CC3100\CC3200 SimpleLink™ Wi-Fi® Network Processor Subsystem

Programmer's Guide

TEXAS INSTRUMENTS

Literature Number: SWRU368
June 2014

Contents

List of Figures

List of Tables

Overview

The SimpleLink™ Wi-Fi CC3100 and CC3200 are the next generation in Embedded Wi-Fi. The CC3100 Internet-on-a-chip™ can add Wi-Fi and Internet to any microcontroller (MCU), such as TI's ultra-low power MSP430$_{TM}$. The CC3200 is a programmable Wi-Fi MCU that enables true, integrated IoT development. The Wi-Fi Network Processor sub-system in both SimpleLink Wi-Fi devices integrates all protocols for Wi-Fi and Internet, greatly minimizing MCU software requirements. With built-in security protocols, SimpleLink Wi-Fi provides a robust and simple security experience.

The SimpleLink Host driver minimizes the host memory footprint requirements, requiring less than 7KB of flash and 700B of RAM memory for a TCP client application. The driver follows strict ANSI C (C89) coding standards, uses industry-standard BSD Sockets and simple APIs reducing integration and development time for software and application developers. The driver is compatible and portable across different MCUs, compilers, operating systems, communication interfaces and use cases.

The architecture of the SimpleLink Host Driver includes a set of six logical and simple API modules:

- **Device API** – Provided to manage hardware-related functionality such as start, stop, set and read device configurations.
- **WLAN API** – Designed to manage WLAN, 802.11 protocol-related functionality such as device mode (station, AP or P2P), setting provisioning method, adding connection profiles and setting connection policy.
- **Socket API** – The most common API set for user applications, and complies with Berkeley socket APIs.
- **NetApp API** – Designed to enable different networking services including HTTP server service, DHCP server service and MDNS Client\Server service.
- **NetCfg API** – Provided for configuring different networking parameters, such as setting the MAC address, acquiring the IP address by DHCP, and setting the static IP address.
- **File System API** – Designed to provide access to the serial flash component, for read and write operations of networking or user proprietary data.

Figure 1-1 shows the host driver anatomy.

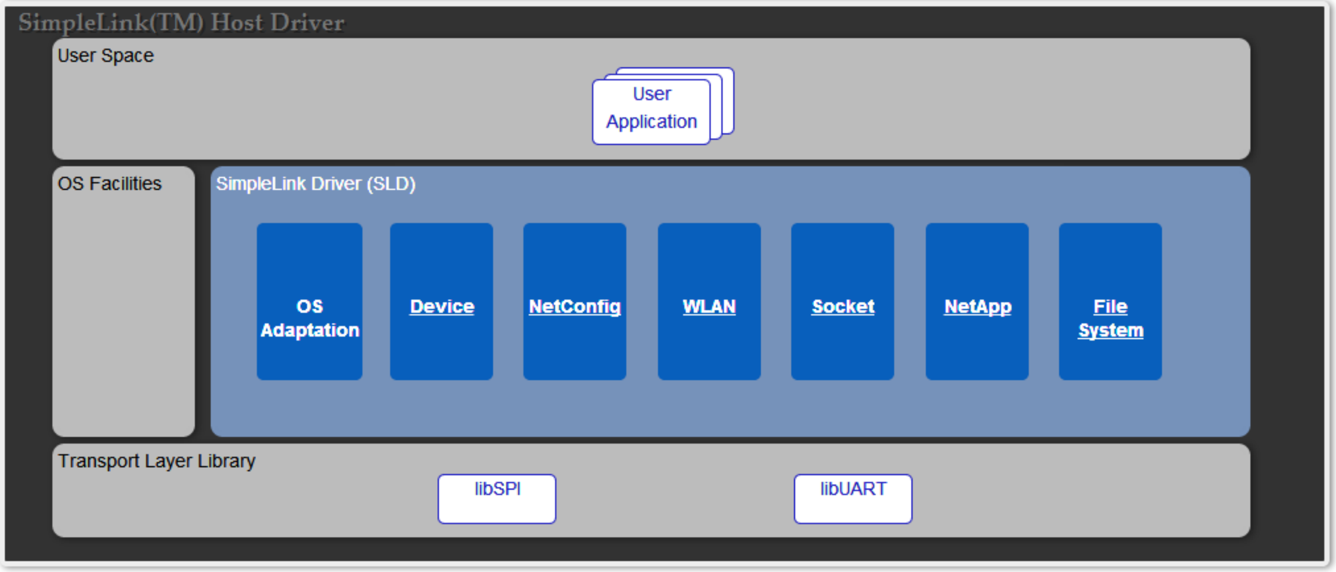


Figure 1-1. Host Driver Anatomy

1.1 Document Scope

The purpose of this document is to provide software programmers who are working with the Wi-Fi subsystem the knowledge of the networking capabilities, and how to use these capabilities through the host driver. This document includes an overview on how to write a networking application, a detailed description of the networking operation modes and features of the device, and a review of each API of the driver, accompanied by source code examples for each topic.

1.2 Host Driver SW Concepts

Before starting to work with the Wi-Fi subsystem host driver, it is important to understand the main architectural concepts:

The host driver supports any standard C compiler (C89):

- The host driver is written in strict ANSI C (C89).
- The host driver does not use pragma or any other extended compiler attributes.

The host driver communicates with the device using messages called *commands*:

- The Wi-Fi subsystem supports the handling of a single command at a given time.
- The Wi-Fi subsystem will send a “Command Complete” message to signal a successful command reception for each command.

The host driver supports asynchronous event handling:

- Some networking commands might take a long time to process (for example, a WLAN connection command). Because of this, the Wi-Fi subsystem uses asynchronous events to signal the host driver of certain status changes.
- In cases of “long” commands, the host driver will get an immediate Command Complete response, followed by an asynchronous event later on, to signal that the process has finished and to return the process results.

The microcontroller of the driver:

- Can run on 8-bit, 16-bit, or 32-bit.
- Can run on any clock speed – No performance or time dependency.
- Supports both big and little endian formats.
- Small memory footprint – Configurable at the time of compiling, the driver requires as low as 7KB of code memory and 700 B of RAM memory.

The standard interface communication port of the driver:

- SPI – Supports standard 4-wire serial peripheral interface:
  - 8-, 16-, or 32-bit word length
  - Default mode 0 (CPOL=0, CPHA=0)
  - SPI clock can be configured up to 20 Mbps.
  - CS is required.
  - Additional IRQ line is required for async operations.
- UART
  - Standard UART with hardware flow control (RTS/CTS) up to 3 Mbps.
  - The default baud rate is 115200 (8 bits, no parity, 1 start/stop bit).

The driver supports systems using or not using OS:

- Simple OS wrapper, requiring only two object wrappers
- Sync Obj (event/binary semaphore)
- Lock Obj (mutex/binary semaphore)
- Built-in logic within the driver for system not running OS

1.3 Common Terminology and References

Table 1-1. Common Terminology and Abbreviations

Abbreviation	Meaning
Host	Host refers to an embedded controller running the SimpleLink driver and using the SimpleLink device as a networking peripheral.

Applicable documents:

- CC3100 data sheet (SWAS031)
- CC3200 data sheet (SWAS031)
- CC31xx Host Driver APIs
- CC31xx Host Interface

Additional Resources:

- www.ti.com/simplelinkwifi.
- CC31xx SimpleLink wiki.

Chapter 2

SWRU368–June 2014

Writing a Simple Networking Application

Topic Page

2.1 Overview

This chapter explains the software blocks needed to build a networking application. In addition, this chapter describes the recommended flow for most applications. The information provided is for guidance only. Programmers have complete flexibility on how to use the various software blocks.

Programs using the SimpleLink device consist of the following software blocks:

- **Wi-Fi subsystem initialization** – Wakes the Wi-Fi subsystem from the hibernate state.
- **Configuration** – Primarily one-time configurations such as cold boot configuration, or infrequently used device configurations. For example, changing the Wi-Fi subsystem from a WLAN STA to WLAN soft AP or WLAN P2P device, or changing the MAC address. After the configuration phase, reboot the Wi-Fi subsystem for the new configurations to take effect.
- **WLAN connection** – The established physical interface is wireless LAN communication (for example, manually connecting to an AP as a wireless station).
  - **DHCP** – An IP address must be received before working with TCP/UDP sockets.
- **Socket connection** – Sets up the TCP/IP layer. This occurs in the following steps:
  - **Creating the socket** – Select either TCP, UDP, or RAW sockets. Select whether the device will be a client or a server socket. Define socket characteristics such as blocking/nonblocking and socket time-outs.
  - **Querying for the server IP address** – When implementing a client-side communication, usually the remote server side IP address required for establishing the socket connection is not known. Use DNS protocol to query the server IP address by using the server name.
  - **Creating socket connection** – TCP socket requires the establishment of a proper socket connection before continuing to perform data transactions.
- **Data transactions** – Once a socket connection is established, data can be transmitted between the client and the server by implementing the application logic.
- **Socket disconnection** – After finishing the required data transactions, the socket communication channel is closed.
- **Wi-Fi subsystem hibernate** – When the Wi-Fi subsystem is inactive for a long period of time, it goes into hibernate state.

2.1.1 Basic Example Code

When implementing a networking application, consider the different application blocks, the host driver software concepts described above and system aspects such as hardware and operating system.

Figure 2-1 shows a state machine diagram that describes the basic software design.

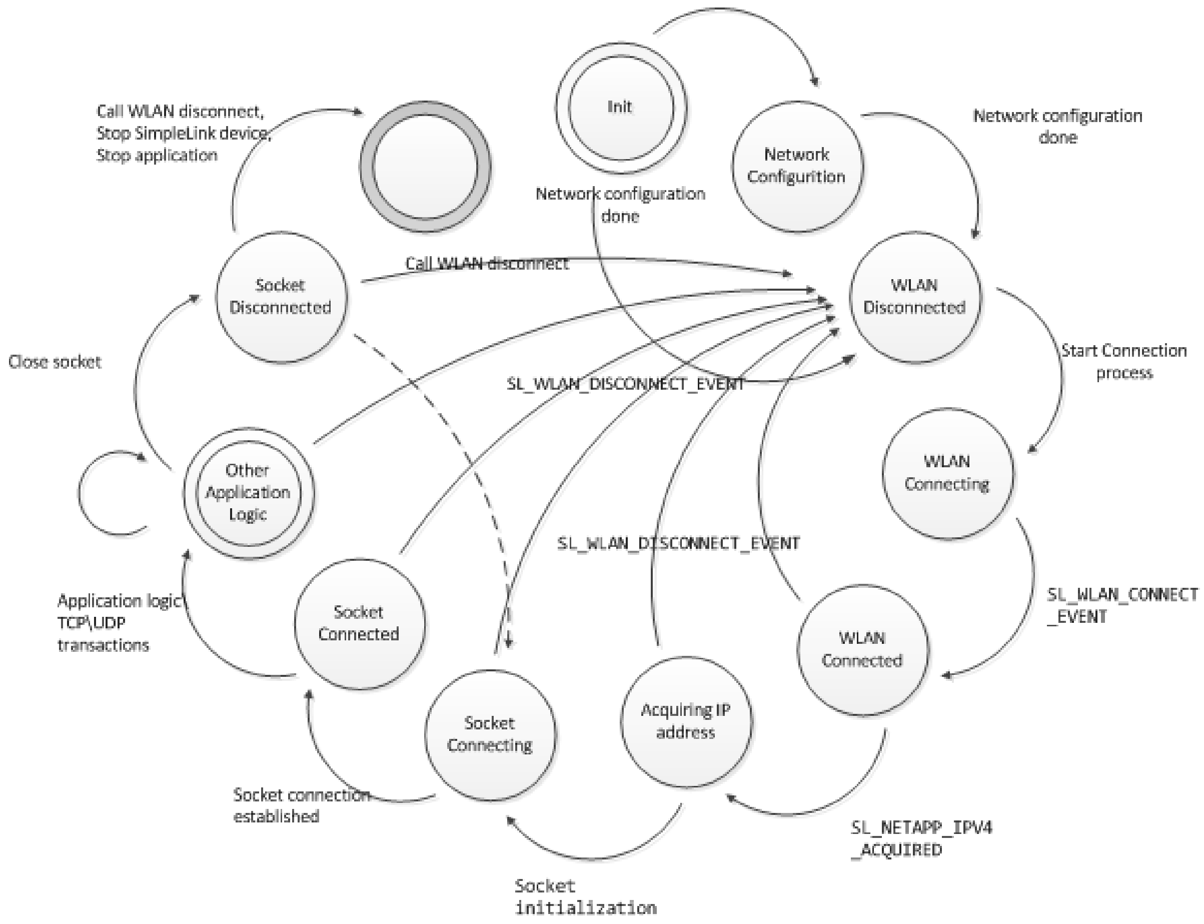


Figure 2-1. Basic Networking Application State Machine

Figure 2-1 shows the different states described in this chapter, the host driver events which trigger the code to move between different states, and basic error-handling events.

An example of the state machine is implemented in the following code:

- **Init state** – Example of initializing the Wi-Fi subsystem as a WLAN station:

```
case INIT:
    status = sl_Start(0, 0, 0);
    if (status == ROLE_STA)
    {
        g_State = CONFIG;
    }
    else
    {
        g_State = SIMPLELINK_ERR;
    }
    break;
```

- **WLAN connection** – Example of WLAN and network event handlers, demonstrating the WLAN connection, waiting for a successful connection and acquiring an IP address:

```
/* SimpleLink WLAN event handler */
void SimpleLinkWlanEventHandler(void *pWlanEvents)
{
    SlWlanEvent_t *pWlan = (SlWlanEvent_t *)pWlanEvents;

    switch(pWlan->Event)
    {
        case SL_WLAN_CONNECT_EVENT:
        g_Event |= EVENT_CONNECTED;
        memcpy(g_AP_Name, pWlan->EventData.STAandP2PModeWlanConnected.ssid_name, pWlan-
>EventData.STAandP2PModeWlanConnected.ssid_len);
        break;

        case SL_WLAN_DISCONNECT_EVENT:
        g_DisconnectionCntr++;
           g_Event |= EVENT_DISCONNECTED;
        g_DisconnectionReason = pWlan->EventData.STAandP2PModeDisconnected.reason_code;
            memcpy(g_AP_Name, pWlan->EventData.STAandP2PModeWlanConnected.ssid_name, pWlan-
>EventData.STAandP2PModeWlanConnected.ssid_len);
            break;

        default:
        break;
    }
}
/* SimpleLink Networking event handler */
void SimpleLinkNetAppEventHandler(void *pNetAppEvent)
{
    SlNetAppEvent_t *pNetApp = (SlNetAppEvent_t *)pNetAppEvent;

    switch( pNetApp->Event )
    {
        case SL_NETAPP_IPV4_ACQUIRED:
            g_Event |= EVENT_IP_ACQUIRED;
        g_Station_Ip = pNetApp->EventData.ipAcquiredV4.ip;
        g_GW_Ip =  pNetApp->EventData.ipAcquiredV4.gateway;
        g_DNS_Ip = pNetApp->EventData.ipAcquiredV4.dns;
        break;

        default:
        break;
    }
}
.
.
.
/* initiating the WLAN connection */
case WLAN_CONNECTION:
    status = sl_WlanConnect(User.SSID,strlen(User.SSID),0,
&amp;secParams, 0);
    if (status == 0)
    {
        g_State = WLAN_CONNECTING;
    }
    else
    {
        g_State = SIMPLELINK_ERR;
    }
/* waiting for SL_WLAN_CONNECT_EVENT to notify on a successful connection */
case WLAN_CONNECTING:
    if (g_Event
&amp;EVENT_CONNECTED)
    {
        printf("Connected to %s\n", g_AP_Name);
        g_State = WLAN_CONNECTED;
    }
    break;

/* waiting for SL_NETAPP_IPV4_ACQUIRED to notify on a receiving an IP address */
case WLAN_CONNECTED:
    if (g_Event
&amp;EVENT_IP_ACQUIRED)
```

```
    {
        printf("Received IP address:%d.%d.%d.%d\n",
(g_Station_Ip>>24)&amp;0xFF,(g_Station_Ip>>16)&amp;0xFF,(g_Station_Ip>>8)&amp;0xFF,(g_Station_Ip&a
mp;0xFF));
        g_State = GET_SERVER_ADDR;
    }
    break;
```

- **Socket connection** – Example of querying for the remote server IP address by using the server name, creating a TCP socket, and connecting to the remote server socket:

```
case GET_SERVER_ADDR:
     status = sl_NetAppDnsGetHostByName(appData.HostName,
                                    strlen(appData.HostName),

&amp;appData.DestinationIP, SL_AF_INET);
    if (status == 0)
    {
        g_State = SOCKET_CONNECTION;
    }
    else
    {
        printf("Unable to reach Host\n");
        g_State = SIMPLELINK_ERR;
    }
    break;
case SOCKET_CONNECTION:
    Addr.sin_family = SL_AF_INET;
    Addr.sin_port = sl_Htons(80);

    /* Change the DestinationIP endianity, to big endian */
    Addr.sin_addr.s_addr = sl_Htonl(appData.DestinationIP);

    AddrSize = sizeof(SlSockAddrIn_t);
    SockId = sl_Socket(SL_AF_INET,SL_SOCK_STREAM, 0);
    if( SockId < 0 )
    {
        printf("Error creating socket\n\r");
        status = SockId;
        g_State = SIMPLELINK_ERR;
    }
    if (SockId >= 0)
    {
        status = sl_Connect(SockId, ( SlSockAddr_t *)
&amp;Addr, AddrSize);
        if( status >= 0 )
                    {
            g_State = SOCKET_CONNECTED;
                    }
        else
                    {
            printf("Error connecting to socket\n\r");
            g_State = SIMPLELINK_ERR;
                    }
    }
    break;
```

- **Data transactions** – Example of sending and receiving TCP data over the open socket:

```
case SOCKET_CONNECTED:
/* Send data to the remote server */
sl_Send(appData.SockID, appData.SendBuff, strlen(appData.SendBuff), 0);

        /* Receive data from the remote server */
        sl_Recv(appData.SockID,
&amp;appData.Recvbuff[0], MAX_RECV_BUFF_SIZE, 0);

break;
```

- **Socket disconnection** – Example of closing a socket:

```
case SOCKET_DISCONNECT:
    sl_Close(appData.SockID);
    /* Reopening the socket */
    g_State = SOCKET_CONNECTION;
    break;
```

- **Device hibernate** – Example of putting the Wi-Fi subsystem into hibernate state:

```
case SIMPLELINK_HIBERNATE:
    sl_Stop();
    g_State = …
    break;
```

Device Initialization

The Wi-Fi subsystem is enabled by calling the **sl_Start()** API. During the initialization, the host driver performs the following key steps:

- Enable the bus interface (in CC3200 – SPI; in CC3100 – SPI or UART).
- Register the asynchronous events handler.
- Enable the Wi-Fi subsystem (in CC3200 this is done by the internal applications microcontroller. In CC3100 it is done by the external host processor).
- Send a synchronization message to the Wi-Fi subsystem and wait for an IRQ in return signaling on completion of the initialization phase.

Figure 3-1 shows the basic initialization flow:

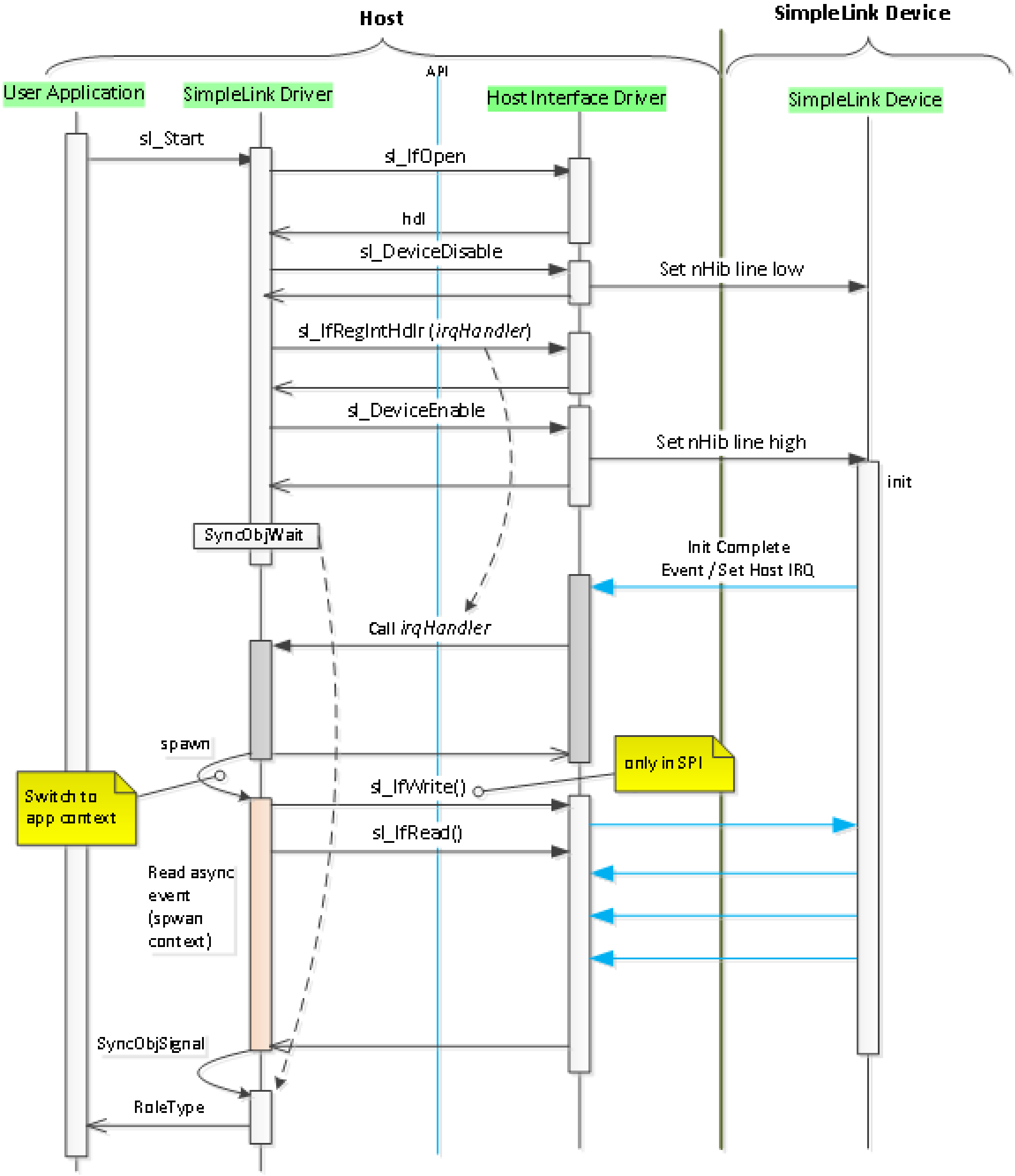


Figure 3-1. Basic Initialization Flow

The Wi-Fi subsystem initialization can take tens of mS to complete. The host driver supports two main options of using the **sl_Start**(const void* pIfHdl, char* pDevName, const P_INIT_CALLBACK pInitCallBack) API:

- **Blocking** – pInitCallBack must be set to NULL. The calling application is blocked until the entire initialization process completes (upon receiving the Init complete interrupt). See the following code example:

```
if( sl_Start(NULL, NULL, NULL) == 0)
{
          LOG("Error opening interface to device\n");
}
```

- **Asynchronous** – pInitCallBack is given a pointer to a function that is called when the initialization process completes. In this case, the call to **sl_Start()** will return immediately. See the following code example:

```
Void InitCallBack(UINT32 Status)
{
  Network_IF_SetMCUMachineState(MCU_SLHost_INIT);
}
.
.
Void Network_IF_InitDriver(void)
{
   ..
  sl_Start(NULL,NULL,InitCallBack);
  while(!(g_usMCUstate & MCU_SLHost_INIT));
  ..
}
```

Chapter 4

SWRU368–June 2014

Device Configurations

Topic Page

4.1 Overview

The Wi-Fi subsystem has several configurable parameters that control its behavior. The host driver uses different APIs to configure these parameters. The parameters are grouped based on their functionality.

Most of the parameters described in this chapter are stored in the serial flash (SFlash). If the parameter values are not set by the user, the Wi-Fi subsystem will use the default values. A value stored in the SFlash is always prioritized over the default value.

An application will usually need to configure its parameters when coming out of cold boot or when a specific configuration change is required.

All the parameters configured in the SFlash take effect only in the next device boot.

This chapter explains all the parameters that the user can configure. This chapter also explains the read-only parameters used for reading the device and networking status.

4.2 Device Parameters

Time and Date: Configures the device internal date and time. For more details, see Section 18.1. **Note**: This parameter is retained in hibernate state but is reset to default in shutdown.

Firmware Version: A read-only parameter that returns the Wi-Fi subsystem firmware version. For more details, see Section 18.1.

Device Status: A read-only parameter that returns status for the last events that recorded in the Wi-Fi subsystem. For more details, see Section 18.1.

Asynchronous events mask: Masked events will not generate asynchronous messages (IRQs) from the Wi-Fi subsystem. For more details, see **sl_EventMaskGet**, **sl_EventMaskSet** in the Section 18.1.

UART configuration: When using the UART interface, the application can set several UART parameters: Baud rate, Flow Control and COM port. For more details, see Section 18.1.

4.3 WLAN Parameters

Device Mode – The Wi-Fi subsystem can operate in several WLAN roles. The different options are:

- WLAN station
- WLAN AP
- WLAN P2P

For more details, see Section 18.2.

AP mode – If set to an Access-point role, the Wi-Fi subsystem has many configurations that can be set:

- **SSID** – AP name
- **Country code**
- **Beacon interval**
- **Operational channel**
- **Hidden SSID** – Enable or Disable
- **DTIM period**
- **Security type** – Possible options are:
  - Open security
  - WEP security
  - WPA security
- **Security password**:
  - For WPA: 8 to 63 characters
  - For WEP: 5 to 13 characters (ASCII)
- **WPS state**

For more details, see Section 8.1.3.

P2P – If set to Peer-to-Peer role, the Wi-Fi subsystem has many configurations that can be set:

- **Device Name**
- **Device type**
- **Operational channels** – Regulatory class determines the listen channel of the device during the P2P find listen phase. Operational channel and regulatory class determines the operating channel preferred by this device (if the device is the group owner, this is the operating channel). Channels should be one of the social channels (1/6/11). If no listen or operational channel is selected, a random 1/6/11 channel will be selected.
- **Information elements** – The application can be set to MAX_PRIVATE_INFO_ELEMENTS_SUPPORTED information elements per role (AP / P2P GO). To delete an information element, use the relevant index and length = 0. The application can be set to MAX_PRIVATE_INFO_ELEMENTS_SUPPORTED to the same role. However, for AP no more than INFO_ELEMENT_MAX_TOTAL_LENGTH_AP bytes can be stored for all information elements. For P2P GO, no more than INFO_ELEMENT_MAX_TOTAL_LENGTH_P2P_GO bytes can be stored for all information elements.
- **Scan channels** – Changes the scan channels and RSSI threshold.

For more details, see Chapter 11.

4.3.1 Advanced

Country code – Sets the Wi-Fi subsystem regulatory domain country code. Relevant for WLAN station and P2P client modes only. For more details, see Section 18.2.

Tx power – Sets the maximal transmit power of the network processor subsystem. For more details, see Section 18.2.

4.4 Network Parameters

MAC address – Sets the MAC address of the device. For more details, see Section 18.5.

IP address – Configures the Wi-Fi subsystem to use DHCP or static IP configuration. In case of static configuration, the user can set the IP address, DNS address, GW address and subnet mask. For more details, see Section 18.5.

4.5 Internet and Networking Services Parameters

HTTP Server: For more details, see Chapter 12.

DHCP Server: For more details, see Section 10.6.

mDNS: For more details, see Chapter 13.

SmartConfig: For more details, see Section 8.1.1.

4.6 Power-Management Parameters

Power management and energy preservation are among the most challenging issues for Wi-Fi systems. Handling power regimes effectively is fundamental for any power-aware solution, particularly in cases where a certain component of the overall solution requires more power than the rest of the system. Such is the case for many embedded Wi-Fi-capable systems.

4.6.1 Power Policy

From host application perspective, only two modes of operation are explicitly selected by the host: hibernate or enabled.

The Wi-Fi subsystem supports predefined power management policies which allow a host application to guide the behavior of power-management algorithm. **sl_PolicySet** API is used to configure the device power-management policy. The available policies are:

- **Normal** (Default) – Features the best tradeoff between traffic delivery time and power performance. For setting normal power-management policy use:

```
sl_WlanPolicySet(SL_POLICY_PM , SL_NORMAL_POLICY, NULL,0)
```

- **Always on** – The Wi-Fi subsystem is kept fully active at all times, providing the best WLAN traffic performance. This policy is user-directed, whereby the user may provide the target latency figure. For setting always-on power-management policy use:

```
    sl_WlanPolicySet(SL_POLICY_PM , SL_ALWAYS_ON_POLICY, NULL,0)
```

- **Long Sleep Interval** – This low power mode comes with a desired max sleep time parameter. The parameter reflects the desired sleep interval between two consecutive wakeups for beacon reception. The Wi-Fi module computes the desired time and wakes up to the next DTIM that does not exceed the specified time. The maximum allowed desired max sleep time parameter is two seconds. **Note**: This policy works in client mode only. It automatically terminates mDNS and internal HTTP server running on the device. TCP/UDP servers initiated by the user application lead to unpredictable system behavior and performance. For setting low-latency power-management policy use:

```
sl_WlanPolicySet(SL_POLICY_PM , SL_LONG_SLEEP_INTERVAL_POLICY, NULL,0)
```

- **Low latency power** – This device power management algorithm exploits opportunities to lower it’s power mode. Tradeoff tends toward power conservation performance (for example, sensor application). **Note**: Low power mode is only supported when the Wi-Fi subsystem is not connected to an AP, and thus mostly relevant to Transceiver mode. For setting low latency power management policy use:

```
sl_WlanPolicySet(SL_POLICY_PM , SL_LOW_LATENCY_POLICY, NULL,0)
```

4.6.2 Advanced

See Section 4.6.

4.7 Scan Parameters

4.7.1 Scan Policy

SL_POLICY_SCAN defines a system scan time interval in case there is no connection. The default interval is 10 minutes. After the scan interval is set, an immediate scan is activated. The next scan will be based on the scan interval settings.

- For example, to set the scan interval to a 1-minute interval, use:

```
 unsigned long intervalInSeconds = 60;
 #define SL_SCAN_ENABLE  1
sl_WlanPolicySet (SL_POLICY_SCAN,SL_SCAN_ENABLE,   (unsigned char
*)&amp;intervalInSeconds,sizeof(intervalInSeconds));
```

- For example, to disable scan:

```
 #define SL_SCAN_DISABLE  0
 sl_WlanPolicySet(SL_POLICY_SCAN,SL_SCAN_DISABLE,0,0);
```

WLAN Connection

Topic **Page**

Connecting to a WLAN network is the first step required before initiating a socket communication. The Wi-Fi subsystem supports two ways of establishing a WLAN connection:

1. Manual connection – The application calls an API that triggers the connection process.
2. Connection using profiles – The Wi-Fi subsystem automatically connects to pre-defined connection profiles.

5.1 Manual Connection

5.1.1 STA

For a manual connection, the user application must implement the following steps:

1. Call to the **sl_WlanConnect** API. This API call accepts the SSID of the Access Point, the security type, and key, if applicable.
2. Implement a callback function to handle the asynchronous connection event **SL_WLAN_CONNECT_EVENT**, signaling the completion of the connection process.

For additional information about these APIs, refer to Section 18.2 or the doxygen API manual.

5.1.2 P2P

For details, see Chapter 11.

5.2 Connection Using Profiles

A WLAN profile provides the information required to connect to a given AP. This includes the SSID, security type and security keys. Each profile refers to a certain AP. The profiles are stored in the NVMEM (nonvolatile memory), and preserved during device reset. The following APIs are available for handling profiles:

- **sl_WlanProfileAdd** – Used for adding a new profile. SSID and security information must be provided, where the returned value refers to the stored index (out of the seven available).
- **sl_WlanProfileDel** – Used for deleting a certain stored profile, or for deleting all profiles at once. Index should be the input parameter.
- **sl_WlanProfileGet** – Used for retrieving information from a specific stored profile. Index should be the input parameter.

For additional information about these APIs, refer to the doxygen API manual.

Download the latest SDK for the complete example code.

```
/* Delete all profiles (0xFF) stored */
sl_WlanProfileDel(0xFF);

/* Add unsecured AP profile with priority 0 (lowest) */
sl_WlanProfileAdd(SL_SEC_TYPE_OPEN, (unsigned char*)UNSEC_SSID_NAME, strlen(UNSEC_SSID_NAME),
g_BSSID, 0, 0, 0, 0);


/* Add WPA2 secured AP profile with priority 1 (0 is lowest) */
sl_WlanProfileAdd(SL_SEC_TYPE_WPA, (unsigned char*)SEC_SSID_NAME, strlen(SEC_SSID_NAME), g_BSSID,
1, (unsigned char*)SEC_SSID_KEY, strlen(SEC_SSID_KEY), 0);
```

5.3 Connection Policies

SL_POLICY_CONNECTION passes the type parameters to the **sl_WlanPolicySet** API to modify or set the connection policies arguments. For additional information about this API, refer to the doxygen API manual. Download the latest SDK for the complete example code.

WLAN connection policy defines five options to connect the SimpleLink device to a given AP. The five options of the connection policy are:

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

```
To set this option, use
sl_WlanPolicySet(SL_POLICY_CONNECTION,SL_CONNECTION_POLICY(1,0,0,0,0),NULL,0)
```

- *Fast* – The device tries to connect to the last connected AP. In this mode "probe request" is not transmitted before "authentication request," as both the SSID and channel are already known.

```
To set this option, use
sl_WlanPolicySet(SL_POLICY_CONNECTION,SL_CONNECTION_POLICY(0,1,0,0,0),NULL,0)
```

- *anyP2P* (relevant for P2P mode only) – The Wi-Fi subsystem tries to automatically connect to the first P2P device available, supporting push-button only.

```
To set this option, use
sl_WlanPolicySet(SL_POLICY_CONNECTION,SL_CONNECTION_POLICY(0,0,0,1,0),NULL,0)
```

- *autoSmartConfig* – For auto SmartConfig upon restart (any command from the host ends this state).

```
To set this option, use
sl_WlanPolicySet(SL_POLICY_CONNECTION,SL_CONNECTION_POLICY(0,0,0,0,1),NULL,0)
```

For setting long sleep interval policy use:

```
unsigned short PolicyBuff[4] = {0,0,800,0}; // PolicyBuff[2] is max sleep time in mSec
sl_WlanPolicySet(SL_POLICY_PM , SL_LONG_SLEEP_INTERVAL_POLICY, PolicyBuff,sizeof(PolicyBuff));
```

5.4 Connection Related Async Events

5.4.1 *WLAN Events*

sl_WlanEvtHdlr is an event handler for WLAN connection or disconnection indication. Possible events are:

- SL_WLAN_CONNECT_EVENT – Indicates WLAN is connected.
- SL_WLAN_DISCONNECT_EVENT – Indicates WLAN is disconnected.

5.4.2 *Network Events*

sl_NetAppEvtHdlr is an event handler for an IP address asynchronous event and is usually accepted after a new WLAN connection. Possible events are:

- SL_NETAPP_IPV4_ACQUIRED – IP address was acquired (DHCP or static).

Chapter 6

SWRU368–June 2014

Socket

Topic **Page**

6.1 Overview

The networking API standard used in SimpleLink is BSD (Berkeley) sockets, upon which the Linux™, POSIX, and Windows™ sockets APIs are based. The main differences are in error codes (return directly without errno) and additional **setsockopt()** options.

- See Simplelink documentation and examples.
- Berkeley sockets on Wikipedia

The content of this page assumes a basic understanding of Internet protocol suite and the differences between TCP and UDP connections. Here are some basic concepts:

6.1.1 TCP

A definition of TCP from Wikipedia follows:

The **Transmission Control Protocol (TCP)** is one of the core protocols of the Internet protocol suite (IP), and is so common that the entire suite is often called TCP/IP. TCP provides reliable, ordered and error-checked delivery of a stream of octets between programs running on computers connected to a local area network, intranet or the public Internet. It resides at the transport layer. Applications that do not require the reliability of a TCP connection may instead use the connectionless User Datagram Protocol (UDP), which emphasizes low-overhead operation and reduced latency rather than error checking and delivery validation.

6.1.2 UDP

A definition of UDP from Wikipedia follows:

The **User Datagram Protocol (UDP)** is one of the core members of the Internet protocol suite (the set of network protocols used for the Internet). With UDP, computer applications can send messages, in this case referred to as datagrams, to other hosts on an Internet Protocol (IP) network without prior communications to set up special transmission channels or data paths. UDP is suitable for purposes where error checking and correction is either not necessary or performed in the application, avoiding the overhead of such processing at the network interface level. Time-sensitive applications often use UDP because dropping packets is preferable to waiting for delayed packets, which may not be an option in a real-time system. If error correction facilities are needed at the network interface level, an application may use the Transmission Control Protocol (TCP) or Stream Control Transmission Protocol (SCTP) which are designed for this purpose.

6.2 Socket Connection Flow

Figure 6-1 describes a general flow of TCP or UDP connection between a server and a client. The overall flow is nearly identical to the Linux implementation.

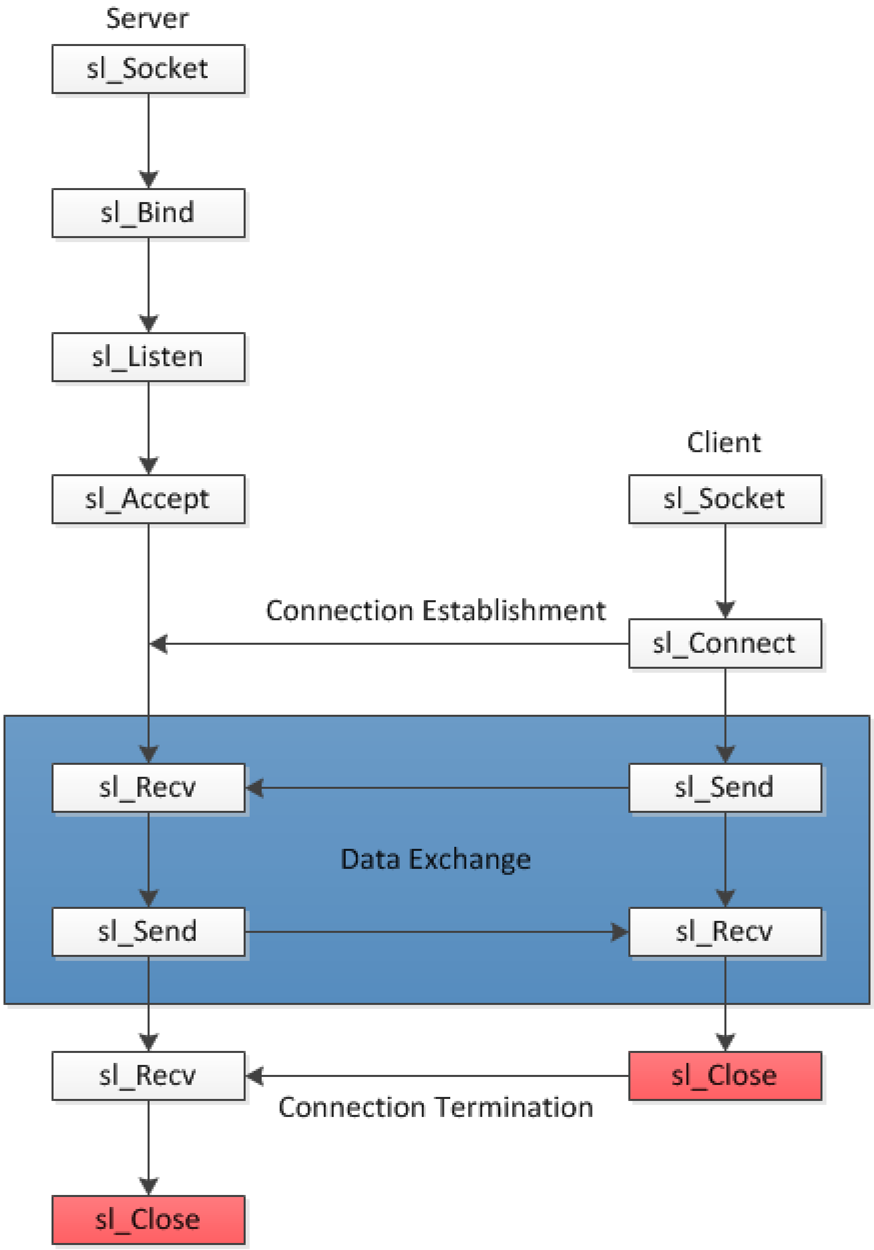


Figure 6-1. Socket Connection Flow

6.3 TCP Connection Flow

The following program structure provides some basic ideas of how to use the SimpleLink API. For a full sample application code, see **tcp_socket** in the SDK examples.

