smxWiFi^{TM} User's Guide

802.11 MAC

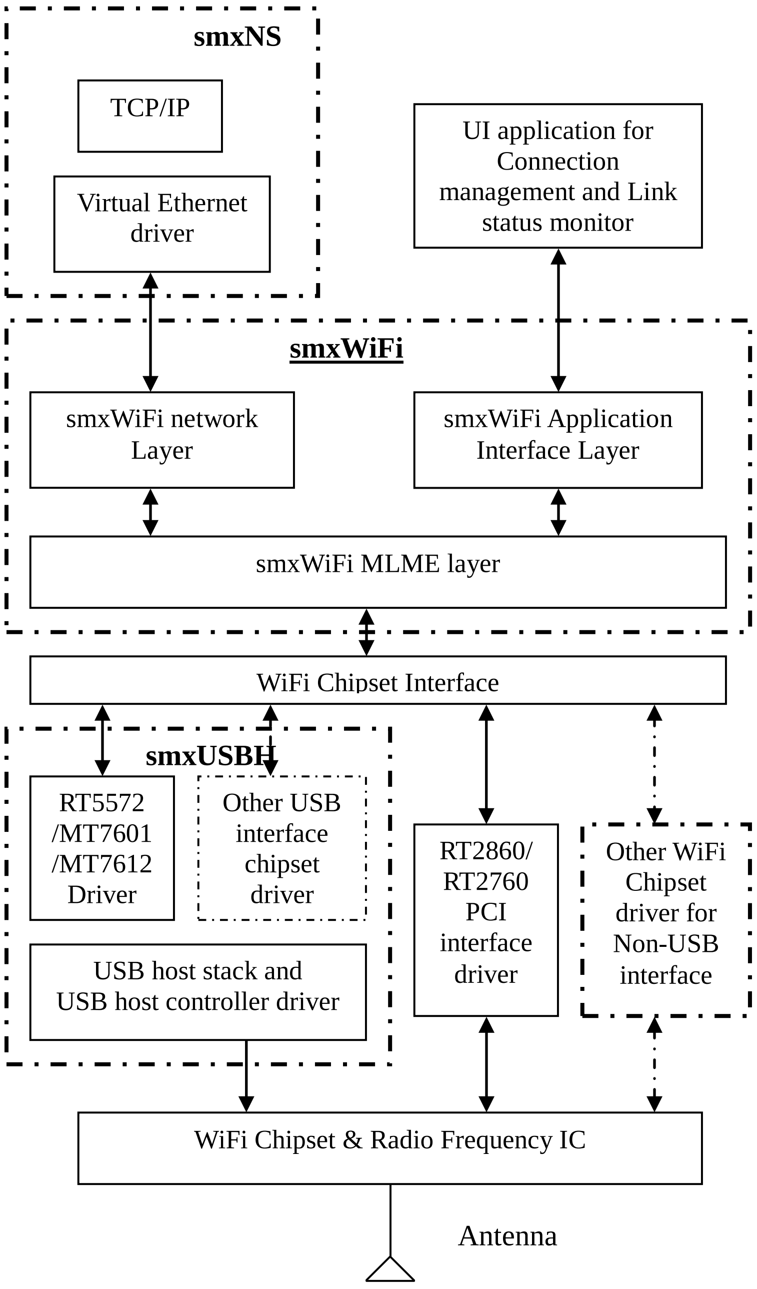
Version 1.60
February 22, 2024
by Yingbo Hu

Table of Contents

1. Overview

smxWiFi is an 802.11 MAC stack for embedded systems. It supports 802.11a/b/g/i/n/ac. It enables the embedded system to add wireless communication capability.

It requires these SMX® RTOS modules: smx for multitasking, smxNS for TCP/IP, and smxUSBH for a USB chipset driver, such as Ralink RT2573, RT2860, RT2870, RT3070, RT3572, RT5370, RT5572, MT7601, and MT7612. The following diagram summarizes the relationships among these components.


smxNS
TCP/IP
Virtual Ethernet driver
UI application for Connection management and Link status monitor
smxWiFi
smxWiFi network Layer
smxWiFi Application Interface Layer
smxWiFi MLME layer
WiFi Chipset Interface
smxUSBH
RT5572 /MT7601 /MT7612 Driver
Other USB interface chipset driver
USB host stack and USB host controller driver
RT2860/ RT2760 PCI interface driver
Other WiFi Chipset driver for Non-USB interface
WiFi Chipset & Radio Frequency IC
Antenna

To learn how protocols supported by smxWiFi correspond to the certification programs of Wi-Fi Alliance, such as **Wi-Fi Direct**™, **Wi-Fi Protected Setup**™, **Wi-Fi Protected Access**®, **WPA**™, and **WPA2**™, see www.smxrtos.com/wificert.htm. Note that the names of functions, settings, and other identifiers from the code that are documented in this manual are suggestive of these terms.

1.1 Types of WiFi Networks

BSS: Basic Service Set. A group of stations that communicate with each other.

Infrastructure BSS: This type of network needs one or more Access Points.

Independent BSS (Adhoc): Stations within the network communicate directly with each other. There is no Access Point. This mode is not commonly used any more. You can enable P2P or SoftAP features to implement peer to peer communication. These are extra cost options.

1.2 Types of Security

There are two components for security. One is the Authentication Type and the other is the Encryption Algorithm/Protocol.

Authentication Types include:

Open System: There is no special authentication. WiFi stations can connect to each other, without any password or key.

Shared Key (obsolete): An old Authentication type, only used with WEP. It is not secure so should not be used anymore.

WPA: WiFi Protected Access. This is a marketing standard put together by the Wi-Fi Alliance. It is based on 802.11i draft version. It may use TKIP as the pairwise/group key encryption so not recommend to use on AP and station.

WPA-PSK: WPA Personal. Based on 802.11i pre-shared key authentication. It may use TKIP as the pairwise/group key encryption so not recommend to use on AP and station.

WPA-EAP: WPA Enterprise. Uses the authenticated key mode that derives keys from the 802.1X standard. It may use TKIP as the pairwise/group key encryption so not recommend to use on AP and station.

WPA2: WPA version 2. It is the same as 802.11i.
WPA2-PSK: Same as WPA-PSK but for WPA version 2 (802.11i)
WPA2-EAP: Same as WPA-EAP but for WPA version 2 (802.11i)

WPA3-SAE: WPA3-Personal, Latest security protocol based on Simultaneous Authentication of Equals (introduced by 802.11s), SAE is a variant of the Dragonfly Key Exchange defined in RFC 7664. smxWiFi support for WPA3 is under development.
WPA3-EAP: Similar to WPA3 but use EAP and optional 192 bit key.

Encryption Protocols include:

WEP (obsolete): Wired Equivalent Privacy. It is the first WiFi encryption algorithm, based on RC4. There are two key lengths, 64-bit and 128-bit. It is not secure so should not be used anymore.

TKIP: Temporal Key Integrity Protocol. This encryption protocol is also based on RC4 but with some improvements, such as add message integrity check (MIC). It is not secure so should not be used anymore.

AES: Advanced Encryption Standard, also called Counter Mode with CBC-MAC Protocol (CCMP).

When discussing WiFi security, these two components are combined. For example:

WEP64 means shared key authentication and WEP 64-bit encryption.
WEP128 means shared key authentication and WEP 128-bit encryption.
WPA-PSK/AES means WPA-PSK authentication and AES encryption protocol.

2. Files

Like other SMX RTOS products, all source code for smxWiFi is stored in its own directory, named XWIFI, under the main SMX directory. Below is a summary of the directory structure.

2.1 Directory Structure

```
SMX
    APP
        DEMO        wifidemo.c (for smxNS and SMX)
    XNS             smxNS TCP/IP stack
    XUSBH           smxUSBH USB Host stack
        CDC         WiFi chipset drivers (e.g. MediaTek MTxxxx).
    XWIFI           WiFi Stack and porting layer files
        wpa_supplicant     Open Source WPA supplicant for the WPA-EAP, P2P, and WPS
        XX.YY       Build directory for SMX releases
```

2.2 Files

FILE	DESCRIPTION
wfcfg.h	WiFi stack configuration file.
wfport.c, h	Porting files.
wfnet.c, h	WiFi APIs, used by the TCP/IP and application to transfer data and control/inquire the status of the stack.
wfdrv.h	Internal data structures and API definitions for the WiFi stack.
wfaction.c	Handler of the WiFi management operation, action.
wf11ac.c	Handler for 802.11ac, such as 11ac only action, VHT capability and operation IE generation, MCS rate calculation etc.
wfact11n.c	Handler of the 802.11n only Action.
wfapeap.c	EAPOL state machine between smxWiFi SoftAP mode and open source WPA supplicant.
wfapwpa.c	WPA key management functions for SoftAP.
wfasso.c	Handler of the WiFi management operation, association.
wfauth.c	Handler of the WiFi management operation, authentication.
wfba.c	Handler of the WiFi Block ACK.
wfbsst.c	Handler of the WiFi BSS table database.
wfconn.c	Handler of the WiFi connection.
wfdisp.c	WiFi Display function between

	smxWiFi and open source WPA supplicant.
wfdls.c	Handler of the direct link setup.
wfdrs.c	Handler for dynamic rate switching.
wfdrv.c	Utility functions of the WiFi stack.
wfeap.c	EAPOL state machine between smxWiFi and open source WPA supplicant.
wfmlme.c	MLME layer of the WiFi stack.
wfsync.c	Handler of WiFi sync.
wfaes.c	Simple AES GTK decryption.
wfwep.c	WEP function (obsolete).
wfhash.c	Key hash function.
wfp2p.c, h, wfp2psm.h	Peer-to-Peer state machine between smxWiFi and open source WPA supplicant.
wfrt2860.c, h	Ralink RT2860/2760 chipset driver.
wfsoftap.c	SoftAP function.
wftkip.c	TKIP MIC function.
wfwpa.c	WPA key management functions for station.
wfwps.c, wfwpssm.h	WiFi Simple Configuration state machine between smxWiFi and open source WPA supplicant.
XUSBH\CDC\umt*.*, urt*.*	Chipset drivers.

3. Using smxWiFi

3.1 Getting Started

3.2 Configuration Settings

3.2.1 wfcfg.h

wfcfg.h contains 802.11 MAC configuration constants that allow you select features and working mode.

SWF_DRV_RT250X_USB
Set to "1" to use Ralink RT250x or RT2573 USB WiFi Chipset driver.

SWF_DRV_RT2870_USB
Set to "1" to use Ralink RT2870 USB WiFi Chipset driver. This is a 2x2 802.11n 2.4GHz band only dongle driver.

SWF_DRV_RT3070_USB
Set to "1" to use Ralink RT3070 USB WiFi Chipset driver. This is a 1x1 802.11n 2.4GHz band only dongle driver.

SWF_DRV_RT3572_USB
Set to "1" to use Ralink RT3572 USB WiFi Chipset driver. This is a 2x2 802.11n dual band 2.4GHz and 5GHz dongle driver.

SWF_DRV_RT5370_USB
Set to "1" to use Ralink RT5370 USB WiFi Chipset driver. This is a 1x1 802.11n 2.4GHz band only dongle driver.

SWF_DRV_RT5572_USB
Set to "1" to use Ralink RT5572 USB WiFi Chipset driver. This is a 2x2 802.11n dual band 2.4GHz and 5GHz dongle driver.

SWF_DRV_MT7601_USB
Set to "1" to use MediaTek MT7601 USB WiFi Chipset driver. This is a 1x1 802.11n 2.4GHz band only dongle driver.

SWF_DRV_MT7612_USB
Set to "1" to use MediaTek MT7612 USB WiFi Chipset driver. This is a 2x2 802.11ac dual band 2.4GHz and 5GHz dongle driver.

SWF_DRV_RT2860_PCI
Set to "1" to use Ralink RT2860 PCI WiFi Chipset driver. This is a 2x2 802.11n dual band card.

Note: As of v1.12, you may enable multiple chipset drivers (those you purchased), and smxWiFi will use the proper driver for the WiFi dongle that is plugged in. smxWiFi currently only supports one interface at a time.

SWF_ENABLE_11N
Set to “1” to enable 802.11n support such as the HT (High Throughput) PHY and BA (Block ACK). This should always be enabled.

SWF_ENABLE_11AC
Set to “1” to enable 802.11ac support such as the VHT (Very High Throughput) PHY. You can disable this feature even if your chipset driver supports 802.11ac; the chipset will then work for 11n mode only. **SWF_ENABLE_11N** must also be enabled

SWF_ENABLE_ADHOC_11N
Set to “1” to enable 11n HT support for adhoc network.

SWF_ENABLE_SECURITY_WEP (obsolete)
SWF_ENABLE_SECURITY_WPA
SWF_ENABLE_SECURITY_EAP
Set to “0” to disable the WEP, WPA-PSK, or WPA-EAP security features. Disabling security will improve the performance and reduce the code size and RAM requirement.

SWF_ENABLE_ENTERPRISE
Set to “1” to enable WiFi Enterprise support. WiFi Enterprise is based on WPA-EAP security. The EAP method for the enterprise we currently support is PEAP/MACHAPv2.

SWF_ENABLE_SOFTAP
SWF_ENABLE_SOFTAP_11N
Set to “1” to enable SoftAP and SoftAP 11n mode support.

SWF_MAX_CLIENT_NUM
Maximum number of clients for SoftAP.

SWF_ENABLE_DRS
Set to “1” to enable software Dynamic Rate Switching. smxWiFi will adjust the transmit rate according to the RSSI value from the AP (signal strength) and Packet Error Ratio (signal quality). Dynamic rate switching is important for VHT at higher MCS values such as MCS8/9(QAM256). If transmit rate is not setup properly, there will be a lot of re-transmit, which will not only slow down the throughput of this station, but also use too much air time, which will also slow down the whole wireless throughput at that frequency.
Some chipsets may have a built-in DRS algorithm. You can disable this setting to reduce the overhead introduced by the smxWiFi algorithm. For the current chipset drivers smxWiFi supports, this needs to be always enabled. For details of this feature, see 5.3.2.1 Dynamic Rate Switching.

