

**Connect-io-n™**  
**RS9110-N-11-22/24/28**  
**API Library Manual for Wi-Fi over SPI Interface**

**Redpine Signals, Inc.**

2107 N. First Street, #680

San Jose, CA 95131.

Tel: (408) 748-3385

Fax: (408) 705-2019

Email: [info@redpinesignals.com](mailto:info@redpinesignals.com)

Website: [www.redpinesignals.com](http://www.redpinesignals.com)

## Table of Contents

<b>1</b>	<b>Introduction .....</b>	<b>5</b>
1.1	Overview .....	5
1.2	Folder Structure.....	5
<b>2</b>	<b>API Library for Wi-Fi over SPI Interface .....</b>	<b>6</b>
2.1	API data structure .....	6
2.1.1	Scan input parameter structure:.....	6
2.1.2	Join input parameter structure: .....	8
2.1.3	IP configuration input parameter structure: .....	10
2.1.4	Socket create input parameter structure: .....	11
2.1.5	DNS query input parameter structure: .....	12
2.1.6	Response scan information data structure: .....	12
2.1.7	Scan response with BSSID and Network type data structure .....	13
2.1.8	Read Packet data structure (From module): .....	14
2.2	API Library .....	23
2.2.1	rsi_spi_bootloader.c.....	23
2.2.2	rsi_spi_band.c.....	24
2.2.3	rsi_spi_init.c .....	24
2.2.4	rsi_spi_scan.c .....	25
2.2.5	rsi_spi_join.c .....	25
2.2.6	rsi_spi_ipparam.c .....	25
2.2.7	rsi_spi_socket.c.....	26
2.2.8	rsi_spi_send_data.c.....	26
2.2.9	rsi_spi_read_packet.c .....	27
2.2.10	rsi_spi_socket_close.c.....	28
2.2.11	rsi_spi_disconnect.c .....	29
2.2.12	rsi_spi_power_mode.c.....	29
2.2.13	rsi_spi_interrupt_handler.c .....	30
2.2.14	rsi_spi_query_conn_status.c .....	31
2.2.15	rsi_spi_query_dhcp_params.c .....	31
2.2.16	rsi_spi_query_fwversion.c.....	32
2.2.17	rsi_spi_query_macaddress.c .....	32
2.2.18	rsi_spi_query_net_parms.c .....	33
2.2.19	rsi_spi_query_rssi.c .....	33
2.2.20	rsi_spi_fwupgrade.c .....	33
2.2.21	rsi_spi_set_listen_interval.c .....	34
2.2.22	rsi_spi_set_mac_addr.c.....	34
2.2.23	rsi_spi_query_dns.c .....	35
2.2.24	rsi_spi_query_bssid_nwtype.c.....	35
2.2.25	rsi_api_sysinit.c.....	36
2.2.26	rsi_interrupt.c .....	36
2.2.27	rsi_spi_feat_select.c.....	39
2.3	Hardware Abstraction Layer (HAL) Files .....	39
<b>3</b>	<b>Applications .....</b>	<b>45</b>
3.1	MCU Applications .....	45
3.2	PC Applications (Remote Applications) .....	50
3.2.1	Steps to Transmit TCP Data to the Wi-Fi Module.....	50

---

3.2.1	Steps to Receive TCP data from the Wi-Fi Module .....	51
3.2.2	Steps to Receive UDP Data from the Wi-Fi Module .....	52
<b>4</b>	<b>Application Notes.....</b>	<b>53</b>
4.1	Typical Usage of APIs .....	53
4.2	Power Mode 1 .....	53
<b>5</b>	<b>Compiling the Driver Source Code .....</b>	<b>55</b>
<b>6</b>	<b>Documentation .....</b>	<b>56</b>

## List of Figures

<b>Figure 1: Application Command Sequence.....</b>	<b>46</b>
<b>Figure 2: Transmit TCP Data to the Wi-Fi Module .....</b>	<b>50</b>
<b>Figure 3: Receive TCP Data from the Wi-Fi Module – 1 .....</b>	<b>51</b>
<b>Figure 4: Receive TCP Data from the Wi-Fi Module – 2.....</b>	<b>51</b>
<b>Figure 5: Receive UDP Data from the Wi-Fi Module – 1 .....</b>	<b>52</b>
<b>Figure 6: Receive UDP Data from the Wi-Fi Module – 2 .....</b>	<b>52</b>

# 1 Introduction

## 1.1 Overview

This document describes the SPI API Library for the RS9110-N-11-22/24/28 Wi-Fi modules of the Connect-io-n family from Redpine Signals. The library is delivered with the intent of providing an easy and quick way to program and configure the modules. This library has to be compiled along with the application(s) and HAL on the MCU to build a wireless communication system with the Wi-Fi module.