6.3.1 Client Side

First, create a socket. The returned socket handler is the most important element of the application. Networking will not work without the returned socket handler.

```
int SockID;
SockID = sl_Socket(SL_AF_INET, SL_SOCK_STREAM, 0);
```

SL_AF_INET indicates using IPv4 and **SL_SOCK_STREAM** indicates using TCP. Definitions for both values are in the **socket.h** header file. The example sets **0** in the third parameter to select a default protocol from the selected domain and type. More detail usage can be found in the online documentation. Definition of some structures and constants is in the **socket.h** header file inside SDK.

As a TCP client, the application executes **sl_Connect()** to connect to a server. The server implementation can be found below.

```
/* IP addressed of server side socket. Should be in long format,
 * E.g: 0xc0a8010a == 192.168.1.10 */
#define IP_ADDR    0xc0a80168

int Status;
int Port = 5001;
SlSockAddrIn_t  Addr;

Addr.sin_family      = SL_AF_INET;
Addr.sin_port        = sl_Htons((UINT16)Port);
Addr.sin_addr.s_addr = sl_Htonl((UINT32)IP_ADDR);

Status = sl_Connect(SockID,  ( SlSockAddr_t *) &amp;Addr, sixeof(SlSockAddrIn_t));
```

The struct **Addr** specifies destination address and relevant information. Because struct type **SlSockAddr** is generic, use **SlSockAddrIn_t** to fill the details and cast it into **SlSockAddr**. Upon successful connection, the **SockID** socket handler is ready to perform data exchange.

sl_Send() and **sl_Recv()** functions can be used for data exchange. Define the buffer size.

```
#define BUF_SIZE 1400
char SendBuf[BUF_SIZE];

/* Write data to your buffer*/
<write buffer action>

Status = sl_Send(SockID, SendBuf, BUF_SIZE, 0 );
char RecvBuf[BUF_SIZE];
Status = sl_Recv(SockID, RecvBuf, BUF_SIZE, 0);
```

Upon completion, close the socket with **sl_Close()** to allow the remaining applications to reuse the resource if needed.

```
sl_Close(SockID);
```

6.3.2 Server Side

Unlike TCP client, a TCP server must establish several things before communication can occur.

- Similar to client implementation, create a TCP-based IPv4 socket.

```
SockID = sl_Socket(SL_AF_INET,SL_SOCK_STREAM, 0);
```

- In a TCP server implementation, the socket must perform **Bind** and **Listen**. **Bind** is used to give the server socket an address. **Listen** puts the socket in listening mode for an incoming client connection.

```
#define PORT_NUM  5001

SlSockAddrIn_t  LocalAddr;

LocalAddr.sin_family      = SL_AF_INET;
LocalAddr.sin_port        = sl_Htons(PORT_NUM);
LocalAddr.sin_addr.s_addr = 0;
Status = sl_Bind(SockID,  (SLSockAdd_t *) &amp;LocalAddr, sizeof(SlSockAddrIn_t));

Status = sl_Listen(SockID, 0);
```

- With the socket now listening, accept any incoming connection request with **sl_Accept()**. There are two ways to perform this: blocking and nonblocking. This example uses the nonblocking mechanism with **sl_SetSockOpt()** and has the **sl_Accept()** placed in a loop to ensure it always retries connection regardless of each failure. Details about blocking and nonblocking can be found in Section 6.5.1.

Upon a successful connection, a new socket handler **newSockID** returns, which is then used for future communication.

```
long nonBlocking = 1;
int  newSockID;
Status = sl_SetSockOpt (SockID, SL_SOL_SOCKET, SL_SO_NONBLOCKING, &amp;nonBlocking,
sizeof(nonBlocking));

while( newSockID < 0 )
{
      newSockID = sl_Accept(SockID, ( struct SlSockAddr_t
*) &amp;Addr, (SlSocklen_t*) &amp;AddrSize) ;
    if( newSockID == SL_EAGAIN )
    {
        /* Wait for 1 ms */
        Delay(1);
    }
else if( newSockID < 0 )
{
        return -1;
}
}
```

Data exchange is exactly the same as implemented in client. The user may need to reverse the order; when one side is sending, the other side must be receiving.

```
#define BUF_SIZE 1400
char SendBuf[BUF_SIZE];

/* Write data to your buffer*/
<write buffer action>

Status = sl_Send(newSockID, SendBuf, BUF_SIZE, 0 );
char RecvBuf[BUF_SIZE];
Status = sl_Recv(newSockID, RecvBuf, BUF_SIZE, 0);
```

At the end, close both sockets with **sl_Close()** to allow the remaining applications run to reuse the resource if needed.

```
sl_Close(newSockID);
sl_Close(SockID);
```

6.4 UDP Connection Flow

The following program structure provides some basic ideas of how to use the SimpleLink API. For a full sample application code, see **udp_socket** in the SDK examples.

6.4.1 Client Side

Similar to the previous TCP example, first create a IPv4-based socket. However, change the second parameter to **SL_SOCK_DGRAM**, which indicates the socket will be used for UDP connection.

```
SockID = sl_Socket(SL_AF_INET, SL_SOCK_DGRAM, 0);
```

Because UDP is a connectionless protocol, the client can start sending data to a specified target address without checking whether the target is alive or not.

```
#define IP_ADDR         0xc0a80164
#define PORT_NUM        5001
```

```
Addr.sin_family      = SL_AF_INET;
Addr.sin_port        = sl_Htons((UINT16)PORT_NUM);
Addr.sin_addr.s_addr = sl_Htonl((UINT32)IP_ADDR);

Status = sl_SendTo(SockID, uBuf.BsdBuf, BUF_SIZE, 0,   (SlSockAddr_t *) &amp;Addr,
sizeof(SlSockAddrIn_t));
```

Finally, close the socket.

```
sl_Close(SockID);
```

6.4.2 Server Side

The server side of the socket is identical to the client side.

```
SockID = sl_Socket(SL_AF_INET,SL_SOCK_DGRAM, 0);
```

Similar to TCP, bind the socket to the local address. No listening is required as UDP is connectionless.

```
#define PORT_NUM        5001

SlSockAddrIn_t  LocalAddr;
AddrSize = sizeof(SlSockAddrIn_t);

TestBufLen  = BUF_SIZE;

LocalAddr.sin_family      = SL_AF_INET;
LocalAddr.sin_port        = sl_Htons((UINT16) PORT_NUM);
LocalAddr.sin_addr.s_addr = 0;
Status = sl_Bind(SockID,  (SlSockAddr_t *) &amp;LocalAddr, AddrSize);
```

The socket now tries to receive information on socket. If the user did not specify the socket option as nonblocking, this command is blocked until an amount of **BUF_SIZE** of data is received. The fifth parameter specifies the source address from which the data is being sent.

```
#define BUF_SIZE 1400
SlSockAddrIn_t  Addr;
char            RecvBuf[BUF_SIZE];

Status = sl+RecvFrom(SockID, RecvBuf, BUF_SIZE, 0, (SlSockAddr_t *) &amp;Addr, (SlSocklen_t*)
&amp;AddrSize );
```

Close the socket once communication is finished.

```
sl_Close(SockID);
```

6.5 Socket Options

6.5.1 Blocking versus NonBlocking

Depending on your implementation, the application can be run with or without OS. Normally when the application is without OS, set the socket option to nonblocking with **sl_SetSockOpt()** with the third parameter as **SL_SO_NONBLOCKING**. An OS-based application, however, has the option to perform multithreading and can handle blocking functions.

```
Status = sl_SetSockOpt(SockID, SL_SOL_SOCKET, SL_SO_NONBLOCKING,
             &amp;nonBlockingValue, sizeof(nonBlockingValue)) ;
```

If the blocking mechanism is used, these functions will block until execution is finished.

If the nonblocking mechanism is used, these functions will return with an error code. The value of the error codes depends the function being used. For details, see the online documentation.

sl_Connect(), **sl_Accept()**, **sl_Aend()**, **sl_Aendto()**, **sl_Recv()**, and **sl_Recvfrom()** are affected by this flag. If not set, the default is blocking.

An example with **sl_Connect()** on a client application:

- Blocking: **sl_Connect()** blocks until connecting to a server, or an error occurs. Do not use blocking if your application is single-threaded and must perform other tasks as well (such as handling multiple sockets to read/write). However, using blocking is fine if the application is OS-based (like FreeRTOS).

```
Status = sl_Connect(SockID, ( SlSockAddr_t *) &amp;Addr, AddrSize);
```

- NonBlocking: **sl_Connect()** returns immediately, regardless of connection. If connection is successful, a value of 0 returns. If not, the function returns **SL_EALREADY** under normal conditions. TI recommends that the function is called in a loop so that the function keeps retrying the connection. The advantage of a nonblocking mechanism is to prevent the application from getting stuck. This is particularly useful if your application must perform other tasks (such as blinking LED, reading sensor data, or having other connections simultaneously) at the same time.

```
while( Status < 0 )
{
Status = sl_Connect(SockID, ( SlSockAddr_t *)&amp;Addr, AddrSize);
    if( Status == SL_EALREADY )
    {
        /* Wait for 1 ms before the next retry */
        Delay(1);
    }
else if( Status < 0 )
                {
        return -1; //Error
                }

    /* Perform other tasks before we retry the connection */
}
```

6.5.2 Secure Sockets

See Section 9.2.

6.6 SimpleLink Supported Socket API

Table 6-1 describes supported BSD sockets and the corresponding SimpleLink implementation.

Table 6-1. SimpleLink Supported Socket API

BSD Socket	Simplelink Implementation	Server/Client Side	TCP/UDP	Description
socket()	sl_Socket()	Both	Both	Creates an endpoint for communication
bind()	sl_Bind()	Server	Both	Assigns a socket to an address
listen()	sl_Listen()	Server	Both	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	TCP	Writes and reads data to and from TCP socket.
write(), read()	Not supported			
sendto(), recvfrom()	sl_SendTo(), sl_RecvFrom()	Both	UDP	Writes and reads data to and from UDP socket

Table 6-1. SimpleLink Supported Socket API (continued)

BSD Socket	Simplelink Implementation	Server/Client Side	TCP/UDP	Description
close()	sl_Close()	Both	Both	Causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.
gethostbyname(), gethostbyaddr()				
select()	sl_Select()	Both	Both	Used to pend, waiting for one or more of a provided list of sockets to be ready to read, ready to write, or that have errors
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

Device Hibernate

Hibernate is the lowest power state of the device. In this state the Wi-Fi subsystem's volatile memory is not maintained. Only the RTC is maintained, for faster boot time and for keeping the system date and time.

The Wi-Fi subsystem goes into hibernate on a call to the **sl_Stop** API. This API receives only one parameter, a timeout parameter that configures the device to use the minimum amount of time to wait before going to hibernate.

For more details, refer to Chapter 18 and the API Doxygen application note.

Provisioning

8.1 Provisioning

8.1.1 SmartConfig

8.1.1.1 General Description

SmartConfig is a method of remotely passing WLAN credentials (SSID, security credentials) to a CC3100/CC3200 device planned to be used as STA.

8.1.1.2 How to Use / API

8.1.1.2.1 Automatic Activation

To enable the feature, start the SimpleLink device. The device should start as STA role. Assuming no profile was added earlier, after a few seconds with no commands SmartConfig should start.

To start the SmartConfig application on a smart phone or PC:

1. Connect the smart phone to any Wi-Fi network.
2. Enter the WLAN credentials (SSID, security credentials).
3. Supply a key (Optional. Used to encrypt the Wi-Fi password).
4. Press the Start button.

The SmartConfig operation should complete in few seconds. However, the operation can take up to two minutes to complete. If the requested network is in the device's proximity, the device will connect to it immediately.

The following topics apply when using automatic activation:

- Any command sent to the device will terminate the SmartConfig operation.
- If a key is stored at the device serial flash, the password will be encrypted and the key must be supplied.
- If the device configuration was changed before SmartConfig was automatically started, this may cause problems. In this case ensure the following:
  - Auto Start policy is set.
  - Auto SmartConfig policy is set.
  - No profile was added earlier.

To verify the configuration, call:

```
sl_WlanPolicyGet(SL_POLICY_CONNECTION, 0, pVal, pValLen);
```

The returned policy will be stored in the allocated buffer pointed by pVal.

Bits 0 and 4 should be set if Auto Start and Auto SmartConfig policies are set.

If this is not the case, set these policies manually by calling:

```
sl_WlanPolicySet(SL_POLICY_CONNECTION,SL_CONNECTION_POLICY(1,0,0,0,1),NULL,0)
```

To ensure no profile is saved, remove all saved profiles by calling:

```
sl_WlanProfileDel(255);
```

After sending these commands, reset the device and SmartConfig should operate successfully.

8.1.1.2.2 Manual Activation

To start SmartConfig manually, send the following command:

```
sl_WlanSmartConfigStart(groupIdBitmask,
                        cipher,
                        publicKeyLen,
                        group1KeyLen,
                        group2KeyLen,
                        publicKey,
                        group1Key,
                        group2Key)
```

Parameters description:

- groupIdBitmask – Use 1 as the default group ID bitmask (group ID 0).

To encrypt the password when the encryption key is not stored in the serial flash of the device, use:

- cipher = 1
- publicKeyLen = 16
- group1KeyLen = 0
- group2KeyLen = 0
- publicKey = put the key here (use a 16-character string)
- group1Key = NULL
- group2Key = NULL

To encrypt the password when the encryption key is stored in the serial flash of the device, use:

- cipher = 0
- publicKeyLen = 0
- group1KeyLen = 0
- group2KeyLen = 0
- publicKey = NULL
- group1Key = NULL
- group2Key = NULL

To avoid encrypting the password use:

- cipher = 1
- publicKeyLen = 0
- group1KeyLen = 0
- group2KeyLen = 0
- publicKey = NULL
- group1Key = NULL
- group2Key = NULL

After sending this command, SmartConfig will start.

When running SmartConfig manually, if a key to encrypt the password is stored in the external serial flash or supplied at the command, it is necessary to supply this key in the SmartConfig Application.

8.1.1.2.3 Stopping Smart Config

To stop the SmartConfig operation call:

```
sl_WlanSmartConfigStop()
```

After the device is connected to the requested network it should receive an IP address from the AP or router.

If the SmartConfig operation does not end successfully, it could be because the Wi-Fi network to which the smart phone is connected is using transmissions modes and rates that are not suitable for SmartConfig. In this case, use the AP Provisioning method.

8.1.2 AP Mode

8.1.2.1 General Description

AP provisioning is a method of passing WLAN credentials (SSID, security credentials) to an accessory.

8.1.2.2 How to Use / API

First start the SimpleLink device in AP Role. There are two methods to start:

- Call **sl_WlanSetMode(mode)** where the mode should be ROLE_AP (2). Reset the device. For additional information see Chapter 10.
- Force the device to start as an AP.

After the device is started as AP, search for it with the smart phone or any other device with Wi-Fi connectivity.

The device should appear in the list containing all available networks. The device name should be mysimplelink-xxyyzz where xxyyzz are the last six digits of the device MAC address.

If the user already changed the device SSID by calling **sl_WlanCfgSet(0, 0, ssid_length, ssid)**, this SSID will be shown on the list.

If the user has not changed the SSID, but changed the Device URN by calling **sl_NetAppSet(16, 0, urn_length, urn)**, then the device name should be urn-xxyyzz where xxyyzz are the last six digits of the device MAC address.

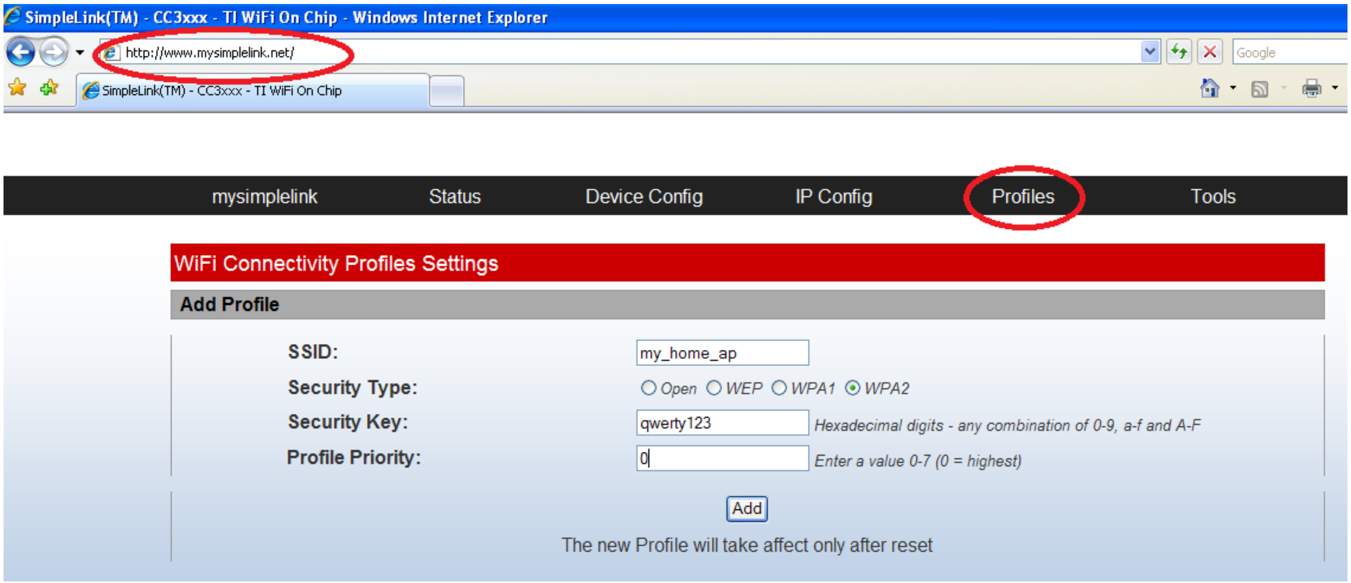
At the device screen, choose the device network and connect to it.

Note: The connection should be unsecured unless the user changed the mode. In this case, supply the password entered when the device requests it.

Once the device is connected, open a browser and go to: http://www.mysimplelink.net.

Go to the Profiles tab.

Enter the WLAN credentials (SSID, security type, and key), select the priority for this profile (any value between 0-7), and press Add. See Figure 8-1.


SimpleLink(TM) - CC3xxx - TI WiFi On Chip - Windows Internet Explorer
http://www.mysimplelink.net/
Google
SimpleLink(TM) - CC3xxx - TI WiFi On Chip
mysimplelink
Status
Device Config
IP Config
Profiles
Tools

WiFi Connectivity Profiles Settings

Add Profile

SSID: my_home_ap

Security Type: ○ Open ○ WEP ○ WPA1 ● WPA2

Security Key: qwerty123 Hexadecimal digits - any combination of 0-9, a-f and A-F

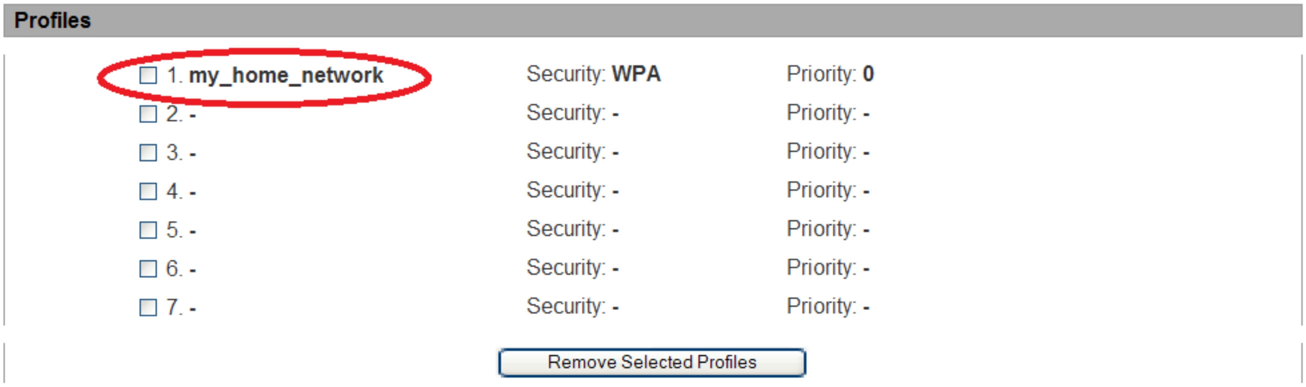
Profile Priority: 0 Enter a value 0-7 (0 = highest)

Add

The new Profile will take affect only after reset

Figure 8-1. AP Mode Connect

Scroll to the bottom of the web page and check that the device is added to the Profiles (the password will not be displayed). See Figure 8-2.

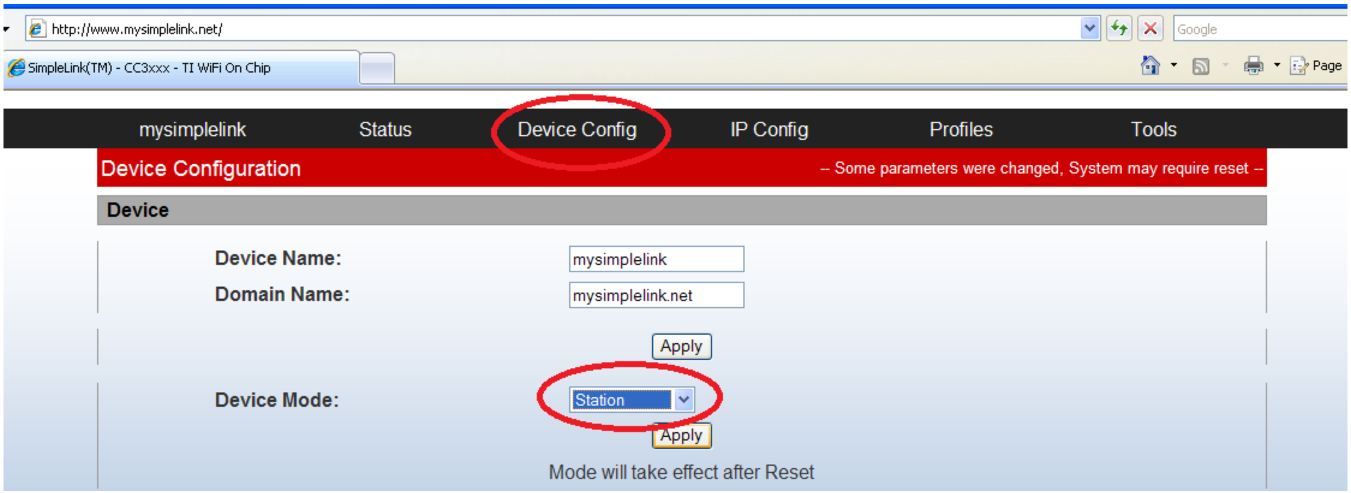


Profiles

□ 1. my_home_network	Security: WPA	Priority: 0
□ 2. -	Security: -	Priority: -
□ 3. -	Security: -	Priority: -
□ 4. -	Security: -	Priority: -
□ 5. -	Security: -	Priority: -
□ 6. -	Security: -	Priority: -
□ 7. -	Security: -	Priority: -

Figure 8-2. Profiles

Now start the device in STA Role. Go to the Device Config tab and set Device Mode to Station (or remove the Force AP constraint) and reset the device. After resetting the device, the device connects automatically to the requested network. See Figure 8-3.



http://www.mysimplelink.net/
Google
SimpleLink(TM) - CC3xxx - TI WiFi On Chip
Page
mysimplelink
Status
Device Config
IP Config
Profiles
Tools

Device Configuration

-- Some parameters were changed, System may require reset --

Device

Device Name: mysimplelink

Domain Name: mysimplelink.net

Device Mode: Station

Mode will take effect after Reset

Figure 8-3. Device Config Tab

8.1.2.3 Things to Note when Configuring AP Provisioning

When connecting with a smart phone to the (AP) device, the device will allocate an IP address as it runs DHCP Server. The smart phone (or other configuring device) should not be using a static IP address.

After entering the WLAN credentials (at the Profiles tab), move to STA mode and reset the device. The device should connect to the requested network and receive an IP address from the AP or router.

8.1.3 WPS

8.1.3.1 General Description

WPS (Wi-Fi protected setup) is a standard for easy and secure wireless network set up and connections, and allows both push-button and PIN-based access to Wi-Fi networks.

Push-button: Push the WPS button in the AP or, if the button is not available, start the WPS process using the GUI of the AP. The AP will enter the WPS provisioning process for 2 minutes. During this period, the SimpleLink device should also enter the WPS provisioning process by calling the **sl_WlanConnect** API with WPS parameters (see Section 8.1.3.3). For example, calling this API can be mapped to a push-button in the MCU. At the end of this process, a wireless network with a network name and security is configured automatically.

PIN-based: Enter the pin generated by the host using the GUI of the AP. The AP will enter WPS provisioning process for 2 minutes. During this period, the SimpleLink device should also enter the WPS provisioning process by calling the **sl_WlanConnect** API with WPS parameters (see Section 8.1.3.3). At the end of this process, a wireless network with a network name and security is configured automatically.

Once the WPS process completes successfully, connection with the AP is established in the correct security setting according to the configuration of the AP (Open, WEP, WPA, or WPA2). The connection parameters are saved as a profile. Using the connection policy AUTO triggers a reconnection after a reset.

8.1.3.2 How to Use / API

```
sl_WlanConnect(char* pName, int NameLen, unsigned char *pMacAddr, SlSecParams_t* pSecParams ,
SlSecParamsExt_t* pSecExtParams);
```

This API with the correct settings can trigger a WPS connection with both configurations: push-button and PIN-based.

Parameter configuration:

- **pName** – NULL
- **NameLen** – 0
- **unsigned char *pMacAddr** – NULL
- **SlSecParamsExt_t* pSecExtParams** – not relevant for WPS, set as NULL
- Push-button:
  - **SlSecParams_t* pSecParams**: Type – SL_SEC_TYPE_WPS_PBC (3)

  Key – NULL
  Key length – set to 0
- **SlSecParams_t* pSecParams**: Type – SL_SEC_TYPE_WPS_PIN (4)
  Key – WPS pin code
  Key length – WPS pin code length

```
int sl_WlanProfileGet(int Index,char* pName, int *pNameLen, unsigned char *pMacAddr,
SlSecParams_t* pSecParams, SlSecParamsExt_t* pEntParams, unsigned long *pPriority)
```

- This API retrieves the profile parameters which were saved during the WPS connection process.

```
sl_WlanProfileDel(int Index)
```

- This API is used to delete a profile. It can be used to delete the profile which was saved during the WPS connection process. Calling this API with index set as 255 erases all stored profiles.

8.1.3.3 Example of Using WPS

```
// Push Button

void main()

{

    SlSecParams_t WPSsecParams;

    Int status;

    Int role;

    role = sl_Start(NULL, NULL, NULL);

    if( 0 > role )

            {

    printf("failed start cc3100\n");
```

```
                  }
        if(role == ROLE_STA)
                  {
    WPSsecParams.Type = SL_SEC_TYPE_WPS_PBC;
        WPSsecParams.Key = NULL;
        WPSsecParams.KeyLen = 0;
    status = sl_WlanConnect(0,0,0,&amp;WPSsecParams,0);
           while (SL_IPEQUIRED != g_SlConnState)
                  {
            Sleep(20);
                  }
                  }
    }
    // PIN-based
    void main()
    {
        SlSecParams_t WPSsecParams;
        Int status;
        Int role;
        role = sl_Start(NULL, NULL, NULL);
        if( 0 > role )
                      {
                          printf("failed start cc3100\n");
                      }
        if(role == ROLE_STA) {
           WPSsecParams.Type = SL_SEC_TYPE_WPS_PIN;
                                WPSsecParams.Key = "34374696";  //example pin code
                                WPSsecParams.KeyLen = 8;
                                status = sl_WlanConnect(0,0,0,&amp;WPSsecParams,0);
           while (SL_IPEQUIRED != g_SlConnState)    {
                                                    Sleep(20);
                              }
                  }
    }
```

Chapter 9

SWRU368–June 2014

Security

Topic Page

9.1 WLAN Security

9.1.1 Personal

The Wi-Fi subsystem supports the Wi-Fi security types AES, TKIP & WEP. The personal security type and personal security key are set in both the manual connection API or profiles connection API by the same parameter type – **SlSecParams_t**. This structure consists of the following fields:

- **Security Type** – The type of security being used. Options include:
  - **SL_SEC_TYPE_OPEN** – No security (default value).
  - **SL_SEC_TYPE_WEP** – WEP security.
  - **SL_SEC_TYPE_WPA** – Used for both WPA\PSK and WPA2\PSK security types, or a mixed mode of WPA\WPA2 PSK security type (for example, TKIP, AES, mixed mode).
  - **SL_SEC_TYPE_WPA_ENT** – WPS security. For more information refer to Section 8.1.3.
  - **SL_SEC_TYPE_WPS_PBC ENT** – Push-button WPS security. For more information refer to Section 8.1.3.
  - **SL_SEC_TYPE_WPS_PIN ENT** – Pin-based WPS security. For more information refer to Section 8.1.3.
- **Key** – A character area containing the pre-shared key (PSK) value.
- **Key length** – The number of characters of the pre-shared key.

Example code for adding a WPA2 secured AP profile:

```
secParams.Type = SL_SEC_TYPE_WPA;
secParams.Key = SEC_SSID_KEY;
secParams.KeyLen = strlen(SEC_SSID_KEY);

sl_WlanProfileAdd((char*)SEC_SSID_NAME, strlen(SEC_SSID_NAME), g_BSSID,  &amp;secParams, 0, 7, 0);
```

9.1.2 Enterprise

9.1.2.1 General Description

The SimpleLink device supports Wi-Fi enterprise connection, according to the 802.1x authentication process. An enterprise connection requires the radio's server behind the AP authenticate the station. The following authentication methods are supported on the device:

- EAP-TLS
- EAP-TTLS with MSCHAP
- EAP-TTLS with TLS
- EAP-TTLS with PSK
- EAP-PEAP with TLS
- EAP-PEAP with MSCHAP
- EAP-PEAP with PSK
- EAP-FAST

After the station has been authenticated, the AP and the station negotiate WPA(1/2) security.

9.1.2.2 How to Use / API

When connecting to an enterprise network, three files are needed:

- **Private Key** – The station (client) RSA private key file in PEM format
- **Client Certificate** – The certificate of the client given by the authenticating network (ensures the public key matches to the private key) in PEM format
- **Server Root Certificate Authority file** – This file authenticates the server. This file must be in PEM format.

These three files must be programmed with the following names so the device will use them:

- Certificate authority: /cert/ca.pem
- Client certificate: /cert/client.pem
- Private key: /cert/private.key

Establishing a connection:

```
sl_WlanConnect(char* pName, int NameLen, unsigned char *pMacAddr, SlSecParams_t* pSecParams ,
SlSecParamsExt_t* pSecExtParams)


sl_WlanProfileAdd(char* pName, int NameLen, unsigned char *pMacAddr, SlSecParams_t* pSecParams ,
SlSecParamsExt_t* pSecExtParams, unsigned long  Priority, unsigned long  Options)
```

The **sl_WlanConnect** and the **sl_WlanProfileAdd** commands are used for different types of Wi-Fi connection. The connect command is for a one-shot connection, and the add profile used when auto connect is on (see the add profile white papers). Use those commands with extra security parameters for the enterprise connection – **SlSecParamsExt_t**.

A short view of the first five parameters of those commands follows. (Learn more of the extra parameters of the add profile command in its white paper.)

1. SSID name – The name of the Wi-Fi network
2. SSID length
3. Flags - Not applicable for enterprise connection
4. Pointer to SlSecParams_t -

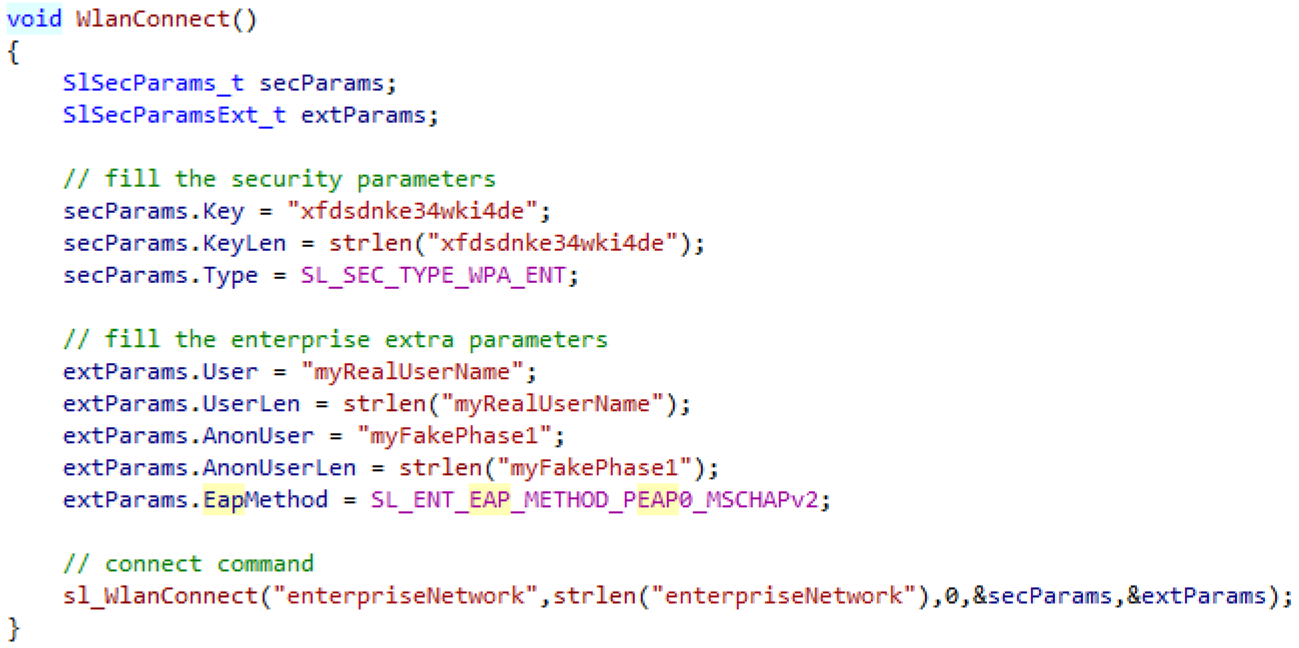
```
typedef struct
{
    unsigned char   Type; - type should be SL_SEC_TYPE_WPA_ENT
    char*                  Key;   - a key password for the enterprise connection that
                                    must have it. MSCHAP, FAST ETC.
    unsigned char   KeyLen;
}SlSecParams_t;
```

5. Pointer to SlSecParamsExt_t-

```
typedef struct
{
    char*                  User;  -               the enterprise user name
    unsigned char   UserLen;
    char*                  AnonUser;  -     the anonymous user name (optional) for two phase
                                                 enterprise connections.
    unsigned char   AnonUserLen;
    unsigned char   CertIndex;  -     not supported
    unsigned long   EapMethod; -
                            SL_ENT_EAP_METHOD_TLS
                            SL_ENT_EAP_METHOD_TTLS_TLS
                            SL_ENT_EAP_METHOD_TTLS_MSCHAPv2
                            SL_ENT_EAP_METHOD_TTLS_PSK
                            SL_ENT_EAP_METHOD_PEAP0_TLS
                            SL_ENT_EAP_METHOD_PEAP0_MSCHAPv2
                            SL_ENT_EAP_METHOD_PEAP0_PSK
                            SL_ENT_EAP_METHOD_PEAP1_TLS
                            SL_ENT_EAP_METHOD_PEAP1_MSCHAPv2
                            SL_ENT_EAP_METHOD_PEAP1_PSK
                            SL_ENT_EAP_METHOD_FAST_AUTH_PROVISIONING
                            SL_ENT_EAP_METHOD_FAST_UNAUTH_PROVISIONING
                            SL_ENT_EAP_METHOD_FAST_NO_PROVISIONING
}SlSecParamsExt_t;
```

9.1.2.3 Example

Figure 9-1 shows an example of a simple WLAN connect command to an enterprise network.



```
void WlanConnect()
{
    SlSecParams_t secParams;
    SlSecParamsExt_t extParams;

    // fill the security parameters
    secParams.Key = "xfdsdnke34wki4de";
    secParams.KeyLen = strlen("xfdsdnke34wki4de");
    secParams.Type = SL_SEC_TYPE_WPA_ENT;

    // fill the enterprise extra parameters
    extParams.User = "myRealUserName";
    extParams.UserLen = strlen("myRealUserName");
    extParams.AnonUser = "myFakePhase1";
    extParams.AnonUserLen = strlen("myFakePhase1");
    extParams.EapMethod = SL_ENT_EAP_METHOD_PEAP0_MSCHAPv2;

    // connect command
    sl_WlanConnect("enterpriseNetwork",strlen("enterpriseNetwork"),0,&secParams,&extParams);
}
```

Figure 9-1. WLAN Connect Command

9.1.2.4 Limitations

There is no command to bind a certificate file to a WLAN enterprise connection. The certificates of the network must be programmed with the names specified in Section 9.1.2.2.

9.2 Secured Socket

9.2.1 General Description

SSL is a secured socket over TCP that lets the user connect to servers (and Internet sites) securely, or to open a secured server.

This chapter covers how to use the socket with the host driver, and how to generate certificates and keys for the SSL.

9.2.2 How to Use / API

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.

Use any standard BSD commands (**sl_Close**, **sl_Listen**, **sl_Accept**, **sl_Bind**, **sl_SetSockOpt**, and so forth) to open client, open server, change socket parameters, and more.