SWF_ENABLE_P2P
Set to “1” to enable WiFi Peer-to-Peer (P2P) support. (Devices with this feature that are certified by the Wi-Fi Alliance can claim support for Wi-Fi Direct^{TM}.) Requires WiFi Simple Configuration, so SWF_ENABLE_WPS should also be set to 1 if you enable this feature. May also require SoftAP. If SoftAP is not enabled, P2P cannot work as Group Owner.

SWF_ENABLE_DISPLAY
Set to “1” to enable Wi-Fi Display support. (Devices with this feature that are certified by the Wi-Fi Alliance can claim support for Wi-Fi Miracast^{TM}.) Requires Wi-Fi Peer-to-Peer, so SWF_ENABLE_P2P should also be set to 1 if you enable this feature.

Note: smxWiFi implements only a small part of what is needed for full WiFi Display capability; it adds only what is needed to supplement P2P. For full support, it is necessary to implement additional protocols for WiFi and TCP/IP, new capabilities in the WiFi chipset driver, video/audio encoder/decoder, and possibly more. As a result, this feature is quite incomplete and only partially tested.

SWF_ENABLE_WPS
Set to “1” to enable WiFi Simple Configuration support. (Devices with this feature that are certified by the Wi-Fi Alliance can claim support for Wi-Fi Protected Setup^{TM}.) It is based on WPA-EAP security, using WSC method. To be the Registrar requires SoftAP.

SWF_DEFAULT_BSS_TYPE_INFRA
Set to “1” to set the BSS type to Infrastructure, which means you need an Access Point. Otherwise, the BSS type will be Independent BSS (Peer-to-Peer connection or Adhoc). It can be changed by swf_SetBssType(). Adhoc mode is not recommended anymore because it lacks sufficient security.

SWF_DEFAULT_CHANNEL
The default channel for this BSS if you are using Adhoc mode. It can be changed by swf_SetAdhocChannel()

SWF_DEBUG_LEVEL
The debug level of the WiFi stack.

SWF_ADHOC_BEACON_LOST_TIME
SWF_INFRA_BEACON_LOST_TIME
SWF_AUTH_TIMEOUT
SWF_ASSOC_TIMEOUT
SWF_DISASSOC_TIMEOUT
SWF_DEAUTH_TIMEOUT
SWF_JOIN_TIMEOUT
SWF_MIN_SCAN_TIMEOUT
SWF_MAX_SCAN_TIMEOUT
SWF_FAST_SCAN_TIMEOUT
SWF_MONITOR_INTERVAL
Timeout settings. You don’t need to change most of them for most processors, but if you are using a slow processor such as ARM7 at 48MHz, you may need to increase some of the timeout values to make sure smxWiFi will not lose important data. For example, if scan timeout is too short and a lot of APs are using that channel, swf_BssScan() may not get all the BSS information because some APs’ beacons may be lost. For detailed information, please contact Micro Digital.

SWF_CONNECT_RETRY
Number of times to retry for authentication and association

SWF_MAX_BSS_TABLE_LEN
Possible BSS networking in your area. If you have more APs than this setting, swf_BssScan() may not report all of them. Increasing it will increase the RAM usage. For detailed information, please contact Micro Digital.

SWF_MAX_EVENT_QUEUE_LEN
Internal event queue size. You don’t need to change these for most processors but if you are using a slow processor such as ARM7 at 48MHz, you may need to increase them to make sure smxWiFi will not lose important data. For detailed information, please contact Micro Digital.

SWF_MLME_TASK_STK_SIZE
MLME task stack size. If WPA-EAP is enabled, more stack is needed to run the whole EAP, WPS, or P2P state machine.

SWF_ENABLE_TX_BURST
Set to “1” to enable transmit burst mode for high performance data transfer. Enabling this feature will require more RAM and system resources. (One more task will be created.)

SWF_ENABLE_BSS_2040_COEXIST
Set to “1” to enable BSS 20/40Mhz coexist feature support. Eanabling it may increase the system’s overhead and may reduce performance. SWF_ENABLE_11N also must be enabled.

SWF_ENABLE_DLS
Set to “1” to enable Direct Link Setup feature support. Enabling it will require an additional 0.5KB of RAM.

SWF_ENABLE_ROAMING
Set to “1” to enable client roaming. Not finished yet so don’t enable it.

SWF_DISABLE_11N_WEP_TKIP
Set to “1” to force disabling 802.11n if the AP is using WEP or TKIP encryption. WEP and TKIP are not recommended for 802.11n.

SWF_CACHE_WPA_KEY
Set to “1” to cache the WPA key. It is useful for slow stations that cannot return the first message of the WPA 4-way handshake within 100ms when the link is up. Enabling it will require an additional 0.5KB of RAM.

SWF_CACHE_EAP_CREDENTIALS
Set to “1” to cache the EAP credentials, such as username, password, certificate’s file name, etc., in the connection profile. Enabling it will require additional RAM.

SWF_KEEP_ALIVE_INTERVAL
This specifies how often the station will send NULL frames during idle time so the AP will not disconnect the station. Unit is milliseconds (ms).

SWF_STAY_AWAKE_INTERVAL
SWF_AWAKE_SETUP_INTERVAL
Power save features. Unit is milliseconds (ms). SWF_KEEP_ALIVE_INTERVAL is the time the station will remain in the awake state after the last data is transmitted. In other words, this station will stay awake even when there is no activity between the station and AP for at least this time. In power save mode, the station still needs to wakeup during each beacon interval to check if the AP buffered any packets for it. Some slow systems may need some time to enter the sleep and wakeup state. The station will only sleep for (beacon interval - SWF_AWAKE_SETUP_INTERVAL) so it will not miss the next beacon.

3.3 Building the Code

After configuring wfcfg.h (see previous section), add the XWIFI source files and paths to the application project, if not already in it.

3.4 Building and Running the Demos

For SMX releases, the demos are stored in \SMX\APP\DEMO. For this product it is:

wifidemo.c	main demo file

The demo file is integrated with the smx Protosystem. It is enabled just like all other SMX module demos, as documented in the SMX Quick Start.

You also need to enable smxNS so you can transfer data through TCP/IP stack. A new virtual Ethernet device has been added to the smxNS to interface with the WiFi stack, so the networking data packet will be transferred from TCP/IP -> WiFi stack -> Ralink RT2573/RT2870 class driver -> USB Host stack -> WiFi Dongle or TCP/IP -> WiFi stack -> Ralink RT2860 chipset driver -> WiFi PCI card.

3.2.2 wfcfg.h

wfport*.h defines and implements hardware related macros and functions, such as byte order swap macros.

4. APIs

This section describes the API for smxWiFi to the TCP/IP stack and the application. All the APIs are defined in smxwifi.h. For example code showing use of these APIs, check wifidemo.c use in an application and smxNS wifi.c for use in a TCP/IP stack.

4.1 APIs for the TCP/IP Stack

The TCP/IP stack needs to call the following functions to transmit and receive Ethernet packets. Note that smxWiFi is already integrated with our TCP/IP stack, smxNS.

int **swf_Init**(void);

int **swf_Open**(void);

int **swf_Close**(void);

int **swf_Release**(void);

void **swf_RegisterNotify**(PWIFIFUNC handler);

int **swf_GetNodeID**(u8 *pMACAddr);

int **swf_SendPacket**(u8 *pData, uint len);

int **swf_IsInserted**(void);

int **swf_IsOpen**(void);

int **swf_Init** (void);

Summary	Initialize the WiFi stack.	
Descr	Call this API to initialize the whole WiFi stack. You should not call any other APIs before this API returns successfully.	
Parameters	None	
Returns	0	WiFi stack has been initialized successfully.
	<0	Error occurred when doing the initialization.
See Also	swf_Release()	

int **swf_Open** (void);

Summary	Open a WiFi interface so it can be used.
Descr	Call this API to open the WiFi interface. You also need to call swf_Connect() to connect this station to an access point or peer station. This function is an asynchronous function. You need to check if the dongle is present first by calling **swf_IsInserted**().
Parameters	None
Returns	0 Interface has been opened for the WiFi stack. <0 Error occurred when opening the interface.
See Also	swf_Close()

int **swf_Close** (void);

Summary	Close a WiFi interface.
Descr	Call this API to close the interface of WiFi stack when you don't need to use it any more.
Parameters	None
Returns	0 Interface closed. <0 Error occurred when closing the interface.
See Also	swf_Open()

int **swf_Release** (void);

Summary	Shutdown the WiFi stack.
Descr	Call this API to shutdown the whole WiFi stack. You should not call any other APIs after this API returns.
Parameters	None
Returns	0 WiFi stack has been shutdown. <0 Error occurred when shutting down the stack.
See Also	swf_Init()

int **swf_RegisterNotify** (SWF_PRECVPKTCBFUNC handler);

Summary Register the receive data call back function.

Descr Can be called any time after **swf_Init**() has been called. The callback function will be called each time the WiFi stack gets a data packet. The TCP/IP stack may need to combine the header and payload data into one Ethernet packet. The callback function should return as soon as possible. Recommended operation is just to copy the incoming packet into the TCP/IP's buffer and notify the TCP/IP stack and then return immediately.

Parameters handler Pointer to the call back function.

The callback function is defined as:

```
typedef int (*SWF_PRECVPKTCBFUNC)(u8 *pHeader, uint HeadLen,
        u8 *pPayload, uint len);
    pHeader is the pointer of the Ethernet packet header.
    HeadLen is the length of the header.
    pPayload is the pointer of the Ethernet payload.
    Len is the length of the payload packet.
```

Returns None

See Also swf_SendPacket()

int **swf_GetNodeID** (u8 *pMACAddr);

Summary Get the hardware's MAC address.

Descr Can be called any time after **swf_Init**() and **swf_Open**() have been called.

Parameters pMACAddr The buffer pointer of the MAC address. Must be at least 6 bytes.

Returns

0	Got the MAC address.
<0	The WiFi device does not exist. For example, a USB dongle is not plugged in.

See Also swf_RegisterNotify()

int **swf_SendPacket** (u8 *pData, uint len, BOOLEAN bLast);

Summary Send an Ethernet packet out through the WiFi stack.

Descr Can be called any time after **swf_Init**() and **swf_Open**() have been called. If SWF_ENABLE_TX_BURST is 0, this API is a blocking function; it will not return until the data have been sent out to the chipset.

Parameters	pData	Pointer to the Ethernet packet buffer.
	Len	The length of the packet.
	bLast	If the TCP/IP stack has more data packets to send, the IP stack may split the large TCP packet into multiple fragments. The WiFi device driver may have the ability to send multiple packets. Normally a burst transfer will get much better performance. Set this parameter to FALSE if you have more data to send.
Returns	0	Data have been sent out successfully.
	<0	Error occurred during the transmission.

See Also swf_RegisterNotify()

int **swf_IsInserted** (void);

Summary Check if the WiFi dongle or device is inserted.

Descr Can be called anytime after **swf_Init**() has been called. After the dongle is inserted, you may need to call swf_Open() when you are ready to open the WiFi interface.

Parameters None

Returns	1	WiFi dongle is present.
	0	WiFi dongle was removed.

See Also swf_Init(), swf_Open()

int **swf_IsOpen** (void);

Summary Check if the WiFi dongle or device has been opened by the TCP/IP stack.

Descr Can be called anytime after **swf_Init**() has been called.

Parameters None

Returns	1	WiFi dongle is open.
	0	WiFi dongle is not open or it is removed.

See Also swf_Init(), swf_Open()

4.2 APIs for the Application

The application can call the following functions to control the connection of the WiFi network and get the status of the link. You must call these APIs after **swf_Open**() returns success.

int **swf_Connect**(char *pNewSSID, BOOLEAN bAutoReconnect, u8 *pNewBSSID, uint iChannelWidth);

int **swf_Disconnect**(void);

int **swf_IsConnected**(void);

int **swf_RegDevEvtCallback**(SWF_PDEVEVTCBFUNC func);

int **swf_GetConnProfile**(SWF_CONN_PROFILE *pConnProfile);

int **swf_GetLinkStatus**(uint *pSignalStrength, uint *pLastRssi, uint *pLinkQuality, uint *pTxRate);

int **swf_ScanBss**(uint iScanType);

int **swf_ScanBssByChannel**(uint iScanType, u8 *pChannelList, uint iChannelListNum);

int **swf_GetScanChannelList**(u8 *pChannelList, uint *piChannelListNum);

int **swf_GetBssNum**(void);

int **swf_GetBssInfo**(uint index, SWF_BSS_INFO *pInfo);

int **swf_SetBssType**(uint iType);

int **swf_SetAdhocChannel**(uint iChannel);

int **swf_SetPowerSaveMode**(BOOLEAN bEnable);

int **swf_Connect** (char *pNewSSID, BOOLEAN bAutoReconnect, u8 *pNewBSSID, uint iChannelWidth);

Summary Connect to a new BSS.

Descr Call this API to connect your station to a new SSID.

Parameters	pNewSSID	String pointer to the new SSID. The length must be less than 32.
	bAutoReconnect	TRUE if you want smxWiFi to re-connect if the connection gets lost. **We don't recommend setting this parameter to TRUE because if the AP changed its security settings, smxWiFi will still use the old settings, and reconnection will always fail.**
	pNewBSSID	Data of the new BSS's BSSID. Must be 6 bytes. If the SSID is unique, you can pass NULL to this parameter. If two BSSes have the same SSID, you may use this parameter to indicate which BSS you want to connect. Choosing which BSS to connect among multiple SSIDs is not done by smxWiFi. The application needs to implement it based on the scan result and desired criteria, such as the highest RSSI value or 5G band preferred over 2.4G band.
	iChannelWidth	Preferred channel width. For 802.11n and 802.11ac, the AP may support wider channel bandwidth, such as 40MHz or 80Mhz,

even 160Mhz, but the client can use narrow bandwidth. It can choose to use either 20Mhz or 40Mhz bandwidth to transmit packets to the AP. Normally wider bandwidth will get better throughput but also is subject to more adjacent channel interference. Using wider bandwidth will NOT always increase throughput, especially in a busy/noisy environment. To match the AP's channel width and adjust by Dynamic Rate Switching, pass SWF_CHAN_WIDTH_AUTO for this parameter. If you know there are a lot of APs (more than 3) on the same channel, by the scan result, we recommend using narrow channel bandwidth. For example, the application can use 20Mhz instead of 40Mhz. If the number of APs at the same channel is more 5, don't use 80Mhz (802.11ac only). Use 20Mhz instead.

