset	
Set SSID for AP mode	sl_WlanSet	Reset	
Set channel for AP mode	sl_WlanSet	Reset	
Set hidden SSID mode for AP mode	sl_WlanSet	Reset	
Set security type for AP mode	sl_WlanSet	Reset	
Set password for for AP mode	sl_WlanSet	Reset	
Set scan parameters	sl_WlanSet	System-persistent	
Set Country Code for AP mode	sl_WlanSet	System-persistent	
Set STA mode Tx power level	sl_WlanSet	System-persistent	
Set AP mode Tx power level	sl_WlanSet	System-persistent	
Set AP mode info element	sl_WlanSet	System-persistent	
Set smart config key	sl_WlanSet	Persistent	
Set P2P device type	sl_WlanSet	System-persistent	
Set P2P channels	sl_WlanSet	System-persistent	
Add new filter rule to the system	sl_WlanRxFilterAdd	Optionally Persistent	
Enable or disable filter in a filter list	sl_WlanSet	Optionally Persistent	
Remove filter from memory	sl_WlanSet	Optionally Persistent	
Save the filters for persistent	sl_WlanSet	Persistent	

Table B-1. Persistency Settings (continued)

Functionality	API	Type	Comments
Update the arguments of existing filter	sl_WlanSet	Optionally Persistent	
Change the default creation of the pre-prepared filters	sl_WlanSet	Optionally Persistent	
Set maximum supported stations	sl_WlanSet	Persistent	
Set AP access list mode	sl_WlanSet	Persistent	
Add station to black list by mac address	sl_WlanSet	Persistent	
Remove station from black list by mac address	sl_WlanSet	Persistent	
Remove station from black list by index	sl_WlanSet	Persistent	

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from D Revision (December 2017) to E Revision	Page
• Updated Calibrations section.	67
• Updated Calibration Modes table.	68
• Updated code in Create a File versus Open for Write section.	112
• Updated code in Open a File for Read section.	118
• Updated code in CC3220 Bundle Aspects section.	124
• Added Note.	164
• Updated Selecting Capabilities table.	234
• Updated Persistency Settings table.	244

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ('TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265