The BSD commands let the user connect without choosing the SSL method (SSLv3 TLS1.0/1.1/1.2) and without choosing the connection cipher suit (those two will be negotiated in the SSL handshake).

In client socket, the certificate of the server is not verified until the root CA is used to verify the server.

In server socket, the user must supply the server certificate and private key.

Use the **setsockopt** command with proprietary options to configure the secured parameters of the socket.

9.2.2.1 Selecting a Method

Use to manually define the SSL method. In CC3100, SSLv3 TLSv 1.0/1.1/1.2 is supported.

SlSockSecureMethod method; method.secureMethod = Choose one of the following defines:

- 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

```
Sl_SetSockOpt(sockID, SL_SOL_SOCKET, SL_SO_SECMETHOD, &amp;method, sizeof(SlSockSecureMethod));
```

9.2.2.2 Selecting a Cipher Suit

Use to manually define the SSL connection and handshake security algorithms, also known as cipher suits.

SlSockSecureMask Mask; mask.secureMask = Choose logic or between the following:

- 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_SECURE_DEFAULT

```
Sl_SetSockOpt(sockID, SL_SOL_SOCKET, SL_SO_SECMETHOD, &amp;mask, sizeof(SlSockSecureMask));
```

9.2.2.3 Selecting the Secured Files for the Socket

Defines which files the socket will use for the connection. There are four files that can be attached to a socket:

- Private Key
- Certificate
- Root CA
- DH file

These are the files in client and server sockets:

Client

- **Private Key, Certificate** – If a client must be authenticated by the server, both private key and certificate are mandatory together. Use two **setsockopt** commands to configure each file.
- **Root CA** – This is the root certificate authority that issued the server certificate and is used to validate that the server is authentic. This file is not mandatory. If the server is not verified, the connection occurs, but the connect command returns an error, SL_ESECSNOVERIFY. This error can be ignored as it is only a warning for an unauthenticated connection.
- **DH file** – No use in client

Server

- **Private Key, Certificate** – Mandatory for server
- **Root CA** – The certificate issued to the client. When file is set, it obligates the client to send their certificate for client authentication.
- **DH file** – Used to support the DH cipher suit – TLS_DHE_RSA_WITH_AES_256_CBC_SHA.

To bind files to a socket, program the file to the device. Then use the **setsockopt** to enter the file name. All secured files must be in DER format.

Setsockopt options for the secured files:

- 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_DH_KEY_FILE_NAME

For example, use the file rootCA.der that is in the device:

```
Sl_SetSockOpt(sockID, SL_SOL_SOCKET, SL_SO_SECURE_FILES_CA_FILE_NAME,”rootCA.der”,
strlen(“rootCA.der”));
```

Note the strlen in the **setsockopt**, and not sizeof.

9.2.3 Example of Using the SSL

```
int CreateConnection(unsigned long DestinationIP)
{
    int Status;
    SlSockAddrIn_t  Addr;
    int AddrSize;
    int SockID = 0;
    SlTimeval_t timeval;

    Addr.sin_family = SL_AF_INET;
    Addr.sin_port = sl_Htons(443);  // secured connection
    Addr.sin_addr.s_addr = sl_Htonl(DestinationIP);

    AddrSize = sizeof(SlSockAddrIn_t);
    SockID = sl_Socket(SL_AF_INET,
                       SL_SOCK_STREAM,
                       SL_SEC_SOCKET);
if( SockID < 0 )
                {
        // error
        while (1);
                }

      Sl_SetSockOpt(sockID,
                   SL_SOL_SOCKET,
                   SL_SO_SECURE_FILES_CA_FILE_NAME,
                   ”rootCA.der”,
                   strlen(“rootCA.der”));

     Status = sl_Connect(SockID,
( SlSockAddr_t *) &amp;Addr,
                         AddrSize);

if( Status < 0 &amp;&amp; Status != SL_ESECSNOVERIFY )
                {
         // error
         while(1);
                }

     return SockID;
}
```

9.2.4 Supported Cryptographic Algorithms

Table 9-1. Supported Cryptographic Algorithms

Cryptographic Algorithms	Standard Protocol	Purpose	Length of Encryption Key
RC4	WEP, TKIP	Data encryption	128 bits
AES	WPA2	Data encryption, authentication	256 bits
DES	–	Data encryption	56 bits
3DES	–	Data encryption	56 bits
SHA1	EAP-SSL/TLS	Authentication	160 bits
MD5	EAP-SSL/TLS	Authentication	128 bits
RSA	EAP-SSL/TLS	Authentication	2048 bits
DHE	EAP-SSL/TLS	Authentication	2048 bits
ECDHE	EAP-SSL/TLS	Authentication	160 bits

9.3 File System Security

SimpleLink uses an advanced file system to store data on the serial flash memory. This file system supports advanced security aspects like data encryption and certificate for file authentication.

The primary security features supported are:

- **Secured FAT** – The file system table used by SimpleLink is encrypted and signed.
- **Secured system files** – The most important networking files are encrypted and signed. For more detailed information, see Section 14.5, under the Chapter 14.
- **Tokens** – The file system has a token mechanism created upon file creation, providing the only allowed operation to the file creator. For more detailed information, see Section 14.8, under Chapter 14.
- **Signature & Certificates** – Secure files can be created with a signature. For more detailed information, see Section 14.9, under Chapter 14.
- **Security Alert** – The file system will lock when identifying an attempt to access the file system without permission. For more detailed information, see Section 14.7, under Chapter 14.

For more detailed information on the file system and specifically the file system security features, see Chapter 14.

AP Mode

Topic **Page**

10.1 General Description

AP (access point) should be set and configured by calling APIs. This chapter describes the parameters that can be configured and how to configure them.

10.2 Setting AP Mode – API

```
sl_WlanSetMode(const unsigned char    mode)
```

Where mode should be ROLE_AP (2).

Note: This change will take affect after reset.

10.3 WLAN Parameters Configuration – API

```
sl_WlanCfgSet (unsigned short ConfigId,
 unsigned short ConfigOpt,
 unsigned short ConfigLen,
 unsigned char *pValues)
```

This function sets the user parameter to configure. The input parameters are:

- **ConfigId** – Should be set to SL_WLAN_CFG_AP_ID (0) or SL_WLAN_CFG_GENERAL_PARAM_ID (1), according to the parameter
- **ConfigOpt** – Identifies the parameter to configure
- **ConfigLen** – Should be the parameter size in bytes
- **pValues** – Pointer to memory containing the parameter

Note: This change will take affect after reset.

Table 10-1 describes how to configure each parameter.

Table 10-1. WLAN Parameters

Parameter	Description	ConfigId	ConfigOpt	ConfigLen
SSID	Service set identifier (SSID) - a 1- to 32-byte string, commonly called the "network name". Default: Composed from the device URN and last 6 digits of the MAC Address. For example – mysimplelink-112233.	SL_WLAN_CFG_AP_ID (0)	0	1-32
Beacon interval	The time interval between beacon transmissions. Range: 15 <= interval <= 65,535 milliseconds. Default value: 100.	SL_WLAN_CFG_AP_ID (0)	2	2
Channel	Operational channel for the AP. The range depends on the country code. Range: US: 1-11 JP: 1-14 EU: 1-13 Default value: 6	SL_WLAN_CFG_AP_ID (0)	3	1
SSID Hidden	Whether to NOT broadcast the SSID inside the Beacon frame. Default value: Disabled	SL_WLAN_CFG_AP_ID (0)	4	1

Table 10-1. WLAN Parameters (continued)

Parameter	Description	ConfigId	ConfigOpt	ConfigLen
DTIM	Indicates the interval of the Delivery Traffic Indication Message (DTIM). A DTIM field is a countdown field informing clients of the next window for listening to broadcast and multicast messages. DTIM field is in the Beacon frame. Range: DTIM > 0 Default value: 2	SL_WLAN_CFG_AP_ID (0)	5	1
Security Type	Security mode for the network. Range: Open (no security) / WEP / WPA Default: Open	SL_WLAN_CFG_AP_ID (0)	6	1
Password	Password to use for the network in case Security Type is not open. Password should be human-readable string. For WEP it should be 5 to 13 characters. For WPA it should be 8 to 63 characters.	SL_WLAN_CFG_AP_ID (0)	7	5-63
WPS State	Wi-Fi Protected Setup (originally Wi-Fi Simple Config) is a network security standard that lets users easily secure a wireless home network. Default value: Disabled	SL_WLAN_CFG_AP_ID (0)	8	1
Country code	Identifies the country code of the AP. Possible values are US (USA) or JP (Japan) or EU (Europe). Default value is “US”.	SL_WLAN_CFG_GENERAL_PARAM_ID (1)	9	1
AP TX Power	AP transmission power. Range: 0 (maximal) – 15 (minimal). Default: 0 (maximal TX power).	SL_WLAN_CFG_GENERAL_PARAM_ID (1)	11	1

10.4 WLAN Parameters Query – API

```
sl_WlanCfgGet(unsigned short ConfigId,
unsigned short *pConfigOpt,
unsigned short *pConfigLen,
unsigned char *pValues)
```

This function gets the parameter the user requested where:

- **ConfigId** – Should be set to SL_WLAN_CFG_AP_ID (0) or SL_WLAN_CFG_GENERAL_PARAM_ID (1) as in the SET function
- **ConfigOpt** – Identifies the parameter to configure. Should be the address of the field. The value of the field is the same as in the SET function.
- **ConfigLen** – Output: pointer to the parameter size returned in bytes
- **pValues** – Output: pointer to memory containing the parameter

10.5 AP Network Configuration

The user must set the AP IP parameters, specifically the IP address, gateway address, DNS address, and network mask.

To set the IP addresses call:

```
sl_NetCfgSet(    unsigned char ConfigId ,
unsigned char ConfigOpt,
unsigned char ConfigLen,
unsigned char *pValues)
```

ConfigId – Should be set to SL_IPV4_AP_P2P_GO_STATIC_ENABLE (7)

ConfigOpt – Should be set to 1

ConfigLen – Should be the parameter size in bytes

pValues – Pointer to memory containing the parameter

This example shows how to configure IP, gateway and DNS addresses to 9.8.7.6 and the subnet to 255.255.255.0:

```
_NetCfgIpV4Args_t ipV4;
ipV4.ipV4         = 0x09080706;
ipV4.ipV4Mask        = 0xFFFFFF00;
ipV4.ipV4Gateway     = 0x09080706;

ipV4.ipV4DnsServer    = 0x09080706;
                                      sl_NetCfgSet(SL_IPV4_AP_P2P_GO_STATIC_ENABLE,
1
,sizeof(_NetCfgIpV4Args_t),
(unsigned char *)
&amp;ipV4);
```

Note: The changes will take affect after reset.

Default values:

ipV4 = 0xc0a80101; /* 192.168.1.1 */

defaultGatewayV4 = 0xc0a80101; /* 192.168.1.1 */

ipV4DnsServer = 0xc0a80101; /* 192.168.1.1 */

subnetV4 = 0xFFFFFF00; /* 255.255.255.0 */

10.6 DHCP Server Configuration

The user must enable the DHCP server and configure DHCP server parameters, DHCP addresses, and lease time.

To start the DHCP server call:

```
sl_NetAppStart(SL_NET_APP_DHCP_SERVER_ID);
```

Where *SL_NET_APP_DHCP_SERVER_ID* is 2.

To stop DHCP server call:

```
sl_NetAppStop(SL_NET_APP_DHCP_SERVER_ID);
```

Default value: DHCP server enabled.

To configure the DHCP parameters use the following API:

```
sl_NetAppSet( unsigned char AppId ,
unsigned char Option,
unsigned char OptionLen,
unsigned char *pOptionValue)
```

AppId – Should be set to SL_NET_APP_DHCP_SERVER_ID (2)

Option – Should be set to NETAPP_SET_DHCP_SRV_BASIC_OPT (0)

OptionLen – Should be the parameter size in bytes

pOptionValue – Pointer to memory containing the parameter

This example shows how to configure the parameters (starting IP address 9.8.7.1, ending IP address 9.8.7.5, lease time = 1000 seconds):

```
SlNetAppDhcpServerBasicOpt_t dhcpParams;

unsigned char outLen = sizeof(SlNetAppDhcpServerBasicOpt_t);
                                  dhcpParams.lease_time      = 1000;
                                          dhcpParams.ipv4_addr_start = 0x09080701;

dhcpParams.ipv4_addr_last  = 0x09080705;

sl_NetAppSet(SL_NET_APP_DHCP_SERVER_ID,
NETAPP_SET_DHCP_SRV_BASIC_OPT,
outLen,
(unsigned char*)
&amp;dhcpParams);
```

Default value:

lease_time = 24 * 3600; /* 24 hours */

ipv4_addr_start = 0xc0a80102; /* 192.168.1.2 */

ipv4_addr_last = 0xc0a801fe; /* 192.168.1.254 */

Note: The DHCP server addresses must be in the subnet of the AP IP address. The changes will take affect after reset.

10.7 Setting Device URN

To set the device name call:

```
sl_NetAppSet (SL_NET_APP_DEVICE_CONFIG_ID,
NETAPP_SET_GET_DEV_CONF_OPT_DEVICE_URN,
strlen(device_urn),
(unsigned char *) device_urn);

Where SL_NET_APP_DEVICE_CONFIG_ID         = 16
NETAPP_SET_GET_DEV_CONF_OPT_DEVICE_URN      = 0
```

Default value: mysimplelink

10.8 Asynchronous Events Sent to the Host

When a station is newly connected or disconnected to or from the AP, an event is sent to the host.

The event opcodes are:

SL_OPCODE_WLAN_STA_CONNECTED 0x082E

SL_OPCODE_WLAN_STA_DISCONNECTED 0x082F

The events include the following parameters:

Table 10-2. Event Parameters

Parameter	Bytes	Remarks
Peer device name	32	Relevant for P2P
Peer MAC address	6	
Peer device name length	1	
WPS device password ID	1	0 – not available
Own SSID	32	Relevant for P2P
Own SSID length	1	

Table 10-2. Event Parameters (continued)

Parameter	Bytes	Remarks
Padding	3	

When the IP address is released to a station, an event is sent to the host.

The event opcode is:

SL_OPCODE_NETAPP_IP_LEASED 0x 182C

The event includes the following parameters:

Table 10-3. Event Parameters

Parameter	Bytes	Remarks
IP address	4	
Lease time	4	In seconds
Peer MAC address	6	
Padding	2	

When an IP address is released by a station, an event is sent to the host.

The event opcode is:

SL_OPCODE_NETAPP_IP_RELEASED 0x 182D

The event includes the following parameters:

Table 10-4. Event Parameters

Parameter	Bytes	Remarks
IP address	4	
Peer MAC address	6	
Reason	2	0 – peer released the IP address 1 – peer declined to this IP address 2 – Lease time was expired

10.9 Example Code

An example code showing how to configure the AP WLAN parameters and network parameters (IP addresses and DHCP parameters) follows. WLAN parameters are also read back.

```
int main()
{
    int SockID;
    unsigned char outLen = sizeof(SlNetAppDhcpServerBasicOpt_t);
    unsigned char   channel, hidden, dtim, sec_type, wps_state, ssid[32],
password[65], country[3];
    unsigned short  beacon_int, config_opt, config_len;
    SlNetAppDhcpServerBasicOpt_t dhcpParams;
    _NetCfgIpV4Args_t ipV4;

    sl_Start(NULL, NULL, NULL);
    Sleep(100);

    // Set AP IP params

ipV4.ipV4         = SL_IPV4_VAL(192,168,1,1);

    ipV4.ipV4Gateway     = SL_IPV4_VAL(192,168,1,1);

    ipV4.ipV4DnsServer    = SL_IPV4_VAL(192,168,1,1);
    ipV4.ipV4Mask        = SL_IPV4_VAL(255,255,255,0);

    sl_NetCfgSet(    SL_IPV4_AP_P2P_GO_STATIC_ENABLE,
            1
            ,sizeof(_NetCfgIpV4Args_t),
            (unsigned char *)
&amp;ipV4);


    //Set AP mode
    sl_WlanSetMode(ROLE_AP);
    //Set AP SSID
    sl_WlanSet(SL_WLAN_CFG_AP_ID, WLAN_AP_OPT_SSID, strlen("cc_ap_test1"),
(unsigned char *)"cc_ap_test1");
    //Set AP country code
    sl_WlanSet(SL_WLAN_CFG_GENERAL_PARAM_ID,
WLAN_GENERAL_PARAM_OPT_COUNTRY_CODE, 2,(unsigned char *)"US");
       //Set AP Beacon interval
    beacon_int = 100;
    sl_WlanSet(SL_WLAN_CFG_AP_ID, WLAN_AP_OPT_BEACON_INT, 2, (unsigned char
*)
&amp;beacon_int);
    //Set AP channel
    channel = 8;
    sl_WlanSet(SL_WLAN_CFG_AP_ID, WLAN_AP_OPT_CHANNEL, 1, (unsigned char
*)
&amp;channel);
    //Set AP hidden/broadcast configuraion
    hidden = 0;
    sl_WlanSet(SL_WLAN_CFG_AP_ID, WLAN_AP_OPT_HIDDEN_SSID, 1, (unsigned char
*)
&amp;hidden);
    //Set AP DTIM period
    dtim = 2;
    sl_WlanSet(SL_WLAN_CFG_AP_ID, WLAN_AP_OPT_DTIM_PERIOD, 1, (unsigned char
*)
&amp;dtim);
        //Set AP security to WPA and password
    sec_type = SL_SEC_TYPE_WPA;
    sl_WlanSet(SL_WLAN_CFG_AP_ID, WLAN_AP_OPT_SECURITY_TYPE, 1, (unsigned
char *)
&amp;sec_type);
    sl_WlanSet(SL_WLAN_CFG_AP_ID, WLAN_AP_OPT_PASSWORD,
strlen("password123"), (unsigned char *)"password123");


    sl_Stop(100);
sl_Start(NULL, NULL, NULL);

    //Retrive all params to confirm setting

    //Get AP SSID
    sendLog("**********************AP parameters**********************\n");
    config_opt = WLAN_AP_OPT_SSID;
    config_len = MAXIMAL_SSID_LENGTH;
    sl_WlanGet(SL_WLAN_CFG_AP_ID,
&amp;config_opt , &amp;config_len, (unsigned
char*) ssid);
    sendLog("SSID: %s\n",ssid);
        //Get AP country code
    config_opt = WLAN_GENERAL_PARAM_OPT_COUNTRY_CODE;
    config_len = 3;
    sl_WlanGet(SL_WLAN_CFG_GENERAL_PARAM_ID,
&amp;config_opt, &amp;config_len,
(unsigned char*) country);
    sendLog("Country code: %s\n",country);
```

```
        //Get AP beacon interval
    config_opt = WLAN_AP_OPT_BEACON_INT;
    config_len = 2;
    sl_WlanGet(SL_WLAN_CFG_AP_ID,
&amp;config_opt, &amp;config_len, (unsigned char*) &amp;beacon_int);
    sendLog("Beacon interval: %d\n",beacon_int);
        //Get AP channel
    config_opt = WLAN_AP_OPT_CHANNEL;
    config_len = 1;
    sl_WlanGet(SL_WLAN_CFG_AP_ID,
&amp;config_opt, &amp;config_len, (unsigned char*) &amp;channel);
    sendLog("Channel: %d\n",channel);
    //Get AP hidden configuraion
    config_opt = WLAN_AP_OPT_HIDDEN_SSID;
    config_len = 1;
    sl_WlanGet(SL_WLAN_CFG_AP_ID,
&amp;config_opt, &amp;config_len, (unsigned char*) &amp;hidden);
    sendLog("Hidden: %d\n",hidden);
    //Get AP DTIM period
    config_opt = WLAN_AP_OPT_DTIM_PERIOD;
    config_len = 1;
      sl_WlanGet(SL_WLAN_CFG_AP_ID,
&amp;config_opt, &amp;config_len, (unsigned char*) &amp;dtim);
    sendLog("DTIM period: %d\n",dtim);
       //Get AP security type
    config_opt = WLAN_AP_OPT_SECURITY_TYPE;
    config_len = 1;
    sl_WlanGet(SL_WLAN_CFG_AP_ID,
&amp;config_opt, &amp;config_len, (unsigned char*) &amp;sec_type);
    sendLog("Security type: %d\n",sec_type);
        //Get AP password
    config_opt = WLAN_AP_OPT_PASSWORD;
    config_len = 64;
    sl_WlanGet(SL_WLAN_CFG_AP_ID,
&amp;config_opt, &amp;config_len, (unsigned char*)
password);
    sendLog("Password: %s\n",password);
        //Get AP WPS state
    config_opt = WLAN_AP_OPT_WPS_STATE;
    config_len = 1;
    sl_WlanGet(SL_WLAN_CFG_AP_ID,
&amp;config_opt, &amp;config_len, (unsigned char*) &amp;wps_state);

       // Set AP DHCP params
    //configure dhcp addresses to: 192.168.1.10 - 192.168.1.20, lease time
4096 seconds
    dhcpParams.lease_time = 4096;
    dhcpParams.ipv4_addr_start =  SL_IPV4_VAL(192,168,1,10);
    dhcpParams.ipv4_addr_last = SL_IPV4_VAL(192,168,1,20);
    outLen = sizeof(SlNetAppDhcpServerBasicOpt_t);
    sl_NetAppStop(SL_NET_APP_DHCP_SERVER_ID);
    sl_NetAppSet(SL_NET_APP_DHCP_SERVER_ID, NETAPP_SET_DHCP_SRV_BASIC_OPT,
outLen, (unsigned char*)
&amp;dhcpParams);
    sl_NetAppStart(SL_NET_APP_DHCP_SERVER_ID);

        // Get AP DHCP params
     sl_NetAppGet(SL_NET_APP_DHCP_SERVER_ID, NETAPP_SET_DHCP_SRV_BASIC_OPT,
&amp;outLen, (unsigned char*)&amp;dhcpParams);

}
```

Chapter 11

SWRU368–June 2014

Peer to Peer (P2P)

Topic Page

11.1 General Description

11.1.1 Scope

P2P mode in CC3100 lets the device connect to other devices without the need for an AP by inheriting the entire station of AP attributes.

11.1.2 Wi-Fi Direct Advantage

- Provides an easy and convenient manner to share, show, print, and synchronize content, whether at home, in public places, while traveling, or at work.
- Provides new connectivity scenarios that include any Wi-Fi devices the user already owns.
- Eliminates the need for routers and Bluetooth for applications that do not rely on low power.
- Delivers an industry-wide peer-to-peer solution based on broadly deployed Wi-Fi technologies.
- Requires only one of the Wi-Fi devices to be compliant with Wi-Fi Direct to establish a peer-to-peer connection that transfers data directly between other devices.
- Uses peer-to-peer opportunistic power-save, allowing a low-power link for both group owner and client.

11.1.3 Support and Abilities of Wi-Fi Direct in CC3100

- P2P configuring device name, device type, listen and operation channels
- P2P device discovery (FULL/SOCIAL)
- P2P negotiation with all intents (0 to 15)
- P2P negotiation Initiator policy – Active / Passive / Random Back Off
- P2P WPS method push-buttonand pin code (keypad and display)
- P2P establish as client role:
- P2P device can join an existing P2P group.
- P2P device can invite to reconnect a persistent group (fast-connect).
- P2P establish as group owner role:
- P2P group owner can accept a join request.
- P2P persistent group owner can respond to invite requests.
- P2P group remove
- P2P connect-disconnect-connect transition, also between different roles (for example GO-CL-GO)
- P2P client legacy PS and NoA support
- Separate IP configuration for P2P role
- Separate Net-Apps configuration on top of P2P-CL/GO role

11.1.4 Limitations

- No service discovery supported.
- No GO-NoA supported.
- Smart configuration for P2P device entry is limited to 15 entries, and is dynamically allocated rather than optimized as a station scan single entry.
- No autonomous group support for the user, although the mode is internally supported and can be opened to the user.
- P2P group owner mode supports single peer (client) connected (similar to AP).

• P2P Find in profile search and connection is infinite, meaning if the remote device is not found the search will continue indefinitely.

11.2 P2P APIs and Configuration

P2P configuration is done by using the host driver APIs that control the CC3100. Different options and modes of P2P configuration can be accessed by using specific APIs.

The configuration is divided into these sections:

- Configure P2P global parameters.
- Configure P2P policy.
- Configure P2P profiles policy.
- Configure manual connection.
- Configure a search for P2P devices and use these devices.
- P2P events.

The next section explains how to configure P2P and how to use its options. Note that:

- Configuration is flushed and can be done only once.
- Not all APIs need to be used. Default parameters are used in the absence of a user configuration.
- All set APIs have a GET operation.

11.2.1 Configuring P2P Global Parameters

This section shows how to configure the device to be in a state of P2P and set its general parameters.

11.2.1.1 Set P2P role

Set the device to P2P mode using an API:

```
sl_WlanSetMode(ROLE_P2P)
```

This API puts the device in P2P mode. All other P2P configurations will not be effected until entering P2P mode.

11.2.1.2 Set P2P Network Configuration

Configure network configuration used by the P2P API:

- P2P client (same API as station):
  - Static IP:

    ```
    sl_NetCfgSet(SL_IPV4_STA_P2P_CL_STATIC_ENABLE,1, sizeof(_NetCfgIpV4Args_t),(unsigned char
    *)&amp;ipV4)
    ```

  - DHCP client:

    ```
    sl_NetCfgSet(SL_IPV4_STA_P2P_CL_DHCP_ENABLE,1,1,&amp;val);
    ```

- P2P GO (same API as AP):
  - GO own static IP:

    ```
    sl_NetCfgSet(SL_IPV4_AP_P2P_GO_STATIC_ENABLE,1, sizeof(_NetCfgIpV4Args_t),(unsigned char
    *)&amp;ipV4)
    ```

This API sets the network configuration used when the P2P role is set.

11.2.1.3 Set P2P Device Name

The following macro sets the P2P device name. Setting a unique P2P device name for every individual device is necessary, because the connection is based on device-name.

Default: TI_SIMPLELINK_P2P_xx (xx = random two characters)

API:

```
sl_NetAppSet (SL_NET_APP_DEVICE_CONFIG_ID,
NETAPP_SET_GET_DEV_CONF_OPT_DEVICE_URN,
strlen(device_name),
(unsigned char *) device_name);
```

11.2.1.4 Set P2P Device Type

The following macro sets the P2P device type. The device type allows P2P discovery parameters to recognize the device.

Default: 1-0050F204-1

API:

```
sl_WlanSet(SL_WLAN_CFG_P2P_PARAM_ID,
        WLAN_P2P_OPT_DEV_TYPE,
           dev_type_len, dev_type);
```

11.2.1.5 Set P2P Listen and Operation Channels

The following macro sets the P2P operation and listen channels. The listen channel is used for the discovery state and can be 1, 6, or 11. The device will be in this channel when waiting for P2P probe requests. The operation channel is used only by the GO. The GO will move to this channel after the negotiation phase.

Default: Random between channels 1,6, or 11

API:

```
sl_WlanSet(SL_WLAN_CFG_P2P_PARAM_ID, WLAN_P2P_OPT_CHANNEL_N_REGS, 4, channels);
```

Note: Regulator domain class should be 81 in 2.4G.

For example:

```
unsigned char  channels [4];
channels [0] = (unsigned char)11; // listen channel
channels [1] = (unsigned char)81; // listen regulatory class
channels [2] = (unsigned char)6;  // oper channel
channels [3] = (unsigned char)81; // oper regulatory class
sl_WlanSet(SL_WLAN_CFG_P2P_PARAM_ID, WLAN_P2P_OPT_CHANNEL_N_REGS,  4, channels);
```

11.2.2 Configuring P2P Policy

This section depicts the P2P policy configuration, including two more P2P working mode options given by the CC3100.

- P2P intent value option – The P2P role of the device (client, GO, or don't care).
- Negotiation initiator option – Value used during P2P negotiation to indicate which side will initiate the negotiation, and which side will passively wait for the remote side to send negotiation and then respond.

11.2.2.1 Configure P2P Intent Value and Negotiation Initiator

This configuration uses macro **SL_P2P_POLICY**, the second parameters sent to function **sl_WlanPolicySet**. For the intent value, three defines options can be used:

- **1SL_P2P_ROLE_CLIENT** (intent 0): Indicates that the device is forced to be P2P client.
- **SL_P2P_ROLE_NEGOTIATE** (intent 7): Indicates that the device can be either P2P client or GO, depending on the P2P negotiation tie-breaker. This is the system default.
- **SL_P2P_ROLE_GROUP_OWNER** (intent 15): Indicates that the device is forced to be P2P GO.

For negotiation initiator, three defines options can be used:

- **SL_P2P_NEG_INITIATOR_ACTIVE**: When the remote peer is found after discovery, the device immediately sends negotiation requests to the peer device.
- **SL_P2P_NEG_INITIATOR_PASSIVE**: When the remote peer is found after discovery, the device passively waits for the peer to start the negotiation before responding.

- **SL_P2P_NEG_INITIATOR_RAND_BACKOFF**: When the remote peer is found after discovery, the device triggers a random timer (1 to 7 seconds). The device waits passively for the peer to negotiate during this period. If the timer expires without negotiation, the device immediately sends negotiation requests to the peer device. This is the system default as two CC3100 devices do not require any negotiation synchronization.

This configuration should be used when working with two CC3100 devices. The user may not have a GUI to start the negotiation, thus this option is offered to prevent the simultaneous negotiation of both devices after discovery.

API:

```
sl_WlanPolicySet(SL_POLICY_P2P,
                 SL_P2P_POLICY(Intent value, negotiation initiator)//macro,
&amp;policyVal,
                 0
                 );
```

For example:

```
sl_WlanPolicySet(SL_POLICY_P2P,
                                 SL_P2P_POLICY(SL_P2P_ROLE_NEGOTIATE,
                                                            SL_P2P_NEG_INITIATOR_RAND_BACKOFF
                                                             ) //macro,
&amp;policyVal,
                                 0
                                 );
```

11.2.3 Configuring P2P Profile Connection Policy

This section discusses the profile connection policy. This policy lets the system connect to a peer without a reset or disconnect operation by the remote peer.

The mechanism describes how the device uses these profiles in relation to P2P automatic connection.

This configuration uses the macro **SL_CONNECTION_POLICY**, the second parameter sent to function **sl_WlanPolicySet**.

There are four connection policy options:

- **Auto Start** – As in STA mode, if the device is not connected it starts P2P find to search for all P2P profiles configured on the device. If at least one candidate is found, the device tries to connect to it. If more than one device is found, the best candidate according to the profiles parameter is chosen.
- **Fast Connect** – This option creates a fast connection after reset, but is dependant on the last connection state. This option has no meaning if no previous connection was performed by the device, because only the last connection parameters are used by the fast-connection option.
  - If the device was a P2P client in its last connection (before a reset or remote disconnect operation), then following reset it will send a p2p_invite to the previously connected GO to perform fast-reconnection.
  - If the device was P2P GO in its last connection (before reset or remote disconnect operation), then following reset it will reinvoke the p2p_group_owner and wait for the previous connected peer to reconnect.
- **OpenAP** – Not relevant for P2P mode
- **AnyP2P policy** – Policy value that makes a connection to any P2P peer device found during discovery. This option does not need a profile. Relevant for negotiation with push-button only.

Each option should be sent or set in this macro as true or false. More than one option can be used; for example, the user can set both the auto-start and the fast connect options to true.

API:

```
sl_WlanPolicySet(SL_POLICY_P2P,
                                 SL_CONNECTION_POLICY(auto start, fast connect,openAp,AnyP2p)
//macro,
&amp;policyVal,
                                 0
                                 );
```

For example:

```
sl_WlanPolicySet(SL_POLICY_P2P,
                                  SL_CONNECTION_POLICY(true, true, false, false) //macro,
&amp;policyVal,
                                  0
                                  );
```

11.2.4 Discovering Remote P2P Peers

This section shows how to start a P2P search and discovery, and how to see the remote P2P devices that are discovered. Discovery is used for:

- Scan and find nearby devices, because P2P connection is based on the remote device name published during discovery phase
- Manually connect by host commands (not using existing profiles)
- Discover the remote peers in the neighborhood and then configure willing profiles, if the neighborhood is unknown and the user wants to set P2P profiles in the system.

11.2.4.1 How to Start P2P Discovery

A scan policy is needed to start the P2P find and discover remote P2P peers. Setting the scan policy to P2P performs a full P2P scan.

The setting of the scan policy should be under the P2P role. P2P discovery is performed as part of any connection, but can also be activated using the SCAN_POLICY.

API:

```
SL_WlanPolicySet(SL_POLICY_SCAN, 1 /*enable scan*/, interval, 0)
```

The second parameter enables the P2P scan operation, and the interval indicates the waiting time between P2P find cycles.

11.2.4.2 How to See/Get P2P Remote Peers (Network P2P List)

There are two ways to see and get P2P remote devices discovered during a P2P find and search operation:

- Listening to the event **SL_WLAN_P2P_DEV_FOUND_EVENT**
- Calling to API **sl_WlanGetNetworkList**

SL_WLAN_P2P_DEV_FOUND_EVENT: This event is sent to each remote P2P that is found. It contains the MAC address, the name, and the name's length of the remote device. By listening to this event the user can find each remote P2P in the neighborhood.

Sl_WlanGetNetworkList: By calling to this API the user gets a list of remote peers found and saved in the device cache. This API is also used in station mode.

API:

```
sl_WlanGetNetworkList(unsigned char Index, unsigned char Count,
                      Sl_WlanNetworkEntry_t *pEntries)
```

Index – Indicates to which index in the list tables the P2P devices will return.

Count - Shows how many peer devices should be returned (if existed).

pEntries – The results are entered in to it. It is allocated by the user.

11.2.5 Negotiation Method

The next sections show how to make a P2P connection manually or automatically by profile, and the negotiation method before the WPS connection.

As stated in Section 11.2.3, the negotiation starts according to the intent and negotiation initiator parameters, but other parameters should be configured to finish this step successfully. These parameters influence the negotiation method and are supplied during the manual connection API command that comes from the host or when setting the profile for automatic connection. The negotiation method is done by the device without user interference.

There are two P2P negotiation methods to indicate the WPS phase that follows the negotiation:

- P2P push-button connection – Both sides negotiate with PBC method. Define **SL_SEC_TYPE_P2P_PBC**.
- P2P pin code connection – Divided to two options. PIN_DISPLAY looks for a pin to be written by its remote P2P. PIN_KEYPAD sends a pin code to its remote P2P.
  - Define **SL_SEC_TYPE_P2P_PIN_KEYPAD**.
  - Define **SL_SEC_TYPE_P2P_ PIN_DISPLAY**.

Note: If no pin code is entered, the NWP auto-generates the pin code from the device MAC using the following method:

1. Take the 7 ISB decimal digits in the device MAC address, and add checksum of those 7 digits to the LSB (total 8 digits). For example, if MAC is 03:4A:22:3B:FA:42
2. Convert to decimal: ….:059:250:066
3. Seven ISB decimal digits are: 9250066
4. WPS Pin Checksum digit: 2
5. Default pin code for this MAC: 92500662

There are two options to configure the negotiation method:

- Setting the value in secParams struct and sending it as a parameter through the manual connection command.
  - For push-button: secParams.Type = **SL_SEC_TYPE_P2P_PBC**
  - For pin code keypad: secParams.Type = **SL_SEC_TYPE_ PIN_KEYPAD secParams.Key** = 12345670
  - For pin code diaplay: secParams.Type = **SL_SEC_TYPE_ PIN_ DISPLAY secParams.Key** = 12345670
- Sending the negotiation method defines and key as parameters through the P2P profile configuration.

11.2.6 Manual P2P Connection

After finding a remote device by getting event **SL_WLAN_P2P_DEV_FOUND_EVENT** or by calling to **get_networkList** API, there is an option to start a connection by using a command originating from the host. This command performs immediate P2P discovery. Once the remote device is found, the negotiation phase is started according to the negotiation initiator policy, method, and intent selected.

This connection is not flashed, so in case of disconnection or reset, a reconnection is done only if the fast-connect policy is on. This connection is stronger than a connection made through profiles, as a P2P connection already exists in the system. The current connection is disconnected in favor of the manual connection.

API:

```
sl_WlanConnect(char* pName, int NameLen, unsigned char *pMacAddr,
               SlSecParams_t* pSecParams ,
               SlSecParamsExt_t* pSecExtParams)
```

pName – The name of the remote device which is known to the user after getting event **SL_WLAN_P2P_DEV_FOUND_EVENT** or by calling to **get_networkList** API.

NameLen – The length of pName

pMacAddr – The option to connect to a remote P2P according to its BSSID. Use {0,0,0,0,0,0} to connect according to MAC address.

pSecParams – See Section 11.2.5.

pSecExtParams – Value should be zero.

For example:

```
sl_WlanConnect("my-tv-p2p-device, 17, {0,0,0,0,0,0},&amp; pSecParams ,0);
```

11.2.7 Manual P2P Disconnection

A manual disconnect option lets the user make a disconnection from its peer through a host command. This command performs a P2P group remove of the current active role, whether it is p2p-device, p2p-group-owner, or p2p-client.

API:

```
sl_WlanDisconnect();
```

11.2.8 P2P Profiles

Profile configuration makes an automatic P2P connection after a reset or disconnection from the remote peer device. This command stores the P2P remote device parameters in flash as a new profile along with profile priority. These profiles are similar to station profiles and have the same automatic connection behavior. The connection depends on the profile policy configuration.

If auto-start policy is on, a P2P discovery is performed. If one or more of the remote devices that are found are adapted to the profiles, a negotiation phase is started according to the negotiation initiator policy, method, and intent selected. The highest priority profile is chosen.

If fast-connect policy is on and there already was a connection, then after a reset or disconnection from the remote peer a fast connection starts according to the last connection parameters, without consulting the P2P profile list. If a manual connection is sent, then the profile connection is stopped by a disconnect command, and manual connection is initiated.

To add a profile with the push-button negotiation method, call to:

```
sl_WlanProfileAdd(char* pName, int NameLen, unsigned char *pMacAddr,
                  SlSecParams_t* pSecParams , SlSecParamsExt_t*
                  pSecExtParams, unsigned long  Priority,
                  unsigned long  Options)
```

An example of adding a profile with the push-button negotiation method:

```
sl_WlanProfileAdd(    SL_SEC_TYPE_P2P_PBC,
            remote_p2p_device,
            strlen(remote_p2p_device),
            bssidEmpty,
            0/*Priority*/,0,0,0);
```

An example of adding a profile with the keypad negotiation method:

```
sl_WlanProfileAdd(    SL_SEC_TYPE_P2P_PIN_ DISPLAY,
            remote_p2p_device,
            strlen(remote_p2p_device),
            bssidEmpty,
            0/*Priority*/,key,8/*keylen*/,0);
```

11.2.9 Removing P2P Profiles

To delete a specific profile or all profiles use the following API:

```
sl_WlanProfileDel(profile_index/*WLAN_DEL_ALL_PROFILES for all profiles*/)
```

11.3 P2P Connection Events

This section describes the P2P connection events. These events are sent by the device during connection and the user decides how to handle them.