Returns	0	WiFi stack is connected to that BSS.
	<0	WiFi stack is not connected.

See Also swf_Disconnect()

int **swf_Disconnect** (void);

Summary Disconnect the current connection

Descr Call this API to disconnect your station from the network

Parameters None

Returns	0	WiFi stack has been disconnected from the BSS.
	<0	WiFi stack is not connected.

See Also swf_Connect()

BOOLEAN **swf_IsConnected** (void);

Summary Check if the station is connected to a BSS.

Descr Call this API to check the connect status. WiFi will try to connect to the BSS when you call **swf_Connect**(). A TRUE return from swf_Connect() does not mean you have fully connected to that AP. Connecting requires a few steps, and it is an asynchronous procedure, for example, the WPA-PSK key 4-way handshake and EAP authentication, so you can poll the status by calling this function or wait for the SWF_DEVEVT_CONNECTED event.

Parameters None

Returns	TRUE	WiFi stack has connected to the BSS.
	FALSE	WiFi stack is not connected.

See Also swf_Connect(), swf_Disconnect()

int **swf_RegDevEvtCallback**(SWF_PDEVEVTCBFUNC func)

Summary Register a callback function for device events such as link up/down event.

Descr Call this API to register a callback function to get the async event. Callback functions are called when the WiFi device is opened or closed, or the link goes up or down. For example, the Access Point wants to disconnect this station or the Access Point disappeared.

Parameters func The callback function pointer. The callback function is defined as:
typedef void (* **SWF_PDEVEVTCBFUNC**)(uint iEvent);
iEvent is one of the following events:
SWF_DEVEVT_OPEN
SWF_DEVEVT_CLOSE
SWF_DEVEVT_LINKUP
SWF_DEVEVT_LINKDN
SWF_DEVEVT_LINKDN_AP
SWF_DEVEVT_CONNECTED
SWF_DEVEVT_EAP_SUCCESS
SWF_DEVEVT_EAP_FAIL
SWF_DEVEVT_WPS_SUCCESS
SWF_DEVEVT_WPS_FAIL
SWF_DEVEVT_SOFTAP_UP
SWF_DEVEVT_SOFTAP_DN
SWF_DEVEVT_SOFTAP_ADD_CLIENT
SWF_DEVEVT_SOFTAP_REMOVE_CLIENT
SWF_DEVEVT_SCAN_CHANNEL

SWF_DEVEVT_OPEN happens when swf_Open() succeeds. If swf_Open() is called in another task, as is done in smxNS, this event is a good signal to inform the user's task it can do the connection procedure.
SWF_DEVEVT_CLOSE happens when swf_Close() is called.
SWF_DEVEVT_LINKUP happens when the station has already associated with the desired AP. That normally happens when swf_Connect() returns success.
SWF_DEVEVT_LINKDN happens when the station loses the AP's beacon signal. For example, the AP's power is off.
SWF_DEVEVT_LINKDN_AP happens when the AP de-authenticates or de-associates this station. For example the AP's settings have been changed, and the AP need to reboot.
SWF_DEVEVT_CONNECTED happens when the connection is ready to be used to transfer network data. After the station associates with the AP, if WPA security is enabled, extra authentication/key management steps will occur. Unless these steps are finished, the link between the station and AP cannot be used to transfer application's network data.
SWF_DEVEVT_EAP_SUCCESS happens when the EAP authentication succeeds.
SWF_DEVEVT_EAP_FAIL happens when the EAP authentication fails, such as if you input the wrong username/password, or the EAP authentication type is not supported by the server, although that is unlikely.
SWF_DEVEVT_WPS_SUCCESS happens when WSC succeeds.
SWF_DEVEVT_WPS_FAIL happens when WSC fails. For example, you enabled

WSC version 2 but set WEP/TKIP as the encryption.
SWF_DEVEVT_SOFTAP_UP happens when SoftAP function is UP. You need to call swf_StartSoftAP () first.
SWF_DEVEVT_SOFTAP_DN happens after you call swf_StopSoftAP () to shut down the SoftAP.
SWF_DEVEVT_SOFTAP_ADD_CLIENT happens when a WiFi station is authenticated to the SoftAP.
SWF_DEVEVT_SOFTAP_ REMOVE_CLIENT happens when a WiFi station is deauthenticated from the SoftAP.
SWF_DEVEVT_SCAN_CHANNEL happens when the wifi station begin to scan Bss on a new channel.

Returns

0	Callback function is registered.
<0	WiFi stack is not initialized yet.

See Also swf_Connect()

int **swf_GetConnProfile** (SWF_CONN_PROFILE *pConnProfile);

Summary Get the current connection profile including BSSID, SSID, Authentication and encryption, WPA passphrase or key, WEP key or default key ID.

Descr Call this API after you have connected to an AP. The profile can be saved and used to speed up the next connection to the same AP.

Parameters pConnProfile Pointer to the profile structure. SWF_CONN_PROFILE is defined as.

```
typedef struct
{
   u8   Bssid[6];
   u8   Channel;
   u8   BssType;

   uint ChannelWidth
   uint AuthMode;
   uint Encryption;
   uint KeyIndex;
   uint KeyLen;

   u8   Key[32];
   char Passphrase[64];

   char Ssid[33]; /* 32 + null */
} SWF_CONN_PROFILE;
```

Bssid is the BSS's MAC address. Normally it is the Access Point's MAC address for an infrastructure BSS network, but this may not always be the case.
Channel is the RF channel of the BSS.
BssType is the BSS type. May be SWF_BSS_TYPE_ADHOC for IBSS or SWF_BSS_TYPE_INFRA for infrastructure BSS.

ChannelWidth is the preferred channel bandwidth: SWF_CHAN_WIDTH_AUTO, SWF_CHAN_WIDTH_20, or SWF_CHAN_WIDTH_40.
AuthMode is the Authentication Mode of the BSS. See **swf_SetAuthMode()** for valid values.
Encryption is the Encryption algorithm of the BSS. See **swf_SetAuthMode()** for valid values.
KeyIndex is the default key index. It is always 0 for WPA.
KeyLen is the length of the key. If WEP is used, the keylen is either 5 or 13. If WPA-PSK is used, KeyLen is string length of the WPA-PSK passphrase, and the real master length is always 32. If WPA_EAP is used. KeyLen is always 32.
Key is the array of either the WEP key (5 or 13 bytes) or WSC master key (always 32 bytes).
Passphrase is the WPA_PSK passphrase. If WPA_EAP is used, this passphrase is null.
Ssid is the name of the BSS.

Returns 0 WiFi stack is connected to the network and the Profile data is valid.
<0 WiFi stack is not connected to an AP yet.

See Also swf_IsConnected()

int **swf_GetLinkStatus** (uint *pSignalStrength, uint *pLastRssi, uint *pLinkQuality, uint *pRxRate, uint *pTxRate);

Summary Get the current signal strength, RSSI, link quality, and transfer rate

Descr Call this API to get the link status of network. May be called after **swf_IsConnected**() returns TRUE

Parameters

pSignalStrength	Signal strength, percentage 0 to 100
pLastRssi	Beacon frame RSSI value, dBm
pLinkQuality	Link quality, percentage 0 to 100. For transmit only, Link Quality was calculated by the Packet Error Ratio
pRxRate	Receiving rate. Unit is 100Kbit/s. For example, if the rate is 54Mbit/s then it will return 540.
pTxRate	Transmit rate. Unit is 100Kbit/s. For example, if the rate is 54Mbit/s it will return 540.

Returns 0 WiFi stack is connected to the network.
<0 WiFi stack is not connected.

See Also swf_IsConnected()

int **swf_ScanBss** (uint iScanType);

Summary Scan all the BSSs.

Descr Call this API to scan all the available WiFi networks (BSSs). This API is a blocking function; it will not return until the scan is done. It will scan both 2.4G and 5G bands according to the chipset PHY configuration and country region settings. If there is no region stored in the chipset, smxWiFi will default to scan channel 1-11 (US/FCC) on 2.4G band and channel 36-48 and 149-165 (US/FCC without DFS) on 5G band. It will also use either active or passive scan which will stay in different dwell time on each channel. You need to call **swf_GetBssNum**() and **swf_GetBssInfo()** after it returns success to get the retrieved BSS information.

Parameters iScanType Scan type for the BSS Scan operation. Should be one of the following macros:

SWF_BSS_SCAN_PASSIVE
SWF_BSS_SCAN_ACTIVE
SWF_BSS_SCAN_FAST_ACTIVE

When **SWF_BSS_SCAN_PASSIVE** is passed as parameter, smxWiFi will only listen to the beacon frame at that channel. The dwell time is SWF_MAX_SCAN_TIMEOUT (200ms) defined in wfcfg.h
When **SWF_BSS_SCAN_ACTIVE** is passed as parameter, smxWiFi will send probe request at that channel and wait for the probe response frame sent back from all the APs. The dwell time is SWF_MIN_SCAN_TIMEOUT (100ms) defined in wfcfg.h.
When **SWF_BSS_SCAN_FAST_ACTIVE** is passed as parameter, smxWiFi will do the similar operation as **SWF_BSS_SCAN_ACTIVE** but stay at that channel a much shorter time. The dwell time is SWF_FAST_SCAN_TIMEOUT (30ms) defined in wfcfg.h. Normally this parameter can be used in roaming mode or there is not too much APs in the environment.

Returns	0	WiFi stack has finished scanning.
	<0	WiFi stack cannot scan the BSS.

See Also swf_GetBssNum(), swf_GetBssInfo()

int **swf_ScanBssByChannel** (uint iScanType, u8 *pChannelList, uint iChannelListNum);

Summary	Scan BSSs on only user-specified channels.
Descr	Call this API to scan available BSSs on user-specified channels. This API is a blocking function; it will not return until the scan is done. You need to call **swf_GetBssNum**() and **swf_GetBssInfo()** after it returns success to get the retrieved BSS information. The API can be used to do a quick scan of the BSS in particular channels.

Parameters

pChannelList	List of channels to scan.
iChannelListNum	Number of channels in the channel list.
iScanType	Scan type for the BSS Scan operation. Use same Macros described in **swf_ScanBss().**

Returns

0	WiFi stack has finished scanning.
<0	WiFi stack cannot scan the BSS.

See Also swf_ScanBss(), swf_GetBssNum(), swf_GetBssInfo()

int **swf_GetScanChannelList** (u8 *pChannelList, uint *piChannelListNum);

Summary	Get scan channel list when you need to call swf_ScanBss().
Descr	Call this API to get the channel list when you call swf_ScanBss(). By calling this API, you will know how many channels you need to scan so you can estimate the time swf_ScanBss() will take. Ensure the list is big enough to hold all the possible channels. The channel number also depends on the country region settings of your WiFi chipset.

Parameters

pChannelList	List of channels to scan.
ipChannelListNum	Pointer to the number of channels in the channel list.

Returns

0	Got channel list.
<0	WiFi device is not open yet.

See Also swf_ScanBss(), swf_GetBssNum(), swf_GetBssInfo()

int **swf_GetBssNum** (void);

Summary Get the number of available BSSs.

Descr Call this API after **swf_ScanBss()** returns success to get the number of available BSSs. Then you can call **swf_GetBssInfo()** multiple times to iterate through the information of each BSS.

Parameters None

Returns

>= 0	Number of available BSSs.
<0	WiFi stack cannot scan the BSS.

See Also swf_ScanBss(), swf_GetBssInfo()

int **swf_GetBssInfo** (uint index, WIFI_BSS_INFO *pInfo);

Summary Get information about the specified BSS.

Descr Call this API after **swf_ScanBss()** returns success, while iterating through the BSSs, to decide which BSS you want to connect to.

Parameters

index Index of the scanned BSS. Should be 0 to **swf_GetBssNum**() – 1.

pInfo Pointer to the information structure. WIFI_BSS_INFO is defined as:

```
typedef struct
{
   u8   Bssid[6];
   u8   Channel;
   u8   BssType;

   uint ChannelWidth;
   uint SignalStrength;
   uint iAuthMode;
   uint iEncryption;
   BOOLEAN bWPSSupported;
   u8   Ssid[33];
} SWF_BSS_INFO;
```

Bssid is the BSS's MAC address. Normally it is the Access Point's MAC address for an infrastructure BSS network, but this may not always be the case.
Channel is the primary RF channel of the BSS.
BssType is the BSS type, may be SWF_BSS_TYPE_ADHOC for IBSS or SWF_BSS_TYPE_INFRA for infrastructure BSS.
ChannelWidth is the BSS's channel bandwidth. It can be SWF_CHAN_WIDTH_20 or SWF_CHAN_WIDTH_40 for 802.11n mode BSS or SWF_CHAN_WIDTH_80 for 802.11ac mode BSS
SignalStrength is the signal strength of the BSS. It only indicates the signal strength at the last scanning time. The value is percentage from 0-100, not dBm.
iAuthMode is the Authentication Mode of the BSS. See **swf_SetAuthMode()** for

valid values. **If the Encryption is WEP, iAuthMode is hard-coded to Shared, but you may need to check what is the exact setting for the APs. Do not pass it directly to swf_SetAuthMode() for the WEP case.**
iEncryption is the Encryption algorithm of the BSS. See **swf_SetAuthMode()** for valid values.
bWPSSupported indicates if this BSS supports WSC (WiFi Simple Configuration)
Ssid is the name of the BSS. If this AP does not broadcast the SSID (Hidden AP), this field will be “”.

Returns	0	Got the BSS information.
	<0	WiFi stack cannot scan the BSS.

See Also swf_ScanBss(), swf_GetBssNum(), swf_SetAuthMode()

int **swf_SetBssType** (uint iType);

Summary Change the current BSS type.

Descr Call this API to change the current BSS type between IBSS and Infrastructure.

Parameters	iType	SWF_BSS_TYPE_ADHOC for IBSS or SWF_BSS_TYPE_INFRA for infrastructure

Returns	0	BSS type changed.
	<0	Cannot change the current BSS type.

See Also swf_Connect(), swf_Disconnect()

int **swf_ SetAdhocChannel** (uint iChannel);

Summary Set the channel for the adhoc network.

Descr Call this API to change the channel for the IBSS. The channel is only valid if this station will create the IBSS. To join an IBSS, it will use the peer’s channel.

Parameters	iChannel	iChannel to be used

Returns	0	Channel set.
	<0	Failed to set the channel.

See Also swf_Connect(), swf_Disconnect()

int **swf_ SetPowerSaveMode** (BOOLEAN bEnable);

Summary Enable the power save mode.

Descr Power save mode is disabled by default, you need to call this API to enable it after swf_Open() return success. You can call this API again to disable power save mode.

Parameters bEnable TRUE for enable and FALSE for disable.

Returns

0	Power Save Mode changed.
<0	Failed to change Power Save Mode.

See Also swf_Open()

4.3 APIs for Security

The application can call the following functions to set the security feature the WiFi network. You must call these APIs after **swf_Open**() returns success.

int **swf_SetAuthMode**(uint iAuthMode, uint iEncrypt);

int **swf_SetWepDefaultKeyID**(uint iKeyId);

int **swf_SetWepSharedKey**(uint iKeyId, u8 *pKey, uint iKeyLen);

int **swf_GenerateWPAKey**(char *pPassphrase, char *Ssid, u8 *pKey);

int **swf_SetWPAKey**(u8 *pKey);

int **swf_SetEAPCredentials**(char *pIdentity, char *pPassword, u8 *pRootCA,
u8 *pClientCertificate, u8 *pPrivateKey, u8 *pPrivateKeyPassword,
uint iMethodVendor, uint iMethodID);

int **swf_WPSStartPBC**(void);

int **swf_WPSStartPIN**(u32 pin);

int **swf_WPSGeneratePIN**(u32 *pin);

int **swf_SetAuthMode** (uint iAuthMode, uint iEncrypt);

Summary Set the authentication mode and encryption algorithm.

Descr The default authentication mode is Open System and no encryption. You can call this API to change it to Shared/WEP or WPA-PSK/TKIP, WPA-PSK/AES
Note: Shared/WEP is obsolete and should not be used anymore. TKIP is also not recommended. If you are using WEP encryption, you need to manually select to use Open or Shared Authentication because the user can select either open or shared authentication, and the beacon does not indicate which one the station should use. Selecting the wrong Authenication will cause connection failure.

Parameters iAuthMode Valid AuthMode are the following:

SWF_AUTH_MODE_OPEN
Security features are disabled if you set this mode. This is the default setting.
SWF_AUTH_MODE_SHARED (obsolete)
Old Authentication Mode. Only works with WEP encryption.
SWF_AUTH_MODE_WPANONE
WPA for Adhoc mode only.
SWF_AUTH_MODE_WPAPSK
WPA Pre-shared Key mode.
SWF_AUTH_MODE_WPA2PSK
WPA2 Pre-shared Key mode. WPA2 is also known as 802.11i.
SWF_AUTH_MODE_WPAEAP
WPA Enterprise.
SWF_AUTH_MODE_WPA2PSK
WPA2 Enterprise

iEncrypt Valid Encryption values:

SWF_ENCRYP_NONE
No encryption. Only pass this parameter with Open System authentication.
SWF_ENCRYP_WEP (obsolete)
Use WEP as the encryption algorithm/protocol. Only pass this parameter with Open or Shared authentication
SWF_ENCRYP_TKIP (not recommended)
Use TKIP as the encryption protocol.
SWF_ENCRYP_AES
Use AES (CCMP) as the encryption protocol.

Returns	0	Authentication/Encryption has been set.
	<0	Cannot set the authentication mode and encryption algorithm.

See Also swf_SetWepDefaultKeyID(), swf_SetWepSharedKey(), swf_GenerateWPAKey(), swf_SetWPAKey()

int **swf_SetWepDefaultKeyID** (uint iKeyId);

Summary Set the default KeyID of WEP (obsolete).

Descr WEP has at most 4 shared keys. Call this API to set the default key index. You need to make sure the default key index is the same as your Access Point's. Call this API after you call **swf_SetAuthMode**();

Parameters iKeyId Default Key Index. Valid values are from 0 to 3.

Returns 0 Default Key Index has been changed.

See Also swf_SetWepSharedKey()

int **swf_SetWepSharedKey** (uint iKeyId, u8 *pKey, uint iKeyLen);

Summary	Set the shared key of WEP (obsolete).
Descr	WEP has at most 4 shared keys. Call this API with a different key index to set each key individually. Call this API after you call **swf_SetAuthMode**(). **Note: You need to know the key length the AP is using. The station has no way to know whether it is 40-bit or 108-bit automatically. The beacon does not indicate this information.**

Parameters

iKeyId	Key Index. Valid values are from 0 to 3.
pKey	Pointer to the key data.
iKeyLen	The length of the key data. If you want to use WEP64 then the data length should be 5 (40 bit), if you want to use WEP128 then the data length should be 13 (108 bit).

Returns

0	Shared key has been set.
< 0	Key is not valid (length is not 5 or 13)

See Also swf_SetWepDefaultKeyID()

int **swf_GenerateWPAKey** (uint char *pPassphrase, char *Ssid, u8 *pKey);

Summary	Generate the WPA-PSK key by passphrase and SSID.
Descr	A WPA-PSK network needs the user to input a passphrase. The WPA-PSK PMK is generated using the passphrase and the network's SSID. Generating the key is a time-consuming job on a slow processor so call this function to generate the key and save it to a configuration file so you don't need to recalculate it. Call this API after you call **swf_SetAuthMode**(). **You may not need to use this function for a WPA-EAP (Enterprise) network. PMK may be generated and transferred by authentication server**

Parameters

pPassphrase	The passphrase string. Must be 8 to 63 bytes, null terminated.
Ssid	The SSID of your network, null terminated.
pKey	Pointer to the buffer to hold the generated key. The buffer must be at least 40 bytes but only the first 32 bytes are used for the key.

Returns

0	Key was generated.
< 0	Current authentication is not WPA-PSK or WPA2-PSK.

See Also swf_SetWPAKey()

int **swf_SetWPAKey** (u8 *pKey);

Summary Set the saved WPA key to the stack.

Descr Set the WiFi stack's WPA-PSK key, previously generated by **swf_GenerateWPAKey()**. If the passphrase and SSID have not changed since the last **swf_GenerateWPAKey()** call, the previously generated WPA key can be used. Call this API after you call **swf_SetAuthMode**().
Don't call this function for a WPA-EAP (Enterprise) network.

Parameters	pKey	Pointer to the buffer holding the generated key. Buffer must be 32 bytes.

Returns	0	Key was set.
	< 0	Current authentication is not WPA-PSK or WPA2-PSK.

See Also swf_GenerateWPAKey()

int **swf_SetEAPCredentials** (char *pIdentity, char *pPassword, u8 *pRootCA, u8 *pClientCertificate,
u8 *pPrivateKey, u8 *pPrivateKeyPassword, uint iMethodVendor, uint iMethodID);

Summary Set the Credentials of the WPA-EAP network.

Descr Set the Credentials of the WPA-EAP network. The WPA key will be transferred by the authenticator during the EAP authentication procedure. Call this API after you call **swf_SetAuthMode**(). **Don't call this function for a WPA-PSK network.**

Parameters	pIdentity	Username of your account. smxWiFi will only save the address of this string, so the memory storing it should not be changed during the whole authentication session.
	pPassword	Password of your account. smxWiFi will only save the address of this string, so the memory storing it should not be changed during the whole authentication session.
	pRootCA	The file name of the root CA. The file must be X.509 in DER (binary) or PEM (Base64 text) format. You may want a file system to save this CA, but a file system is not required. Check with Micro Digital about how to do it without one. Certificates require RTC support to compare the certificate's valid date. The code is in the wpa_supplicant porting layer for smxWiFi os_xbase.c
	pClientCertificate	The file name of the client certificate. A file system is not necessary. See pRootCA. **This feature is not tested yet so always pass NULL to it.**
	pPrivateKey	Key for the client certificate. **This feature is not tested yet so always pass NULL to it.**
	pPrivateKeyPassword	The private key password. **This feature is not tested yet so always pass NULL to it.**
	iMethodVendor	Desired EAP authentication method Vendor. **Normally pass 0**.
	iMethodID	Desired EAP authentication method ID. Only MD5, MSCHAPV2, and PEAP are tested. For use of WPA-EAP that requires a key, **use 25 for PEAP**.

Returns	0	Credentials were set.
	< 0	Current authentication is not WPA-EAP or WPA2-EAP.

See Also swf_SetAuthMode()

4.4 APIs for WiFi Simple Configuration (WSC)

The application can call the following functions to do WiFi Simple Configuration (WSC) operations. (Devices with this feature that are certified by the Wi-Fi Alliance can claim support for Wi-Fi Protected Setup^{TM}.) You must call these APIs after **swf_Open**() returns success.

int **swf_WPSStartPBC**(BOOLEAN bAP, u8 *pPeerAddr);

int **swf_WPSStartPIN**(BOOLEAN bAP, u32 pin);

int **swf_WPSGeneratePIN**(u32 *pin);

int **swf_WPSStartPBC** (BOOLEAN bAP , u8 *pPeerAddr);

Summary Start WSC PBC (Push Button Configuration) protocol.

Descr Try to start the WSC Push Button Configuration. Normally you should push the actual button on the AP or start the AP's PBC via the configuration page of that AP first. If bAP is FALSE, your device is WSC Enrollee. You also need to call **swf_ScanBss()** first before you call this function. This API will check the scanned result returned by the swf_ScanBss() to make sure there is at least one AP which is in the active Push Button Configuration mode. If bAP is TRUE, your device is WSC Registrar. Normally you should call swf_StartSoftAP() first to let your device work in SoftAP mode.

Parameters	bAP	If WSC is Registrar for the SoftAP mode.
	pPeerAddr	When this device is WSC Registrar, this parameter is the peer WSC Enrolle MAC address for the P2P Group Owner and SoftAP. Should pass NULL if you don't care about the enrollee's MAC address or bAP is FALSE

Returns	0	Start PBC succeeded.
	< 0	No APs have the active Push Button Configuration in the latest scan result, or for the SoftAP mode, no client connected to the AP during the WSC walk time period.

See Also swf_Connect()

int **swf_WPSStartPIN** (BOOLEAN bAP, u32 pin);

Summary	Start WSC PIN (PIN configuration) protocol.	
Descr	Try to start the WSC PIN configuration. Normally you input the 8 digit number pin on the AP’s configuration page first. If bAP is FALSE, your device is WSC Enrollee and you also need to call **swf_ScanBss()** first before you call this function. This API will check the scanned result returned by the swf_ScanBss() to make sure there is at least one AP which is in the active PIN Configuration mode. PIN can be permanent, such as on a label, or generated dynamically and displayed on your device’s display panel. It is the application’s job to retrieve the permancent pin from the storage of your device. The PIN is 7-digit PIN and 1-digit checksum. You need to use a special code to generate the permanent PIN. See our swf_WPSGeneratePIN() code as an example. If bAP is TRUE, your device is WSC Registrar and you need to call swf_StartSoftAP() first to let your device work in SoftAP mode.	
Parameters	bAP	If this device is WSC Registrar for SoftAP mode.
Returns	0	Start PIN succeeded.
	< 0	No APs have the active PIN configuration in the latest scan result or for the SoftAP mode, no client connected to the AP for WSC.
See Also	swf_Connect()	

int **swf_WPSGeneratePIN** (u32 *pin);

Summary	Generate new WSC PIN.	
Descr	Call this API to generate the random 8-digit PIN.	
Parameters	Pointer to the PIN.	
Returns	0	PIN generated.
	< 0	Device is not open yet.
See Also	swf_Connect()	

4.5 APIs for WiFi Peer-to-Peer (P2P)

The application can call the following functions to do WiFi Peer-to-Peer (P2P) operations. (Devices with this feature that are certified by the Wi-Fi Alliance can claim support for Wi-Fi Direct^{TM}.) You must call these APIs after **swf_Open**() returns success.

> int **swf_P2PStartFind**(uint iTimeout);
>
> int **swf_P2PStopFind**(void);
>
> void **swf_P2PRegEvtCallback**(SWF_PP2PEVTCBFUNC handler);
>
> int **swf_P2PDoFormation**(const u8 *DevAddr, u8 *pDevIntendedAddr, u32 pin,
> u8 *pOpChannel, uint iGOIntent);
>
> int **swf_P2PGetDeviceNum**(void);
>
> int **swf_P2PGetDeviceInfo**(uint index, SWF_P2P_DEVICE_INFO *pDevInfo);
>
> int **swf_P2PPrepareGroupOwner**(char *Passphrase, char *Ssid);

int **swf_P2PStartFind**(uint iTimeout);

Summary Start P2P find procedure.

Descr Try to start the P2P find procedure. The device will first scan all the P2P social channels (1, 6 and 11) to find if there is any P2P device around. Then it will listen at the listen channel (set in the wfcfg.h) at random intervals to let other P2P devices find it. This procedure will continue until it reaches the timeout, but this function will return immediately.

Parameters The timeout value the find procedure should last, in seconds.

Returns

0	Start find succeeded.
< 0	This device has already connected to a device, or another find procedure is in progress.

See Also swf_P2PStopFind()

int **swf_P2PStopFind**(void);

Summary Stop P2P find procedure.

Descr Stop the find procedure that is in progress. You need to call this function to stop the find procedure before the user wants to connect to the found P2P device or you get the event that another P2P device wants to connect to this device.

Parameters none

Returns	0	Stop find succeeded.
	< 0	No find procedure is in progress for this device.

See Also swf_P2PStartFind()

void **swf_P2PRegEvtCallback**(SWF_PP2PEVTCBFUNC handler);

Summary Register the P2P device event callback function.

Descr Register the P2P device event callback function so you will get the P2P device async event. P2P needs user input so those async events will allow the application to do the interactive communication between the user's application (GUI most likely) and the WiFi stack. You must call this function before you call swf_P2PStartFind().