• **SL_WLAN_P2P_NEG_REQ_RECEIVED_EVENT** – This event is sent if the negotiation is ended successfully. It contains:

- Device peer MAC address

- Device peer name
- Length of device peer name
- Device peer WPS password ID

• **SL_WLAN_CONNECTION_FAILED_EVENT** – This event is sent if the connection fails, with the failure reason. It contains:

- Failure reason

• **SL_WLAN_CONNECT_EVENT** – This event is shared by P2P client and station. Indicates that the P2P connection has ended successfully and that the device is a P2P client. It contains:

- Remote device parameters

• **SL_WLAN_CONNECT_EVENT** – This event is shared for P2P GO and AP. Indicates that the P2P connection has ended successfully and that the device is P2P GO. It contains:

- Remote device parameters
- Information about the group owner parameters

11.4 Use Cases and Configuration

This section describes common P2P use cases, their meanings, and how to configure them.

11.4.1 Case 1 – Nailed P2P Client Low-Power Profile

Device is a P2P client by automatic connection, using fast connection after each reset or disconnection. The device stores the parameters of the last connection.

At least one profile should be configured in the system.

On fast connect, add the device to P2P supplicant to avoid find.

Send P2P-Invite to remote persistent GO with network parameters.

Note: In case fast connection failed, fall back to profile P2P find and perform the regular connection method (join or full negotiation).

1. Configure P2P global parameters.
2. Configure P2P profile policy with **SL_P2P_ROLE_CLIENT (intent 0)**, and with **SL_P2P_NEG_INITIATOR_RAND_BACKOFF (negotiation initiator don't care)**. **SL_P2P_POLICY(SL_P2P_ROLE_CLIENT, SL_P2P_NEG_INITIATOR_RAND_BACKOFF)**
3. Configure P2P profile policy connection with auto-start and fast connection. **SL_CONNECTION_POLICY(true, true, false, false)**
4. Configure P2P profile according to the willing parameters of the remote device such as name, MAC address, and so forth.
5. If the peer parameters are unknown, operate a discover P2P operation, find P2P peers, and then configure profiles.

11.4.2 Case 2 – Mobile Client Low-Power Profile

Device is a P2P client by automatic connection, with no fast connection after each reset or disconnection. The device does not store the parameters of the last connection.

At least one profile should be configured in the system.

As in the first case, perform a profile P2P find with stored networks, examine the results and send a negotiation request according to remote device GO capabilities (group owner).

1. Configure P2P global parameters.
2. Configure P2P profile policy with **SL_P2P_ROLE_CLIENT (intent 0)**, and with **SL_P2P_NEG_INITIATOR_RAND_BACKOFF (negotiation initiator don't care)**. **SL_P2P_POLICY(SL_P2P_ROLE_CLIENT, SL_P2P_NEG_INITIATOR_RAND_BACKOFF)**

3. Configure P2P profile policy connection with auto-start and fast connection. **SL_CONNECTION_POLICY(true, false, false, false)**
4. Configure P2P profile according to the willing parameters of the remote device such as name, MAC address, and so forth
5. If the peer parameters are unknown, operate a discover P2P operation, find P2P peers, and then configure profiles.

11.4.3 Case 3 – Nailed Center Plugged-in Profile

Device is a P2P GO by automatic connection, with fast connection after each reset or disconnection. The device stores the parameters of the last connection.

At least one profile should be configured in the system.

- HARD RESET ONLY – Launch Group (as GO) with previous persistent network parameters stored in fast-connect, set WPS method and set timer.
- PEER DISCONNECT – Set WPS method and set timer.
- WPS timer expired with no peers connected – Tear down group, go back and perform a profile P2P find with stored network. Examine the results and initiate a negotiation request with the policy parameters (negotiate with intent = 15 results as GO).

1. Configure P2P global parameters.
2. Configure P2P profile policy with **SL_P2P_ROLE_GROUP_OWNER (intent 15)**, and with **SL_P2P_NEG_INITIATOR_RAND_BACKOFF (negotiation initiator don't care)**. **SL_P2P_POLICY(SL_P2P_ROLE_GO, SL_P2P_NEG_INITIATOR_RAND_BACKOFF)**
3. Configure P2P profile policy connection with auto-start and fast connection. **SL_CONNECTION_POLICY(true, true, false, false)**
4. Configure P2P profile according to the willing parameters of the remote device such as name, MAC address, and so forth.
5. If the peer parameters are unknown, operate a discover P2P operation, find P2P peers, and then configure profiles.

11.4.4 Case 4 – Mobile Center Profile

Perform a group formation and launch as GO (nonpersistent) by automatic connection with no fast connection after each reset or disconnection. The device does not store the parameters of the last connection. At least one profile should be configured in the system.

- DISCONNECT – Tear down the group immediately.
- Hard reset or after tearing down – Perform a profile P2P find with the stored network, examine the results and initiate a negotiation request with the policy parameters (negotiate with intent = 15 results as GO).

1. Configure P2P global parameters.
2. Configure P2P profile policy with **SL_P2P_ROLE_GROUP_OWNER (intent 15)**, and with **SL_P2P_NEG_INITIATOR_RAND_BACKOFF (negotiation initiator don't care)**. **SL_P2P_POLICY(SL_P2P_ROLE_GO, SL_P2P_NEG_INITIATOR_RAND_BACKOFF)**
3. Configure P2P profile policy connection with auto-start and fast connection. **SL_CONNECTION_POLICY(true, false, false, false)**
4. Configure P2P profile according to the willing parameters of the remote device such as name, MAC address, and so forth.
5. If the peer parameters are unknown, operate a discover P2P operation, find P2P peers, and then configure profiles.

11.4.5 Case 5 – Mobile General-Purpose Profile

Perform a group formation and launch as GO (nonpersistent) or CL, without storing any parameter. At least one profile should be configured in the system.

- DISCONNECT – If GO, then tear down the group immediately.
- Hard reset or after tearing down – Perform a profile P2P find with the stored network, examine the results and initiate a negotiation request with the policy parameters (negotiate with intent = 7 results as GO or CL).

1. Configure P2P global parameters.
2. Configure P2P profile policy with **SL_P2P_ROLE_NEGOTIATE (intent 7)**, and with **SL_P2P_NEG_INITIATOR_RAND_BACKOFF (negotiation initiator don't care)**. **SL_P2P_POLICY(SL_P2P_ROLE_GO, SL_P2P_NEG_INITIATOR_RAND_BACKOFF)**
3. Configure P2P profile policy connection with auto-start and fast connection. **SL_CONNECTION_POLICY(true, false, false, false)**
4. Configure P2P profile according to the willing parameters of the remote device such as name, MAC address, and so forth.
5. If the peer parameters are unknown, operate a discover P2P operation, find P2P peers, and then configure profiles.

11.5 Example Code

The following is sample code of a simple profile connection, configuring the device to act as a P2P client device using a profile connection with a known remote GO device name (without MAC), using push-button method.

```
unsigned char val = 1;
unsigned char policyVal;
unsigned char my_p2p_device[33];
unsigned char *remote_p2p_device = "Remote_GO_Device_XX";
unsigned char bssidEmpty[6] = {0,0,0,0,0,0};
sl_Start(NULL, NULL, NULL);
//Set P2P as active role
sl_WlanSetMode(3/*P2P_ROLE*/);
//Set P2P client dhcp enable (assuming remote GO running DHCP server)
sl_NetCfgSet(SL_IPV4_STA_P2P_CL_DHCP_ENABLE,1,1,
&amp;val);
//Set Device Name
strcpy(my_p2p_device,"jacky_sl_p2p_device");
sl_NetAppSet (SL_NET_APP_DEVICE_CONFIG_ID,
    NETAPP_SET_GET_DEV_CONF_OPT_DEVICE_URN, strlen(my_p2p_device),
      (unsigned char *) my_p2p_device);

//set connection policy Auto-Connect
sl_WlanPolicySet( SL_POLICY_CONNECTION,
SL_CONNECTION_POLICY(1/*Auto*/,0/*Fast*/,
0/*OpenAP*/,0/*AnyP2P*/),
&amp;policyVal, 0 /*PolicyValLen*/
                );

//set P2P Policy - intent 0, random backoff
sl_WlanPolicySet( SL_POLICY_P2P,
              SL_P2P_POLICY(SL_P2P_ROLE_CLIENT/*Intent 0 - Client*/,
    SL_P2P_NEG_INITIATOR_RAND_BACKOFF/*Negotiation initiator – random backoff*/),
&amp;policyVal,0 /*PolicyValLen*/
 );

sl_WlanProfileAdd(
        SL_SEC_TYPE_P2P_PBC,
        remote_p2p_device,
        strlen(remote_p2p_device),
        bssidEmpty,
        0, //unsigned long   Priority,
        0,//unsigned char   *pKey,
        0,//unsigned long   KeyLen,
        0//unsigned long   Options)
        );
```

```
sl_Stop(1);
sl_Start(NULL, NULL, NULL);
```

Chapter 12

SWRU368–June 2014

HTTP Server

Topic **Page**

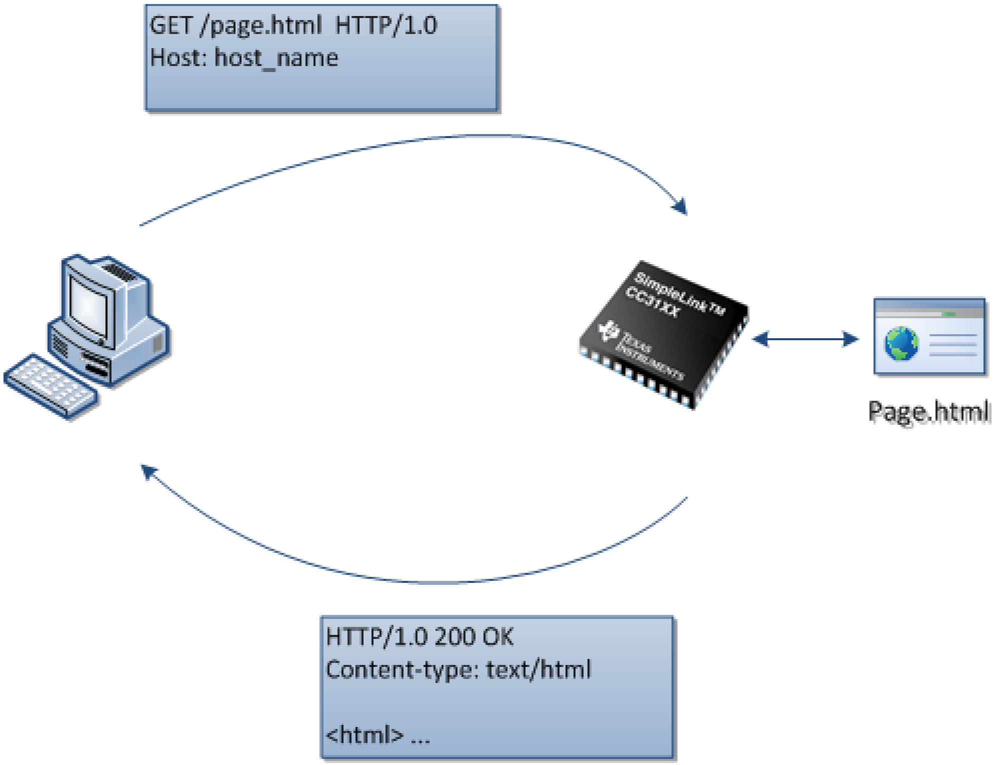
12.1 Overview

The HTTP web server allows end-users to remotely communicate with the SimpleLink device using a standard web browser.

The HTTP web server enables the following functions:

- Device configuration
- Device status and diagnostic
- Application-specific functionality

HTTP stands for Hypertext Transfer Protocol. HTTP is a client/server protocol used to deliver hypertext resources (HTML web pages, images, query results, and so forth) to the client side. HTTP works on top of a predefined TCP/IP socket, usually port 80.



```
GET /page.html  HTTP/1.0
Host: host_name
```

```
HTTP/1.0 200 OK
Content-type: text/html

<html> ...
```

Figure 12-1. HTTP GET Request

The SimpleLink HTTP server handles the HTTP request. The server listens on the HTTP socket (default is 80). According to the request type, such as HTTP GET or HTTP POST, the server handles the request URI resource and content. The server then composes a correct HTTP response and returns it to the client.

The SimpleLink server communicates with the serial flash file system, which hosts the web page files. The files are saved in the serial flash under their own filename. Filenames can include the full path to achieve a directory structure-like behavior. For security purposes, the ability of the web server to access the file system is limited to the following root folders:

(a) **www/**

(b) **www/safe/**

Important: One of these two root folders should be prefixed to the filename when downloading files to the file system.

The www/safe/ folder is also used in Force AP mode. For details, see Section 12.5.

The SimpleLink server holds an internal set of web pages statically, on ROM, which serves as an internal default web page. The internal web page provides out-of-the-box device configuration, status, and basic analysis tools.

The SimpleLink server also has an interface to the host (through the SimpleLink commands and events dispatcher module) in order to implement the user API.

12.2 HTTP GET Processing

12.2.1 Overview

When the HTTP web server gets an HTTP GET request, it first checks the resource URN of the HTTP request for the name of the requested resource. The server then checks if this resource exists in the serial flash. If the resource exists, the server returns it as part of the HTTP response.

If the server does not find the requested resource on the flash, the server checks if this resource is one of the files of the internal web page in the ROM. If the resource is a file of the web page, the server returns the resource; if not, an HTTP error message is sent (HTTP/1.0 404 Not Found response).

12.2.2 Default Web Page

In case the HTTP GET does not contain any resource name (for example, /), the HTTP web server looks for the following filenames in this order:

1. Index.html
2. Main.html

The server first checks them on the serial flash, and then in the internal web page.

12.2.3 SimpleLink GET Tokens

To support HTML pages with data that is generated dynamically by the server, the HTTP web server supports a set of predefined tokens replaced on the fly by the server with dynamically generated content.

- The tokens have a fixed length of 10 characters.
- The token prefix is identical to all tokens and seven characters in length.
- The token prefix is: **__SL_G_**
- A typical token should look like this: **__SL_G_XYZ** where XYZ is one of the predefined tokens.

For a complete list of the predefined tokens, see Section 12.10.

12.2.4 User-Defined Tokens

The user can define new tokens that are not known by the HTTP web server.

The token should follow the same rules as the predefined tokens:

- The tokens have a fixed length of 10 characters.
- The token prefix is identical to all tokens and seven characters in length.
- The token prefix is: **__SL_G_**
- A typical token should look like this: **__SL_G_XYZ** where XYZ is user-defined and can contain any character or number.

If the HTTP web server scans the HTML file and finds a token that is not in the predefined list of tokens, the server generates a **get_token_value** asynchronous event with the token name, requesting the token value from the host.

The host should respond with a **send_token_value** command, with the token value. The HTTP web server will use this token value and return it to the client.

Important: The maximum length of the token value is 64 bytes.

If the host is not responding to the **get_token_value** request, the server implements a time-out of two seconds. After the time-out, the Not Available string is used as the token value (and is eventually be displayed by the browser).

Important: To prevent user tokens from colliding with the internal tokens, use tokens with the following structure: **__SL_G_UXX**, where XX can be any character or number.

12.2.5 HTML Sample Code with Dynamic HTML Content

In the following code the token __SL_G_N.A will be replaced by the actual IP address:

<tr>

<td dir=LTR> IP Address: </td>

<td dir=LTR>__SL_G_N.A </td>

</tr>

12.3 HTTP POST Processing

12.3.1 Overview

The client uses HTTP POST requests to update data in the server. The SimpleLink HTTP web server supports HTML forms with content type of application/x-www-form-urlencoded. The POST information that is sent by the browser includes the form action name and one or more pairs of variable name and variable value.

12.3.2 SimpleLink POST Tokens

In SimpleLink the variable name in the POST should follow the same rules of the GET tokens.

- The tokens have a fixed length of 10 characters.
- The token prefix is identical to all tokens and seven characters in length.
- The token prefix is: **__SL_P_**
- A typical token should look like this: **__SL_P_XYZ** where XYZ is user-defined and can contain any character or number.

When the HTTP web server receives an HTTP POST request, the server first checks the form action name to understand if this POST should be handled internally. The server then goes over the parameters list, and checks each variable name to see if it matches one of the known predefined tokens. If the variable names match the predefined tokens, the server processes the values.

12.3.3 SimpleLink POST Actions

In SimpleLink the variable name in the POST should follow the same rules of the GET tokens.

- The tokens have a fixed length of 10 characters.
- The token prefix is identical to all tokens and seven characters in length.
- The token prefix is: **__SL_P_**
- A typical token should look like this: **__SL_P_XYZ** where XYZ is user-defined, and can contain any character or number.

When the HTTP web server receives an HTTP POST request, the server first checks the form action name to understand if this POST should be handled internally. The server then goes over the parameters list, and checks each variable name to see if it matches one of the known predefined tokens. If the variable names match predefined tokens, the server processes the value.

12.3.4 SimpleLink POST Actions

There are two types of POST operations: simple actions and complex actions.

In simple POST actions, the server processes the list of POST parameters and saves the new information (such as set domain name). In this case, the action value is not important and the server does not identify it.

In complex POST actions, the server should gather all the needed POST parameters and then trigger a specific action (such as add profile). In this case, the action is identified by the action value in the HTTP POST command.

12.3.5 User-Defined Tokens

User-defined tokens are used in POST requests to send information to the host.

If the HTTP web server receives an HTTP POST request that contains tokens that are not in the predefined list of tokens, the server generates a **post_token_value** asynchronous event to the host, which will contain the following information: form action name, token name, and token value. The host can then process the required information.

Important: To prevent user tokens from colliding with the internal tokens, use tokens with the following structure: **__SL_P_UXX**, where XX can be any character or number.

12.3.6 Redirect after POST

In SimpleLink, the user redirects the browser to a different web page after the POST submission.

After the POST is processed, the HTTP web server checks the action-URI received in the POST request. If the action-URI includes a valid web page in the SimpleLink (serial flash or ROM), the server issues an HTTP 302 Found response with the action-URI value, to perform redirection.

If the action-URI does not contain a valid web page, the HTTP web server issues an HTTP 204 No content response and the browser remains on the current web page.

12.3.7 HTML Sample Code with POST and Dynamic HTML Content

In the following POST example, once the user clicks on the submit button the POST request includes the profiles_add.html as the action resource and the variables __SL_P_P.A and __SL_P_P.B with the values that the user requested.

```
<form method="POST" name="SimpleLink Configuration" action="profiles_add.html">
<tr>
<td dir=LTR> SSID: </td>
<td dir=LTR><input type="text" maxlength="32" name="__SL_P_P.A" /> Enter any value of up to 32 characters</td>
</tr>
<tr>
<td dir=LTR> Security Type: </td>
<td dir=LTR> <input type="radio name="__SL_P_P.B" value="0" checked />Open
<input type="radio" name="__SL_P.P.B" value="1" />WEP
<input type="radio" name="__SL_P_P.B" value="2" />WPA1
<input type="radio" name="__SL_P_P.B" value="3" />WPA2 </td>
</tr>
<tr>
<td colspan=2 align=center><input type="submit" value="add"/></td>
</tr>
</form>
```

In the following example, when the page is displayed (HTTP GET), the __SL_G_N.A is replaced by the HTTP web server with the current IP address value, displayed in the input box. When the user changes and submits the IP address, the new value is sent with the __SL_P_N.A variable.

```
<form method="POST" name="SimpleLink Configuration action" action="ip_config.html">
<tr>
<td dir=LTR> IP Address: <td>
<td dir=LTR><input type="text" maxlength="15" name="__SL_P_N.A" value="__SL_G_N.A"/> </td>
</tr>
<tr>
<td colspan=2 align=center><input type="submit" value="Apply"/></td>
</tr>
</form>
```

12.4 Internal Web Page

The SimpleLink device has a default web page already embedded in ROM. This web page can be used to perform the following:

- Get versions and general information about the device
- IP configuration
- Add or remove Wi-Fi profiles
- Enable or disable ping test

Access to the internal web page is configured through the API. By default, access is enabled.

The web page is composed of the following files:

- about.html
- image001.png
- ip_config.html
- Logo.gif
- main.html
- ping.html
- profiles_config.html
- simple_link.css
- status.html
- tools.html

12.5 Force AP Mode Support

The Force AP mode returns the SimpleLink to its default configuration. This mode is entered with a special external GPIO.

When the SimpleLink enters Force AP mode, the HTTP server behaves as follows:

1. The server only permits access to the *www/safe/* folder in the file system. This lets the user put a set of web pages in the *www/* folder that will not be accessible in Force AP mode.
2. In this mode, a Clear all Profiles button appears in the internal configuration web pages, enabling the user to clear all the saved profiles from the device.

12.6 Accessing the Web Page

12.6.1 SimpleLink in Station Mode

When the SimpleLink is in station mode, the user can access the web page from the browser using the IP address. The HTTP service is also published by the mDNS server, so the IP address can be acquired from the mDNS publications.

12.6.2 SimpleLink in AP Mode

When in AP mode, access to the web page is performed using the domain name. The domain name is configured with the host API.

In AP mode, the SimpleLink is also accessed using the IP address.

The default domain name is **mysimplelink.net.**

Accessing the web page can be performed with either **mysimplelink.net** or **www.mysimplelink.net**.

When using the www. prefix in the domain name when the domain name does not match, the server internally removes this prefix and tries to search without it.

Important: Use the www. prefix in the domain name search because commercial browsers behave better from a DNS perspective.

12.7 HTTP Authentication Check

If enabled, the SimpleLink performs an authentication check when the client first connects to the server. The authentication check can be enabled or disabled using the host API.

Authentication user name, password, and realm can also be configured by the host API.

By default, the authentication is disabled.

The default authentication values are:

- Authentication name: **admin**
- Authentication password: **admin**
- Authentication realm: **Simple Link CC31xx**

A description of the host interface can be found in paragraph: Host / HTTP web server API.

12.8 Handling HTTP Events in Host Using the SimpleLink Driver

When the HTTP server locates user tokens in the HTML files, the server generates **get_token_value** (for GET tokens) or **post_token_value** (for post tokens) events to the host for the user to correctly handle them.

When the host gets a **get_token_value** event with a specific token name, the server returns the token value for this token name by using the **send_token_value** command.

If the host does not have any token value to return, the server uses zero as the length of the token value.

When the user gets a **post_token_value** event with the token name and value, the user must save this new token value.

In the SimpleLink driver, when one of the preceeding events is generated the driver calls a predefined callback called **SimpleLinkHttpServerCallback();**

The callback is defined as follows:

```
void SimpleLinkHttpServerCallback(SlHttpServerEvent_t *pHttpServerEvent, SlHttpServerResponse_t
*pHttpServerResponse)
```

Where serverEvent and serverResponse are defined as follows:

```
typedef struct
{

unsigned long             Event;

SlHttpServerEventData_u        EventData;

}SlHttpServerEvent_t;
typedef struct
{
```

```
unsigned long            Response;

SlHttpServerResponsedata_u    ResponseData;

}SlHttpServerResponse_t;
typedef union
{

slHttpServerString_t          httpTokenName; /* SL_NETAPP_HTTPGETTOKENVALUE */

slHttpServerPostData_t       httpPostData;  /* SL_NETAPP_HTTPPOSTTOKENVALUE */
} SlHttpServerEventData_u;
typedef union
{

slHttpServerString_t         token_value;  /*
< 64 bytes*/
} SlHttpServerResponsedata_u;
typedef struct _slHttpServerString_t
{

UINT8                 len;

UINT8                 *data;
} slHttpServerString_t;
typedef struct _slHttpServerPostData_t
{

slHttpServerString_t            action;

slHttpServerString_t          token_name;

slHttpServerString_t            token_value;
}slHttpServerPostData_t;
```

The following is a sample code for the user callback:

```
/*HTTP Server Callback example */
void SimpleLinkHttpServerCallback(SlHttpServerEvent_t *pHttpServerEvent,
                                  SlHttpServerResponse_t *pHttpServerResponse)
{
  switch (pHttpServerEvent->Event)
            {
    /* Handle Get token value */
    case SL_NETAPP_HTTPGETTOKENVALUE:
                        {
char * tokenValue;

 tokenValue = GetTokenValue (pHttpServerEvent >EventData.httpTokenName);

        /* Response using driver memory - Copy the token value to the Event response
Important - Token value len should be < MAX_TOKEN_VALUE_LEN (64 bytes) */

        strcpy (pHttpServerResponse->ResponseData.token_value.data, tokenValue);
```

```
            pHttpServerResponse->ResponseData.token_value.len = strlen (tokenValue);
                            }
            break;

        /* Handle Post token */
    case SL_NETAPP_HTTPPOSTTOKENVALUE:
                            {

    HandleTokenPost (pHttpServerEvent->EventData.httpPostData.action,
                     pHttpServerEvent->EventData.httpPostData.token_name,
                     pHttpServerEvent->EventData.httpPostData.token_value);

                            }
            break;
        default:
            break;
                }
    }
```

Important: For the HTTP callback to work the following line should be uncommented in *user.h*:

```
#define sl_HttpServerCallback SimpleLinkHttpServerCallback
```

12.9 SimpleLink Driver Interface the HTTP Web Server

The SimpleLink driver supplies an API to access and configure the HTTP server.

The API definition can be found in *netapp.h.*

12.9.1 Enable or Disable HTTP Server

Functions used for enabling or disabling the HTTP server. By default, the server is enabled.

Table 12-1. Enable or Disable HTTP Server

Function name	Description	Parameters
void sl_NetAppStart(UINT32 appId)	Starts the HTTP server	appId = SL_NET_APP_HTTP_SERVER_ID
void sl_NetAppStop(UINT32 appId)	Stops the HTTP server	appId = SL_NET_APP_HTTP_SERVER_ID

12.9.2 Configure HTTP Port Number

Functions used for configuring the port number.

Table 12-2. Configure HTTP Port Number

Function name	Description	Parameters

`long sl_NetAppSet (unsigned char appId ,`
`unsigned char Option,`
`unsigned char OptionLen,`
`unsigned char *pOptionValue)`
	Sets the port number on which the HTTP server will listen.
	Port number is UINT16.
		appId = SL_NET_APP_HTTP_SERVER_ID
		Option = NETAPP_SET_GET_HTTP_OPT_PORT_NUMBER
		pOptionLen = 2 (size of UINT16)
		pOptionValue = pointer to port number (UINT16)

Table 12-2. Configure HTTP Port Number (continued)

Function name	Description	Parameters

`long sl_NetAppGet (unsigned char appId,`
`unsigned char Option,`
`unsigned char *pOptionLen,`
`unsigned char *pOptionValue)`
	Gets the current HTTP server port number.
	Port number is UINT16.
		appId = SL_NET_APP_HTTP_SERVER_ID
		Option = NETAPP_SET_GET_HTTP_OPT_PORT_NUMBER
		pOptionLen = user-supplied pointer for the returned value len
		pOptionValue = user-supplied pointer for the returned value

Important: After setting a new port number, restart the server for the configuration to occur.

12.9.3 Enable or Disable Authentication Check

Functions used for enabling or disabling authentication check. By default, authentication check is disabled.

Table 12-3. Enable or Disable Authentication Check

Function name	Description	Parameters

`long sl_NetAppSet (unsigned char appId ,`
`unsigned char Option,`
`unsigned char OptionLen,`
`unsigned char *pOptionValue)`
	Enables or disables the HTTP server authentication check
	Auth_enable value is true/false.
		appId = SL_NET_APP_HTTP_SERVER_ID
		Option = NETAPP_SET_GET_HTTP_OPT_AUTH_CHECK
		pOptionLen = 1 (size of UINT8)
		pOptionValue = pointer to auth_enable (true/false)

`long sl_NetAppGet (unsigned char appId,`
`unsigned char Option,`
`unsigned char *pOptionLen,`
`unsigned char *pOptionValue)`
	Gets the current authentication status.
	Return auth_enable value is true/false.
		appId = SL_NET_APP_HTTP_SERVER_ID
		Option = NETAPP_SET_GET_HTTP_OPT_AUTH_CHECK
		pOptionLen = user-supplied pointer for the returned value len
		pOptionValue = user-supplied pointer for the returned value

12.9.4 Set or Get Authentication Name, Password, and Realm

Functions used to set or get the authentication name, password, and realm.

Table 12-4. Set or Get Authentication Name

Function name	Description	Parameters

`long sl_NetAppSet (unsigned char appId ,`
`unsigned char Option,`
`unsigned char OptionLen,`
`unsigned char *pOptionValue)`
	Sets authentication name.
	Name format is string.
		appId = SL_NET_APP_HTTP_SERVER_ID
		Option = NETAPP_SET_GET_HTTP_OPT_AUTH_NAME
		OptionLen = authentication name length
		pOptionValue = pointer to authentication name

`long sl_NetAppGet (unsigned char appId,`
`unsigned char Option,`
`unsigned char *pOptionLen,`
`unsigned char *pOptionValue)`
	Gets current authentication name.
	Name format is string (not null terminated).
		appId = SL_NET_APP_HTTP_SERVER_ID
		Option = NETAPP_SET_GET_HTTP_OPT_AUTH_NAME
		pOptionLen = user-supplied pointer for the returned name len
		pOptionValue = user-supplied pointer for the returned name

Table 12-5. Set or Get Authentication Password

Function name	Description	Parameters

`long sl_NetAppSet (unsigned char appId ,`
`unsigned char Option,`
`unsigned char OptionLen,`
`unsigned char *pOptionValue)`
	Sets authentication password. Password format is string.	appId = SL_NET_APP_HTTP_SERVER_ID
		Option = NETAPP_SET_GET_HTTP_OPT_AUTH_PASSWORD
		OptionLen = authentication password length
		pOptionValue = pointer to authentication password

`long sl_NetAppGet (unsigned char appId,`
`unsigned char Option,`
`unsigned char *pOptionLen,`
`unsigned char *pOptionValue)`
	Gets current authentication password. Password format is string (not null terminated).	appId = SL_NET_APP_HTTP_SERVER_ID
		Option = NETAPP_SET_GET_HTTP_OPT_AUTH_PASSWORD
		pOptionLen = user-supplied pointer for the returned password len
		pOptionValue = user-supplied pointer for the returned password

Table 12-6. Set or Get Authentication Realm

Function name	Description	Parameters

`long sl_NetAppSet (unsigned char appId ,`
`unsigned char Option,`
`unsigned char OptionLen,`
`unsigned char *pOptionValue)`
	Sets authentication realm. Realm format is string.	appId = SL_NET_APP_HTTP_SERVER_ID
		Option = NETAPP_SET_GET_HTTP_OPT_AUTH_REALM
		OptionLen = authentication realm length
		pOptionValue = pointer to authentication realm

`long sl_NetAppGet (unsigned char appId,`
`unsigned char Option,`
`unsigned char *pOptionLen,`
`unsigned char *pOptionValue)`
	Gets current authentication realm. Realm format is string (not null terminated).	appId = SL_NET_APP_HTTP_SERVER_ID
		Option = NETAPP_SET_GET_HTTP_OPT_AUTH_REALM
		pOptionLen = user-supplied pointer for the returned realm len
		pOptionValue = user-supplied pointer for the returned realm

12.9.5 Set or Get Domain Name

Functions used to set or get the domain name (used for accessing the web server in AP mode).

Table 12-7. Set or Get Domain Name

Function name	Description	Parameters

`long sl_NetAppSet (unsigned char appId ,`
`unsigned char Option,`
`unsigned char OptionLen,`
`unsigned char *pOptionValue)`
	Sets the domain name. Domain name format is string.	appId = SL_NET_APP_DEVICE_CONFIG_ID
		Option = NETAPP_SET_GET_DEV_CONF_OPT_DOMAIN_NAME
		OptionLen = domain name length
		pOptionValue = pointer to domain name

Table 12-7. Set or Get Domain Name (continued)

Function name	Description	Parameters

`long sl_NetAppGet (unsigned char appId,`
`unsigned char Option,`
`unsigned char *pOptionLen,`
`unsigned char *pOptionValue)`
	Gets current domain name.
	Domain name format is string (not null terminated).
		appId = SL_NET_APP_DEVICE_CONFIG_ID
		Option = NETAPP_SET_GET_DEV_CONF_OPT_DOMAIN_NAME
		pOptionLen = user-supplied pointer for the returned domain name len
		pOptionValue = user-supplied pointer for the returned domain name

12.9.6 Set or Get URN Name

Functions used to set or get the device unique URN name.

Table 12-8. Set or Get URN Name

Function name	Description	Parameters

`long sl_NetAppSet (unsigned char appId ,`
`unsigned char Option,`
`unsigned char OptionLen,`
`unsigned char *pOptionValue)`
	Sets the URN name.
	URN name format is string.
		appId = SL_NET_APP_DEVICE_CONFIG_ID
		Option = NETAPP_SET_GET_DEV_CONF_OPT_DEVICE_URN
		OptionLen = URN name length
		pOptionValue = pointer to URN name

`long sl_NetAppGet (unsigned char appId,`
`unsigned char Option,`
`unsigned char *pOptionLen,`
`unsigned char *pOptionValue)`
	Gets current URN name.
	URN name format is string (not null terminated).
		appId = SL_NET_APP_DEVICE_CONFIG_ID
		Option = NETAPP_SET_GET_DEV_CONF_OPT_DEVICE_URN
		pOptionLen = user-supplied pointer for the returned URN name len
		pOptionValue = user-supplied pointer for the returned URN name

12.9.7 Enable or Disable ROM Web Pages Access

Functions used to enable or disable the access to the ROM internal web pages. By default, web access is enabled.

Table 12-9. Enable or Disable ROM Web Pages Access

Function name	Description	Parameters

`long sl_NetAppSet (unsigned char appId ,`
`unsigned char Option,`
`unsigned char OptionLen,`
`unsigned char *pOptionValue)`
	Enables or disables the HTTP server ROM pages access.
	Value is true or false.
		appId = SL_NET_APP_HTTP_SERVER_ID
		Option = NETAPP_SET_GET_HTTP_OPT_ROM_PAGES_ACCESS
		OptionLen = 1 (sizeof UINT8)
		pOptionValue = pointer to boolean (true/false)

Table 12-9. Enable or Disable ROM Web Pages Access (continued)

Function name	Description	Parameters

`long sl_NetAppGet (unsigned char appId,`
`unsigned char Option,`
`unsigned char *pOptionLen,`
`unsigned char *pOptionValue)`
	Gets the current ROM pages access status.
	Return value is true or false.
		appId = SL_NET_APP_HTTP_SERVER_ID
		Option = NETAPP_SET_GET_HTTP_OPT_ROM_PAGES_ACCESS
		pOptionLen = user-supplied pointer for the returned value len
		pOptionValue = user-supplied pointer for the returned value

12.10 SimpleLink Predefined Tokens

This section contains information about the predefined internal tokens, and describes the tokens for the GET operations, POST operations, and the POST actions handled internally.

Important: To prevent user tokens from colliding with the internal tokens, use tokens with the following structure:

- For GET operations: **__SL_G_UXX**, where XX can be any character or number.
- For POST operations: **__SL_P_UXX**, where XX can be any character or number.

12.10.1 GET Values

GET system information values:

Table 12-10. System Information

Token	Name	Value / 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 the device name
__SL_G_S.C	Domain name	Returns the domain name
__SL_G_S.D	Device mode (role)	Returns the device role. Values: Station, Access Point, P2P
__SL_G_S.E	Device role station	Drop-down menu select/not select Returns “selected” if the device is a station, otherwise it returns “not_selected.”
__SL_G_S.F	Device role AP	Drop-down menu select/not select Returns “selected” if the device is an AP, otherwse it returns “not_selected.”
__SL_G_S.G	Device role P2P	Drop-down menu select/not select Returns “selected” if the device is in P2P, otherwise it returns “not_selected.”
__SL_G_S.H	Device name URN (truncated to 16 bytes)	Returns the URN string name with up to 16-byte len. Longer names are truncated.
__SL_G_S.I	System requires reset (after parameters change)	If the system requires a reset, the return value is the following string: -- Some parameters were changed, System may require reset --, otherwise 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 the device is in safe mode it returns “Safe Mode”, if not it returns an empty string.

GET version information:

Table 12-11. Version Information

Token	Name	Value / 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

GET network information:

Table 12-12. Network Information

Token	Name	Value / Usage
Station (and P2P client)		
__SL_G_N.A	STA IP address	String format: xxx.yyy.zzz.ttt
__SL_G_N.B	STA subnet mask	String format: xxx.yyy.zzz.ttt
__SL_G_N.C	STA default gateway	String format: xxx.yyy.zzz.ttt
__SL_G_N.D	MAC address	String format: 00:11:22:33:44:55
__SL_G_N.E	STA DHCP state	Returns value: Enabled or Disabled
__SL_G_N.F	STA DHCP disable state	If DHCP is disabled, returns Checked, otherwise it returns Not_Checked. Used in the DHCP radio button.
__SL_G_N.G	STA DHCP enable state	If DHCP is enabled, returns Checked, otherwise it returns Not_Checked. Used in the DHCP radio button.
__SL_G_N.H	STA DNS server	String format: xxx.yyy.zzz.ttt
DHCP server		
__SL_G_N.I	DHCP start address	String format: xxx.yyy.zzz.ttt
__SL_G_N.J	DHCP last address	String format: xxx.yyy.zzz.ttt
__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_W.A	Channel # in AP mode	
__SL_G_W.B	SSID	
__SL_G_W.C	Security type	Returned values: Open, WEP, WPA
__SL_G_W.D	Security type Open	If the security type is open, it returns Checked, otherwise it returns Not_Checked. Used in the security type radio button check/not checked.
__SL_G_W.E	Security type WEP	If the security type is WEP, returns Checked, otherwise it returns Not_Checked. Used in the security type radio button check/not checked.
__SL_G_W.F	Security type WPA	If the security type is WPA, it returns Checked, otherwise it returns Not_Checked. Used in the security type radio button check/not checked.

GET tools:

Table 12-13. Tools

Token	Name	Value / Usage
Ping test results		
__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.E	Ping test duration	In seconds

GET connection policy status:

Table 12-14. Connection Policy Status

Token	Name	Value / Usage
__SL_G_P.E	Auto connect	If auto connect is enabled, returns Checked, otherwise it returns Not_Checked. Used in the auto connect check box.
__SL_G_P.F	Fast connect	If fast connect is enabled, returns Checked, otherwise it returns Not_Checked. Used in the fast connect check box.
__SL_G_P.G	Any P2P	If any P2P is enabled, returns Checked, otherwise it returns Not_Checked. Used in the any P2P checkbox.
__SL_G_P.P	Auto SmartConfig	If auto SmartConfig is enabled, returns Checked, otherwise it returns Not_Checked. Used in the auto SmartConfig checkbox.

GET display profiles information:

Table 12-15. Display Profiles Information

Token	Name	Value / Usage
__SL_G_PN1	Return profile 1 SSID	SSID string
__SL_G_PN2	Return profile 2 SSID	SSID string
__SL_G_PN3	Return profile 3 SSID	SSID string
__SL_G_PN4	Return profile 4 SSID	SSID string
__SL_G_PN5	Return profile 5 SSID	SSID string
__SL_G_PN6	Return profile 6 SSID	SSID string
__SL_G_PN7	Return profile 7 SSID	SSID string
__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	Returned values: Open, WEP, WPA, WPS, ENT, P2P_PBC, P2P_PIN or – for empty profile.
__SL_G_PS3	Return profile 3 security status	Returned values: Open, WEP, WPA, WPS, ENT, P2P_PBC, P2P_PIN or – for empty profile.
__SL_G_PS4	Return profile 4 security status	Returned values: Open, WEP, WPA, WPS, ENT, P2P_PBC, P2P_PIN or – for empty profile.
__SL_G_PS5	Return profile 5 security status	Returned values: Open, WEP, WPA, WPS, ENT, P2P_PBC, P2P_PIN or – for empty profile.

Table 12-15. Display Profiles Information (continued)

Token	Name	Value / Usage
__SL_G_PS6	Return profile 6 security status	Returned values: Open, WEP, WPA, WPS, ENT, P2P_PBC, P2P_PIN or – for empty profile.
__SL_G_PS7	Return profile 7 security status	Returned values: Open, WEP, WPA, WPS, ENT, P2P_PBC, P2P_PIN or – for empty profile.
__SL_G_PP1	Return profile 1 priority	Profile priority: 0-7
__SL_G_PP2	Return profile 2 priority	Profile priority: 0-7
__SL_G_PP3	Return profile 3 priority	Profile priority: 0-7
__SL_G_PP4	Return profile 4 priority	Profile priority: 0-7
__SL_G_PP5	Return profile 5 priority	Profile priority: 0-7
__SL_G_PP6	Return profile 6 priority	Profile priority: 0-7
__SL_G_PP7	Return profile 7 priority	Profile priority: 0-7

GET P2P information:

Table 12-16. P2P Information

Token	Name	Value / 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 the current listen channel is 1, returns Selected, otherwise it returns Not_selected. Used for the drop-down menu of the listen channel.
__SL_G_R.U	Listen channel 6	If the current listen channel is 6, returns Selected, otherwise it returns Not_selected. Used for the drop-down menu of the listen channel.
__SL_G_R.V	Listen channel 11	If the current listen channel is 11, returns Selected, otherwise it 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 the current operational channel is 1, returns Selected, otherwise it returns Not_selected. Used for the drop-down menu of the operational channel.
__SL_G_R.X	Operational channel 6	If the current operational channel is 6, returns Selected, otherwise it returns Not_selected. Used for the drop-down menu of the operational channel.
__SL_G_R.Y	Operational channel 11	If the current operational channel is 11, returns Selected, otherwise it 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

Table 12-16. P2P Information (continued)

Token	Name	Value / Usage
__SL_G_R.M	Role group owner	If the intent is Group Owner, it returns Checked, otherwise it returns Not_Checked. Used for the negotiation intent radio button.
__SL_G_R.N	Role negotiate	If the intent is Negotiate, it returns Checked, otherwise it returns Not_Checked. Used for the negotiation intent radio button.
__SL_G_R.O	Role client	If the intent is Client, it returns Checked, otherwise it returns Not_Checked. Used for the 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 the negotiation initiator policy is Active, it returns Checked, otherwise it returns Not_Checked. Used for the negotiation initiator policy radio button.
__SL_G_R.R	Neg initiator passive	If the negotiation initiator policy is Passive, it returns Checked, otherwise it returns Not_Checked. Used for the negotiation initiator policy radio button.
__SL_G_R.S	Neg initiator random backoff	If the negotiation initiator policy is Random Backoff, it returns Checked, otherwise it returns Not_Checked. Used for the negotiation initiator policy radio button.

12.10.2 POST Values

POST system configuration

Table 12-17. System Configuration

Token	Name	Value / Usage
__SL_P_S.B	Device name (URN)	Sets device name
__SL_P_S.C	Domain name	Sets domain name
__SL_P_S.D	Device mode (role)	Sets device mode Values: Station, AP, P2P
__SL_P_S.J	Post system time and date	Sets system time and date. The value is a string with the following format: Year, month, day, hours, minutes, seconds
__SL_P_S.R	Redirect after post	Value should contain a valid web page. If the page exists, the web server issues an HTTP 302 response to redirect to the web page. Can be used for redirection after submitting a form (with HTTP post).

POST network configurations:

Table 12-18. Network Configurations

Token	Name	Values / Usage
	Station (and P2P client)	
__SL_P_N.A	STA IP address	Sets STA IP address. Value format: xxx.yyy.zzz.ttt
__SL_P_N.B	STA subnet mask	Sets STA subnet mask. Value format: xxx.yyy.zzz.ttt
__SL_P_N.C	STA default gateway	Sets STA default gateway. Value format: xxx.yyy.zzz.ttt
__SL_P_N.D	STA DHCP state (must be disabled for the IP setting to take affect)	Enables or disables DHCP state. If value is Enable, then DHCP is enabled, any other value disables the DHCP.
__SL_P_N.H	STA DNS server	Sets STA DNS server address. Value format: xxx.yyy.zzz.ttt
	DHCP server	
__SL_P_N.I	DHCP start address	Sets start address. Value format: xxx.yyy.zzz.ttt
__SL_P_N.J	DHCP last address	Sets last address. Value format: xxx.yyy.zzz.ttt
__SL_P_N.K	DHCP lease time	Sets lease time, in seconds
	AP (and P2P Go)	
__SL_P_N.P	AP IP address	Sets AP IP address. Value format: xxx.yyy.zzz.ttt
__SL_P_W.A	Channel # in AP mode	Sets channel number. Values: 1 to 13
__SL_P_W.B	SSID	Sets SSID
__SL_P_W.C	Security type	Sets security type: 0 for Open, 1 for WEP, 2 for WPA.
__SL_P_W.G	Password	Sets password

POST connection policy configuration:

Table 12-19. Connection Policy Configuration

Token	Name	Values / Usage
Connection policy configuration		Used with the connection policy form (policy_config.html action)
__SL_P_P.E	Auto connect	Enable or Disable If this parameter exists in the POST (with any value), this policy is set. If this parameter does not exist in the POST, this policy flag is cleared.
__SL_P_P.F	Fast connect	Enable or Disable If this parameter exists in the POST (with any value), this policy is set. If this parameter does not exist in the POST, this policy flag is cleared.
__SL_P_P.G	Any P2P	Enable or Disable If this parameter exists in the POST (with any value), this policy is set. If this parameter does not exist in the POST, this policy flag is cleared.
__SL_P_P.P	Auto SmartConfig	Enable or Disable If this parameter exists in the POST (with any value), this policy is set. If this parameter does not exist in the POST, this policy flag is cleared.

POST profiles configuration:

Table 12-20. Profiles Configuration

Token	Name	Values / Usage
Add new Profile		Used with the add new profile form (profiles_add.html action)
__SL_P_P.A	SSID	SSID string
__SL_P_P.B	Security type	Security type: 0 for Open, 1 for WEP, 2 for WPA1, 3 for WPA2
__SL_P_P.C	Security key	Smaller than 32 characters
__SL_P_P.D	Profile priority	0 to 7
Add P2P Profile		Used with the add P2P profile form (p2p_profiles_add action)
__SL_P_P.A	P2P Remote device name	String
__SL_P_P.B	P2P Security type	Security type: 6 for push-button, 7 for PIN keypad, 8 for PIN display
__SL_P_P.C	P2P PIN code	Digits only
__SL_P_P.D	P2P Profile priority	0 to 7
Add Enterprise Profile		Used with the add enterprise profile form (eap_profiles_add action)
__SL_P_P.H	SSID	String
__SL_P_P.I	Identity	String
__SL_P_P.J	Anonymous identity	String
__SL_P_P.K	Password	String
__SL_P_P.L	Profile priority	0 to 7
__SL_P_P.M	EAP method	Values: TLS, TTLS, PEAP0, PEAP1, FAST
__SL_P_P.N	PHASE 2 Authentication	Values: TLS, MSCHAPV2, PSK
__SL_P_P.O	Provisioning (0,1, 2) (for fast method only)	Values: None, 0, 1, 2, 3 Relevant for fast method only (values 0 to 3) For other methods, use None.
Profile remove		
__SL_P_PRR	Remove profile	Remove selected profile Value: 1 to 7

POST tools:

Table 12-21. Tools

Token	Name	Values / Usage
Start ping test		
__SL_P_T.A	IP address	IP address of the remote device
__SL_P_T.B	Packet size	In bytes (32 to 1472)
__SL_P_T.C	Number of pings	0 to unlimited, 1 to 255

POST P2P configuration:

Table 12-22. P2P Configuration

Token	Name	Values / Usage
__SL_P_R.E	P2P Channel (operational)	Set P2P operational channel. Values: 1, 6, 11
__SL_P_R.L	Negotiation intent value	Set Negotiation intent value. Values: CL for client, NEG for negotiate, GO'for group owner

12.10.3 POST Actions

Table 12-23 describes the POST actions handled internally as complex actions and the respective token parameters used in each post. All other remaining post parameters are handled by the server by updating their respective value.

Table 12-23. POST Actions

Action Name	Description	POST Tokens Parameters
sta_ip_config	Station network configuration	• STA IP address • STA subnet mask • STA default gateway • STA DHCP state • STA DNS server
ap_ip_config	Access point network configuration	• AP IP address • DHCP start address • DHCP last address • DHCP lease time
profiles_add.html	Add new profile	• SSID • Security type • Security key • Profile priority
p2p_profiles_add	Add peer to peer profile	• P2P Remote device name • P2P Security type • P2P PIN code • P2P Profile priority
eap_profiles_add	Add Enterprise profile	• SSID • Identity • Anonymous identity • Password • Profile priority • EAP method • PHASE 2 Authentication • Provisioning
remove_all_profiles	Remove all profiles	Not relevant Note: Keep at least one parameter in the HTML so the HTTP POST will not be empty. The server will not check the parameter value.
ping.html	Start the ping test	• IP address • Packet size • Number of pings
ping_stop	Stop the ping test	Not relevant Note: Have at least one parameter in the HTML so the HTTP POST will not be empty. The server will not check the parameter value.
policy_config.html	Connection policy configuration	• Auto connect • Fast connect • Any P2P • Auto SmartConfig

Chapter 13

SWRU368–June 2014

mDNS

Topic Page

13.1 Overview

The mDNS and DNS-SD feature provides the ability to find and advertise services on its local network, and resolve host names to IP addresses without using a local name server.

By supporting RFCs 6762 and 6763, the mDNS module advertises services, sends query responses to peer queries, and listens to peer advertisements.

mDNS uses the same programming interfaces, packet formats, and operating semantics as the unicast Domain Name System (DNS), except that mDNS uses IP multicast User Datagram Protocol (UDP) packets.

By default, mDNS exclusively resolves host names ending with the .local top-level domain (TLD).

The mDNS Ethernet frame is a multicast UDP packet to:

- MAC address 01:00:5E:00:00:FB
- IPv4 address 224.0.0.251 or IPv6 address FF02::FB
- UDP port 5353

13.2 Services – How to Find Them

Services are found using RR queries. RRs store a large variety of information about a domain:

- IP address
- Name server
- Mail exchanger
- Alias
- Host name
- Geo-location
- Service discovery
- Certificates
- Arbitrary text

A DNS zone database is a collection of resource records. Each resource record specifies information about a particular object. For example, address mapping records a host name to an IP address.

RRS queries with types of PTR, SRV, TXT, and A are needed for discovering the full-service details.

- PTR RR returns the URN/full-service name.
- SRV RR is used for the discovery services provided by the hosts and returns the service types, including the domain name.
- TXT RR gets the arbitrary text associated with a domain, or depicts the service. A record maps the host names to an IPV4 address.

All answers must be sent in a single response to a query. For example:

When an mDNS client must resolve a host name after receiving a PTR record which includes a service in the interest of the application:

1. The client sends an IP multicast query message asking the host with that name to identify it.
2. That target machine then multicasts a message that includes its IP address.
3. All machines in that subnet can then use that information to update their mDNS caches.

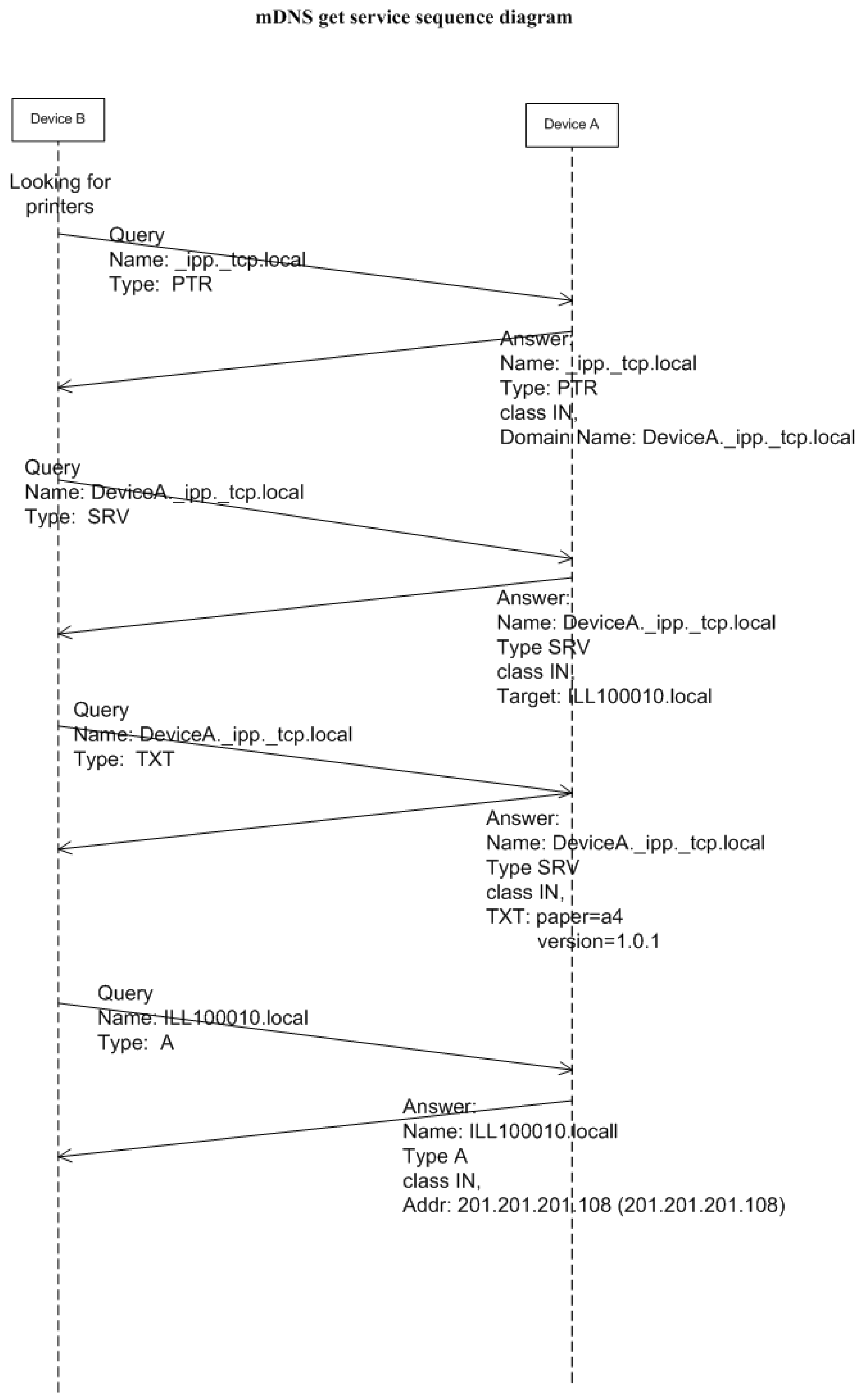


Figure 13-1. mDNS Get Service Sequence

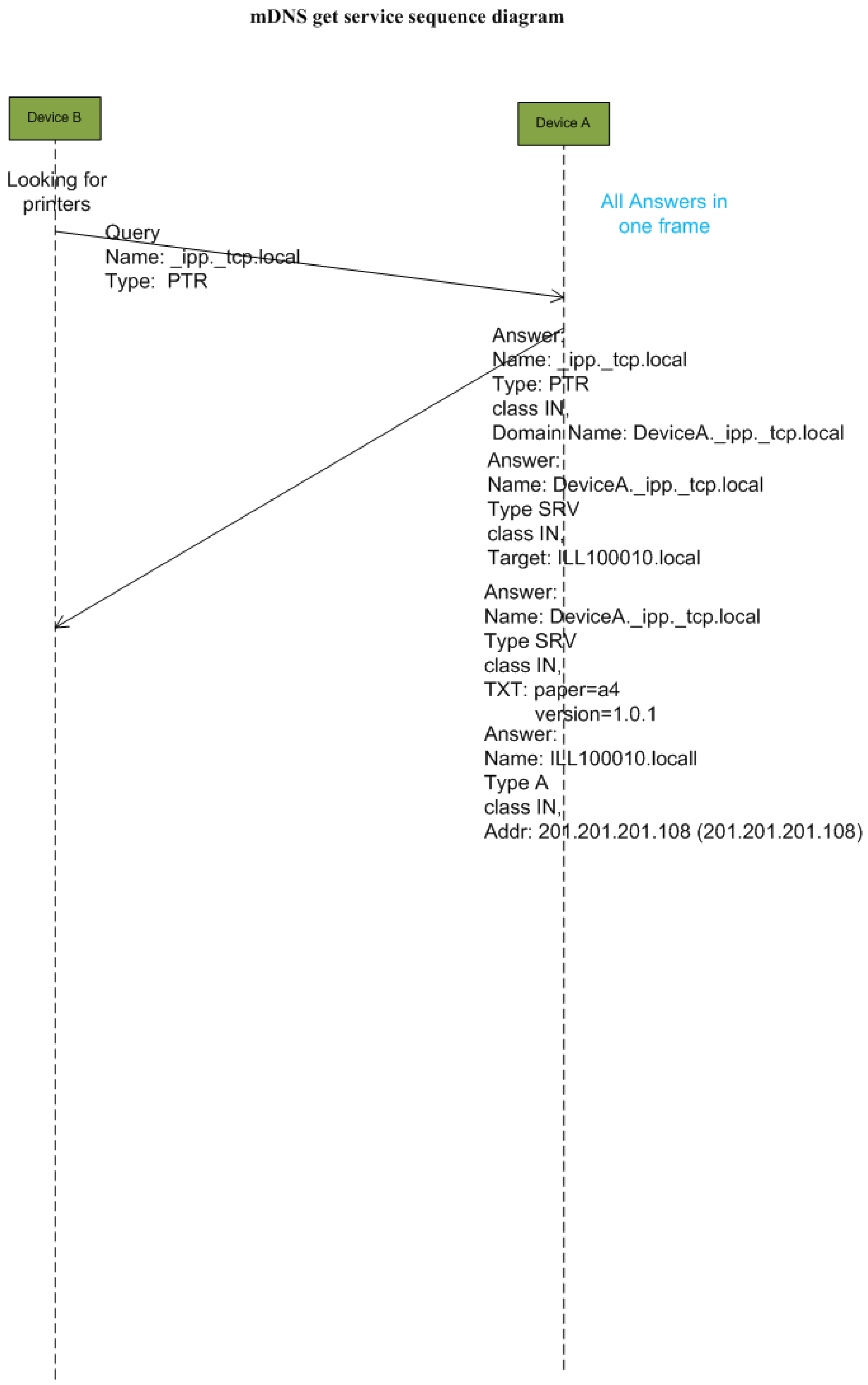


Figure 13-2. Find Full Service After Query

13.3 Start and Stop mDNS

By default, the mDNS is started. To stop the mDNS, use the API **sl_ sl_NetAppStop** with **SL_NET_APP_MDNS_ID**.

For example:

```
sl_ sl_NetAppStop(SL_NET_APP_MDNS_ID);
```

To start the mDNS, use the API **sl_NetAppStart** with **SL_NET_APP_MDNS_ID**.

For example:

```
sl_NetAppStart(SL_NET_APP_MDNS_ID);
```

The mDNS is stopped and started only for the current role. Other mDNS configurations are common to all roles.

The mDNS can be configured if the role has no interface (for example, if the station is not connected or P2P is not upped). The configuration is the same for all roles and is not related to a specific role (other than start and stop).

mDNS frames (advertise, response to queries) are sent only if there is a valid IP address.

mDNS works if one of the following conditions is met:

- Station is connected
- P2P is upped (GO or client)
- AP is upped

13.4 Typical Operation Methods

This section recommends three operations to find and register services.

13.4.1 Find Service RRs (Parameters) – By One-Shot Query

1. Mask, just once, all unwanted services to save space in the peer cache.
2. Use the API GET host by service using a full name or partial name containing the type of service.
3. Wait for an answer from one adapted service.
4. For more answers, wait and then use the API GET service list to find all the answers of the adapted services.

13.4.2 Find Service RRs (Parameters) – By Continuous Query

1. Mask, just once, all unwanted services to save space in the peer cache.
2. Set, just once, a continuous query with a full or partial name that contains the type of service. The query is saved and used after each reset (if mDNS is started with IP).
3. Wait and use the API get service list to find all the answers of the adapted services.

13.4.3 Register Service

- Register, just once, a specific service. The service is saved and advertised after each reset (if mDNS is started and STA with IP or P2P/AP up).
- Responses to queries that contain the service name or type are also sent.
- Set, just once, the advertising timing parameters if the default parameters are not required.

13.5 Detailed APIs

This section describes the mDNS APIs.

13.5.1 API – Get Host by Service

Description: Finds service resource records such as IP address, port, and text by a given full-service name or by a given type of service.

The user sets a service name Full or Part (see the following example) and should get:

- IP of service
- The port of service
- The text of service

Similar to the get host by name method, a single-shot query with PTR type on the service name can make a connection to the specific service and use it.

Note: The user only gets resource records of one adapted service.

Because all services adapted to the query send their answers at different times, use the API **sl_NetAppGetServiceList** to see the RR of all answers.

Return:

0 – Success.

<0 – A kind of error

API name:

```
long sl_NetAppDnsGetHostByService(char                     *pServiceName,
                   unsigned char     ServiceLen,
                   unsigned char     Family,
                   unsigned long      pAddr[],
                   unsigned long     *pPort,
                   unsigned short     *pTextLen,
                   char                    *pText
                  );
```

Parameters:

Table 13-1. Parameters

Type	Name	In/Out	Description
Char*	pServiceName	In	Partial or full name of the looking service. This name is set in the field name of the query. Example of full-service name: • PC1._ipp._tcp.local • PC2._http._tcp.local Example of partial service name: • PC1._ipp._tcp.local • PC2._http._tcp.local
unsigned char	ServiceLen	In	The length of the pServiceName
unsigned char	Family	In	Should be IPv4. Use the variable **SL_AF_INET .**
unsigned long*	pAddr	Out	Contains the IP address of the looking service
unsigned long*	pPort	Out	Contains the port address of the looking service

Table 13-1. Parameters (continued)

Type	Name	In/Out	Description
unsigned short*	pTextLen	In_Out	The length of the text of the looking service. Used for input and output Input: The maximum length that the user wants to get. If the real text length is bigger than the input value, the text is cut to the input length and the user loses part of the real text. Output: The length of the text of the looking service (this length depends on the input length). If the output length is equal to input length it may indicate that the text was cut) Note: The input value helps in tiny hosts because it saves space, but risks not getting the full text and should be used carefully. To get the full text, the value of the parameter must be the maximum of text size (256).
Char*	pText		Contains the text of the looking service

Table 13-2 lists the defines for this API

Table 13-2. Defines for API

Name	Description
SL_AF_INET	Should be used for the Family input parameter
NETAPP_MAX_SERVICE_TEXT_SIZE	Max text length = 256

Example:

```
long             Status = 0;

char             out_pText[NETAPP_MAX_SERVICE_TEXT_SIZE];
short            inout_TextLen;
unsigned long    IPv4Addr;
unsigned long    out_pPort;
unsigned char    ServiceNameIPP[50] = "_ipp._tcp.local";

while (Status != 0)
{
  //Find IPP service
  inout_TextLen = NETAPP_MAX_SERVICE_TEXT_SIZE;
  Status = sl_NetAppDnsGetHostByService( ServiceNameIPP,
                     strlen(ServiceNameIPP),
                     SL_AF_INET,
&amp;IPv4Addr,
&amp;out_pPort,
&amp;inout_TextLen,
&amp;out_pText[0]
);
```

```
    printf("one shot query on %s\n",ServiceNameff3);
    printf("status %d\n",Status);
    printf("port %d\n",out_pPort);
    printf("text len %d\n",inout_TextLen);
    printf("IP%d.%d.%d.%d\n",SL_IPV4_BYTE(IPv4Addr,3),
                             SL_IPV4_BYTE(IPv4Addr,2),
                             SL_IPV4_BYTE(IPv4Addr,1),
                             SL_IPV4_BYTE(IPv4Addr,0));
    printf("Text %.*s\n\n",inout_TextLen,out_pText);

  }
```

13.5.2 API - Get Service List

Description: The get service list returns a list of services according to the parameters of the command. The list is inserted into the out buffer (see Table 13-3).

The services are taken from the peer cache of the device. The peer cache is an internal memory containing all peer services received by advertising or by responding. The device keeps the RRs of all services in this peer cache (the services of up to eight peers are supported).

Three types of lists are available; each list contains different kinds of services and different types of RRs. The type of the list is set as an input parameter to the API.

Possible types of list input parameters:

- Full-service parameters with text (service name, IP, port, text name)
- Full-service parameters (service name, IP, port)
- Short-service parameters (port and IP only), especially for tiny hosts

The different types of lists give the possibility to use the API with tiny hosts.

The user sets the number of return services and the start index in the peer cache (see Table 13-3).

Note: The services in the peer cache are updated frequently.

Return:

= 0 – No services were found

>0 – Number finding of services

<0 – A kind of error

API name:

```
int sl_NetAppGetServiceList(unsigned char  IndexOffest,
               unsigned char  MaxServiceCount,
               unsigned char  Flags,
               char                *pBuffer,
               unsigned long  RxBufferLength
);
```

Parameters:

Table 13-3. Parameters

Type	Name	In/Out	Description
unsigned char	IndexOffest	In	Mentions the start row (entry) in the peer cache. From this entry the services are returned. For example, the number 2 means that the first service that is returned is in row number two.
unsigned char	MaxServiceCount	In	The maximum services that are returned (if they exist) If the return number is smaller than this value, all services were found and there are no more services. If the return number is equal to this value, there can be more services in the peer cache.
unsigned char	Flags	In	Mentions the type (RRs) of the services that are set in the list: full service with text, full service, or short service. It is an ENUM number that is taken from SlNetAppGetServiceListType_e (see Table 13-4).
Char*	pBuffer	Out	The services are inserted into this buffer. The buffer contains an array of services according to the return number (if bigger than zero). The type and kind of each service is according to the value of the flags.
unsigned long	RxBufferLength	In	The maximum length of pBuffer. If the returned data is bigger or should be bigger than this length, then a specific error is returned (see error section). In this case, the user must decrease the maximum returned services number (MaxServiceCount), or change the type of the list to get a smaller service. The length of the data that is returned must be smaller than NWP RX packet (about 1480), or an error is returned. The returned length is a multiply of MaxServiceCount and the size of service struct (that is set according to flags value).

Table 13-4 lists the user defines for this API:

Table 13-4. User Defines

Name	Description

typedef enum
{
  SL_NET_APP_FULL_SERVICE_WITH_TEXT_IPV4_TYPE = 1,
  SL_NET_APP_FULL_SERVICE_IPV4_TYPE,
  SL_NET_APP_SHORT_SERVICE_IPV4_TYPE,
 	
} SlNetAppGetServiceListType_e;
	Should be used for the Flags parameter.
	Indicates the type of the services that are returned.

typedef struct
{
    unsigned long   service_ipv4;
    unsigned short service_port;
    unsigned short Reserved;
}SlNetAppGetShortServiceIpv4List_t;
	Type of short service

typedef struct
{
    unsigned long   service_ipv4;
    unsigned short service_port;
    unsigned short Reserved;
    unsigned char service_name[NETAPP_MAX_SERVICE_NAME_SIZE];
    unsigned char service_host[NETAPP_MAX_SERVICE_HOST_NAME_SIZE];
}SlNetAppGetFullServiceIpv4List_t;
	Type of full service

typedef struct
{
    unsigned long   service_ipv4;
    unsigned short service_port;
    unsigned short Reserved;
    unsigned char service_name[NETAPP_MAX_SERVICE_NAME_SIZE];
    unsigned char service_host[NETAPP_MAX_SERVICE_HOST_NAME_SIZE];
    unsigned char service_text[NETAPP_MAX_SERVICE_TEXT_SIZE];
}SlNetAppGetFullServiceWithTextIpv4List_t;
	Type of short service with text

Example:

```
char                 BufferList[1500];
int                 Status;
int                 index,serviceIndex,serviceCount;
unsigned int   IPv4Addr;
SlNetAppGetShortServiceIpv4List_t            *ShortService;
SlNetAppGetFullServiceIpv4List_t             *FullService;
SlNetAppGetFullServiceWithTextIpv4List_t     *FullServiceWithText;

/***************************************************************************************************
***************/
//Get all full services that are in the peer cache.

//buffer with  get service list
memset(BufferList,0,sizeof(BufferList));

//Full service list with text index 0 count 8
serviceIndex = 0;
serviceCount = 9;
Status =
sl_NetAppGetServiceList(serviceIndex,serviceCount,SL_NET_APP_FULL_SERVICE_IPV4_TYPE,BufferList,150
0);
printf("getServiceList StartServiceIndex %d\n num of requested services %d\n service struct type
%d\n num of returned services %d\n",
        serviceIndex,serviceCount,SL_NET_APP_FULL_SERVICE_IPV4_TYPE,Status);

printf("service list is depicted below: \n ");

if(Status >0)
{
   for(index = 0; index < Status; index ++)
            {
      FullService =
(SlNetAppGetFullServiceIpv4List_t*)(&amp;BufferList[index*sizeof(SlNetAppGetFullServiceIpv4List_t)
]);

      printf("index %d\n",index);
      IPv4Addr = FullService->service_ipv4;
      printf("IP
%d.%d.%d.%d\n",SL_IPV4_BYTE(IPv4Addr,3),SL_IPV4_BYTE(IPv4Addr,2),SL_IPV4_BYTE(IPv4Addr,1),SL_IPV4_
BYTE(IPv4Addr,0));
      //printf("service_ipv4 %x\n",FullService->service_ipv4);
      printf("service_port %d\n",FullService->service_port);
      printf("service_name %s\n",FullService->service_name);
      printf("service_host %s\n",FullService->service_host);
      printf("\n\n");

            }
}

/*************************************************************************************************
***************/

/*************************************************************************************************
***************/
//Get short services that are in the peer cache.

//buffer with  get service list
memset(BufferList,0,sizeof(BufferList));

//Short service list with text index 0 count 10
serviceIndex = 0;
serviceCount = 10;
Status =
sl_NetAppGetServiceList(serviceIndex,serviceCount,SL_NET_APP_SHORT_SERVICE_IPV4_TYPE,BufferList,15
00);
printf("getServiceList StartServiceIndex %d\n num of requested services %d\n service struct type
%d\n num of returned services
       serviceIndex,serviceCount,SL_NET_APP_SHORT_SERVICE_IPV4_TYPE,Status);

printf("service list is depicted below: \n ");

if(Status > 0)
{
for(index = 0; index < Status ; index ++)
            {
shortService =
(SlNetAppGetShortServiceIpv4List_t*)(&amp;BufferList[index*sizeof(SlNetAppGetShortServiceIpv4List_
t)]);

    printf("index %d\n",index);
    IPv4Addr = ShortService->service_ipv4;
    printf("IP
\n",SL_IPV4_BYTE(IPv4Addr,3),SL_IPV4_BYTE(IPv4Addr,2),SL_IPV4_BYTE(IPv4Addr,1),SL_IPV4_BYTE(IPv4Ad
dr,0));
    //printf("service_ipv4 %x\n",ShortService->service_ipv4);
    printf("service_port %d\n",ShortService->service_port);
    printf("\n\n");

            }
}

/***************************************************************************************************
***************/

/***************************************************************************************************
***************/

//Get full services with text with loop that are in the peer cache.

//buffer with  get service list
memset(BufferList,0,sizeof(BufferList));

//Full service list with text index 0 count 3
serviceIndex = 0;
serviceCount = 3;
Status = 0;

do
{
  serviceIndex = serviceIndex + Status;

  Status =
sl_NetAppGetServiceList(serviceIndex,serviceCount,SL_NET_APP_FULL_SERVICE_WITH_TEXT_IPV4_TYPE,Buff
erList,1500);
  printf("getServiceList StartServiceIndex %d\n num of requested services %d\n service struct
type %d\n num of returned %d\n",
       serviceIndex,serviceCount,SL_NET_APP_FULL_SERVICE_WITH_TEXT_IPV4_TYPE,Status);

  printf("service list is depicted below: \n ");

  if(Status > 0 )
            {
for(index = 0; index < Status ; index ++)
                        {
FullServiceWithText =
erviceWithTextIpv4List_t*)(&amp;BufferList[index*sizeof(SlNetAppGetFullServiceWithTextIpv4List_t)]
);

        printf("index %d\n",index);
        IPv4Addr = FullServiceWithText->service_ipv4;
        printf("IP
_IPV4_BYTE(IPv4Addr,3),SL_IPV4_BYTE(IPv4Addr,2),SL_IPV4_BYTE(IPv4Addr,1),SL_IPV4_BYTE(IPv4Addr,0))
;
        //printf("service_ipv4 %x\n",FullServiceWithText->service_ipv4);
        printf("service_port %d\n",FullServiceWithText->service_port);
```

```
                printf("service_name %s\n",FullServiceWithText->service_name);
                printf("service_host %s\n",FullServiceWithText->service_host);
                printf("service_text %s\n",FullServiceWithText->service_text);
                printf("\n\n");

                                }
                    }

        }while( (Status == serviceCount ));
```

13.5.3 API – Register Service

Description: This API registers a service and advertises it if mDNS is started and the device has an IP address (STA) or up (P2P or AP). The service is kept in the mDNS database; only one registration is needed. This registered service is offered by the application.

The service name should be the full-service name according to DNS-SD RFC, meaning the value in the name field of the SRV answer.

Example of a service name:

- PC1._ipp._tcp.local
- PC2_server._ftp._tcp.local

If the service is unique (see the Options parameter in Table 13-5), mDNS probes the service name to ensure its uniqueness before starting to announce the service on the network. If it is not unique, a number will be added to identify it.

Return

= 0 – Success

<0 – A kind of error

API name

```
int sl_NetAppMDNSRegisterService( const char*     pServiceName,
                                  unsigned char   ServiceNameLen,
                                  const char*     pText,
                                  unsigned char   TextLen,
                                  unsigned short  Port,
                                  unsigned long   TTL,
                                  unsigned long   Options);
```

Parameters:

Table 13-5. Parameters

Type	Name	In/Out	Description
const char*	pServiceName	In	The service name. Example of service name: • PC1._ipp._tcp.local • PC2_server._ftp._tcp.local
unsigned char	ServiceNameLen	In	The length of pServiceName
const char*	pText	In	The description of the service; should be as mentioned in the RFC (according to type of the service IPP, FTP, and so forth)
unsigned char	TextLen	In	The length of pText
unsigned short	Port	In	The port on the target host port
unsigned long	TTL	In	The TTL of the service

Table 13-5. Parameters (continued)

Type	Name	In/Out	Description
unsigned long	Options	In	Bitwise parameters: **Bit 0** - Service is unique (the service must be unique). **Bit 31** - For internal use if the service should be added or deleted (set means ADD) **Bit 1-30** for future use

Example:

```
unsigned char AddService1[40]    = "SimpleLinkPrinter553321._ipp._tcp.local";

//Register IPP service
Status =  sl_NetAppMDNSRegisterService(AddService1,
      strlen(AddService1),
                                      "payper=A3;size=5",
                                      strlen("payper=A3;size=5"),
                                      1000,120,1);
printf("Register service  %s \n\rstatus %d\n\n",AddService1,Status);
```

13.5.4 API – Unregister Service

Description: This API unregisters mDNS service, deletes the service from mDNS DB, and sends a goodbye frame if the mDNS is started and has an IP address / UP.

The unregistered mDNS service is a service that the application no longer needs to provide. The name of the deleted service should be the same as the name of the service that was previously registered

Note: If the service name is null, all services are deleted and the mDNS machine stops and starts, then the peer cache is deleted. Other services which were not deleted are retained in the system, and they do not need to be registered again.

Return:

= 0 – Success

<0 – A kind of error

API name:

```
int sl_NetAppMDNSUnRegisterService( const char      *pServiceName,
                                    unsigned char    ServiceNameLen);
```

Parameters:

Table 13-6. Parameters

Type	Name	In/Out	Description
const char*	pServiceName	In	The service name that should be deleted. Example of service name: • PC1._ipp._tcp.local • PC2_server._ftp._tcp.local Null name means delete all services.
unsigned char	ServiceNameLen	In	The length of *pServiceName*

Example:

```
unsigned char AddService1[40]    = "SimpleLinkPrinter553321._ipp._tcp.local";

//Register IPP service
Status =  sl_NetAppMDNSRegisterService(AddService1,
      strlen(AddService1),
                                      "payper=A3;size=5",
                                      strlen("payper=A3;size=5"),
                                      1000,120,1);
printf("Register service  %s \n\rstatus %d\n\n",AddService1,Status);
```

13.5.5 API – Set Masking Receive Services

Description: This API indicates which user-defined service types to ignore. The mask is a 32-bit ULONG type. Each bit represents a service type (see Table 13-7). If a bit is set, the corresponding service type specified will not be set in the peer cache and will not be shown to the user.

The peer cache contains up to eight services. By using the masking, the user can prevent unwanted services from entering the peer cache, or save places for desired services.

For example, if the user is interested, RAOP and Airplay services should mask all the other services. Default is zero, with no event mask.

Return:

= 0 – Success

<0 – A kind of error

API name:

```
long sl_NetAppSet(unsigned char AppId ,
                                unsigned char Option,
                                unsigned char OptionLen,

unsigned char *pOptionValue);
```

Table 13-7 lists the defines for this API:

Table 13-7. Defines for API

Name	Description
SL_NET_APP_MDNS_ID	Used as AppId parameter in sl_NetAppSet function
NETAPP_SET_GET_MDNS_QEVETN_MASK_OPT	Used as Option parameter in sl_NetAppSet function

Table 13-7. Defines for API (continued)

Name	Description
#define SL_NET_APP_MASK_IPP_TYPE_OF_SERVICE 0x00000001 #define SL_NET_APP_MASK_DEVICE_INFO_TYPE_OF_SERVICE 0x00000002 #define SL_NET_APP_MASK_HTTP_TYPE_OF_SERVICE 0x00000004 #define SL_NET_APP_MASK_HTTPS_TYPE_OF_SERVICE 0x00000008 #define SL_NET_APP_MASK_WORKSATION_TYPE_OF_SERVICE 0x00000010 #define SL_NET_APP_MASK_GUID_TYPE_OF_SERVICE 0x00000020 #define SL_NET_APP_MASK_H323_TYPE_OF_SERVICE 0x00000040 #define SL_NET_APP_MASK_NTP_TYPE_OF_SERVICE 0x00000080 #define SL_NET_APP_MASK_OBJECITVE_TYPE_OF_SERVICE 0x00000100 #define SL_NET_APP_MASK_RDP_TYPE_OF_SERVICE 0x00000200 #define SL_NET_APP_MASK_REMOTE_TYPE_OF_SERVICE 0x00000400 #define SL_NET_APP_MASK_RTSP_TYPE_OF_SERVICE 0x00000800 #define SL_NET_APP_MASK_SIP_TYPE_OF_SERVICE 0x00001000 #define SL_NET_APP_MASK_SMB_TYPE_OF_SERVICE 0x00002000 #define SL_NET_APP_MASK_SOAP_TYPE_OF_SERVICE 0x00004000 #define SL_NET_APP_MASK_SSH_TYPE_OF_SERVICE 0x00008000 #define SL_NET_APP_MASK_TELNET_TYPE_OF_SERVICE 0x00010000 #define SL_NET_APP_MASK_TFTP_TYPE_OF_SERVICE 0x00020000 #define SL_NET_APP_MASK_XMPP_CLIENT_TYPE_OF_SERVICE 0x00040000 #define SL_NET_APP_MASK_RAOP_TYPE_OF_SERVICE 0x00080000 #define SL_NET_APP_MASK_ALL_TYPE_OF_SERVICE 0xFFFFFFFF	Masking types

Example:

```
//mask IPP service
unsigned int EventMask = SL_NET_APP_MASK_IPP_TYPE_OF_SERVICE;

Status = sl_NetAppSet(SL_NET_APP_MDNS_ID,
                      NETAPP_SET_GET_MDNS_QEVETN_MASK_OPT,
                      sizeof(EventMask), (unsigned char*)&amp;EventMask );
printf("event Mask status %d\n",Status);
```

13.5.6 API – Set Continuous Query

Description: This API sets a continuous query. The query can be on full service or on type of service. Default is null – no query.