Parameters handler P2P device event callback function. It is defined as
typedef int (* **SWF_PP2PEVTCBFUNC**)(uint iEvent, void *pInfo);

iEvent may be one of the following macros:
SWF_P2PEVT_FOUND_DEVICE
This event will happen when the find procedure finds a new device.
SWF_P2PEVT_CONNECT_REQUEST
This event will happen when this WiFi device detects a connect request from another P2P device.

pInfo is a pointer to **SWF_P2P_DEVICE_INFO** which contains the basic information of the P2P device which triggered this event.
SWF_P2P_DEVICE_INFO is defined as

```
typedef struct
{
   u8   DevAddr[6];
   u8   PriDevType[8];
   char DeviceName[33];
   char Manufacturer[65];
   char ModelName[33];
   char ModelNumber[33];
   char SerialNumber[33];
   u16  ConfigMethods;
}SWF_P2P_DEVICE_INFO;
```

DevAddr is the Device Address (MAC) of this P2P device.
PriDevType is the primary device type. The first two bytes are the category; 0050F204 is the Wi-Fi Alliance OUI; and the last two bytes are the Sub Category. For details about the Device Type see the comments about SWF_WPS_DEVICE_TYPE in wfcfg.h
DeviceName is the Device's name.
Manufacturer is the Manufacturer's name.
ModelName is the Model's name.
ModelNumber is the Model Number (text).

SerialNumber is the Serial Number (text).
ConfigMethods is the supported Configuration Methods (bitmap) of this device. Valid bitmaps are

SWF_P2P_CONFIG_METHOD_PBC
SWF_P2P_CONFIG_METHOD_PIN_DISPLAY
SWF_P2P_CONFIG_METHOD_PIN_KEYPAD

Returns none

See Also swf_P2PStartFind(), swf_GetP2PDeviceInfo()

int **swf_P2PDoFormation**(const u8 *DevAddr, u8 *pDevIntendedAddr, u32 pin, u8 *pOpChannel, uint iGOIntent);

Summary Do P2P Group Formation.

Descr After the user decides to connect to the found P2P device or after this P2P device gets a connect request from another P2P Device, the smxWiFi stack needs to do the Group Formation first by using this function. You need to call swf_P2PStartFind() first to find the nearby P2P device and also let the other device find this P2P device.
After the Group Formation is done and succeeds, if the role of this device is client, this function will also do the WSC enroll for the application. After that, it is still necessary to call swf_Connect() to do the real connection. It is the same procedure as for a normal WSC Enrollee. If P2P device will become Group Owner, it needs to call swf_StartSoftAP() to switch to SoftAP mode and then start the WSC session and finally wait for the client to connect to this P2P Group Owner.

Parameters

DevAddr	The MAC address of the peer device. You can get the address by calling swf_P2PGetDevInfo().
pDevIntendedAddr	Pointer to the intended address of the peer device. This function will fill the buffer of this intended address during the group formation procedure.
pOpChannel	Pointer to the operation channel of the peer device. This function will fill the channel during the group formation procedure.
iGOIntent	This P2P device Group Formation Group Owner intent. The intent is a value from 0 to 15. 15 means this device must be Group Owner. 0 means this device does not want to become group owner. Go intent normal should be a random number but if SWF_ENABLE_SOFTAP is 0, Group Owner Intent will be reset to 0 internally.
pin	The WSC pin during the Group Formation. If you want to use PushButton method, pass 0 for the pin.

Returns

> 0	P2P Group Formation is done and this device will become Group Owner.
0	P2P Group Formation is done and this device will become client.
< 0	P2P Group Formation failed.

See Also swf_WPSStartPCB(),swf_WPSStartPIN()

int **swf_P2PGetDeviceNum**(void);

Summary	Get the total P2P device number smxWiFi found during the find procedure.
Descr	Call this function to get the total device we found during the find procedure.
Parameters	none
Returns	The number of the P2P device we found
See Also	swf_P2PStartFind(), swf_P2PGetDeviceInfo()

int **swf_P2PGetDeviceInfo**(uint index, SWF_P2P_DEVICE_INFO *pDevInfo);

Summary Get the detailed P2P device information.

Descr Call this function to get the detailed P2P device information. For the details of structure SWF_P2P_DEVICE_INFO, see the comments in function **swf_P2PRegEvtCallback()**

Parameters

index	The device index, must be 0 to **GetP2PDeviceNum() – 1**
pDevInfo	Pointer to the SWF_P2P_DEVICE_INFO.

Returns

0	smxWIFi gets the required P2P Device Information
< 0	smxWiFi cannot get the required P2P Device Information.

See Also swf_P2PStartFind(), swf_P2PGetDeviceInfo()

int **swf_P2PPrepareGroupOwner**(char *Passphrase, char *Ssid);

Summary Prepare the Group Owner information.

Descr Call this function if swf_P2PDoFormation() result is this P2P device need to become Group Owner. The application passes the WPA passphrase of the group owner. This function will generate the required WPA key by the passphrase. This function will also generate the SSID of this Group Owner by following rule of the P2P spec.

Parameters

Passphrase	Pass phrase for the WPA key of this group owner. It is NULL terminated string, string length must be 8 to 63.
Ssid	Pointer to the generated SSID, Buffer size must be larger than 32 byte.

Returns

0	smxWIFi Prepared the information required to start the group owner.
< 0	smxWiFi cannot get the required P2P Device Information.

See Also swf_P2PDoFormation()

4.6 APIs for SoftAP

The application can call the following functions to start and stop the SoftAP. You must call these APIs after **swf_Open**() returns success.

int **swf_StartSoftAP** (char *pSSID, uint iChannel, uint iBandWidth, uint iGroupKeyUpdateInterval);

int **swf_StopSoftAP** (void);

int **swf_StartSoftAP** (char *pSSID, uint iChannel, uint iBandWidth, uint iGroupKeyUpdateInterval);

Summary Start SoftAP.

Descr Try to start the SoftAP. The parameter of this function indicates the SSID and channel you want to use for this BSS. If WPA security is used, you also need to indicate the group key update interval. You need to call swf_SetAuthMode() to step up the authentication and encryption of this BSS. You also need to call swf_GenerateWPAKey()/swf_SetWPAKey() if WPA is used. You also need to call swf_SetWepSharedKey()/swf_SetWepDefaultKeyID() if WEP is used.

Parameters

pSSID	NUL-terminated SSID of this BSS, at most 32 bytes.
iChannel	Operation channel of this BSS
iBandWidth	Bandwidth of the channel, 1 for 40MHz and 0 for 20MHz.
iGroupKeyUpdateInterval	If WPA is used, this parameter is used to specify the group key update interval, in seconds. If 0, SoftAP will not update group key.

Returns

0	Start SoftAP success.
< 0	Start SoftAP fail.

See Also swf_Open(), swf_StopSoftAP()

int **swf_StopSoftAP** (void);

Summary Stop SoftAP.

Descr Try to stop the SoftAP.

Parameters none

Returns

0	Stop SoftAP success.
< 0	Stop SoftAP failed or SoftAP has not been started yet.

See Also swf_Close(), swf_StartSoftAP()

4.7 APIs for ATE Testing

The application can call the following functions to do ATE testing.

```
int  swf_ATESendRawData(u8 *pData, uint size);
int  swf_ATESetTxContMode(BOOLEAN bEnable);
int  swf_ATESetTxPower(uint iPercentage);
int  swf_ATESetTxRate(uint iMode, uint MCS, uint iBandWidth, uint iGapInterval, uint NSS);
int  swf_ATESwitchChannel(uint iChannel);
```

Here is an example of the testing to transfer some data @ 130Mbps at channel 6 with 100% Tx power.

```
swf_ATESetTxPower(100);
swf_ATESetTxRate(2, 15, 0, 0, 1);/* HTMIX, MCS15, 20MHz, LongGI, 2SS */
swf_ATESwitchChannel(6);

while(1)
  swf_ATESendRawData((u8 *)"12345678901234567890", 20);
```

int **swf_ATESendRawData** (u8 *pData, uint size);

Summary	Send raw ATE data packet.
Descr	Send the data buffer you provide during ATE testing. An 802.11 header will still be added to the packet by the WiFi stack.

Parameters

pData	Data packet buffer pointer
size	Size of the data packet

Returns

0	Data sent.
< 0	Device is not open yet.

See Also swf_ATESwitchChannel()

int **swf_ATESetTxContMode**(BOOLEAN bEnable);

Summary	Enable/Disable the Tx continues mode.
Descr	Let the dongle continue to send data packets.

Parameters

bEnable	Enable/Disable

Returns	0	Mode set.
	< 0	Device is not open yet.
See Also	ATESendRawData()	

int **swf_ATESetTxPower** (uint iPercentage);

Summary Set the Tx power percentage.

Descr Set the power percentage. You need to call swf_ATESwitchChannel() again to make the power setting take effect.

Parameters	iPercentage	Percentage of Tx power, 1-100

Returns	0	Percentage set.
	< 0	Device is not open yet.

See Also swf_ATESwitchChannel()

int **swf_ATESetTxRate**(uint iTxRate, uint iBandWidth, uint iGapInterval);

Summary Set the Tx rate.

Descr Set the Tx rate. Check the 802.11 spec for the relationships between Rate, Bandwidth, and Gap Interval.

Parameters	iTxRate	TxRate. Unit is 100kbps. For example 54mbps is 540.
	iBandWidth	0 for 20 Mhz, 1 for 40Mhz. It is ignored if the TxRate is 11b/g rate.
	iGapInterval	Short or long GI. 0 for long GI, 1 short GI. It is ignored if the is 11b/g rate.

Returns	0	Tx rate is set.
	< 0	Device is not open yet or the Tx Rate/Bandwidth/GI combination is invalid.

See Also swf_ATESwitchChannel()

int **swf_ATESwitchChannel**(uint iChannel);

Summary Switch the channel for the ATE test.

Descr Switch the channel you want to use to do the ATE test.

Parameters	iChannel	Channel number

Returns	0	Channel is changed.
	< 0	Device is not open yet.

See Also swf_ATESetTxRate ()

4.8 Chipset Driver Interface

smxWiFi defines a set of APIs to interface with the chipset so the core stack is independent of the hardware. New chipset drivers must follow this interface and add it to the smxWiFi stack in the WiFiInit() function.

Micro Digital does not recommend that anyone other than a Micro Digital developer write a new chipset driver. The interface may not be suitable for some chipsets.

The chipset (hardware) interface is defined as:

```
typedef struct
{
   int (*Init) (SWF_SHARED_INFO *);
   int (*Release) (void);
   int (*Reset) (void);
   int (*Start) (void);
   int (*Stop) (void);
   int (*GetMACAddr)(u8 *pMACAddr);
   int (*SendMgmtPacket)(u8 *pBuffer, uint iLen, u8 Mode, u8 MCS, BOOLEAN AckRequired);
   int (*SendDataPacket)( u8 *Header, uint iHeaderLen, u8 *pPayload, uint iPayloadLen, u8 Mode, u8
                          MCS, BOOLEAN bAckRequired,BOOLEAN bRTSCTSFrame, BOOLEAN
                          bFrag);
   int (*SendRTSCTSPacket)(u8 *Frame, uint iLen, u8 Mode, u8 MCS, u8 FrameGap, u8 Type);
   int (*SendNullFrame)(u8 *pHeader, uint size, u8 Mode, u8 MCS);
   int (*EnableIbssSync)(u8 *pBeaconBuf, uint FrameLen, u8 Mode, u8 MCS, uint BeaconPeriod);
   int (*SwitchChannel)(uint iChannel, s8 TxPower);
   void (*RegRecvPacket)(RECVPACKETNOTIFY pNotify);
   int (*SetLEDStatus)(u8 bStatus);
   int (*SetSignalLED)(long Dbm);
   int (*SetSlotTime)(BOOLEAN bUseShortSlotTime);
   int (*SetBSSID)(u8 *pBssid);
   int (*SetPowerSaveMode)(uint psm);
   int (*ForceWakeup)(void);
   int (*SleepThenAutoWakeup)(u16 TbttNumToNextWakeUp, uint BeaconPeriod);
   int (*EnableBssSync)(uint BeaconPeriod);
   int (*DisableSync)(void);
   int (*SuspendMsduXmit)(void);
   int (*ResumeMsduXmit)(void);
   int (*SetTxPreamble)(u16 TxPreamble);
   int (*PrepareChannel)(uint iChannel);
   int (*LinkUp)(u8 BssType, BOOLEAN bUseShortSlotTime, BOOLEAN bShortPreamble);
   int (*LinkDown)(uint channel);
   int (*SetTxRate)(u32 NewBasicRateBitmap);
   int (*SetRadio)(BOOLEAN bOn);
   int (*EnableRX)(void);
   int (*GetTimeStamp)(u32 *pdwLow, u32 *pdwHigh);
   int (*ScanFinishResetRF)(void);
   int (*ScanAdjustRF)(uint channel);
   int (*TuningRF)(BOOLEAN IsIdle);
   int (*GetCounters)(void);
```

```
    int (*AdjustTxPower)(u32 PeriodicRound, s8 TxPwer);
    int (*ResetRawCounters)(void);
    int (*LostBeaconAction)(void);
    int (*AdjustRTSProtection)(s16 dbm);
    int (*RSSI2SignalStrength)(u8 Rssi);
    int (*AddSharedKey)(uint KeyID, uint Wcid);
    int (*RemoveSharedKey)(uint KeyID, uint Wcid);
    int (*AddPairwiseKey)(uint WCID);
    int (*RemovePairwiseKey)(uint WCID);
    int (*RxAntEvalAction)(void);
    int (*AddBASession)(uint Wcid, uint TID);
    int (*RemoveBASession)(uint Wcid, uint TID);
    int (*SetTxContMode)(BOOLEAN bEnable);
#if SWF_ENABLE_SOFTAP
    int (*GetGroupKeyWcid)(void);
    int (*AddClient)(u8 *pMACAddr, uint Aid);
    int (*RemoveClient)(uint Aid);
#endif
}SWF_WLAN_HW_OPER_T;
```

5. Built-In Features

smxWiFi has the following features built-in.