For example:

- For full service, the name should be PC1._ipp._tcp.local.
- For a type of IPP services, the name should be _ipp._tcp.local.
- For stopping the continuous query, the name should be null.

Note:

- Only one continuous query can be set. A new setting stops the old query and sets a new one. If the name is null, the old query is stopped and no continuous query is configured.
- The answers are not given automatically to the user. To see the received services the user should use the get service list operation (see Section 13.5.2).

Return:

= 0 – Success

<0 – A kind of error

API name:

```
long sl_NetAppSet(    unsigned char AppId ,
unsigned char Option,
unsigned char OptionLen,
unsigned char *pOptionValue);
```

Table 13-8 lists the defines for this API:

Table 13-8. Defines for API

Name	Description
SL_NET_APP_MDNS_ID	Used as AppId parameter in sl_NetAppSet function
NETAPP_SET_GET_MDNS_CONT_QUERY_OPT	Used as Option parameter in sl_NetAppSet function

Example:

```
unsigned char name[40]  = "_ipp._tcp.local";

//looking for IPP services with continuous query
Status = sl_NetAppSet(SL_NET_APP_MDNS_ID ,NETAPP_SET_GET_MDNS_CONT_QUERY_OPT,strlen(name), name);
printf("cont query status %d\n",Status);
```

13.5.7 API – Set Timing Parameters for Advertising

Description: This API reconfigures the timing parameters employed by mDNS when sending service announcements. The published period starts from t ticks and can be expanded telescopically with 2 to the power of k factor. The number of repetitions per advertisement is p, the interval between each repeated advertisement is interval ticks, and the number of announcement period is max_time.

By default, the initial period is set to 1 second, with k = 1 (the period doubles each time), p = 1 (no repetition), retrans_interval = 0 (no time interval), period_interval = 0xFFFF FFFF (max period interval), and max_time = 3 (number of advertisement).

Return:

= 0 – Success

<0 – A kind of error

API name:

```
long sl_NetAppSet(    unsigned char AppId ,
unsigned char Option,
unsigned char OptionLen,
unsigned char *pOptionValue);
```

Table 13-9 lists the defines for this API:

Table 13-9. Defines for API

Name	Description
SL_NET_APP_MDNS_ID	Used as *AppId parameter* in sl_NetAppSet function
NETAPP_SET_GET_MDNS_TIMING_PARAMS_OPT	Used as *option parameter* in sl_NetAppSet function

typedef struct
{
unsigned long t
unsigned long p
unsigned long k;
unsigned long RetransInterval
unsigned long Maxinterval;
unsigned long max_time;
}SlNetAppServiceAdvertiseTimingParameters_t;
	• **t** - Number of ticks for the initial period. Default is 100 ticks for 1 second.
	• **p** - Number of repetitions. Default value is 1.
	• **K** - Telescopic factor. Default value is 1.
	• **retrans_interval** - Number of ticks to wait before sending out repeated announcement messages. Default value is 0.
	• **period_interval** - Number of ticks between two announcement periods. Default value is 0xFFFF FFFF.
	• **max_time** - Number of announcement period to use for the advertisement. Default value is 3.

Example:

```
SlNetAppServiceAdvertiseTimingParameters_t   TimingParams;

TimingParams.t = 200;
TimingParams.p = 2;
TimingParams.k = 2;
TimingParams.RetransInterval = 0;
TimingParams.Maxinterval = 0xffffffff;
TimingParams.max_time = 5;

// announcement timing set
Status = sl_NetAppSet(SL_NET_APP_MDNS_ID
,NETAPP_SET_GET_MDNS_TIMING_PARAMS_OPT,sizeof(TimingParams),(unsigned char*)&amp;TimingParams );
printf("timing status %d\n",Status);
```

13.5.8 API – Get Event Mask

Description: This API gets the configured event mask.

Return:

= 0 – Success

<0 – A kind of error

API name:

```
long sl_NetAppGet(    unsigned char AppId ,
unsigned char Option,
unsigned char OptionLen,
unsigned char *pOptionValue);
```

Example:

```
char          BufferList[1500];
unsigned char     GetLen = 100;
unsigned int     *GetEventMask;
```

```
//get event mask
memset(BufferList,0,sizeof(BufferList));
GetLen = 100;
Status = sl_NetAppGet(SL_NET_APP_MDNS_ID
,NETAPP_SET_GET_MDNS_QEVETN_MASK_OPT,&amp;GetLen,BufferList );
GetEventMask = (unsigned int*)(BufferList);
printf("Get event Mask:\n status %d\n event mask %d\n",Status,*GetEventMask);
```

13.5.9 API – Get Continuous Query

Description: This API gets the configured continuous query.

Return:

= 0 – Success

<0 – A kind of error

API name:

```
long sl_NetAppGet(    unsigned char AppId ,
unsigned char Option,
unsigned char OptionLen,
unsigned char *pOptionValue);
```

Example:

```
char         BufferList[1500];
unsigned char      GetLen = 100;

//get continuous query
memset(BufferList,0,sizeof(BufferList));
GetLen = 100;
Status = sl_NetAppGet(SL_NET_APP_MDNS_ID
,NETAPP_SET_GET_MDNS_CONT_QUERY_OPT,&amp;GetLen,BufferList );
if(GetLen > 0)
{
  printf("Get continuous query:\n status %d\n query %s\n",Status,BufferList);
}
else
{
  printf("Get continuous query:\n status %d\n no query \n",Status);

}
```

13.5.10 API – Get Timing Parameters for Advertising

Description: This API gets the configured timing parameters.

Return:

= 0 – Success

<0 – A kind of error

API name:

```
long sl_NetAppGet(    unsigned char AppId ,
unsigned char Option,
unsigned char OptionLen,
unsigned char *pOptionValue);
```

Example:

```
char                       BufferList[1500];
unsigned char                   GetLen = 100;
SlNetAppServiceAdvertiseTimingParameters_t   *GetTimingParams;

//get timing parameters
memset(BufferList,0,sizeof(BufferList));
GetLen = 100;
Status = sl_NetAppGet(SL_NET_APP_MDNS_ID
,NETAPP_SET_GET_MDNS_TIMING_PARAMS_OPT,&amp;GetLen,BufferList );
GetTimingParams = (SlNetAppServiceAdvertiseTimingParameters_t *)(BufferList);
printf("    Get timing parameters:\n status %d\n start time(ticks) %d\n Numbr of packets each
cycle %d\n Factor %d\n Retransmit on number of packets each cycle %d\n Max Interval between cycle
%u\n Number of cycles %d\n",
            Status,GetTimingParams->t,
GetTimingParams->p,
GetTimingParams->k,
GetTimingParams->RetransInterval,
GetTimingParams->Maxinterval,
GetTimingParams->max_time);
```

Serial Flash File System

Topic Page

14.1 Overview

In a secure file system:

- The file system tables are encrypted.
- The creation of secure files is enabled.
- The maximum number of files supported is 128.
- The file system is blocked after three security alerts.

Secure files characteristics:

- Kept encrypted on the storage
- Authenticated
- Read or write access only allowed to authorized users

14.2 File Download and Creation

Execute the file create command with the following parameters:

- File name
- Maximum file size
- Creation flags
- For a secure file, a master token, which is an in/out parameter
- Return file handle, which is used to write the file

Execute the file write command with the following parameters:

- File handle
- Buffer to write
- Size to write

Execute the file close command with the following parameters:

Note: For a secure file, which requires a signature, the certificate file should be downloaded before the close command.

- File handle
- For a secure file with a signature test:
  - Certificate file name (the certificate file should exist on the storage).
  - Signature (256 or 128 bytes)

14.3 File Download, Open for Write

Execute the file open for write command with the following parameters:

- File name
- For a secure file, a write token, which is an in/out parameter
- Return file handle, which is used to write the file

Execute the file write command with the following parameters:

- File handle
- Buffer to write
- Size to write

Execute the file close command with the following parameters:

- File handle
- For a secure file with a signature test:
  - Certificate file name (the certificate file should exist on the storage).
  - Signature

14.4 File Open for Read

Execute the file open for read command with the following parameters:

- File name
- For a secure file, a read token, which is an in/out parameter
- Return file handle, which is used to write the file

Execute the file read command with the following parameters:

- File handle
- Buffer to read
- Size to read

Execute the file close command with the following parameter:

- File handle

14.5 Secure System Files

These system files are secured:

- Service pack
- SLINK_FILE_PAC_FILE_ID
- SLINK_FILE_FAST_CONNECT_FRAME
- SLINK_FILE_PREFERRED_NETWORKS
- CONFIG_TYPE_SMART_CONFIG_KEYS.

14.6 Commit Creation Flag

When creating a file, the commit flag makes the file fail-safe by creating a mirror. Using the commit flag doubles the file storage size.

When opening a file for read, the last valid file mirror is chosen as the active mirror.

14.7 Security Alert

If the device detects a security intrusion, it creates a security alert. Three security alerts lock the file system, which can only be opened by formatting the storage. In the R1 release, the lock does not execute.

Some security alerts wipe the storage, although not in R1.

14.8 Tokens

Tokens are relevant only for secure files. Secure files can only be created on a secured chip; otherwise the secure file options are ignored.

When a secure file is created, the following tokens are created:

- Master
- Read/Write
- Read only
- Write only

The master token is returned upon file creation (the create function has an in/out parameter which is the token). The other tokens can be received by invoking the GetInfo function with the token as the parameter. If the GetInfo function is invoked with the master token, it returns the other three tokens. The function always returns tokens which have a level lower than the input token.

Without an appropriate token the user cannot open the file for read or write or delete the file. Deleting a file requires the master token.

Using the read token, the user can open the file for read and not for write. This could be used in the case of a logo file, which may be displayed on the web pages but not changed.

Trying to open a secure file with an invalid token (or NULL token) creates a security alert.

When the file is opened for write, all the tokens other than the master token are recreated (the file creator can define different behavior when creating the file = STATIC option).

More token creation options:

- Creating a secure file with public read permission (no token required for read) or public write (no token required for write).
- When creating the file, the master token may be set by the creator (vendor option).

14.9 Signature

Secure files are created with a signature test, by using the *no_signature_test_flag*, or no file authentication occurs.

The file signature is supplied as part of the file close function.

A file created with a signature test authenticates during the close function and every time the file is opened for read/write.

The signature is RSA-SHA1 (key can be sized 1024 or 2048). Certificate chain is also supported.

To sign the files the following steps are required:

1. Create a private key: *\\OpenSSL-Win32\\bin\\openssl genrsa -out [PrivateKey.PEM] [1024|2048]*
2. Transform the private key to DER format: *\\OpenSSL-Win32\\bin\\openssl rsa -in [PrivateKey.PEM] -inform PEM -out [PrivateKey.DER] -outform DER*
3. Create a certificate request using the private key DER format: *\\OpenSSL-Win32\\bin\\openssl req –new –key [PrivateKey.DER] –out [certific.pem]*
4. Send the certificate request *[Certfile.pem]* file to TI, to receive a TI certificate.
5. Transform the certificate from .pem format to .der format: *"C:\\OpenSSL-Win32\\bin\\openssl x509 -in [Certfile.pem] -inform PEM -out [Certfile.der] -outform DER*
6. Download the TI certificate *[Certfile.der]* as a nonsecure file to the device.
7. The private key can be used to create the signature for the secure files. *\\OpenSSL-Win32\\bin\\openssl dgst -binary -sha1 -sign [PrivateKey.PEM] -out [SecureFile1_Signature.bin] [InputFile.txt]*

14.10 Option for File Creation

A definition of a possible option for file creation follows:

```
typedef enum
{
       _FS_MODE_OPEN_READ            = 0,
       _FS_MODE_OPEN_WRITE,
       _FS_MODE_OPEN_CREATE,
       _FS_MODE_OPEN_WRITE_CREATE_IF_NOT_EXIST
}SlFsFileOpenAccessType_e;
```

A definition of the possible flags for file creation follows:

```
typedef enum
{
   _FS_FILE_OPEN_FLAG_COMMIT  =  0x1, //MIRROR - for fail safe
   _FS_FILE_OPEN_FLAG_SECURE  =  0x2, //SECURE
   _FS_FILE_OPEN_FLAG_NO_SIGNATURE_TEST = 0x4, //Relevant to secure file only
   _FS_FILE_OPEN_FLAG_STATIC  =           0x8,  // Relevant to secure file only
   _FS_FILE_OPEN_FLAG_VENDOR  =           0x10, // Relevant to secure file only
   _FS_FILE_PUBLIC_WRITE=                 0x20, //Relevant to secure file only, the file can be
opened for write without Token
   _FS_FILE_PUBLIC_READ =                 0x40  //Relevant to secure file only, the file can be
opened for read without Token
}SlFileOpenFlags_e;
```

14.11 Code Example

```
//Open for write non-secure file with size 63K
RetVal = sl_FsOpen((unsigned char *)DeviceFileName,
                        FS_MODE_OPEN_CREATE(63*1024 , 0 ),
&amp;MasterToken,
&amp;FileHandle);

//Open for write secure file with size 63K
RetVal = sl_FsOpen((unsigned char *)DeviceFileName,
                        FS_MODE_OPEN_CREATE(63*1024 , _FS_FILE_OPEN_FLAG_SECURE |
_FS_FILE_OPEN_FLAG_NO_SIGNATURE_TEST),
&amp;MasterToken,
&amp;FileHandle);

//Open for write secure file with signature with size 63K
RetVal = sl_FsOpen((unsigned char *)DeviceFileName,
                        FS_MODE_OPEN_CREATE(63*1024 , _FS_FILE_OPEN_FLAG_SECURE),
&amp;MasterToken,
&amp;FileHandle);

//Write to file
RetVal = sl_FsWrite(FileHandle,
                    (unsigned int)Offset
                    (unsigned char *)Buffer, BufferSize);

//Close
RetVal = sl_FsClose( FileHandle, 0, 0, 0);

//Close with signature
RetVal = sl_FsClose(DeviceFileHandle, pCeritificateFileName, pSignature , SignatureLen);
```

Chapter 15

SWRU368–June 2014

Rx Filter

Topic Page

15.1 Overview

The Rx-filters module lets the user define which of the received frames will be transferred by the CC3100 to the host and which frames will be dropped. The Rx-filters can be activated during AP-connection and during promiscuous mode (disconnect mode).

15.2 Detailed Description

Every received frame traverses through a series of decision trees that determine how the frame is treated.

The decision trees are composed of filter nodes. Each node has its filter rule, action, and trigger. The tree traversal process starts with the trees' root nodes: if the filter rule and trigger of the root node are TRUE, the action of the root note is performed and the frame continues to the child nodes. Filter rules are specific protocol header values:

- MAC layer: Frame type, frame sub-type, BSSID, source MAC address, destination MAC address, and frame length
- LLC layer: Protocol type
- Upper layers: IP version, IP protocol, source IP address, destination IP address, ARP operation, ARP target IP address, source port number, and destination port number

Possible triggers:

- When role is… (station/AP/promiscuous)
- When connection state is… (connected/disconnected)
- When counter reaches X

Possible actions:

- Drop the packet (do not pass it to the host)
- Auto reply using pre-defined Tx template
- Increase or decrease the counter value

Trees traversal is stopped when the frame reaches a DROP action in one of the trees. Traversing is done layer by layer among all the trees.

The user can define a combined-filter node. This node has two parent nodes (unlike a regular node which only has one), and is checked only if one or both (user-defined) of its two parent nodes is TRUE. For example: if (node_1 OR node_2).

15.3 Examples

Example 1 – Supposing a user has the following requirements:

- Receive WLAN data broadcast frames only from two specific MAC addresses
- Receive all WLAN unicast frames, except for frames with a certain SRC_IP address range
- If a unicast frame is received from MAC address AA.AA.AA, increase counter_1.
- If a unicast frame is received from MAC address BB.BB.BB, increase counter_2.
- If a unicast UDP frame is received from MAC address AA.AA.AA or BB.BB.BB, pass only packets from port 5001.

The following trees should be created: Figure 15-1

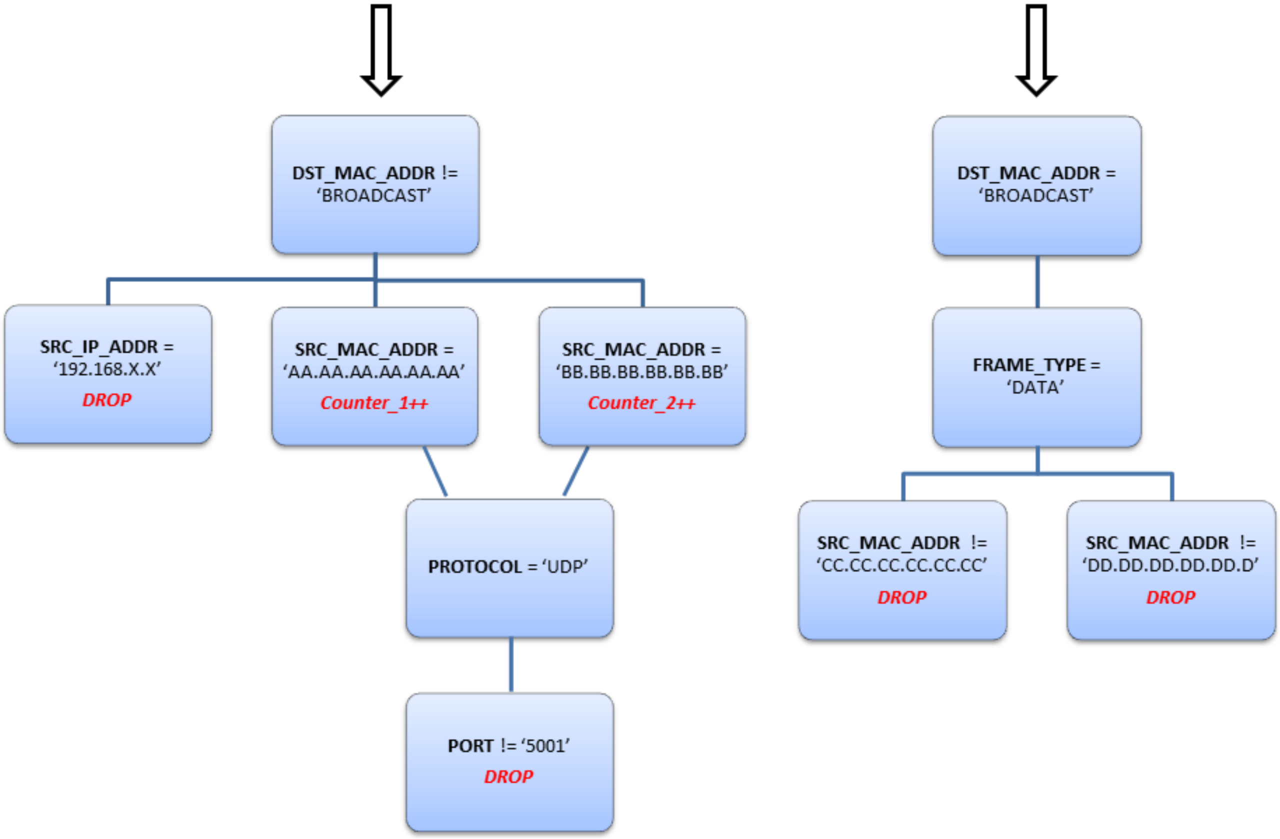

DST_MAC_ADDR != 'BROADCAST'
SRC_IP_ADDR = '192.168.X.X'
DROP
SRC_MAC_ADDR = 'AA.AA.AA.AA.AA.AA'
Counter_1++
SRC_MAC_ADDR = 'BB.BB.BB.BB.BB.BB'
Counter_2++
PROTOCOL = 'UDP'
PORT != '5001'
DROP
DST_MAC_ADDR = 'BROADCAST'
FRAME_TYPE = 'DATA'
SRC_MAC_ADDR != 'CC.CC.CC.CC.CC.CC'
DROP
SRC_MAC_ADDR != 'DD.DD.DD.DD.DD.D'
DROP

Figure 15-1. Trees Example 1

Example 2 – Supposing a user has the following requirements:

- Receive only WLAN management beacon frames from all MAC addresses

The following trees should be created:Figure 15-2

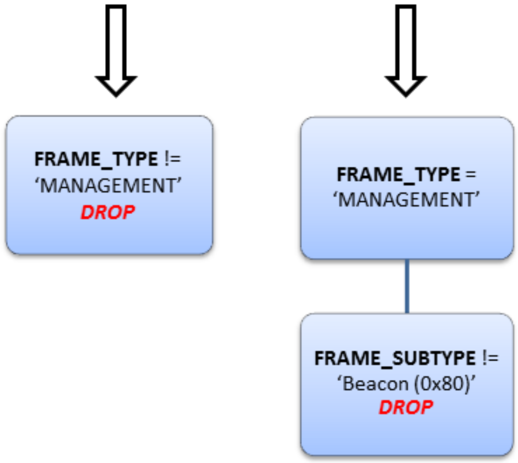


Figure 15-2. Trees Example 2

15.4 Creating Trees

- Trees are created by the user. The user adds the filter nodes individually and defines the filter tree hierarchy.
- Trees can also be created and applied internally by the system.
- Filters can be created as persistent filters that are saved in the FLASH memory and loaded at the system startup.
- The maximal number of filter nodes is 64. 14 filter nodes are used by the system, and the remaining 50 nodes are for the user.
- Filters can be created, removed, enabled, and disabled. After filters are created, they must be enabled to start filtering.

15.5 Rx Filter API

WlanRxFilterAdd – Adds a new filter to the system

sl_WlanRxFilterEnableDisable – Enables or disables filters

sl_WlanRxFilterRemove – Removes filter

sl_WlanRxFilterGetGeneralStatisticsInfo – Retrieves general statistics information regarding the filters

sl_WlanRxFilterGetNodeStatisticsInfo – Retrieves statistics information regarding a specific filter

sl_WlanRxFilterSaveToFlash – Saves the persistent filters to flash

sl_WlanRxFilterUpdateFilterArgs – Updates Args of an existing filter

sl_WlanRxFilterPrePreparedFiltersOperation – Changes or retrieves the internal filter creation default

15.5.1 Code Example

```
#include "datatypes.h"
#include "simplelink.h"
#include "protocol.h"
#include "driver.h"
#include "debug.h"
```

```
void RxFiltersExample( char input )

{
SlrxFilterID_t          FilterId;
    SlrxFilterRuleType_t    RuleType;
    SlrxFilterFlags_t       FilterFlags;
    SlrxFilterRule_t        Rule;
    SlrxFilterTrigger_t     Trigger;
    SlrxFilterAction_t      Action;
    SlrxFilterOperationResult_t RetVal;
    unsigned char MyLabCompIPAddress[4] = {0x0A,0x01,0x04,0x0A};
    unsigned char MyMAsk[4]      = {0xFF,0xFF,0xFF,0xFF};

    switch(input)
    {
        case '1': //Create filter
            //Rule definition
            RuleType = HEADER;
            FilterFlags.IntRepresentation = RX_FILTER_BINARY ;
            Rule.HeaderType.RuleHeaderfield = IPV4_SRC_ADRRESS_FIELD;
            Rule.HeaderType.RuleCompareFunc = COMPARE_FUNC_EQUAL;
            memcpy(
Rule.HeaderType.RuleHeaderArgsAndMask.RuleHeaderArgs.RxFilterDB4BytesRuleArgs[0],
MyLabCompIPAddress , 4 );
            memcpy( Rule.HeaderType.RuleHeaderArgsAndMask.RuleHeaderArgsMask, MyMAsk , 4 );
            //parent
            Trigger.ParentFilterID = 0;
            //When RX_FILTER_COUNTER7 is bigger than 0
            Trigger.Trigger = RX_FILTER_COUNTER7;
            Trigger.TriggerCompareFunction = TRIGGER_COMPARE_FUNC_EQUAL;
            Trigger.TriggerArg = 0;
            //connection state and role
            Trigger.TriggerArgConnectionState.IntRepresentation =
RX_FILTER_CONNECTION_STATE_STA_HAS_IP;
            Trigger.TriggerArgRoleStatus.IntRepresentation = RX_FILTER_ROLE_STA;
            //Action
            Action.ActionType.IntRepresentation = RX_FILTER_ACTION_ON_REG_INCREASE;
            Action.ActionArg[ 0 ]  = RX_FILTER_COUNTER6;


RetVal = (SlrxFilterOperationResult_t)sl_WlanRxFilterAdd(   RuleType,
                                            FilterFlags,

&amp;Rule,

&amp;Trigger,

&amp;Action,

&amp;FilterId);

            break;
        case '2' : //remove filter
            {
                SlrxFilterIdMask_t      RxFilterIdMask ;
                //remove all
                memset( RxFilterIdMask, 0xFF , 16 );
                RetVal = sl_WlanRxFilterRemove( RxFilterIdMask );
            }
            break;
        case '3' : //enable/disable filter
            {
                SlrxFilterIdMask_t      RxFilterIdMask ;
                //Enable All
                memset( RxFilterIdMask, 0xFF , 16 );
                RetVal = sl_WlanRxFilterEnableDisable( RxFilterIdMask);
            }
            break;
case '4'://Get filters statistics
            {
                SlrxFilterStatisticsOperation_t ResetStatistics = 0;
                unsigned long  StatsInputCounter;
                unsigned long  StatsMatchCounter;
                unsigned long  StatsMaxProcessingTime[2];
                unsigned long  StatsAverageProcessingTime[2];

                RetVal = sl_WlanRxFilterGetGeneralStatisticsInfo( ResetStatistics,
&amp;StatsInputCounter, &amp;StatsMatchCounter, StatsMaxProcessingTime,
StatsAverageProcessingTime );
            }
            break;
        case '5': //save the persistent filters to flash
            RetVal =  sl_WlanRxFilterSaveToFlash();
            break;
case '6' : //change the creation default of the internal filters
            {
                SlrxFilterPrePreparedFilterstOperation_t FilterPrePreparedFiltersOperation =
SL_FILTER_PRE_PREPARED_SET_CREATE_REMOVE_STATE;
               SlrxFilterPrePreparedFiltersMask_t  FilterPrePreparedFiltersMask ;

                memset(FilterPrePreparedFiltersMask, 0, sizeof(FilterPrePreparedFiltersMask));
                FilterPrePreparedFiltersMask[ 0 ] = 0xF0;
                RetVal =  sl_WlanRxFilterPrePreparedFiltersOperation(
FilterPrePreparedFiltersOperation ,  FilterPrePreparedFiltersMask );
           }
           break;
       case '7' : //update filter args
           {
               SlrxFilterRuleHeaderArgsAndMask_t  FilterRuleHeaderArgsAndMask;
                unsigned char BinaryRepresentation = 1;
                FilterId = 9;
                memcpy( FilterRuleHeaderArgsAndMask.RuleHeaderArgs.RxFilterDB4BytesRuleArgs,
MyLabCompIPAddress , 4 );
                memcpy( FilterRuleHeaderArgsAndMask.RuleHeaderArgsMask, MyMAsk , 4 );
                FilterRuleHeaderArgsAndMask.RuleHeaderArgs.RxFilterDB4BytesRuleArgs[0][0] = 0xBB;

                FilterRuleHeaderArgsAndMask.RuleHeaderArgsMask[ 0 ] = 0xDD;
                RetVal = sl_WlanRxFilterUpdateFilterArgs( FilterId ,
&amp;FilterRuleHeaderArgsAndMask , BinaryRepresentation );
            }
            break;

    }

}
//******************************************************************************
//    RX filters
//******************************************************************************

int main( int argc, char *argv[] )
{
    sl_Start(NULL, NULL, NULL);

    RxFiltersExample( '1' );  //create filter
    RxFiltersExample( '3' );  //enable all
    RxFiltersExample( '4' );  //Get statistics
    RxFiltersExample( '6' );  //change prepared
    RxFiltersExample( '7' );  //Update
    RxFiltersExample( '5' );  //SaveTo flash
    while (1);
}
```

Transceiver Mode

Topic **Page**

16.1 General Description

The 802.11 transceiver mode is a powerful tool for sending and receiving raw data in either Layer 2 (MAC) or Layer 1 (Physical).

There are eight network layers:

Table 16-1. Network Layers

Application
Presentation
Session
Transport
Network
LLC – Logical Link Control
MAC – Medium Access Control
Physical

The user could use the entire frame's space starting from the 802.11 header (excluding the duration field) to receive and transmit data.

In transceiver mode, there are no frame acknowledgments or retries. Therefore there are no guarantees that the frame will reach its destination. When working in L1 mode, there may be collisions with other frames or energy.

Figure 16-1 illustrates the 802.11 frame structure. The white sections are user-configurable, and the grayed part cannot be overwritten.

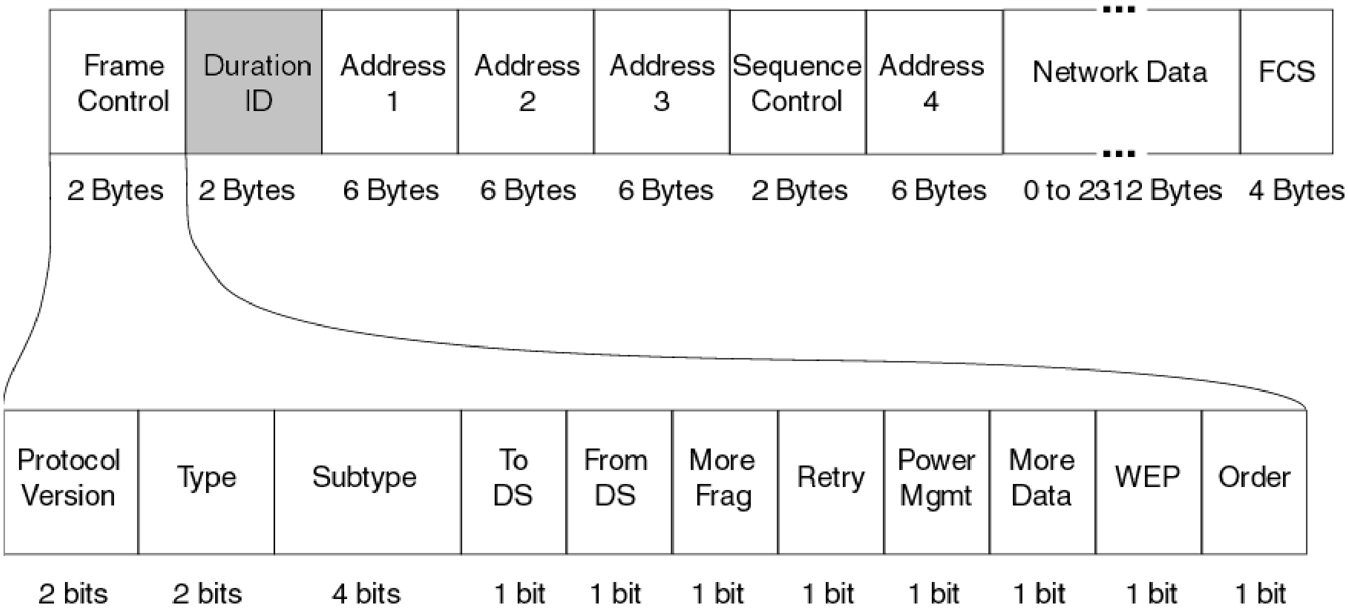


Figure 16-1. 802.11 Frame Structure

16.2 How to Use / API

Using the host driver that controls the CC3100 (which includes the SimpleLink studio), the user needs no more than five commands to work with the abilities of the 802.11 transceiver. As the CC3100 is a networking device that complies with BSD socket implementation, use the command **sl_Socket** with the following arguments to start the transceiver.

```
soc = sl_Socket(SL_AF_RF ,SL_SOCK_RAW/SL_SOCK_DGRAM,channel);
```

SL_AF_RF – Indicates from what level of network the user can override the frame.

SL_SOCK_RAW – Indicates an L1 raw socket (no respect for 802.11 medium access policy (CCA))

SL_SOCK_DGRAM – Indicates an L2 raw socket (respecting 802.11 medium access policies)

channel – Configures the operational channel to start receiving or transmitting traffic. Use 0 to keep the default channel.

This command returns a socket ID, a two byte integer that will be used to reference the socket. If there is a problem with the socket, the command will return an error code.

To close the socket, use the command **sl_Close**:

```
sl_Close(soc);
```

16.3 Sending and Receiving

The user can open and close the transceiver using two commands. To start the traffic flow, use the **sl_Send** command to transmit and the **sl_Recv** command to receive.

```
sl_Send(soc ,RawData ,len ,flags);
```

soc – Socket ID

RawData – The char *array holds the data to send, beginning with the first byte of the 802.11 MAC header.

len – The size of the data in bytes

flags – Usually the user sets this parameter to 0, but the user can change any of the default channel/rate/tx-power/11b-preamble parameters using this parameter. Use the **SL_RAW_RF_TX_PARAMS** macro to specify the mentioned parameters.

Returns – The number of bytes sent

For example, to transmit a frame on channel 1 using the 1MBPS data rate with Tx power setting of 1 and short preamble (valid only for 11b), use:

```
sl_Send(soc,buf,len,SL_RAW_RF_TX_PARAMS(CHANNEL_1, RATE_1M,1, TI_SHORT_PREAMBLE));
```

```
sl_Recv(soc,buffer,500,0);
```

soc – Socket ID

buffer – The char *array used for containing the received packet

500 – The maximum size of the packet received. The max size is 1472: if the packet is longer, the rest will be discarded.

The last argument should always be 0, which indicates no flags.

Return – The number of bytes sent

16.4 Changing Socket Properties

The command **sl_SetSockOpt** changes the socket properties (after opening the socket).

Examples:

Change the operating channel:

```
sl_SetSockOpt(soc, SL_SOL_SOCKET,  SO_CHANGE_CHANNEL, &amp;channel,1);
```

Change the default PHY data rate:

```
sl_SetSockOpt(soc, SL_SOL_PHY_OPT,  SO_PHY_RATE, &amp;rate,1);
```

Change the default Tx power:

```
sl_SetSockOpt(soc, SL_SOL_PHY_OPT,  SO_PHY_TX_POWER, &amp;power,1);
```

Change the number of frames to transmit (see Section 16.5):

```
sl_SetSockOpt(soc, SL_SOL_PHY_OPT,  SO_PHY_NUM_FRAMES_TO_TX, &amp;numFrames,1);
```

Change the 802.11b preamble:

```
sl_SetSockOpt(soc, SL_SOL_PHY_OPT,  SO_PHY_PREAMBLE,&amp;preamble,,1);
```

16.5 Internal Packet Generator

For testing purposes, there is an internal packet generator in the CC3100 capable of repeating a pre-defined pattern of data.

To use, **before** calling **sl_Send**, set the number of frames using the **sl_SetSockOpt** to either 0 (an infinite number of frames) or to the given number of frames needed to transmit.

Following that, use a single call to **sl_Send** API to trigger the frames transmission.

The CC3100 will keep transmitting until it has sent all the requested frames, or until the socket is closed or another socket property changes (in case an infinite number of frames were used).

16.6 Transmitting CW (Carrier-Wave)

To transmit a carrier-wave signal, use the **sl_Send** API with NULL buffer and 0 (zero) length.

Use the flags parameter in the **sl_Send** API to signal the tone offset (-25 to 25).

Stopping CW transmission is done by triggering another **sl_Send** with flags = 128 (decimal) as follows: **sl_Send (soc,NULL,0,128);**

16.7 Connection Policies and Transceiver Mode

To use transceiver mode, disable previous connection policies that might attempt to automatically connect to an access-point.

Example:

```
sl_WlanPolicySet(SL_POLICY_CONNECTION,SL_CONNECTION_POLICY(0,0,0,0,0),NULL,0);
sl_WlanDisconnect();
```

16.8 Notes about Receiving and Transmitting

16.8.1 Receiving

The packet being received has a SimpleLink proprietary radio header attached to it. The header has some information about the packet. This is the structure of the header.

The rate is an index from 0 to 20 in the following order:

RATE 1M = 0

RATE 2M = 1

RATE 5.5M = 2

RATE 11M = 3

RATE 6M = 4

RATE 9M = 6

RATE 12M = 7

RATE 18M = 8

RATE 24M = 9

RATE 36M = 10

RATE 48M = 11

RATE 54M = 12

RATE MCS_0 = 13

RATE MCS_1 = 14

RATE MCS_2 = 15

RATE MCS_3 = 16

RATE MCS_4 = 17

RATE MCS_5 = 18

RATE MCS_6 = 19

RATE MCS_7 = 20

The channel is 1 to 11.

If using the **sl_Recv** command resulted in a frame in a buffer, extract the header by casting the start of the buffer to a pointer variable in type of *TransceiverRxOverHead_t*

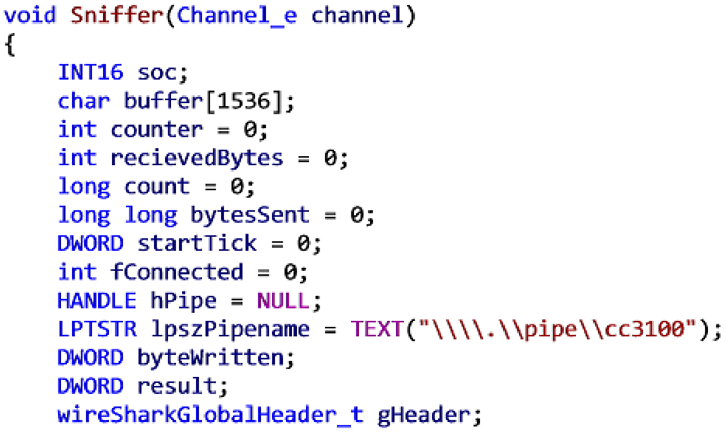
```
frameRadioHeader = (TransceiverRxOverHead_t *)buffer;
```

16.9 Use Cases

The following key applications can be developed using this feature.