5.1. Country/Region

WiFi needs to run under certain regulatory domains, which the application may need to set up first. smxWiFi has a simple built-in country region database in wfdrv.c. Regulatory agencies may or may not allow end users to change this setting. For example, FCC will not allow any end user to change country settings. Regulatory domain information can also be programmed to the wireless hardware module you are using. smxWiFi will first read country settings from the hardware/chipset. If there is no country set in the chipset, smxWiFi will default to use the FCC/US region but no DFS channels. For 2.4GHz, it will use channel 1-11. For 5GHz band, it will use channel 36-48 and 149-165. These are UNII-1 and UNII-3 bands. After smxWiFi gets the country information, it generates the allowed channel list and Tx power table according to the country region database. So, the settings in the hardware/chipset will take priority.

This default regions settings can be changed in function WiFiInitData() in wfnet.c by the following code:

```
/* CountryRegion for BG band, default to US/FCC */
pWiFi->SharedInfo.CountryRegion     = SWF_REGION_0_BAND_BG;
/* CountryRegionABand, default to US/FCC without DFS  */
pWiFi->SharedInfo.CountryRegionForABand = SWF_REGION_10_BAND_A;
```

5.2. BSS Scanning

smxWiFi can trigger the BSS scan by calling swf_ScanBss(). This scan can be active or passive. Passive scan will only listen for the beacon frame of each SSID at that channel. Active scan will send probe request to all APs first and then wait for the probe response from the APs at that channel. The dwell time for this station to stay at that channel depends on the scan type. SWF_MAX_SCAN_TIMEOUT is for passive scan, SWF_MIN_SCAN_TIMEOUT is for active scan, and SWF_FAST_SCAN_TIMEOUT is for fast active scan which should be used for most likely roaming purpose. All the dwell times are configurable.

Active scan will save time. The station does not need to stay at that channel at least one beacon interval (at least 102.4ms for most AP) to get beacon information. If the beacon interval on a certain AP is not 102.4ms, waiting 200ms may not be enough to get that AP's information. The scan result may not have that AP at all. 100 ms should be enough for active scan. But you still need to make sure the processor is fast enough to handle all the probe responses (maybe 5-10, depending on how many APs are there) within that dwell period. DFS channels will not allow active scan. smxWiFi will force to use passive scan when it needs to do scanning on those DFS channels.

You can also call swf_ScanBssByChannel() to scan only the channel(s) of interest, such as 5G channels only. The default channel list is built based on the country/region settings. Don't pass channels that are not allowed in the country/region.

After BSS scanning is done, you normally need to call swf_GetBssNum() and swf_GetBssInfo() to get the list of APs in your environment. BssInfo structure returned by swf_GetBssInfo() includes basic information about that AP, such as SSID, BSSID, primary channel, signal strength, channel bandwidth, authentication/encryption. WPS supported, etc. Multiple APs may have the same SSID but each AP's BSSID should be unique.

5.3. MAC Layer Management Entity

smxWiFi will do most MLME tasks, such as connect to certain BSSID by either user's selection (after BSS scan) or saved profile information.

5.3.1 Start Connection

Multiple APs with the same SSID may exist. Normally this means those APs belong to the same ESSID. Your application needs to decide which AP to connect. The application may choose the one with strongest signal, for example. If multiple APs with same SSID exist, BSSID need to be passed in swf_Connect() so smxWiFi will know which one to connect.

smxWiFi will first switch to that channel and make sure it will still get beacon frame from that BSSID. Then it will start the connect handshake:

Authentication
Association
4-way handshake when RSN/WPA enabled

After the handshake is done, smxWiFi will send the SWF_DEVEVT_LINKUP event to the application by the callback function.

5.3.2 Maintain Connection

smxWiFi will also sync the BSS information by continually tracking the beacon frame. If there is no beacon coming in within 2 seconds (hard coded), smxWiFi will believe that AP is gone and notify the application by the SWF_DEVEVT_LINKDN event.

smxWiFi will also monitor any DEAUTH or DEASSOC frames from the AP. If smxWiFi receives any of these management frames, it will notify application by the SWF_DEVEVT_LINKDN_AP event.

smxWiFi also has very simple power management in station mode. It can let the chipset go to sleep mode and periodically wake up to check activities with the AP. The main MCU will not be in sleep mode. Only the wireless chipset will go to sleep mode to save power.

smxWiFi also has a monitor task that runs once a second for additional tasks, which includes RF tuning (depends on chipset function), transmit power adjustment, dynamic rate switching, radar detection, software antenna diversity evaluation, station roaming evaluation, etc.

5.3.2.1 Dynamic Rate Switching

smxWiFi will automatically adjust the transmit rate according to packet error ratio. Basic logic is:

1) After the client connects to the AP, smxWiFi will check the RSSI from the beacon frame

and decide which transmit rate it should use. RSSI vs. rate table is listed in wfdrs.c, which comes from the 802.11 spec and common knowledge about the required SNR and RSSI value for each MCS rate.

2) After setting the initial rate by RSSI, the rate will be changed when number of transmit packets is larger than 15/second. The main factor to decide the rate is the packet error ratio. It is a percentage value defined as:

   total transmit retry counter/total tx counter.

   smxWiFi DRS code has multiple rate tables for each mode (CCK/OFDM, HT and VHT). Bandwidth (20/40/80 MHz) and number of spatial stream.
   For each table, it contains multiple entries. Each entry will have the following fields:

   STBC: Use or not use STBC
   ShortGI: 800ns GI or 400ns GI
   BW: 20M/40M/80MHz channel bandwidth
   Mode: CCK, OFDM, HTMIX, HTGF or VHT PHY mode.
   MCS: Modulation and Coding Scheme index value this entry should use.
   NSS: HT/VHT PHY only, Number of Spatial Stream
   UpThld: The PER threshold that should increase the transmit rate. For example, if PER value is less than 8(%), smxWiFi will increase the transmit rate to next level.
   DnThld: The PER threshold that should decrease the transmit rate. For example, if PER value is more than 15(%), smxWiFi will decrease the transmit rate to previous level.

   Here is a sample of the table, 802.11ac 2SS:

```
#define VHT2S_INIT_INDEX 6/* The index that maps to close -65dBm */
const static SWF_RATE_TABLE_ENTRY RateTableVht2S[] =
{
/*                  [Mode]                                  [MCS/NSS]    [UpThld]  [DnThld] */
    {SWF_STBC_NONE,SWF_GAP_INTERVAL_800,SWF_BAND_WIDTH_20,SWF_MODE_VHT,0,  SWF_MCS_0,1,
        20,       100},   /* 0  VHT MCS0, 20MHz, 2SS, 13Mbps */
    {SWF_STBC_NONE,SWF_GAP_INTERVAL_800,SWF_BAND_WIDTH_20,SWF_MODE_VHT,0,  SWF_MCS_1,1,
        20,        50},   /* 1  VHT MCS1, 20MHz, 2SS, 26Mbps  */
    {SWF_STBC_NONE,SWF_GAP_INTERVAL_800,SWF_BAND_WIDTH_40,SWF_MODE_VHT,0,  SWF_MCS_1,1,
        20,        30},   /* 2  VHT MCS1, 40MHz if supported, 2SS, 54Mbps */
    {SWF_STBC_NONE,SWF_GAP_INTERVAL_800,SWF_BAND_WIDTH_40,SWF_MODE_VHT,0,  SWF_MCS_2,1,
        15,        30},   /* 3  VHT MCS2, 40MHz if supported, 2SS, 81Mbps */
    {SWF_STBC_NONE,SWF_GAP_INTERVAL_800,SWF_BAND_WIDTH_40,SWF_MODE_VHT,0,  SWF_MCS_3,1,
        15,        25},   /* 4  VHT MCS3, 40MHz if supported, 2SS, 108Mbps */
    {SWF_STBC_NONE,SWF_GAP_INTERVAL_800,SWF_BAND_WIDTH_80,SWF_MODE_VHT,0,  SWF_MCS_3,1,
        15,        25},   /* 5  VHT MCS3, 80MHz if supported, 2SS, 234Mbps */
    {SWF_STBC_NONE,SWF_GAP_INTERVAL_800,SWF_BAND_WIDTH_80,SWF_MODE_VHT,0,  SWF_MCS_4,1,
        10,        25},   /* 6  VHT MCS4, 80MHz if supported, 2SS, 351Mbps */
    {SWF_STBC_NONE,SWF_GAP_INTERVAL_800,SWF_BAND_WIDTH_80,SWF_MODE_VHT,0,  SWF_MCS_5,1,
        10,        20},   /* 7  VHT MCS5, 80MHz if supported, 2SS, 468Mbps */
    {SWF_STBC_NONE,SWF_GAP_INTERVAL_800,SWF_BAND_WIDTH_80,SWF_MODE_VHT,0,  SWF_MCS_6,1,
        10,        20},   /* 8  VHT MCS6, 80MHz if supported, 2SS, 562.5Mbps */
    {SWF_STBC_NONE,SWF_GAP_INTERVAL_800,SWF_BAND_WIDTH_80,SWF_MODE_VHT,0,  SWF_MCS_7,1,
        10,        20},   /* 9  VHT MCS7, 80MHz if supported, 2SS, 585Mbps */
    {SWF_STBC_NONE,SWF_GAP_INTERVAL_800,SWF_BAND_WIDTH_80,SWF_MODE_VHT,0,  SWF_MCS_8,1,
        8,        15},   /* 10 VHT MCS8, 80MHz if supported, 2SS, 702Mbps */
    {SWF_STBC_NONE,SWF_GAP_INTERVAL_800,SWF_BAND_WIDTH_80,SWF_MODE_VHT,0,  SWF_MCS_9,1,
        8,        15},   /* 11 VHT MCS9, 80MHz if supported, 2SS, 780Mbps */
    {SWF_STBC_NONE,SWF_GAP_INTERVAL_400,SWF_BAND_WIDTH_80,SWF_MODE_VHT,0,  SWF_MCS_9,1,
        8,        15},   /* 12 VHT MCS9, ShortGI, 80MHz if supported, 2SS, 866.7Mbps */
};
```

smxWiFi will use the MCS index which maps to –65dBm as the default rate. It is 6 in this case. It will use VHT mode PHY, LongGI, STBC is 0, 80MHz bandwidth. MCS value 4, the rate will be 351Mbps. After one second, if the PER is 10, which did not hit either the Up or Down threshold, it will remain the same transmit rate, if the PER is 4, then smxWiFi will increase to the level to entry 7, which is 468Mbps (use MCS 5 instead of 4). If the PER is 30, smxWiFi will decrease the level to entry 5, which is 234Mbps (use MCS 3 instead of 4). Transmit rate will keep changing if the PER remains in the Up or Down range until it reached the highest or lowest rate defined in the table. In this sample table, the highest rate is MCS9 with ShortGI.

Note: DRS is disabled for SoftAP mode. The algorithm needs to be improved to support SoftAP mode.

5.3.2.2 Transmit Power Adjust

smxWiFi will also automatically adjust the transmit power. Most of the code is implemented in the chipset driver because transmit power most likely depends on antenna and PA design of the hardware/chipset.

5.3.2.3. Radar Detection

smxWiFi has also built-in radar channel database. If the current connection is on a radar channel, it will keep doing radar scan to comply the regulatory requirement. This feature has not been fully tested. To use DFS channel, the device needs to get certified. Special testing needs to be done in order to get that certificate.

5.3.2.4. Software Antenna Diversity Evaluation

smxWiFi can also do periodic antenna diversity evaluation if chipset supports this feature. Details are implemented in the chipset driver. For example, select the receive path number according to RSSI value. If RSSI is lower, the chipset driver will use less receive path to boost the signal.

5.3.2.5. Station Roaming

An smxWiFi station can do roaming if the signal from the current AP is not good. It can re-associate to another BSS with the same SSID but better signal. This feature has not been fully tested and is disabled by default.

6. WPA Supplicant Support

smxWiFi WPA supplicant support is an optional feature based on the open source wpa_supplicant v2.0. This code has a BSD license.

Only EAP-MD5, EAP-MSCHAPV2, EAP-PEAP have been tested. EAP_WSC has been tested for WSC.

The code for most other EAP types can be compiled, but you need to test it. See the comments in build_config.h.

6.1. WiFi Peer-to-Peer (P2P)

WiFi Peer-to-Peer (P2P) is an optional feature built on WPA supplicant support. It depends on WiFi Simple Configuration and WPA2 security. It is a protocol designed to replace the legacy ad-hoc (IBSS) protocol for interconnecting WiFi devices. Improvements over ad-hoc include easier connection and the latest security, WPA2. When used with SoftAP, the device can be either the Group Owner (server) or client; otherwise, it can only be the client. It is based on the P2P protocol specification from the Wi-Fi Alliance.

1. SWF_ENABLE_SOFTAP also must be set to 1 to allow this WiFi device to become Group Owner.
2. wfcfg.h includes some configuration related to your device, such as operation and listen channel and postfix of the Group Owner SSID. You may change it if desired. See the comments in that file.
3. Optional features such as service discovery are not support yet.
4. P2P has been tested against Android tablet/phone.

6.2. WiFi Display

WiFi Display is an optional feature built on WPA supplicant support. It depends on WiFi Peer-to-Peer.

1. SWF_ENABLE_P2P also must be set to 1.
2. Optional features such as service discovery are not support yet.
3. smxWiFi implements only a small part of what is needed for full WiFi Display capability; it adds only what is needed to supplement P2P. For full support, it is necessary to implement additional protocols for WiFi and TCP/IP, new capabilities in the WiFi chipset driver, video/audio encoder/decoder, and possibly more. As a result, this feature is quite incomplete and only partially tested.

6.3. WiFi Enterprise (PEAP/MSCHAPv2)

MD5 and MSCHAP cannot be used to derive the WPA key, so only PEAP has been tested to work with smxWiFi in this version. Because Windows only supports SmartCard and PEAP/MSCHAPV2, if your Windows PC works fine with your authentication server, smxWiFi should also work.

1. Some versions of the PEAP spec use “client PEAP encryption” as the label to calculate TK. The latest spec (v10) uses “client EAP encryption” so we hardcode the label as “client EAP

encryption".
2. We hardcode the PEAP inner method to MSCHAPV2.

If you want to dynamically configure these two variables, contact Micro Digital.

6.4. WiFi Enterprise Test Environment

A RADIUS server is required to test WPA Enterprise. Your Access Point needs to set the Wi-Fi security to WPA-Enterprise. You also need to know the RADIUS server's IP address, the port number the RADIUS server is listening to (default is 1812), and the secret between the AP and your RADIUS server. You need to ask your network administrator for this information.

You can also build your RADIUS server on any Linux PC using FreeRADIUS server (http://freeradius.org/).

More detailed information can be found at the FreeRADIUS Server FAQ http://wiki.freeradius.org/guide/faq#How-to-Find-and-Install-FreeRADIUS?

Remember to set up your Linux PC's firewall to enable port 1812.

Here set up steps we used for FreeRADIUS Server on Fedora 13:

1. Download freeradius-2.1.10-1.fc13.i686.rpm and all the depends rpm and install it.

2. FreeRADIUS server config file is in /etc/raddb

3. Open /etc/raddb/users, add the following line at the top of that file to add the user

wifitester Cleartext-Password := "wifitester"

4. Open /etc/raddb/clients.conf, and add the following lines at the end of the file to add the AP:

```
client 192.168.1.1 {
        secret          = ap
        shortname       = ap
}
```

5. Set your Linux PC's IP address to static 192.168.1.30.

6. On the AP WiFi security settings page, input the RADIUS server address as '192.168.1.30', the secret as 'ap', and port as default 1812. Save the settings.

7. On the Linux PC, run radiusd –X to start the server.

8. Begin the WiFi Enterprise testing.

6.5. WiFi Simple Configuration (WSC) / EAP-WSC

WiFi Simple Configuration (WSC) is an optional feature built on WPA supplicant support. The method EAP-WSC is mainly just for this. This simplifies connection to an AP or other device. (Devices with this feature that are certified by the Wi-Fi Alliance can claim support for Wi-Fi Protected Setup^{TM}.)

1. By default, WSC version 2 is enabled, so WEP and TKIP may not be set up properly. In some old APs, WEP and TKIP can be used even if WSC is enabled. To support this kind of AP, you

need to comment out CONFIG_WPS2 in build_config.h

2. We only generate an 8-digit PIN.
3. wfcfg.h includes some information about your device, such as the name of your device and manufacturer. You may change it as desired. See the comments in that file.
4. This WSC implementation is only tested as Enrollee and through the in-band channel.
5. Both Enrollee and Registrar are tested. Registrar needs the SoftAP feature.
6. WSC key exchange proceduce needs some DH Key generation and derivation operations, so low speed ARM7 or ARM Cotex-M without instruction and data cache is not recommended for WSC, especially as Registrar (AP), because it may not meet the timing requirement.

6.6. Requirement for Real Time Clock

RTC is highly recommended for the Enterprise support because the random generation code os_get_random() in os_xbase.c needs the real time as the random number seed. Also when checking the server certificate, wpa_supplicant also needs real time to validate the certificate. If you have a RTC on the board, please set CONFIG_RTC in build_config.h and implement the function os_get_time() in os_xbase.c to get it.

We provide some workaround code in os_xbase.c to use the build date/time of the smxWiFi code as the current time. This approach may not work for some RADIUS servers' certificates such as the FreeRADIUS Server default server certificate. For that case, we highly recommend you add a RTC to your hardware. Otherwise, do not enable the definition of CONFIG_RTC in build_config.h

7. Optional Features

Also see section 6. WPA Supplicant Support.

7.1 SoftAP Support

SoftAP is an optional feature that provides simple AP function support to allow connection from other WiFi devices. It cannot be used to replace a full-function Access Point. SoftAP is an advanced feature and involves some complicated key generation and data encryption/decryption operations, so we recommend using a processor which is more powerful than a 200MHz ARM9. Otherwise performance may be bad, and the client may time out for various reasons.

SoftAP supports the following features:

1. Open, shared, WPA-PSK/WPA2-PSK authentication, and TKIP and AES encryption.
2. 802.11 a/b/g/n band.

Limitations in the current version:

1. Client power save is not supported.
2. Dynamic rate switching is not supported.
3. Communication between clients (routing) needs the TCP/IP stack's support. Currently smxNS does not support it so you cannot even ping from one client to another client. However, each client can communicate with the SoftAP. This limitation is due to smxNS having only one WiFi interface, yet it would handle data from multiple devices. smxNS would need to know about all the clients connected to that interface and route data to the right one.

8. Application Notes

8.1 Connecting to a Hidden AP

smxWiFi allows you to connect to a hidden AP (one which does not broadcast the SSID information in its beacon). There are two methods to connect to a hidden AP:

1. If you know the BSSID (MAC address) of the AP, you can call swf_Connect() and pass “” as SSID and pass the BSSID information to it. This is the fastest way. It is useful for connecting to a hidden AP. Note: You can get the security settings of the AP by comparing the BSSID and the BSSID Information returned by swf_GetBssInfo(). swf_SetAuthMode() needs to be called before swf_Connect(). This method can also be used when you try to re-connect to a saved hidden AP.

2. If you know the SSID, you can still pass the SSID to swf_Connect(). smxWiFi will check if there are any APs in the BSS table which already have that SSID. If there is no such named AP and there are any hidden AP(s) in the scanned BSS table, smxWiFi will try to connect to each hidden AP one by one. So the connect time may be a little longer than a normal AP. Note: Because we don’t know which AP is the one you really need to connect to at this time, the caller (application) needs to know the security settings of the AP. Normally that requires user’s input. swf_SetAuthMode() needs to be called before swf_Connect(). For details, check our sample code in wifidemo.c, TEST_HIDDEN conditional sections.

8.2 Improving Performance

smxWiFi performance is not only related to hardware speed. It is also related to the environment, Access point configuration, and how the TCP/IP stack calls the smxWiFi API to send data packets. The real world environment and shielded room are different. Multiple Access Points may be using the same channel, which will dramatically reduce performance. Also if one AP is configured to support both 802.11b/g/n, then low speed 802.11b may also reduce performance. Our testing shows that simply disabling b or g support will improve the n dongle performance by about 10%.

Sending bigger data packets by TCP/IP normally will improve performance because the WiFi chipset driver has the ability to send burst packets. One big TCP packet may become multiple IP fragments but if the TCP/IP stack can tell the WiFi stack there are more data pending (through the bLast parameter of the swf_SendPacket() function) then the chipset function may hold those packets and send them together to the hardware to get better performance. Our testing shows this kind of burst sending can improve performance by almost 100% under some conditions. We modified our Ralink dongle drivers to accumulate as much as possible until the transmit buffer is full, then send out one big packet. SWF_ENABLE_TX_BURST need to be set to 1 and the Ralink dongle driver may need an additional 40KB RAM to support this feature.

8.3 Switching between APs

The correct steps to switch between APs are:

1. Call swf_Disconnect() to disconnect the station from the AP.
2. Call swf_ScanBss() to get the latest BSS information. Make sure the desired AP is within the resulting BSS table. Do not depend on the saved information because the AP’s settings may

be changed, such as the channel and security settings.

3. Set up the correct AuthMode and Key, according to the information in the BSS table. Make sure to pass the correct AuthMode and key length for the WEP case.
4. Call swf_Connect() again with the new SSID.

8.4 Reconnecting to an AP

There are two conditions under which you may want to reconnect to an AP:

1. The AP is still on but your application needs to temporarily disconnect from the network and will connect to the same AP again later. Do similar steps to those in section 8.3 Switching between APs.
2. The application gets the SWF_DEVEVT_LINKDN or SWF_DEVEVT_LINKDN_AP event. Your application needs to keep calling swf_ScanBss() until it finds the AP's SSID again in the table, then do the steps 3 and 4 in section 8.3 Switching between APs. The application also needs to set up a timeout value for the scan BSS operation, such as two minutes. If that AP still does not show up, then the application should stop trying and report to the user the link was lost. The user then can select another AP or go check the AP.

Do not use the bAutoReconnect parameter for the swf_Connect() unless you know for certain that the AP will always be started with the same settings. In the normal case, a user could have changed the AP settings, and then this reconnect will always fail.

8.5 RAM Usage of the WiFi Stack

Global handles are allocated when you call swf_Init() and released when you call swf_Release(). That memory will be used during the whole life of the WiFi stack. Handle's size is about 8KB – 12KB, depending upon the features you enabled, such as 11n or security options. The WiFi stack will also need additional dynamic memory during the connection procedure, such as the WPA key generation, WPA 4-way handshake, and EAP authentication. This phase may need an additional 10KB of RAM, and that memory will be released to heap after the connection is done. If your system has tight memory constraints, you can try to release some application memory before the WiFi connection phase or you can start some network server only after the WiFi connection is established. The whole WiFi connection procedure, even for Enterprise or WSC, should only take at most 10 seconds.

8.6 WiFi Simple Configuration (WSC) Steps

WiFi Simple Configuration needs user's action at both the AP and the station. See the demo code in wifidemo.c, TEST_WPS_PBC and TEST_WPS_PIN, for the code for the station.

1. For Push Button Configuration (PBC), normally the user needs to do something on the AP to activate PBC mode. This is done by pressing the physical WSC/WSP button or clicking the virtual WSC/WSP button in the AP's configuration UI (typically through a web browser interface). The AP will wait for 120 seconds for the device to connect to it. During this time, the station also needs to go to the PBC mode. From the application's point of view, the application need to:
    a. Call swf_ScanBss() to get the latest BSS information.
    b. Call swf_GetBssInfo() to get BSS information for all APs and check if the

    bWPSSupported flag is set to TRUE for at least one scanned BSS. If none of the bWPSSupported flags is set, it reports an error to the user that WSC is not enabled in any the APs in this area.

   c. If at least one AP's bWPSSupported flag is set, call swf_WPSStartPBC() to start PBC. If this function returns a negative value, report to the user that no AP is in the active PBC mode. User may need to press the physical WSC/WPS button on the AP again.
   d. swf_WPSStartPBC() will not return until the WSC protocol is finished. If it returns 0, call swf_GetConnProfile() to get the AP's settings. The application can save the profile for future use. This profile already contains all the necessary information for the station to connect to that AP later. Sometimes the AP will disconnect (Deauthentication) first. If the application wants to connect to that AP immediately, just call swf_Connect(ConnProfile.Ssid, FALSE, ConnProfile.Bssid). The application does not need to scan the BSS or set up the security features again if connecting to that AP immediately after the WSC procedure.
   e. If the swf_WPSStartPBC() returns fail, the application needs to call swf_Disconnect() and then report an error to user. A possible reason for the failure is WEP is enabled while using WSC version 2, or WSC timed out.

2. For PIN configuration, normally the user should get the PIN of the device first. The application can use the preset PIN on the label/display of the station or generate a dynamic PIN every time it wants to do WSC. swf_WPSGeneratePIN() can be used to generate the dynamic PIN. This function is also useful to create a function to create the preset PINs for all devices. Then the user needs to input the PIN in the AP's configuration UI (typically through a web browser interface) and start the PIN configuration. The AP will wait for 120 seconds for the device to connect to it. During this time, the station also needs to go to the PIN mode. From the application's point of view, the application needs to:
   a. Call swf_ScanBss() to get the latest BSS information.
   b. Call swf_GetBssInfo() to get BSS information for all APs and check if the bWPSSupported flag is set to TRUE for at least one scanned BSS. If none of the bWPSSupported flags is set, it reports an error to the user that WSC is not enabled in any the APs in this area.
   c. If at least one AP's bWPSSupported is set, call swf_WPSStartPIN(pin) to start the PIN configuration, if this function returns a negative value, report to the user that no AP is in the active PIN mode. The user may need to re-input the PIN and restart the procedure on the AP again.
   d. If swf_WPSStartPIN() will not return until the WSC protocol is finished. If it returns 0, call swf_GetConnProfile() to get the AP's settings. The application can save the profile for future use. This profile already contains all the necessary information for the station to connect to that AP later. The application also needs to call swf_Disconnect() so it can established the connection with the new information. Sometimes the AP will disconnect (Deauthentication) first. If the application wants to connect to that AP immediately, just call swf_Connect(ConnProfile.Ssid, FALSE, ConnProfile.Bssid). The application does not need to scan the BSS or set up the security features again if connecting to that AP immediately after the WSC procedure.
   e. If swf_WPSStartPIN() returns fail, the application needs to call swf_Disconnect() and then report an error to user. A possible reason for the failure is WEP is enabled while using WSC version 2, or it timed out.

8.7 WiFi Peer-to-Peer (P2P) Steps

WiFi Peer-to-Peer needs user action at both devices. See the demo code in wifidemo.c, in the TEST_P2P and TEST_P2P_CONNECT (TEST_P2P_PBC/ TEST_P2P_LABEL/ TEST_P2P_PIN) case.

1. When smxWiFi is idle (device is open but has not connected to any BSS yet), call swf_P2PStartFind() to start the find procedure. A 30-second timeout value may be a common find procedure time. After this function returns 0 (success), the user's task should keep monitoring the SWF_P2PEVT_CONNECT_REQUEST and SWF_P2PEVT_FOUND_DEVICE events. When the application gets the SWF_P2PEVT_FOUND_DEVICE event, it may need to display the found device so the user can select it and try to connect to this P2P device. If the application gets the SWF_P2PEVT_CONNECT_REQUEST event, it means one of the peer devices is trying to connect to the application. The user application should pop up a prompt (window) to let the user to accept it or not. If the user does not want to accept it, just ignore this event and do nothing.
2. Either the user wants to connect to one of the found P2P devices or accept the connect request from the found P2P device. The application needs to call swf_P2PStopFind() first and then swf_P2PDoFormation() to do the group formation of these two P2P devices.
3. If swf_P2PDoFormation() succeeds
    a. If the role of this device is client, the application must call swf_GetConnectProfile() to get the required profile information and then call swf_Connect() to do the normal connect to the Group Owner.
    b. If the role of the device is Group Owner, it needs to call swf_P2PPreapreGroupOwner() to generate the WPA key and get the Group Owner's SSID. Then the application needs to start the SoftAP by calling swf_StartSoftAP() and also call swf_WPSStartPBC() or swf_WPSStartPin() to start the WSC Registrar. If the WSC Enrollee connects to the registrar within the 120 second time limit, swf_WPSStartPBC() or swf_WPSStartPIN() will return success and the Group Owner should wait for the client to do a normal connection. If swf_P2PDoFormation() failed, the application should report this to the user and do the P2P find again.
4. If the device is Group Owner and the peer device leaves, the application should stop the SoftAP and return to normal state, so this WiFi device can start a new P2P session, become a SoftAP, or connect to an AP.