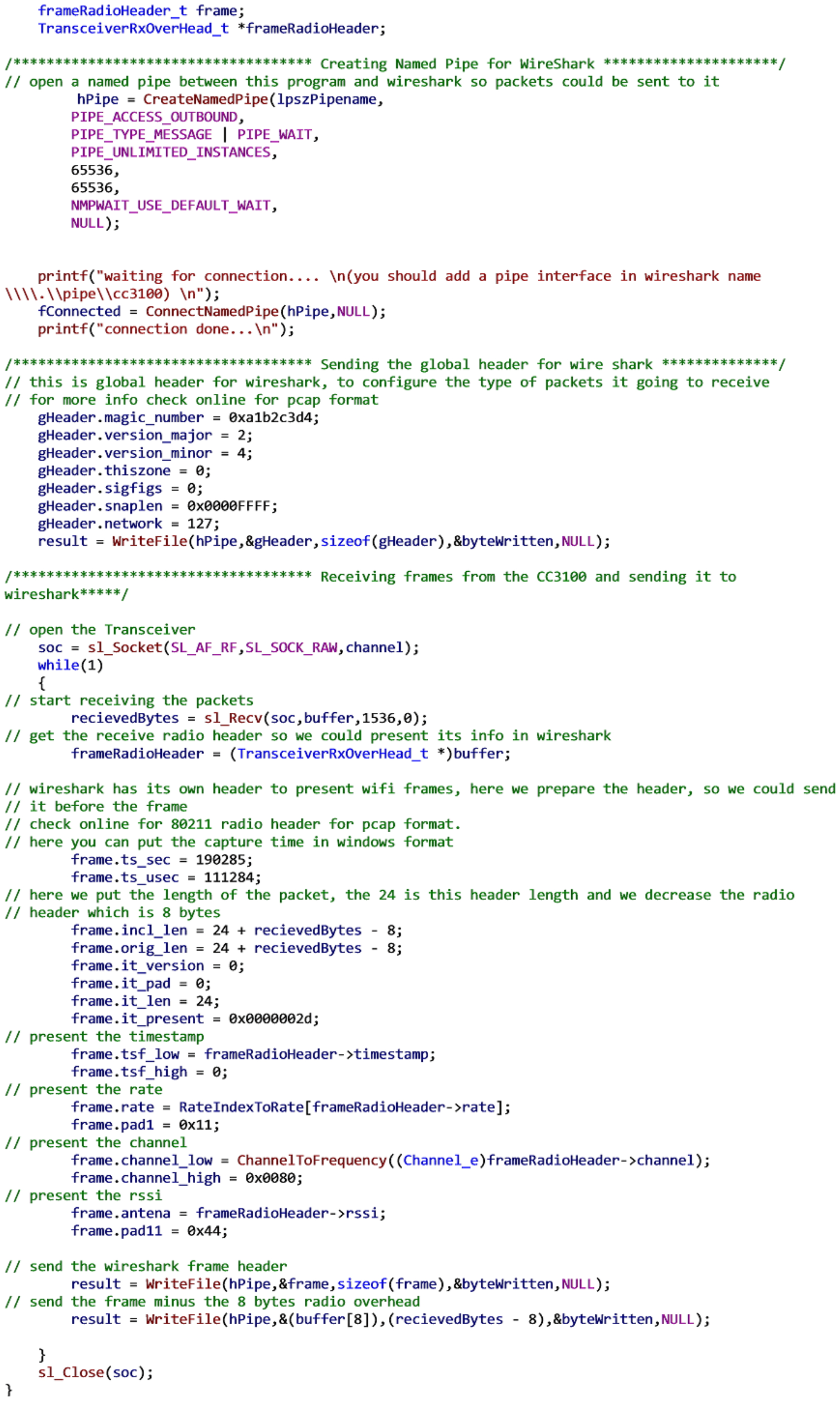
16.9.1 Sniffer

The transceiver can be used as a sniffer. Open a socket and receive packets in a loop using the **sl_Recv** command. The following code describes how to capture frames and present them in wireshark:



```
void Sniffer(Channel_e channel)
{
    INT16 soc;
    char buffer[1536];
    int counter = 0;
    int recievedBytes = 0;
    long count = 0;
    long long bytesSent = 0;
    DWORD startTick = 0;
    int fConnected = 0;
    HANDLE hPipe = NULL;
    LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\cc3100");
    DWORD byteWritten;
    DWORD result;
    wireSharkGlobalHeader_t gHeader;
```

Figure 16-2. Sniffer

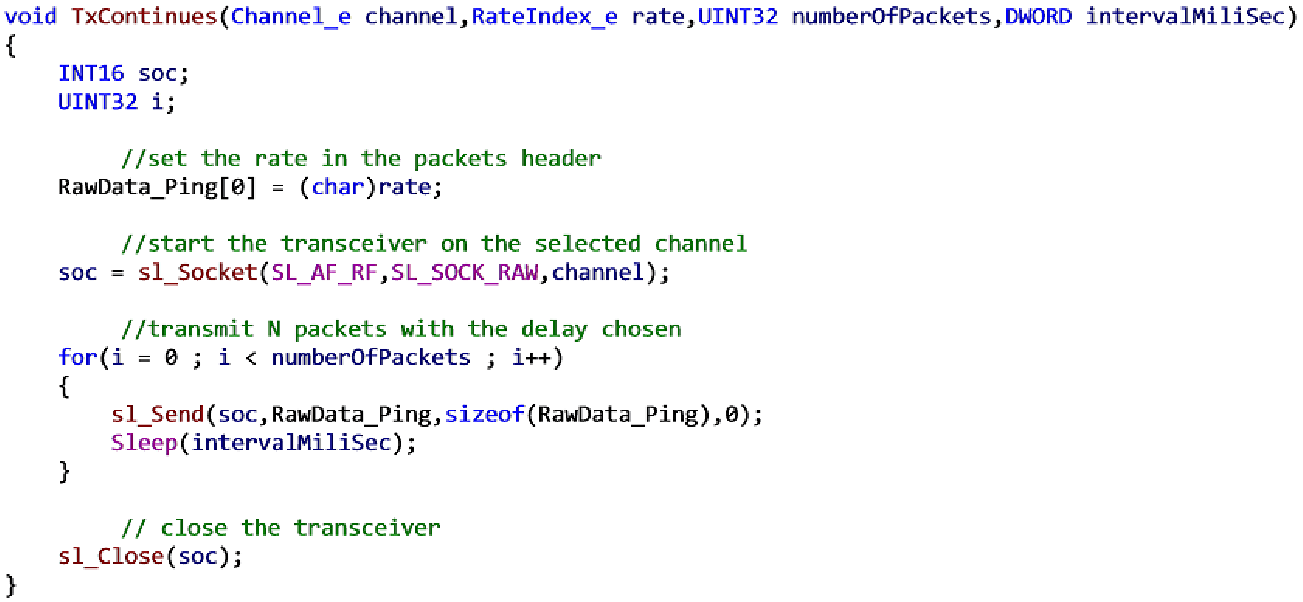


```
    frameRadioHeader_t frame;
    TransceiverRxOverHead_t *frameRadioHeader;

/************************************ Creating Named Pipe for WireShark *********************/
// open a named pipe between this program and wireshark so packets could be sent to it
         hPipe = CreateNamedPipe(lpszPipename,
        PIPE_ACCESS_OUTBOUND,
        PIPE_TYPE_MESSAGE | PIPE_WAIT,
        PIPE_UNLIMITED_INSTANCES,
        65536,
        65536,
        NMPWAIT_USE_DEFAULT_WAIT,
        NULL);


    printf("waiting for connection.... \n(you should add a pipe interface in wireshark name
\\\\.\\pipe\\cc3100) \n");
    fConnected = ConnectNamedPipe(hPipe,NULL);
    printf("connection done...\n");

/************************************ Sending the global header for wire shark **************/
// this is global header for wireshark, to configure the type of packets it going to receive
// for more info check online for pcap format
    gHeader.magic_number = 0xa1b2c3d4;
    gHeader.version_major = 2;
    gHeader.version_minor = 4;
    gHeader.thiszone = 0;
    gHeader.sigfigs = 0;
    gHeader.snaplen = 0x0000FFFF;
    gHeader.network = 127;
    result = WriteFile(hPipe,&gHeader,sizeof(gHeader),&byteWritten,NULL);

/************************************ Receiving frames from the CC3100 and sending it to
wireshark*****/

// open the Transceiver
    soc = sl_Socket(SL_AF_RF,SL_SOCK_RAW,channel);
    while(1)
    {
// start receiving the packets
        recievedBytes = sl_Recv(soc,buffer,1536,0);
// get the receive radio header so we could present its info in wireshark
        frameRadioHeader = (TransceiverRxOverHead_t *)buffer;

// wireshark has its own header to present wifi frames, here we prepare the header, so we could send
// it before the frame
// check online for 80211 radio header for pcap format.
// here you can put the capture time in windows format
        frame.ts_sec = 190285;
        frame.ts_usec = 111284;
// here we put the length of the packet, the 24 is this header length and we decrease the radio
// header which is 8 bytes
        frame.incl_len = 24 + recievedBytes - 8;
        frame.orig_len = 24 + recievedBytes - 8;
        frame.it_version = 0;
        frame.it_pad = 0;
        frame.it_len = 24;
        frame.it_present = 0x0000002d;
// present the timestamp
        frame.tsf_low = frameRadioHeader->timestamp;
        frame.tsf_high = 0;
// present the rate
        frame.rate = RateIndexToRate[frameRadioHeader->rate];
        frame.pad1 = 0x11;
// present the channel
        frame.channel_low = ChannelToFrequency((Channel_e)frameRadioHeader->channel);
        frame.channel_high = 0x0080;
// present the rssi
        frame.antena = frameRadioHeader->rssi;
        frame.pad11 = 0x44;

// send the wireshark frame header
        result = WriteFile(hPipe,&frame,sizeof(frame),&byteWritten,NULL);
// send the frame minus the 8 bytes radio overhead
        result = WriteFile(hPipe,&(buffer[8]),(recievedBytes - 8),&byteWritten,NULL);

    }
    sl_Close(soc);
}
```

Figure 16-3. Sniffer

16.10 TX Continues

This application transmits the same packet in continues. This application tags and measures loss using the Rx Statistics feature. The following code shows how to use this feature:



```
void TxContinues(Channel_e channel,RateIndex_e rate,UINT32 numberOfPackets,DWORD intervalMiliSec)
{
    INT16 soc;
    UINT32 i;

         //set the rate in the packets header
    RawData_Ping[0] = (char)rate;

         //start the transceiver on the selected channel
    soc = sl_Socket(SL_AF_RF,SL_SOCK_RAW,channel);

         //transmit N packets with the delay chosen
    for(i = 0 ; i < numberOfPackets ; i++)
    {
        sl_Send(soc,RawData_Ping,sizeof(RawData_Ping),0);
        Sleep(intervalMiliSec);
    }

         // close the transceiver
    sl_Close(soc);
}
```

Figure 16-4. Tx Continues

Rx Statistics

17.1 General Description

The Rx Statistics feature is used to determine certain important parameters about the medium and the CC3100 RX mechanism. Rx Statistics provides data about:

- RSSI – Receive power in dbm units.
  - RSSI histogram, between -40 and -87 dbm in resolution of 8 dbm.
  - Average RSSI divided to DATA + CTRL / MANAGEMENT.
- Received frames – Divided into valid frames, FCS error and PLCP error frames.
- Rate histogram – Creates a histogram of all BGN rates 1mbps-MCS7.

17.2 How to Use / API

Three commands get the needed statistics. The first command is **sl_WlanRxStatStart ()**, which starts collecting the data about all Rx frames.

sl_WlanRxStatStop(), stops collecting the data.

sl_WlanRxStatGet(), gets the statistics that have been collected and returns them in a variable type **SlGetRxStatResponse_t**. The command is used as follows:

```
SlGetRxStatResponse_t rxStatResp;
sl_WlanRxStatGet
(&amp;rxStatResp , 0 );
```

The second parameter is flagged and it is not currently in use.

The **SlGetRxStatResponse_t** holds the following variables:

```
typedef struct
{
    UINT32  ReceivedValidPacketsNumber;
    UINT32  ReceivedFcsErrorPacketsNumber;
    UINT32  ReceivedPlcpErrorPacketsNumber;
    INT16   AvarageDataCtrlRssi;
    INT16   AvarageMgMntRssi;
    UINT16  RateHistogram[NUM_OF_RATE_INDEXES];
    UINT16  RssiHistogram[SIZE_OF_RSSI_HISTOGRAM];
    UINT16  Padding[1];
    UINT32  StartTimeStamp;
    UINT32  GetTimeStamp;
}_GetRXStatResponse_t;
```

ReceivedValidPacketsNumber – Holds the number of valid packets received

ReceivedFcsErrorPacketsNumber – Holds the number of FCS error packets dropped

ReceivedPlcpErrorPacketsNumber – Holds the number of PLCP error packets dropped

avarageDataCtrlRssi – Holds the average data + ctrl frames RSSI

avarageMgMntRssi – Holds the average management frames RSSI

RateHistogram[NUM_OF_RATE_INDEXES] – Histogram of all rates of the received valid frames. The rates are sorted as follows:

RATE_1M = 0 …

RATE_2M

RATE_5_5M

RATE_11M

RATE_6M

RATE_9M

RATE_12M

RATE_18M

RATE_24M

RATE_36M

RATE_48M

RATE_54M

RATE_MCS_0

RATE_MCS_1

RATE_MCS_2

RATE_MCS_3

RATE_MCS_4

RATE_MCS_5

RATE_MCS_6

RATE_MCS_7

NUM_OF_RATE_INDEXES is 21.

RssiHistogram[SIZE_OF_RSSI_HISTOGRAM] – Histogram that holds the accumulated RSSI of all received packets between -40 and -87 dbm every 8 dbm.

SIZE_OF_RSSI_HISTOGRAM is 6

Padding[1]

StartTimeStamp – Holds the start collecting time in microsec

GetTimeStamp – Holds the statistics get time in microsec

17.3 Notes about Receiving and Transmitting

Every packet that exceeds the upper or lower boundary of the RSSI histogram in dbm (-40 or -87) is accumulated in the higher or lower cell respectively.

Every call to **sl_WlanRxStatGet** resets the statistics database.

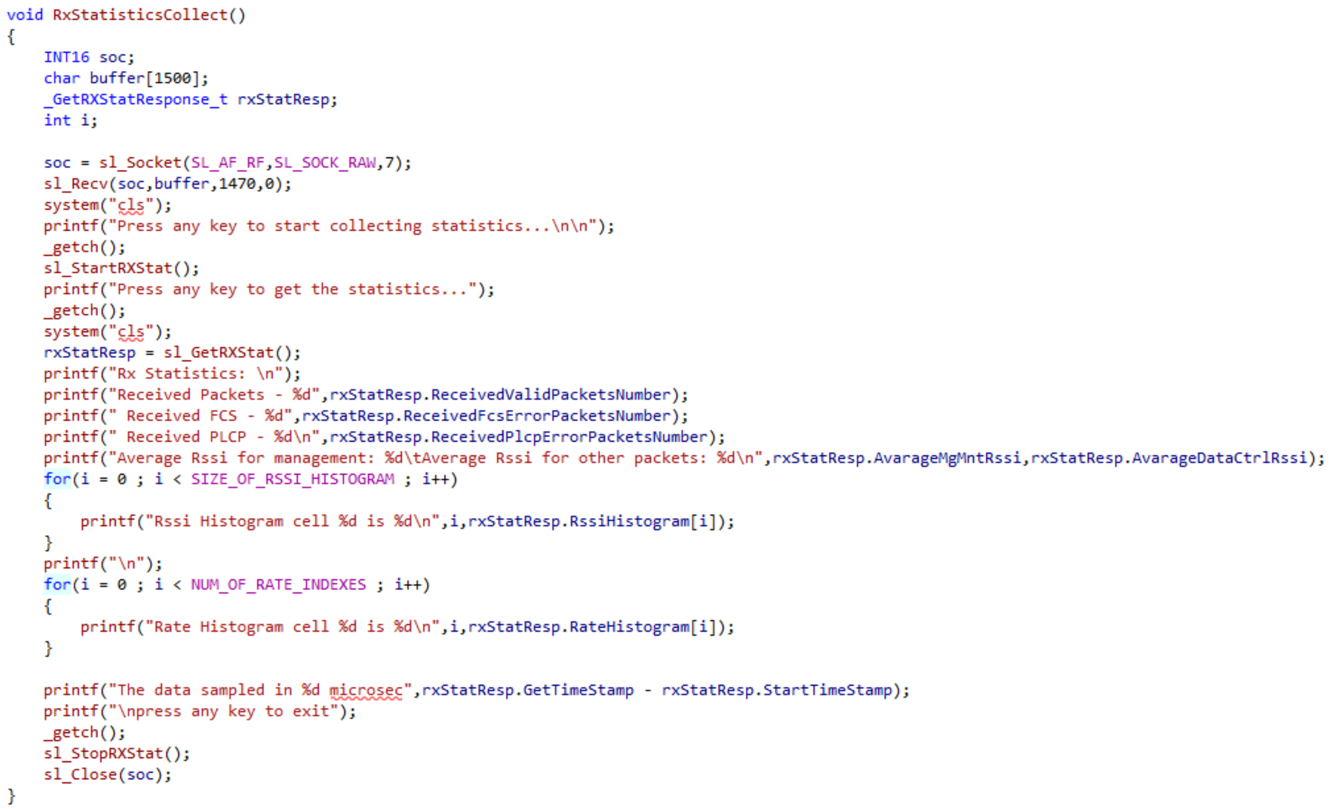
When calling only **sl_WlanRxStatGet**, without the start command, only the RSSI is available.

17.4 Use Cases

The Rx Statistics feature inspects the medium in terms of congestion and distance, validates the RF hardware, and uses the RSSI information to position the CC3100 in an ideal location.

Example:

Connect the SimpleLink device to an AP, run a UDP flow of packets to the device from the AP and use the following code with the SimpleLink Studio to get Rx statistics.



```
void RxStatisticsCollect()
{
    INT16 soc;
    char buffer[1500];
    _GetRXStatResponse_t rxStatResp;
    int i;

    soc = sl_Socket(SL_AF_RF,SL_SOCK_RAW,7);
    sl_Recv(soc,buffer,1470,0);
    system("cls");
    printf("Press any key to start collecting statistics...\n\n");
    _getch();
    sl_StartRXStat();
    printf("Press any key to get the statistics...");
    _getch();
    system("cls");
    rxStatResp = sl_GetRXStat();
    printf("Rx Statistics: \n");
    printf("Received Packets - %d",rxStatResp.ReceivedValidPacketsNumber);
    printf(" Received FCS - %d",rxStatResp.ReceivedFcsErrorPacketsNumber);
    printf(" Received PLCP - %d\n",rxStatResp.ReceivedPlcpErrorPacketsNumber);
    printf("Average Rssi for management: %d\tAverage Rssi for other packets: %d\n",rxStatResp.AvarageMgMntRssi,rxStatResp.AvarageDataCtrlRssi);
    for(i = 0 ; i < SIZE_OF_RSSI_HISTOGRAM ; i++)
    {
        printf("Rssi Histogram cell %d is %d\n",i,rxStatResp.RssiHistogram[i]);
    }
    printf("\n");
    for(i = 0 ; i < NUM_OF_RATE_INDEXES ; i++)
    {
        printf("Rate Histogram cell %d is %d\n",i,rxStatResp.RateHistogram[i]);
    }

    printf("The data sampled in %d microsec",rxStatResp.GetTimeStamp - rxStatResp.StartTimeStamp);
    printf("\npress any key to exit");
    _getch();
    sl_StopRXStat();
    sl_Close(soc);
}
```

Figure 17-1. Use Cases

Chapter 18

SWRU368–June 2014

API Overview

Topic **Page**

This chapter discusses the different SimpleLink APIs. The chapter does not cover the number of parameters, parameter types, or return values. For that information, refer to the API doxygen guide.

The APIs are separated into six main groups:

- Device
- NetConfig
- WLAN
- Socket
- NetApp
- File System

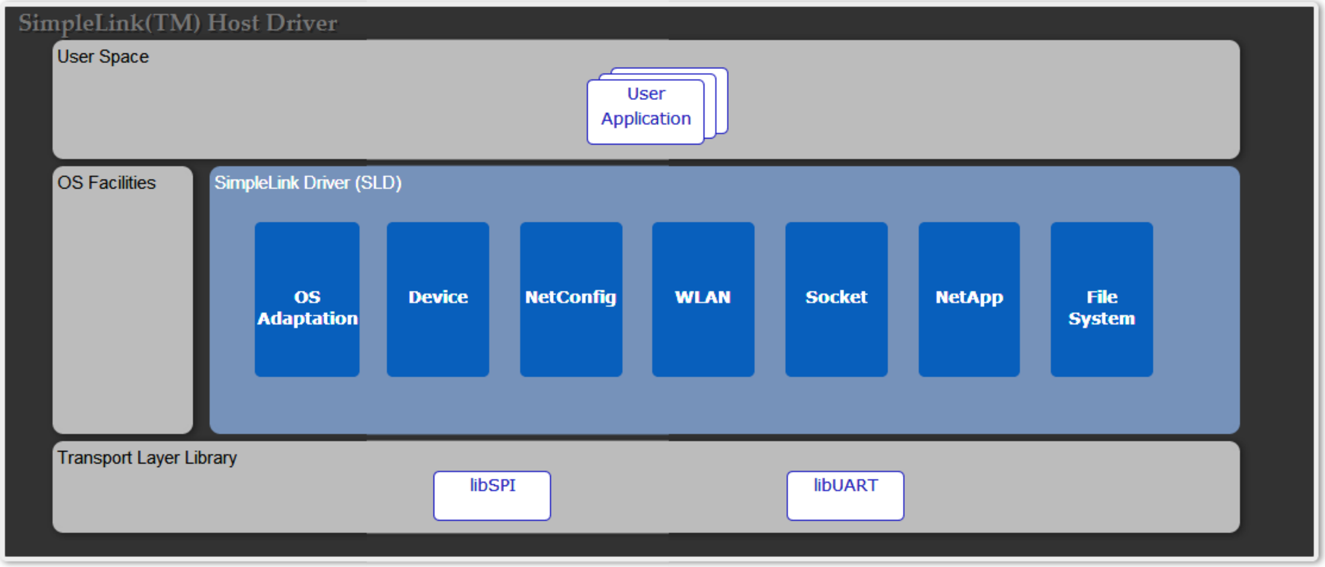


Figure 18-1. Host Driver API Silos

18.1 Device

The device APIs handle the device power and general configuration.

Sl_Start – This function initializes the communication interface, sets the enable pin of the device, and calls to the init complete callback. If no callback function is provided, the function is blocking until the device finishes the initialization process. The device returns to its functional role in case of success: ROLE_STA, ROLE_AP, ROLE_P2P. Otherwise, in case of a failure it returns: ROLE_STA_ERR, ROLE_AP_ERR, ROLE_P2P_ERR

Sl_Stop – This function clears the enable pin of the device, closes the communication interface, and invokes the stop complete callback (if it exists). The timeout parameter enables the user to control the hibernate timing:

- 0 – Enter to hibernate immediately
- 0xFFFF – The host waits for the device to respond before hibernating, without timeout protection.
- 0 < Timeout[msec] < 0xFFFF – The host waits for the device to respond before hibernating, with a defined timeout protection. This timeout defines the maximum time to wait. The NWP response can be sent earlier than this timeout.

sl_DevSet – This function configures different device parameters. The main parameters used are the DeviceSetID and Option. The possible DeviceSetID and Option combinations are:

- SL_DEVICE_GENERAL_CONFIGURATION – The general configuration options are:
  - SL_DEVICE_GENERAL_CONFIGURATION_DATE_TIME – Configures the device internal date and time. Note that the time parameter is retained in hibernate mode but will reset in shutdown.

Setting device time and date example:

```
SlDateTime_t dateTime= {0};
dateTime.sl_tm_day =   (unsigned long)23;          // Day of month (DD format) range 1-13
dateTime.sl_tm_mon =   (unsigned long)6;           // Month (MM format) in the range of 1-12
dateTime.sl_tm_year =  (unsigned long)2014;        // Year (YYYY format)
dateTime.sl_tm_hour =  (unsigned long)17;          // Hours in the range of 0-23
dateTime.sl_tm_min =   (unsigned long)55;          // Minutes in the range of 0-59
dateTime.sl_tm_sec =   (unsigned long)22;          // Seconds in the range of  0-59
sl_DevSet(SL_DEVICE_GENERAL_CONFIGURATION,SL_DEVICE_GENERAL_CONFIGURATION_DATE_TIME,sizeof(SlDateT
ime_t),(unsigned char *
) (&amp;dateTime));
```

sl_DevGet – This function enables the user to read different device parameters. The main parameters used are the DeviceSetID and Option parameter. The possible DeviceSetID and Option combinations are:

- SL_DEVICE_GENERAL_VERSION – Returns the device firmware versions
- SL_DEVICE_STATUS – The device status options are:
  - SL_EVENT_CLASS_DEVICE – Possible values are:
    - EVENT_DROPPED_DEVICE_ASYNC_GENERAL_ERROR – General system error, please check your system configuration.
    - STATUS_DEVICE_SMART_CONFIG_ACTIVE – Device in SmartConfig mode.
  - SL_EVENT_CLASS_WLAN – Possible values are:
    - EVENT_DROPPED_WLAN_WLANASYNCONNECTEDRESPONSE
    - EVENT_DROPPED_WLAN_WLANASYNCDISCONNECTEDRESPONSE
    - EVENT_DROPPED_WLAN_STA_CONNECTED
    - EVENT_DROPPED_WLAN_STA_DISCONNECTED
    - STATUS_WLAN_STA_CONNECTED
- SL_EVENT_CLASS_BSD – Possible values are:
  - EVENT_DROPPED_SOCKET_TXFAILEDASYNCRESPONSE
- SL_EVENT_CLASS_NETAPP – Possible values are:
  - EVENT_DROPPED_NETAPP_IPACQUIRED
  - EVENT_DROPPED_NETAPP_IP_LEASED
  - EVENT_DROPPED_NETAPP_IP_RELEASED
- SL_EVENT_CLASS_NETCFG
- SL_EVENT_CLASS_NVMEM

Example for getting version:

```
SlVersionFull ver;
pConfigOpt = SL_DEVICE_GENERAL_VERSION;
sl_DevGet(SL_DEVICE_GENERAL_CONFIGURATION, &amp;pConfigOpt,&amp;pConfigLen,(unsigned char
*)(&amp;ver));
printf("CHIP %d\nMAC 31.%d.%d.%d.%d\nPHY %d.%d.%d.%d\nNWP %d.%d.%d.%d\nROM %d\nHOST
%d.%d.%d.%d\n",ver.ChipFwAndPhyVersion.ChipId,
ver.ChipFwAndPhyVersion.FwVersion[0],ver.ChipFwAndPhyVersion.FwVersion[1],
ver.ChipFwAndPhyVersion.FwVersion[2],ver.ChipFwAndPhyVersion.FwVersion[3],
ver.ChipFwAndPhyVersion.PhyVersion[0],ver.ChipFwAndPhyVersion.PhyVersion[1],
ver.ChipFwAndPhyVersion.PhyVersion[2],ver.ChipFwAndPhyVersion.PhyVersion[3],
ver.NwpVersion[0],ver.NwpVersion[1],ver.NwpVersion[2],ver.NwpVersion[3],
ver.RomVersion, SL_MAJOR_VERSION_NUM, SL_MINOR_VERSION_NUM, SL_VERSION_NUM, SL_SUB_VERSION_NUM);
```

sl_EventMaskSet – Masks asynchronous events from the device. Masked events do not generate asynchronous messages from the device. This function receives an EventClass and a bit mask. The events and mask options are:

- SL_EVENT_CLASS_WLAN user events:
  - SL_WLAN_CONNECT_EVENT
  - SL_WLAN_DISCONNECT_EVENT

  - SL_WLAN_STA_CONNECTED_EVENT
  - SL_WLAN_STA_DISCONNECTED_EVENT
- SmartConfig events:
  - SL_WLAN_SMART_CONFIG_START_EVENT
  - SL_WLAN_SMART_CONFIG_STOP_EVENT
- SL_EVENT_CLASS_DEVICE user events:
  - SL_DEVICE_FATAL_ERROR_EVENT
- SL_EVENT_CLASS_BSD user events:
  - SL_SOCKET_TX_FAILED_EVENT
  - SL_SOCKET_SSL_ACCEPT_EVENT
- SL_EVENT_CLASS_NETAPP user events:
  - SL_NETAPP_IPACQUIRED_EVENT
  - SL_NETAPP_IPACQUIRED_V6_EVENT

An example of masking out connection and disconnection from WLAN class:

```
sl_EventMaskSet(SL_EVENT_CLASS_WLAN, (SL_WLAN_CONNECT_EVENT | SL_WLAN_DISCONNECT_EVENT) );
```

sl_EventMaskGet – Returns the events bit mask from the device. If that event is masked, the device does not send this event. The function is similar to **sl_EventMaskSet**.

An example of getting an event mask for WLAN class:

```
unsigned long maskWlan;
 sl_StatusGet(SL_EVENT_CLASS_WLAN,&amp;maskWlan);
```

sl_Task – This function must be called from the main loop or from dedicated thread in the following cases:

- **Non-Os Platform** – Should be called from the main loop
- **Multi Threaded Platform** – When the user does not implement the external spawn functions, the function should be called from a dedicated thread allocated to the SimpleLink driver. In this mode the function never returns.

sl_UartSetMode – This function should be used if the user's chosen host interface is UART. The function is responsible for setting the user's UART configuration:

- Baud rate
- Flow control
- COM port

18.2 WLAN

sl_WlanSetMode – The WLAN device has several WLAN modes of operation. By default the device acts as a WLAN station, but it can also act in other WLAN roles. The different options are:

- ROLE_STA – For WLAN station mode
- ROLE_AP – For WLAN AP mode
- ROLE_P2P – For WLAN P2P mode

Note: The set mode functionality only takes effect in the next device boot.

An example of switching from any role to WLAN Access point roles:

```
sl_WlanSetMode(ROLE_AP);

/* Turning the device off and on in order for the roles change to take effect */
sl_Stop(0);
sl_Start(NULL,NULL,NULL);
```

sl_WlanSet – Enables the user to configure different WLAN related parameters. The main parameters used are ConfigID and ConfigOpt. The possible ConfigID and ConfigOpt combinations are:

- SL_WLAN_CFG_GENERAL_PARAM_ID – The different general WLAN parameters are:

  - WLAN_GENERAL_PARAM_OPT_COUNTRY_CODE
  - WLAN_GENERAL_PARAM_OPT_STA_TX_POWER – Sets STA mode Tx power level, a number from 0 to 15, as the dB offset from max power (0 will set maximum power).
  - WLAN_GENERAL_PARAM_OPT_AP_TX_POWER – Sets AP mode Tx power level, a number from 0 to 15, as the dB offset from max power (0 will set maximum power).
- SL_WLAN_CFG_AP_ID – The different AP configuration options are:
  - WLAN_AP_OPT_SSID
  - WLAN_AP_OPT_COUNTRY_CODE
  - WLAN_AP_OPT_BEACON_INT – Sets the beacon interval
  - WLAN_AP_OPT_CHANNEL
  - WLAN_AP_OPT_HIDDEN_SSID – Sets the AP to be hidden or not hidden
  - WLAN_AP_OPT_DTIM_PERIOD
  - WLAN_AP_OPT_SECURITY_TYPE – Possible options are:
    - Open security: SL_SEC_TYPE_OPEN
    - WEP security: SL_SEC_TYPE_WEP
    - WPA security: SL_SEC_TYPE_WPA
  - WLAN_AP_OPT_PASSWORD – Sets the security password for AP mode:
    - For WPA: 8 to 63 characters
    - For WEP: 5 to 13 characters (ASCII)
  - WLAN_AP_OPT_WPS_STATE
- SL_WLAN_CFG_P2P_PARAM_ID
  - WLAN_P2P_OPT_DEV_NAME
  - WLAN_P2P_OPT_DEV_TYPE
  - WLAN_P2P_OPT_CHANNEL_N_REGS – The listen channel and regulatory class determine the device listen channel during the P2P find and listen phase. The operational channel and regulatory class determines the operating channel preferred by the device (if the device is the group owner, this is the operating channel). Channels should be one of the social channels (1/6/11). If no listen or operational channel is selected, a random 1/6/11 will be selected.
  - WLAN_GENERAL_PARAM_OPT_INFO_ELEMENT – The application sets up to MAX_PRIVATE_INFO_ELEMENTS_SUPPORTED info elements per role (AP / P2P GO). To delete an info element, use the relevant index and length = 0. The application sets up to MAX_PRIVATE_INFO_ELEMENTS_SUPPORTED to the same role. However, for AP no more than INFO_ELEMENT_MAX_TOTAL_LENGTH_AP bytes are stored for all info elements. For P2P GO no more than INFO_ELEMENT_MAX_TOTAL_LENGTH_P2P_GO bytes are stored for all info elements.
  - WLAN_GENERAL_PARAM_OPT_SCAN_PARAMS – Changes the scan channels and RSSI threshold

An example of setting SSID for AP mode:

```
unsigned char  str[33];

memset(str, 0, 33);
memcpy(str, ssid, len);  // ssid string of 32 characters
sl_WlanSet(SL_WLAN_CFG_AP_ID, WLAN_AP_OPT_SSID, strlen(ssid), str);
```

sl_WlanGet – Enables the user to configure different WLAN-related parameters. The main parameters used are ConfigID and ConfigOpt. The usage of **sl_WlanGet** is similar to **sl_WlanSet**.

sl_WlanConnect – Manually connects to a WLAN network

sl_WlanDisconnect – Disconnects WLAN connection

sl_WlanProfileAdd – When auto start connection policy is enabled, the device connects to an AP from the profiles table. Up to seven profiles are supported. If several profiles are configured, the device selects the highest priority profile. Within each priority group, the device choses the profile based on the following parameters in descending priority: security policy, signal strength.

sl_WlanProfileGet – Reads a WLAN profile from the device

sl_WlanProfileDel – Deletes an existing profile

sl_WlanPolicySet – Manages the configuration of the following WLAN functionalities:

- **SL_POLICY_CONNECTION** – SL_POLICY_CONNECTION type defines three options available to connect the CC31xx device to the AP:
  - **Auto Connect** – The CC31xx device tries to automatically reconnect to one of its stored profiles each time the connection fails or the device is rebooted. To set this option, use:

    ```
    sl_WlanPolicySet(SL_POLICY_CONNECTION,SL_CONNECTION_POLICY(1,0,0,0,0),NULL,0)
    ```
  - **Fast Connect** – The CC31xx device tries to establish a fast connection to AP. To set this option, use:

    ```
    sl_WlanPolicySet(SL_POLICY_CONNECTION,SL_CONNECTION_POLICY(0,1,0,0,0),NULL,0)
    ```
  - **P2P Connect** – If Any P2P mode is set, CC31xx device tries to automatically connect to the first P2P device available, supporting push button only. To set this option, use:

    ```
    sl_WlanPolicySet(SL_POLICY_CONNECTION,SL_CONNECTION_POLICY(0,0,0,1,0),NULL,0)
    ```
  - **Auto smart config upon restart** – The device wakes up in SmartConfig mode. **Note**: Any command from the host ends this state. To set this option use:

    ```
    sl_WlanPolicySet(SL_POLICY_CONNECTION,SL_CONNECTION_POLICY(0,0,0,0,1),NULL,0)
    ```
- **SL_POLICY_SCAN** – Defines the system scan time interval if there is no connection. The default interval is 10 minutes. After the settings scan interval, an immediate scan is activated. The next scan is based on the interval settings. For setting the scan interval to one minute interval use the following example:

  ```
  unsigned long intervalInSeconds = 60;
  #define SL_SCAN_ENABLE  1
  sl_WlanPolicySet(SL_POLICY_SCAN,SL_SCAN_ENABLE, (unsigned char *)
  &amp;intervalInSeconds,sizeof(intervalInSeconds));
  ```

  To disable the scan, use:

  ```
  #define SL_SCAN_DISABLE  0

  sl_WlanPolicySet(SL_POLICY_SCAN,SL_SCAN_DISABLE,0,0);
  ```
- **SL_POLICY_PM** – Defines a power management policy for station mode only. There are four power policies available:
  - **SL_NORMAL_POLICY** (default) – For setting normal power management policy use:

    ```
    sl_WlanPolicySet(SL_POLICY_PM , SL_NORMAL_POLICY, NULL,0)
    ```
  - **SL_LOW_LATENCY_POLICY** – For setting low latency power management policy use:

    ```
    sl_WlanPolicySet(SL_POLICY_PM , SL_LOW_LATENCY_POLICY, NULL,0)
    ```
  - **SL_LOW_POWER_POLICY** – For setting low power management policy use:

    ```
    sl_WlanPolicySet(SL_POLICY_PM , SL_LOW_POWER_POLICY, NULL,0)
    ```
  - **SL_ALWAYS_ON_POLICY** – For setting always on power management policy use:

    ```
    sl_WlanPolicySet(SL_POLICY_PM , SL_ALWAYS_ON_POLICY, NULL,0)
    ```
  - **SL_LONG_SLEEP_INTERVAL_POLICY** – For setting long sleep interval policy use:

    ```
    unsigned short PolicyBuff[4] = {0,0,800,0}; // 800 is max sleep time in mSec
    sl_WlanPolicySet(SL_POLICY_PM , SL_LONG_SLEEP_INTERVAL_POLICY,
    PolicyBuff,sizeof(PolicyBuff));
    ```
- **SL_POLICY_P2P** – Defines P2P negotiation policy parameters for a P2P role. To set the intent

negotiation value, set one of the following:

- – **SL_P2P_ROLE_NEGOTIATE** – intent 3
- – **SL_P2P_ROLE_GROUP_OWNER** – intent 15
- – **SL_P2P_ROLE_CLIENT** – intent 0

To set the negotiation initiator value (the initiator policy of the first negotiation action frame), set one of the following:

- – **SL_P2P_NEG_INITIATOR_ACTIVE**
- – **SL_P2P_NEG_INITIATOR_PASSIVE**
- – **SL_P2P_NEG_INITIATOR_RAND_BACKOFF**

For example:

```
set sl_WlanPolicySet(SL_POLICY_P2P,
SL_P2P_POLICY(SL_P2P_ROLE_NEGOTIATE,SL_P2P_NEG_INITIATOR_RAND_BACKOFF),NULL,0);
```

sl_WlanPolicyGet – Reads the different WLAN policy settings. The possible options are:

- SL_POLICY_CONNECTION
- SL_POLICY_SCAN
- SL_POLICY_PM

sl_WlanGetNetworkList – Gets the latest WLAN scan results

sl_WlanSmartConfigStart – Puts the device into SmartConfig state. Once SmartConfig has ended successfully, an asynchronous event will be received: SL_OPCODE_WLAN_SMART_CONFIG_START_ASYNC_RESPONSE. The event holds the SSID and an extra field that might also have been delivered (for example, device name).

sl_WlanSmartConfigStop – Stops the SmartConfig procedure. Once SmartConfig is stopped, an asynchronous event is received: SL_OPCODE_WLAN_SMART_CONFIG_STOP_ASYNC_RESPONSE

sl_WlanRxStatStart – Starts collecting WLAN Rx statistics (unlimited time)

sl_WlanRxStatStop – Stops collecting WLAN Rx statistics

sl_WlanRxStatGet – Gets WLAN Rx statistics. Upon calling this command, the statistics counters are cleared. The statistics returned are:

- Received valid packets – Sum of the packets received correctly (including filtered packets)
- Received FCS Error packets – Sum of the packets dropped due to FCS error
- Received PLCP error packets – Sum of the packets dropped due to PLCP error
- Average data RSSI – Average RSSI for all valid data packets received
- Average management RSSI – Average RSSI for all valid management packets received
- Rate histogram – Rate histogram for all valid packets received
- RSSI histogram – RSSI histogram from -40 until -87 (all values below and above RSSI appear in the first and last cells)
- Start time stamp – The timestamp of starting to collect the statistics in uSec
- Get time stamp – The timestamp of reading the statistics in uSec

18.3 Socket

sl_Socket – Creates a new socket of a socket type identified by an integer number, and allocates system resources to the socket. The supported socket types are:

- SOCK_STREAM (TCP – Reliable stream-oriented service or Stream Sockets)
- SOCK_DGRAM (UDP – Datagram service or Datagram Sockets)
- SOCK_RAW (Raw protocols atop the network layer)

sl_Close – Gracefully closes socket. This function causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.

sl_Accept – This function is used with connection-based socket types (SOCK_STREAM) to extract the first connection request on the queue of pending connections, creates a new connected socket, and returns a new file descriptor referring to that socket. The newly created socket is not in the listening state. The original socket sd is unaffected by this call.

sl_Bind – Gives the socket the local address *addr*. *addr* is *addrlen* bytes long. Traditionally, this function is called when a socket is created, exists in a name space (address family) but has no name assigned. A local address must be assigned before a SOCK_STREAM socket receives connections.

sl_Listen – Specifies the willingness to accept incoming connections and a queue limit for incoming connections. The **listen()** call applies only to sockets of type SOCK_STREAM, and the backlog parameter defines the maximum length for the queue of pending connections.

sl_Connect – Connects the socket referred to by the socket descriptor *sd* to the address specified by *addr*. The *addrlen* argument specifies the size of *addr*. The format of the address in *addr* is determined by the address space of the socket. If the socket is of type SOCK_DGRAM, this call specifies the peer with which the socket is associated. Datagrams should be sent to this address, the only address from which datagrams should be received. If the socket is of type SOCK_STREAM, this call tries to make a connection to another socket. The other socket is specified by an address in the communications space of the socket.

sl_Select – Allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become ready for a class of I/O operation. **sl_Select** has several sub functions to set the file descriptor options:

- **SL_FD_SET** – Selects **SlFdSet_t** SET function. Sets the current socket descriptor on the **SlFdSet_t** container
- **SL_FD_CLR** – Selects **SlFdSet_t** CLR function. Clears the current socket descriptor on the **SlFdSet_t** container
- **SL_FD_ISSET** – Selects **SlFdSet_t** ISSET function. Checks if the current socket descriptor is set (TRUE/FALSE)
- **SL_FD_ZERO** – Selects **SlFdSet_t** ZERO function. Clears all socket descriptors from **SlFdSet_t**

sl_SetSockOpt – Manipulates the options associated with a socket. Options exist at multiple protocol levels and are always present at the uppermost socket level. The supported socket options are:

- **SL_SO_KEEPALIVE** – Keeps TCP connections active by enabling the periodic transmission of messages; Enable/Disable, periodic keep alive. *Default: Enabled, keep alive timeout 300 seconds.*
- **SL_SO_RCVTIMEO** – Sets the timeout value that specifies the maximum amount of time an input function waits until it completes. *Default: No timeout*
- **SL_SO_RCVBUF** – Sets TCP max receive window
- **SL_SO_NONBLOCKING** – Sets the socket to nonblocking operation. *Impact on: connect, accept, send, sendto, recv and recvfrom. Default: Blocking.*
- **SL_SO_SECMETHOD** – Sets the method to the TCP-secured socket (SL_SEC_SOCKET). *Default: SL_SO_SEC_METHOD_SSLv3_TLSV1_2* .
- **SL_SO_SECURE_MASK** – Sets a specific cipher to the TCP-secured socket (SL_SEC_SOCKET). *Default: "Best" cipher suitable to method*
- **SL_SO_SECURE_FILES** – Maps programmed files to the secured socket (SL_SEC_SOCKET)
- **SL_SO_CHANGE_CHANNEL** – Sets the channel in transceiver mode
- **SL_IP_MULTICAST_TTL** – Sets the time-to-live value of the outgoing multicast packets for the socket
- **SL_IP_RAW_RX_NO_HEADER** – Raw socket; removes the IP header from received data. *Default: data includes IP header.*
- **SL_IP_HDRINCL** – RAW socket only; the IPv4 layer generates an IP header when sending a packet unless the IP_HDRINCL socket option is enabled on the socket. When it is enabled, the packet must contain an IP header. *Default: disabled, IPv4 header generated by Network Stack*.
- **SL_IP_ADD_MEMBERSHIP** – UDP socket; joins a multicast group.
- **SL_IP_DROP_MEMBERSHIP** – UDP socket; leaves a multicast group.
- **SL_SO_PHY_RATE** – RAW socket; sets WLAN PHY transmit rate.

- **SL_SO_PHY_TX_POWER** – RAW socket; sets WLAN PHY Tx power.
- **SL_SO_PHY_NUM_FRAMES_TO_TX** – RAW socket; sets the number of frames to transmit in transceiver mode.
- **SL_SO_PHY_PREAMBLE** – RAW socket; sets WLAN PHY preamble.

sl_GetSockOpt – Manipulates the options associated with a socket. Options may exist at multiple protocol levels and are always present at the uppermost socket level. The socket options are the same as in **sl_SetSockOpt**.

sl_Recv – Reads data from the TCP socket

sl_RecvFrom – Reads data from the UDP socket

sl_Send – Writes data to the TCP socket. Returns immediately after sending data to device. In case of TCP failure, an async event SL_NETAPP_SOCKET_TX_FAILED will be received. In case of a RAW socket (transceiver mode), an extra four bytes should be reserved at the end of the frame data buffer for WLAN FCS.

sl_SendTo – Writes data to the UDP socket. This function transmits a message to another socket (connectionless socket SOCK_DGRAM, SOCK_RAW). Returns immediately after sending data to the device. In case of transmission failure, an async event SL_NETAPP_SOCKET_TX_FAILED is received.

sl_Htonl – Reorders the bytes of a 32-bit unsigned value from processor order to network order.

sl_Htons – Reorders the bytes of a 16-bit unsigned value from processor order to network order.

18.4 NetApp

sl_NetAppStart – Enables or starts different networking services. Could be one or a combination of the following:

- SL_NET_APP_HTTP_SERVER_ID – HTTP server service
- SL_NET_APP_DHCP_SERVER_ID – DHCP server service (DHCP client is always supported)
- SL_NET_APP_MDNS_ID – MDNS Client\Server service

sl_NetAppStop – Disables or stops a networking service. Similar options as in **sl_NetAppStart**.

sl_NetAppSet

sl_NetAppGet

sl_NetAppDnsGetHostByName – Obtains the IP address of a machine on the network, by machine name. Example:

```
unsigned long DestinationIP;
 sl_NetAppDnsGetHostByName("www.ti.com", strlen("www.ti.com"), &amp;DestinationIP,SL_AF_INET);

Addr.sin_family = SL_AF_INET;
Addr.sin_port = sl_Htons(80);
Addr.sin_addr.s_addr = sl_Htonl(DestinationIP);
AddrSize = sizeof(SlSockAddrIn_t);
SockID = sl_Socket(SL_AF_INET,SL_SOCK_STREAM, 0);
```

sl_NetAppDnsGetHostByService – Returns service attributes such as IP address, port, and text according to the service name. The user sets a service name full/part (see the following example), and should get:

- The IP of service
- The port of service
- The text of service

This is similar to the get host by name method, and is done with a single shot query with PTR type on the service name. An example for full service name:

- PC1._ipp._tcp.local
- PC2_server._ftp._tcp.local

An example for partial service name:

- _ipp._tcp.local
- _ftp._tcp.local

sl_NetAppGetServiceList – Gets the list of peer services. The list is in a form of service structure. The user chooses the type of the service structure. The supported structures are:

- Full service parameters with text
- Full service parameters
- Short service parameters (port and IP only), especially for tiny hosts

Note: The different types of structures exist to save memory in the host.

sl_NetAppMDNSRegisterService – Registers a new mDNS service to the mDNS package and the DB. This registered service is offered by the application. The service name should be a full service name according to DNS-SD RFC, meaning the value in the name field in the SRV answer.

Example for service name:

- PC1._ipp._tcp.local
- PC2_server._ftp._tcp.local

If the option is_unique is set, mDNS probes the service name to ensure it is unique before announcing the service on the network.

sl_NetAppMDNSUnRegisterService – Deletes the mDNS service from the mDNS package and the database.

sl_NetAppPingStart – Sends an ICMP ECHO_REQUEST (or ping) to the network hosts. An example of sending 20 ping requests and reporting the results to a callback routine when all requests are sent:

```
// callback routine
void pingRes(SlPingReport_t* pReport)
{
             // handle ping results
}

// ping activation
void PingTest()
{
                 SlPingReport_t report;
                 SlPingStartCommand_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;       // max number of ping requests. 0 -
 forever
                 pingCommand.Flags = 0;                        // report only when finished

                 sl_NetAppPingStart( &amp;pingCommand, SL_AF_INET, &amp;report, pingRes ) ;
}
```

18.5 NetCfg

sl_NetCfgSet – Manages the configuration of the following networking functionalities:

- **SL_MAC_ADDRESS_SET** – The new MAC address overrides the default MAC address and is saved in the SFlash file system.
- **SL_IPV4_STA_P2P_CL_DHCP_ENABLE** – Sets the device to acquire an IP address by DHCP when in WLAN sta mode or P2P client. This is the default mode of the system for acquiring an IP address after a WLAN connection.
- **SL_IPV4_STA_P2P_CL_STATIC_ENABLE** – Sets a static IP address to the device working in STA

mode or P2P client. The IP address is stored in the SFlash file system. To disable the static IP and get the address assigned from DHCP, use **SL_STA_P2P_CL_IPV4_DHCP_SET**.

- **SL_IPV4_AP_P2P_GO_STATIC_ENABLE** – Sets a static IP address to the device working in AP mode or P2P go. The IP address is stored in the SFlash file system.
  Example:

```
_NetCfgIpV4Args_t ipV4;
ipV4.ipV4          = (unsigned long)SL_IPV4_VAL(10,1,1,201);            // unsigned long IP
address
ipV4.ipV4Mask      = (unsigned long)SL_IPV4_VAL(255,255,255,0);         // unsigned long
Subnet mask for this AP/P2P
ipV4.ipV4Gateway   = (unsigned long)SL_IPV4_VAL(10,1,1,1);              // unsigned long
Default gateway address
ipV4.ipV4DnsServer = (unsigned long)SL_IPV4_VAL(8,16,32,64);            // unsigned long DNS
server address
        sl_NetCfgSet(SL_IPV4_AP_P2P_GO_STATIC_ENABLE,1,sizeof(_NetCfgIpV4Args_t),(unsigned
char *)
&amp;ipV4);
sl_Stop(0);
sl_Start(NULL,NULL,NULL);
```

  Note: AP mode must use static IP settings.

  Note: All set functions require system restart in order for changes to take effect.

sl_NetCfgGet – Reads the network configurations. The options are:

- SL_MAC_ADDRESS_GET
- SL_IPV4_STA_P2P_CL_GET_INFO – Gets an IP address from the WLAN station or P2P client. A DHCP flag is returned to indicate if the IP address is static or from DHCP.
- o SL_IPV4_AP_P2P_GO_GET_INFO – Returns the IP address of the AP.

An example of getting an IP address from a WLAN station or P2P client:

```
unsigned char len = sizeof(_NetCfgIpV4Args_t);
unsigned char dhcpIsOn = 0;
_NetCfgIpV4Args_t ipV4 = {0};

sl_NetCfgGet(SL_IPV4_STA_P2P_CL_GET_INFO,&amp;dhcpIsOn,&amp;len,(unsigned char *)&amp;ipV4);

printf("DHCP is %s IP %d.%d.%d.%d MASK %d.%d.%d.%d GW %d.%d.%d.%d DNS %d.%d.%d.%d\n",
(dhcpIsOn
 > 0) ? "ON":"OFF",
SL_IPV4_BYTE(ipV4.ipV4,3),
SL_IPV4_BYTE(ipV4.ipV4,2),
SL_IPV4_BYTE(ipV4.ipV4,1),
SL_IPV4_BYTE(ipV4.ipV4,0),
SL_IPV4_BYTE(ipV4.ipV4Mask,3),
SL_IPV4_BYTE(ipV4.ipV4Mask,2),
SL_IPV4_BYTE(ipV4.ipV4Mask,1),
SL_IPV4_BYTE(ipV4.ipV4Mask,0),
SL_IPV4_BYTE(ipV4.ipV4Gateway,3),
SL_IPV4_BYTE(ipV4.ipV4Gateway,2),
SL_IPV4_BYTE(ipV4.ipV4Gateway,1),
SL_IPV4_BYTE(ipV4.ipV4Gateway,0),
SL_IPV4_BYTE(ipV4.ipV4DnsServer,3),
SL_IPV4_BYTE(ipV4.ipV4DnsServer,2),
SL_IPV4_BYTE(ipV4.ipV4DnsServer,1),
SL_IPV4_BYTE(ipV4.ipV4DnsServer,0));
```

18.6 File System

sl_FsOpen – Opens a file for read or write from or to Sflash storage AccessModeAndMaxSize. Possible inputs are:

- **FS_MODE_OPEN_READ** – Reads a file
- **FS_MODE_OPEN_WRITE** – Opens an existing file for write

- **FS_MODE_OPEN_CREATE**(maxSizeInBytes,accessModeFlags) – Opens for creating a new file. The maximum file size is defined in bytes. For optimal file system size, use max size in 4K-512 bytes (for example, 3584,7680). Several access modes can be combined together from **SlFileOpenFlags_e**.

Example:

```
sl_FsOpen("FileName.html",FS_MODE_OPEN_CREATE(3584,_FS_FILE_OPEN_FLAG_COMMIT|_FS_FILE_PUBLIC_WRITE
) ,NULL, &amp;FileHandle);
```

sl_FsClose – Closes file in Sflash storage

sl_FsRead – Reads a block of data from a file

sl_FsWrite – Writes a block of data to a file

sl_FsDel – Deletes a specific file from Sflash storage or all files (format)

sl_FsGetInfo – Returns the file information: flags, file size, allocated size, and tokens

Chapter 19
SWRU368–June 2014

Asynchronous Events

Topic Page

The SimpleLink host driver interacts with the SimpleLink device through commands transmitted to the device over the SPI or UART bus interface. Because some of the commands might trigger long processes which can take several hundred milliseconds or even seconds, the device and the host driver support a mechanism of asynchronous events sent from the device to the host driver.

Events notify on process completion such as a SmartConfig process done, notify on device status changes such as a WLAN disconnection event, or notify on errors such as a fatal error event.

These events can be classified in the following logical sections:

- WLAN events
- Network application events
- Socket events
- General device events

19.1 WLAN

SL_WLAN_CONNECT_EVENT – Notifies that the device is connected to the AP. Event parameters:

- connection_type
- ssid_len
- ssid_name
- go_peer_device_name_len – Relevant for P2P
- go_peer_device_name
- bssid

SL_WLAN_DISCONNECT_EVENT – Notifies that the device is disconnected from the AP. Event parameters:

- connection_type
- ssid_len
- ssid_name
- go_peer_device_name_len – Relevant for P2P
- go_peer_device_name
- bssid
- reason_code – WLAN disconnection reason

SL_WLAN_SMART_CONFIG_START_EVENT – Notifies the host that SmartConfig has ended. Event parameters:

- Status
- ssid_len
- ssid
- private_token_len
- private_token

SL_WLAN_SMART_CONFIG_STOP_EVENT – Notifies the host that SmartConfig has stopped. Event parameters:

- status

SL_WLAN_STA_CONNECTED_EVENT – Notifies that STA is connected; relevant in AP mode or P2P GO. Event parameters:

- status
- go_peer_device_name
- mac
- go_peer_device_name_len
- wps_dev_password_id

- own_ssid
- own_ssid_len

SL_WLAN_STA_DISCONNECTED_EVENT – Notifies that STA is disconnected; relevant in AP mode or P2P GO. Event parameters:

- Status
- go_peer_device_name
- mac
- go_peer_device_name_len
- wps_dev_password_id
- own_ssid
- own_ssid_len

SL_WLAN_P2P_DEV_FOUND_EVENT – Notifies that the device is found; relevant in P2P mode. Event parameters:

- go_peer_device_name
- mac
- go_peer_device_name_len
- wps_dev_password_id
- own_ssid
- own_ssid_len

SL_WLAN_P2P_NEG_REQ_RECEIVED_EVENT – Notifies that the negotiation request received an event; relevant in P2P mode. Event parameters:

- go_peer_device_name
- mac
- go_peer_device_name_len
- wps_dev_password_id
- own_ssid
- own_ssid_len

SL_WLAN_CONNECTION_FAILED_EVENT – Notifies negotiation failure; relevant in P2P mode. Event parameters:

- status

19.2 Netapp

SL_NETAPP_IPV4_IPACQUIRED_EVENT – Notifies IPv4 enquired. Event parameters:

- ip
- gateway
- dns

SL_NETAPP_IP_LEASED_EVENT – Notifies STA IP lease; relevant in AP or P2P GO mode. Event parameters:

- ip_address
- lease_time
- mac

SL_NETAPP_IP_RELEASED_EVENT – Notifies STA IP release; relevant in AP or P2P GO mode. Event parameters:

- ip_address
- mac
- reason

SL_NETAPP_HTTPGETTOKENVALUE_EVENT – Notifies token is missing. Tries to retrieve this value from host. Event parameters:

- httpTokenName
- httpTokenName length

SL_NETAPP_HTTPPOSTTOKENVALUE_EVENT – Notifies a new post with the included parameters. Event parameters:

- action
- action length
- token_name
- token_name length
- token_value
- token_value length

19.3 Socket

SL_SOCKET_TX_FAILED_EVENT – Notifies of Tx failure. Event parameters:

- Status
- Sd

SL_SOCKET_ASYNC_EVENT – Notifies of asynchronous event. Event parameters:

- sd
- type – The event type can be one of the following:
  - **SL_SOCKET_ASYNC_EVENT_TYPE_SSL_ACCEPT** – Accept failed due to SSL issue (TCP pass)
  - **SL_SOCKET_ASYNC_EVENT_TYPE_RX_FRAGMENTATION_TOO_BIG** – Connectionless mode, Rx packet fragmentation > 16K, packet is released.
  - **SL_SOCKET_ASYNC_EVENT_TYPE_OTHER_SIDE_CLOSE_SSL_DATA_NOT_ENCRYPTED** – Remote side down from secure to unsecure
- value

19.4 Device

SL_DEVICE_FATAL_ERROR_EVENT – Notifies of fatal error; needs to perform device reset. Event parameters:

- Status
- sender

Host Driver Architecture

A.1 Overview

From a software perspective, the system can be separated into four parts:

- User application
- CC3100 Host driver – Platform independent part
  - Host driver API
  - Main driver logic and flow
- CC3100 Host driver – Platform dependent part
  - OS wrapper implementation
  - Transport layer (SPI/UART) implementation
- Figure A-1

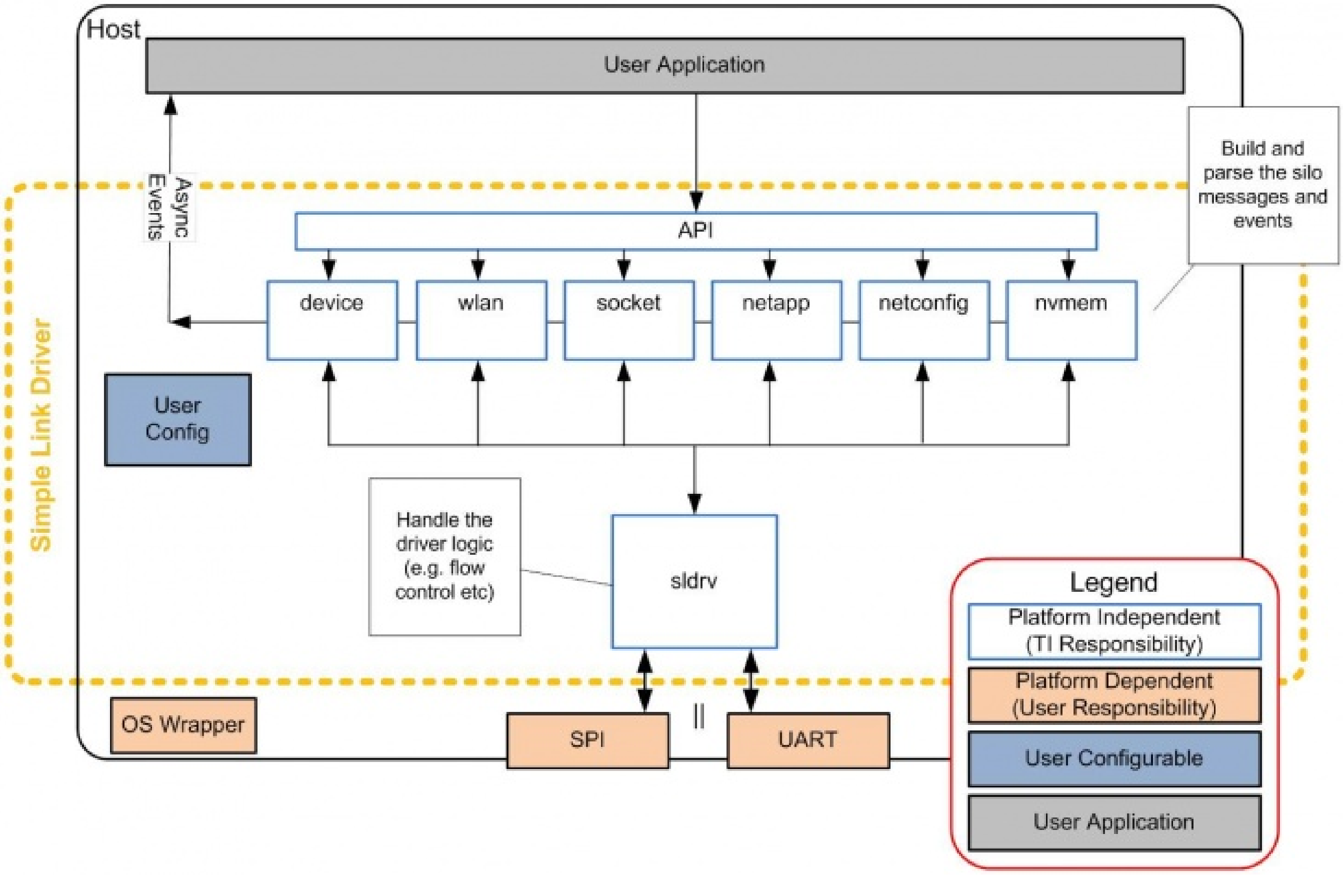

Host
User Application
Async Events
Simple Link Driver
Build and parse the silo messages and events
API
device
wlan
socket
netapp
netconfig
nvmem
User Config
Handle the driver logic (e.g. flow control etc)
sldrv
OS Wrapper
SPI
||
UART
Legend
Platform Independent (TI Responsibility)
Platform Dependent (User Responsibility)
User Configurable
User Application

Figure A-1. CC3100 Driver Configuration

A.1.1 CC3100 Host Driver - Platform Independent Part

The host driver implementation includes the driver API, CC3100 initialization, CC3100 commands, commands response handling, asynchronous event handling, data flow, and transport layer interface. All are platform-independent and OS-independent code provided by TI. The driver APIs are organized into six silos reflecting six different logical API types:

- **Device API** – Handles the HW related API
- **WLAN API** – Handles the WLAN, 802.11 protocol related functionality
- **Socket API** – The most common API set to be used by the user application. The CC3100 socket API complies with the Berkeley socket APIs.
- **NetApp API** – Handles additional networking protocols and functionality, delivered as a complementary part of the on-chip content.
- **NetCfg API** – Handles configuration of different networking parameters
- **NVMem API** – Handles access to the SFlash component, for read and write operations of networking or user proprietary data

A.1.2 CC3100 Host Driver - Platform Dependent Part

The driver has two SW components supplied by the user:

- Transport layer implementation – *Mandatory*
  The driver code provides the required transport and bus interfaces.
  It is up to the user to provide the function implementation per the transport layer chosen (SPI or UART), and the platform used.
- OS Wrapper – *Optional*
  The driver code provides the required OS wrapper interface.
  To use an OS, first provide the function implementation per the OS and platform used.

A.1.3 CC3100 Driver Configuration

The driver provides a configuration file, allowing the user to control the supported API sets, memory allocation module, OS usage and more. In addition, some pre-configuration options are provided. A pre-configuration is a set of customizations and settings of the driver already made and tested by TI for a specific scenario.

A.1.4 User Application

The user application is developed and owned by the user. The application interfaces with the CC3100 driver using the driver APIs and asynchronous driver events.

A.2 Driver Data Flows

A.2.1 Transport Layer Protocol

The types of messages used by the host driver are:

- Command [Host -> CC3100]
- Command Complete [CC3100 -> Host]
- Async Event [CC3100 -> Host]
- Data [Host <-/-> CC3100]

A.2.2 Command and Command Complete

A command is any message from the host to CC3100 which isn't a data packet (send or receive). The host driver supports a single command at a time. Before sending the next command, the driver waits for a command complete message received from the CC3100. To avoid blocking the driver for long periods of time in terms of RT and embedded systems, in some commands the CC3100 sends the command complete immediately after the command is received and validated, and sends an asynchronous event when the execution of the command is completed.

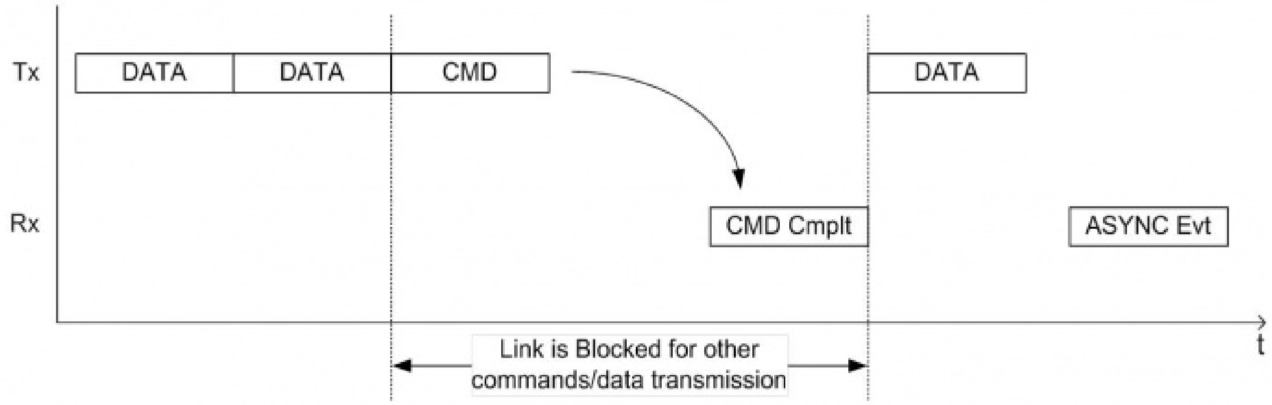


Figure A-2. Blocked Link

A.2.3 Data Transactions

A data message is any information (usually TCP or UDP packets) sent to or from the user using the socket send and receive APIs.

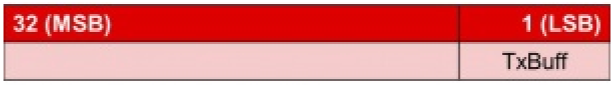
A.2.3.1 Data Send (From Host to CC3100)

Sending data to the CC3100 is similar to a command flow. The difference is that there is no command complete message for a data packet, thus the host is not blocked or waiting for a message. Data packets can be sent sequentially from the user application; the driver is responsible for avoiding buffer overrun by using data flow control logic.

A.2.3.2 Data Flow Control

Status Field – Part of each message (async event) from CC3100 to the host:

- TxBuff – Indicates the number of available Tx buffers within the CC3100 device.



32 (MSB)	1 (LSB)
	TxBuff

Figure A-3. Data Flow Control

- The host can send up to TxBuff packets (<1500B) without waiting for a response from the CC3100 device.
- The CC3100 device generates a dummy event if the number is changed (threshold from last update / timeout if change is smaller than the threshold).

A.2.3.3 Data Receive (From CC3100 to Host)

Receiving data from the CC3100 is done in two manners: blocking and nonblocking.

A.2.3.4 Blocking Receive

Upon a call to a blocking **sl_Recv**, the host driver issues a command to the CC3100. The driver is blocked, waiting until an async event notifies the driver of a data packet ready to be read within the CC3100 device. After receiving this async event, the driver continues to read the data packet from the host.

A.2.3.5 Non-Blocking Receive

If the command complete status returns with a status notifying that no data is waiting for the host, the command returns with pending return status (SL_EAGAIN). If data is waiting for the host, the driver continues to read the data packet from the host.

Appendix B
SWRU368–June 2014

HTTP Server Supported Features and Limitation

B.1 Supported Features

1. HTTP version support: 1.0

2. HTTP requests: GET, POST

3. Supported file types: .html, .htm, .css, .xml, .png, .gif

4. HTML form submission of data uses POST method.

5. Supported content-type of POST request: application/x-www-form-urlencoded

6. HTTP port number can be configured – default is port 80.

7. HTTP web server authentication:
(a) Can be enabled or disabled (disabled by default).
(b) Authentication name, password and realm are configurable.

8. SimpleLink domain name (in AP mode) can be configured.

9. Built-in default page that provides device configuration, status, and analyzing tools (with no configuration necessary from the user side).

10. Option to set the device configuration as part of user-provided pages.

11. For security purposes, the web server can access only the following root folders on the file system:
(a) *www/*
(b) *www/safe/*

12. Force AP support mode – In this mode the HTTP server will behave as follows:
(a) The server will permit access only to the *www/safe/* folder in the file system.
(b) Special Clear all Profiles button in the internal configuration web pages.

13. Default values:

- Domain name: **www.mysimplelink.net** or **mysimplelink.net**
- Authentication Name: **admin**
- Authentication Password: **admin**
- Authentication Realm: **Simple Link CC31xx**

B.2 Limitations

1. HTTPS is currently not supported.

2. The HTTP web server can host a single domain only.

3. HTML form submission using GET method is not supported.

Appendix C

SWRU368–June 2014

SSL Limitations

1. Only two SSL connections are allowed.

How to Generate Certificates, Public Keys and CA's

1. Download and install the latest package of OpenSSL (either Windows or Linux).

2. In the installation path \bin library, find openssl.exe.

Private Key

To create a new private key for a certificate, use:

```
openssl genrsa -out privkey.pem 2048
```

Notes:

- The default key size is 2048, but the user can use any desired protocol key size (1024, 2048, 4096…).
- The name of the file is replaceable.
- The default format is PEM which is in ASCII form. In many systems the binary format, DER, is more popular due to its smaller size. To convert between the formats use: *openssl rsa -in privkey.pem –inform PEM –out privkey.der –outform DER*

Certificate and CA

The CA (Certificate Authority) is a certificate which is self-signed and used for signing other certificates.

To generate a CA, use the following command and insert the desired values:

```
openssl req -new -x509 -days 3650 -key privkey.pem -out root-ca.pem
```

Notes:

- The days argument determines how long the certificate is valid.
- The key is generated in the Private Key section of this document, in PEM format.
- The output is PEM format. To convert from PEM to DER use:

```
openssl x509 -in input.crt -inform PEM –out output.crt
                -outform DER
```

To generate a certificate, prepare the certificate document first. Similar to making a CA, fill the desired values such as country code name and so forth with the command:

```
openssl req -new -key privkey.pem -out cert.pem
```

The private key is different from the one used for the CA. Each certificate should have its own private key.

After generating a certificate form (also called certificate request), sign it with another certificate. The form is usually signed with the CA but to make a chain, sign it with another certificate.

To do the signing process use:

```
openssl x509 -req -days 730 -in cert.pem -CA ca.pem -CAkey CAPrivate.pem             -
set_serial 01 -out cert.pem
```

Notes:

- The example uses the CA to sign on the generated certificate.
- The key in the example is the CA private key.
- The days argument determines how long this certificate is valid.
- -set_serial 01 is needed.

To generate a CA and a certificate signed by the CA do the following:

1. Generate a private key for the CA.
2. Generate a private key for the certificate.

3. Create a CA with its own private key.
4. Create a certificate request with its own private key.
5. Sign the certificate with the CA and the CA private key.
6. To create a chain, create another private key and certificate request and sign it with the first certificate.

A sha1 can be signed with a private key. To make a sha1 code out of data.txt file use:

```
openssl dgst -sha1 data.txt > hash
```

To RSA sign the sha1 code with a private key, use:

```
openssl dgst –binary –out signature.bin -sha1  -sign privatekey.pem BufferToSign.bin
```

Transceiver Mode Limitations

1. The user can open one transceiver socket in the system.

2. The length of a received packet is trimmed if it exceeds 1472 bytes.

3. Cannot transmit frame over 1472 bytes or below 14 bytes.

4. If using the SimpleLink studio, the maximum throughput of the SPI is 5~6Mbps: in a crowded Wi-Fi medium there is packet loss.

5. The transceiver will not open if in connection or connected mode. Auto connection mode is also considered as connected mode, even if not connected.

Rx Statistics Limitations

1. The maximum histogram cell capacity is 65535 for both RSSI and rate. If more packets are received, the accumulation will stop.

2. All other statistics are held in unsigned integers and wraparound when exceeding the maximum.

mDNS Supported Features and Limitations

G.1 Supported Features

1. Advertises:

- Register and advertise services
- External services advertise:
  - Any service a user is willing to add.
  - Services using host API.
  - Services written in the mDNS DB.
- Internal services advertise:
  - Services that are advertised by default without a relation to mDNS DB.
  - The internal capabilities of the product.
  - HTTP server – Peers can find the IP and browse without a DNS server.

2. Responds:

- Response to queries.
- Ignores queries that are already familiar with services.

3. Looks for services in the local network by using:

- One shot query
- Continuous query

4. Gathers the known information on peer services using the Get service list API:

- Full services with text – name, host, IP, port, text.
- Full services without text – name, host, IP, port.
- Short services – IP, port.

5. Masks peer services received in responses or peer advertisements.

6. Sets advertising timing. Reconfigures the timing parameters employed by mDNS when sending the service announcements: how many times and when a service is advertised (not related to responding queries).

G.2 Specific Behavior and Assumptions

1. Advertise up to five external services.

2. Advertise one internal service – HTTP.

3. Support or receive up to eight different services in the peer cache (use filter for storing only the required services).

4. Default names:

(a) Default value of target name – MAC-mysimplelink.local the target name is the name that should be in ‘A’ query (query for getting the IP). The name is assembled from the MAC address of the target, the word mysimplelink, and the domain of the local network. This name can be changed by the user by setting the URN name or by a smart-configure operation.

(b) Default name of internal services: MAC@mysimplelink._type._local. For example, MAC@mysimplelink._http._local.

5. By default the mDNS is started.

6. In cases of registering services, there is a partial check if the name is legal. The user must send the legal service name.

7. Some of the API parameters use in_out. Set these parameters again in case of an API recall.

G.3 Limitations

The following are the known limitations of the mDNS feature.

1. The mDNS machine stops and starts when all services are deleted; the peer cache is deleted.

2. If the user registers a unique service but a service with the same name already exists in the network, then the service name is changed to "name (number)," for example PC1 (2)_ipp._tcp.local. In this case, the name in the DB is the original name but the advertising uses the new name.

3. Deleting a unique name that was changed because of mismatch between the names (the advertising name and the DB name) causes the mDNS machine to stop and start, and deletes the peer cache.

4. If there is a one shot query but the peer cache is full, there will be no place to set the query. The peer cache will be deleted, and then the query sent.

5. There is a partial check if the service name that is registered is legal. The user must send a legal name.

6. When using get host by service, only one answer is returned. To see all the answers, wait for all peer answers to be sent and received, then read the answers by using the API get service list.

7. The max buffer list size of the API get service list is about 1500 bytes. Requests for a list bigger than this size returns an error.

G.4 Errors Numbers and Corrections

The table below depicts error numbers that can be returned, their meaning, and possible corrections. The error number should be smaller than zero.

Table G-1. Error Numbers

Error number	Meaning	Correction
-200	Maximum advertise services are already configured, registering for another external service is not allowed.	Delete another existing registered service, then register the new service again.
-201	Trying to register a service that already exists.	Don't register this service.
-203	Trying to delete a service that does not exist.	Don't unregister this service.
-204/-179	Illegal service name (the name is not allowed according to the RFC).	Change the service name, and register the service again.
-205	Buffer resource problem in the NWP (list cannot be returned).	Wait, and call the API again.
-206	List size buffer is bigger than internally allowed in the NWP (API get service list).	Change the parameters of the API to decrease the size of the list.
-207	Illegal length of one of the mDNS Set functions.	Change the length value and respectively the set parameter.
-208	Illegal value of flags parameters in API get service list. Flags – indicates the Type of the services that are returned.	Use value from ENUM – SlNetAppGetServiceListType_e
-230	Returned list buffer is bigger than the user allocated buffer (API get service list).	The user increases the size of the allocated buffer or requests for a smaller list.
-161	mDNS is not operational as the device has no IP.	Connect the device to an AP to get an IP address.
-162	mDNS parameters errors	Check the API parameters

Table G-1. Error Numbers (continued)

Error number	Meaning	Correction
-163/-182	mDNS cache error	Stop the mDNS and restart it.
-164 to -176	Internal mDNS error	
-177	No service is found (API –get host by service)	Request again until service is found (if exists).
-178	Adding a service is not allowed as it already exists (duplicate service).	Delete all services by using API unregister service with NULL, then add services again.
-180	mDNS is not started	Start the mDNS
-181	Host name error. Host name format is not allowed according to RFC 1033,1034,1035, 6763.	Change the host name and start again the mDNS.

Socket Limitations

There are eight regular (un-secured) sockets and two secured* sockets available on CC3100. Table H-1 shows the number of available sockets, divided by client or server.

** Secure sockets indicate a SSL/TLS connection.*

Table H-1. Available Sockets

Role	Number of Normal Sockets	Number of Secure Sockets
Client	8	2
Server	8 - number of listening sockets	2

The client side can use eight normal sockets and two secure sockets, or all ten simultaneously.

The number of available sockets for communication depends on the number of listening sockets. If one socket is reserved for public socket and used to listen to incoming client requests, this leaves seven private sockets for actual client communication. If there are two server sockets for listening, only six private sockets will remain for communication.

The number of available server sockets for UDP connection remains eight as UDP is a connectionless socket. Since it does not require a socket to be in listening mode, all eight can be used for client communication.

Server side secure socket (SSL/TLS) connection number isn't affected because a regular server socket can be used for listening. After accepting a new client connection, the user can switch to a secure socket.

H.1 Important Notice

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability. TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements. CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE–SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK. In order to minimize risks associated with the customer's applications, the customer to minimize inherent or procedural hazards must provide adequate design and operating safeguards. TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers’ products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers’ products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information 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.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265