Instructions for running the P2P demo code:

1. In wifidemo.c set TEST_P2P, TEST_P2P_CONNECT, and TEST_P2P_PIN to 1. All other TEST_xxx macros should be 0.
2. After you start the code and plug in the wifi dongle, the demo will keep scanning the P2P device. You will see "Scanning P2P" on the terminal screen.
3. On your P2P-enabled Android device, go to Settings->Wireless and network, and tap the Wi-Fi Direct button.
4. The Android device will try to scan for available devices. It will display "My device name" such as "Android_f6b4"" and also available devices such as "smxWiFi Device".
5. At the same time, this device will display "Found Device Android_f6b4".
6. Wait for about 20 seconds, and this device will try to automatically connect to the first P2P device it found, such as "Android_f6b4".
7. On the Android device, it will pop up dialog to let the user choose to accept this request. Select

Accept.

8. This device will display a few status updates, and then will display “Link is Connected” (Client case) or “Client connected to SoftAP” (GroupOwner case) and the Android device will display Connected.
9. If web server is enabled in nsdemo.c, you can use the Android device’s web browser to access it by inputting http://192.168.1.1
10. You can also use the Android device’s terminal app to access the telnet server at 192.168.1.1 if it is enabled in the nsdemo.c.
11. If you set TEST_P2P_CONNECT to 0, you need to initiate the connection on the Android device. Just tap the “smxWiFi Device”. This device will display “Waiting for User to Accept” and wait 5 seconds to simulate the user’s action. It will always accept the request.

9. Limitations

The current version of smxWiFi has some limitations:

- It requires a multitasking environment; standalone operation is not supported.
- Roaming is untested.
- Protected management frame (PMF) is not supported yet.

It also has limitations in its 802.11n support:

- For the Action request, we only support Block Ack, BSS 20/40MHz Public, and DLS.

It also has limitations in its 802.11ac support:

- 160Mhz (or 80MHz+80MHz) channel bandwidth support is not complete because we don’t have hardware that supports it.
- Tx Beamforming and MU-MIMO are not supported.

Appendix A. Memory Usage and Performance Summary

A.1 Size

A.1.1 Code Size

Code size varies widely depending upon CPU, compiler, and optimization level. The figures below are intended as an example.

Component	**ARM Thumb-2 IAR v6.10**	**ARM IAR v6.10**	**CF CW v7.1**
smxWiFi core stack 802.11abg	30.5 KB	47.5 KB	53 KB
smxWiFi 802.11n	4 KB	7.5 KB	7.5 KB
smxWiFi 802.11ac	2 KB	3 KB	N/A
smxWiFi WEP + WPA security	21.5 KB	27.5 KB	25 KB
smxWiFi Display (partial)	4 KB	5KB	N/A
smxWiFi Enterprise	58 KB	92.5 KB	106 KB
smxWiFi Peer-to-Peer (P2P)	57 KB	82 KB	N/A
smxWiFi Simple Configuration	24 KB	38 KB	49 KB
smxWiFi SoftAP	12 KB	20 KB	N/A
MediaTek MT7601 chipset driver	76 KB	90 KB	N/A
MediaTek MT7612 chipset driver	160 KB	190 KB	N/A
Ralink RT2573 chipset driver	20.5 KB	23 KB	31 KB
Ralink RT2860 chipset driver	N/A	N/A	N/A
Ralink RT2870 chipset driver	31.5 KB	37 KB	44 KB
Ralink RT3070 chipset driver	34.5 KB	40 KB	49 KB
Ralink RT3572 chipset driver	32.5 KB	39 KB	47 KB
Ralink RT5370 chipset driver	43.5 KB	50 KB	61 KB
Ralink RT5572 chipset driver	36 KB	46 KB	55 KB

A.1.2 Data Size (RAM Requirement)

Component	**Size**
smxWiFi core stack 802.11abg	9 KB
smxWiFi 802.11n	3 KB
smxWiFi 802.11ac	0.5 KB
smxWiFi WEP + WPA security	4 KB
smxWiFi Display (partial)	1 KB
smxWiFi Enterprise	5 KB
smxWiFi Peer-to-Peer (P2P)	6 KB
smxWiFi Simple Configuration	4 KB
smxWiFi SoftAP	(2+2*num clients) KB
MediaTek MT7601 chipset driver	12 KB
MediaTek MT7612 chipset driver	20 KB
Ralink RT2573 chipset driver	8 KB
Ralink RT2860 chipset driver	20 KB
Ralink RT2870 chipset driver	12 KB
Ralink RT3070 chipset driver	12 KB
Ralink RT3572 chipset driver	12 KB
Ralink RT5370 chipset driver	12 KB
Ralink RT5572 chipset driver	12 KB

Core Stack RAM requirement is related the settings in wfcfg.h. Please contact us if you need to reduce the RAM requirement.

A.2 Performance

For raw data performance testing, the code sends 1 or 10 burst UDP packets by swf_SendPacket() function. The payload size of the UDP packet is 1472. The code is in wifidemo.c.

For smxNS performance testing, the code sends 1 or 10 burst UDP packets by sendto() function. The payload size of the UDP packet is 1472. The code is in wifidemo.c.

The AP was working for 11n only and used 5Hz Channel 44. There were no other APs using any 5GHz channel during the testing.

A.2.1 Ralink RT3572 Chipset Driver Raw UDP Data without TCP/IP Stack

USB Host Controller	10 packet burst send	1 packet send
AT91SAM9M10 (400MHz EHCI)	12500 KB/s (100Mbps)	5900 KB/s (47.2Mbps)

A.2.2 Ralink RT3572 Chipset Driver UDP with smxNS

USB Host Controller	10 packet burst send	1 packet send
AT91SAM9M10 (400MHz EHCI)	11900 KB/s (95Mbps)	5800 KB/s (46.4Mbps)

A.2.3 Ralink RT3572 Chipset Driver TCP with smxNS

USB Host Controller	64KB burst send	1500 bytes send
AT91SAM9M10 (400MHz EHCI)	6300 KB/s (50Mbps)	1700 KB/s (13.6Mbps)

A.3 Connection Time

A WiFi station needs some time to connect to an Access Point or other station. The following are the testing results. Security is disabled. 802.11n support is disabled.

Time to connect to access point and/or other station:

USB Host Controller, Board, Speed	WiFi Chip	Open Dev	Scan BSS	Connect
AT91SAM9X35-EK Atmel board (400MHz)	MT7612	0.5 s	4 s (dual band, passive) 2s (dual band, active)	0.2 s
AT91SAM9XE512-EK Atmel board (200MHz)	RT2573	1 s	2 s	0.5 s
AT91SAM9XE512-EK Atmel board (200MHz)	RT3070	5 s	4 s	0.5 s
LM3S9B96-DK TI board (80MHz)	RT2573	1.5 s	2.6 s	0.5 s
LM3S9B96-DK TI board (80MHz)	RT3070	2 s	3 s	0.5 s
LPC2388 Keil board (72MHz)	RT2573	3.5 s	5.8 s	1 s
LPC2388 Keil board (72MHz)	RT3070	12 s	5 s	1.5 s
LPC2468 Embedded Artists board (48MHz)	RT2573	2 s	3 s	1 s
LPC2468 Embedded Artists board (48MHz)	RT3070	14 s	6 s	1.5s
LPC2939 Hitex board (120MHz)	RT2573	3 s	6 s	0.5 s
LPC2939 Hitex board (120MHz)	RT3070	11 s	5 s	1 s

MCF5329EVB Freescale board (200MHz)	RT2573	1 s	2 s	0.5 s
MCF5329EVB Freescale board (200MHz)	RT3070	5 s	5 s	0.5 s
LPC1788EA board (120MHz)	MT7601	1.5 s	4 s	0.5 s

RT2870 has not been tested but should be similar to RT3070. RT3070 times are longer than RT2573 because the firmware to download to the dongle is much larger than for RT2573 and there are more registers to set up. Also the protocol is less efficient (three USB control request packets are sent instead of two for each four bytes of data). The times for processors with OHCI controllers are much higher than for those with EHCI or proprietary controllers because the OHCI controller is less efficient for sending control requests.

Appendix B. Tested Devices

B.1 WiFi Dongles

Caution:

Vendors may change the chipset used in a particular model of WiFi dongle and just mark it with a different version number. For example, Linksys WUSB54GC uses RT2573 but WUSB54GC-EU v3 uses RT3070. Linksys WUSB600N uses RT2870 but WUSB600N ver.2 uses RT3572. Belkin F5D7050 v300x uses a Ralink chipset but other Belkin dongles use an Atheros chipset, which is not supported by smxWiFi yet. Please make sure your dongle is exactly the same as listed below.

Also even if your dongle uses one of the supported MediaTek/Ralink chipsets, it may be necessary to add a new entry to the table in the driver with the vendor ID and product ID. These IDs can be determined by plugging the dongle into a Windows PC. In Device Manager, expand Network Adapters. Right click on the line for the WiFi dongle and select Properties. On the Details tab, select Device Instance Id from the drop list.

- AmbiCom M600N-USB (RT2870)
- AmbiCom WL150N-nUSB (RT3070)
- AmbiCom WL150N-USBx (RT3070)
- Asus USB-N13 (RT3072)
- Belkin F5D7050 v3002 (RT2573)
- Belkin F5D8053 v3001 (RT2870)
- Buffalo WLI-UC-G300N (RT2870)
- D-Link DWA-140 (RT2870)
- D-Link DWA-160B2 (RT5572)
- D-Link EWUGRL2700 (RT2573)
- Linksys AE1000 (RT3572)
- Linksys WUSB54GC (RT2573)
- Linksys WUSB54GC v3 (RT3070)
- Linksys WUSB600N ver.2 (RT3572)
- MediaTek 7601 (MT7601)
- NETGEAR A6210 AC1200 (MT7612)
- Ralink 2070 (RT2573)
- Ralink 3070 WS-WNU682N (RT3070)
- Ralink 5370 OEM dongle (RT5370)

- Samsung, WIS09ABGN LinkStick USB Adapter (RT2870)
- SparkLAN WUBM-273ACN dongle (MT7612)
- SparkLAN WUBR-170GN dongle (RT3370)
- SparkLAN WUBR-507N dongle (RT3572)
- SparkLAN WUBR-508N dongle (RT5572)
- W483 802.11n PCI Card (RT2860)

B.2 Access Points

- Apple AirPort Extreme A1034 Router
- Apple AirPort Extreme A1354 802.11n Router
- ASUS RT-AX58U Wireless-AX Dual-Band Wi-Fi Router
- Belkin N150 Wireless Router
- Cisco 861W Router
- Cisco AIR-AP1262N-A-K9
- D-Link DIR-601 Router
- D-Link DIR-615 Router
- D-Link DIR-625 RangeBooster N Router (Fails. Ours defective or problem for all?)
- D-Link DIR-655 Extreme N Router
- FRITZ!Box Fon WLAN 7270
- Linksys E1200 Wireless-N Router
- Linksys E4200 Wireless-N Router
- Linksys E4500 Wireless-N Router
- Linksys E5400 Dual-Band WiFi 5 Router
- Linksys WRT54G Wireless Router
- Linksys WRT160N Wireless-N Router
- Linksys WRT320N Dual-Band Wireless-N Router
- Linksys WRT400N Dual-Band Wireless-N Router
- NETGEAR WGR614 v6 Wireless Router
- NETGEAR WNDR3300 RangeMax Dual Band Wireless-N Router
- NETGEAR WNR2000 N300 Wireless Router
- Pakedge WA-2200 802.11ac Wave 2 AP

- Pakedge WR-1 802.11ac Wave 2 Router
- TP-Link Archer A6 MU-MIMO Router
- TP-Link TL-WR841ND Wireless N Router
- TRENDnet TEW-432BRP 11g Wireless Router

B.3 RADIUS Servers

- Cisco ACS v5.2 (tested by a customer)
- FreeRADIUS Server v2.1.10 on Fedora 13
- ZyXEL NWA3160-N Built-In RADIUS Server

B.4 WiFi Peer-to-Peer (P2P) Devices

- Android 4.0.4, 4.1, 4.2.2 Samsung Note Tablet and Phone
- Android 4.2.2 Asus Memo 7 HD Tablet
- Android 9.0 Samsung Tablet 2 (2017)
- Netgear Push2TV3000

B.5 Clients for SoftAP

- Android 4.0.4, 4.1, 4.2.2 Samsung Note Tablet and Phone
- Android 4.2.2 Asus Memo 7 HD Tablet
- Android 9.0 Samsung Tablet 2 (2017)
- Apple iBook G4 laptop OS X 10.5.8
- Apple iPad 2nd Generation iOS 9.0
- Apple MacBook Pro 2017 version
- Linux with with Ralink WiFi chipset driver
- Microsoft Windows XP with Ralink, Realtek, and Atheros WiFi chipset driver
- Microsoft Windows 10 with Intel Ax200, Broadcom WiFi chipset driver

Appendix C. References

- *ANSI/IEEE Std 802.11, 2016 Edition*
- *ANSI/IEEE Std 802.11, 2012 Edition*
- *IEEE Std 802.11b-1999*
- *IEEE Std 802.11g-2003*
- *IEEE Std 802.11i-2004*
- *IEEE Std 802.11n-2009*
- *IEEE Std 802.1X-2004*
- *Wi-Fi Protected Setup Specification Version 1.0h December 2006*
- *Wi-Fi Peer-to-Peer (P2P) Technical Specification Version 1.1*
- *Wi-Fi Display Technical Specification Version 1.0.0*
- *RFC 1321 The MD5 Message-Digest Algorithm*
- *RFC 2898 The PKCS #5: Password-Based Cryptography Specification Version 2.0*
- *RFC 3174 US Secure Hash Algorithm 1 (SHA1)*
- *RFC 3748 Extensible Authentication Protocol (EAP)*
- *draft-josefsson-pppext-eap-tls-eap-10.txt Protected EAP Protocol (PEAP) Version 2*
- *Federal Information Processing Standards Publication 197, Announcing the ADVANCED ENCRYPTION STANDARD (AES)*
- *802.11 Wireless Networks The Definitive Guide (2nd_Edition)*