The library is based on the Software Programming Reference Manual (PRM) for these modules and forms a subset of the features mentioned in the Software PRM – please refer to the Release Notes of the library to know which commands are not supported. The APIs provide a simple way to setup a wireless connection and transmit/receive data from remote clients.

The library is platform/OS independent and is written in simple C language. The document also discusses the requirements of the MCU's HAL APIs and its memory. Please note that general SPI based functionality of the module is described in the Programming Reference Manual (PRM). This document cannot be used as a substitute for the PRM, it should be used as a reference to the sample driver provided.

## 1.2 Folder Structure

The folder structure and contents of this library are as follows:

1. **API\_Lib**: The source code of the API Library for Wi-Fi over SPI interface.
2. **Applications**: Contains sample applications for MCU and PC
  - a. **MCU**: Contains dummy main.c file along with the rsi\_config\_init.c, rsi\_config.h and rsi\_global.h files which contain the initial configuration and global macros for the API Library. The rsi\_api\_util files contain utility functions needed for configuration initialization and debug prints.
  - b. **PC (Linux/Windows)**: Contains example applications which can be executed on a PC connected to the Access Point and exchange data with the MCU+Wi-Fi Module system.
3. **Documentation\HTML**: Contains the documentation for the API Library's source code in HTML form.

Each of these contents is explained in more detail in the subsequent sections.

## 2 API Library for Wi-Fi over SPI Interface

This section discusses the API Library and the requirements of the HAL of the MCU.

### 2.1 API data structure

This section contains important data structure used by the Application as in/out parameter to the API's.

Following are the data structures used as in parameter for different API's.

#### 2.1.1 Scan input parameter structure:

This data structure is passed as an input parameter to `rsi_scan(rsi_uScan *uScanFrame)` API.

```
typedef union
{
    struct {
        uint8 channel[4];
        uint8 ssid[RSI_SSID_LEN];
    } scanFrameSnd;
    uint8 uScanBuf[RSI_SSID_LEN + 4];
} rsi\_uScan;
```

Structure Member Name	Member Type	Description
channel[4]	uint8	Channel Number of the Access Point. This value can be one of many values, as listed.
ssid[RSI_SSID_LEN]	uint8	SSID of the Access Point

Table1: Scan input parameter data structure

uchannelNo parameter for 2.4 Ghz

Actual Channel Number	uChannelNo parameter
All Channels	0
1	1
2	2
3	3

4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	13
14	14

Table 2: Channel Number Parameter (2.4 GHz)

uchannel parameter for 5 Ghz<sup>1</sup>

Channel Number	chan_num parameter
All channels	0
36	1
40	2
44	3
48	4
52	5
56	6
60	7
64	8
100	9
104	10
108	11
112	12
116	13
120	14
124	15
128	16
132	17
136	18
140	19
149	20
153	21
157	22

<sup>1</sup> DFS is not supported; it is advised to not use channels from 52 to 140 if the environment is expected to co-exist with Radars.

161	23
165	24

Table 3: Channel Number Parameter (5 GHz)

### 2.1.2 Join input parameter structure:

This data structure is used to pass as an input parameter to

```
typedef union {
    struct {
        uint8 nwType;
        uint8 securityType;
        uint8 dataRate;
        uint8 powerLevel;
        uint8 psk[RSI_PSK_LEN];
        uint8 ssid[RSI_SSID_LEN];
        uint8 ibssMode;
        uint8 ibssChannel;
#ifdef RSI_63BYTE_PSK_SUPPORT
        uint8 padding[3];
#else
        uint8 padding[2];
#endif
    } joinFrameSnd;
#ifdef RSI_63BYTE_PSK_SUPPORT
        uint8 uJoinBuf[RSI_SSID_LEN + RSI_PSK_LEN + 9];
#else
        uint8 uJoinBuf[RSI_SSID_LEN + RSI_PSK_LEN + 8];
#endif
    } rsi uJoin;
```

Structure Member Name	Structure Member Type	Description
-----------------------	-----------------------	-------------



Structure Member Name	Structure Member Type	Description
nwType	uint8	Network type 0 – IBSS (ad-hoc, open mode) <sup>1</sup> 1 – Infrastructure 2 – IBSS (ad-hoc) with WEP security
securityType	uint8	Security type <sup>2</sup> 0 – OPEN 1 – WPA1 2 – WPA2 3 – WEP
dataRate	uint8	Transmission data rate. Rate at which the data has to be transmitted. For Channel 14, only 11b rates (1, 2, 5.5 and 11 Mbps) are allowed
powerlevel	uint8	This fixes the Transmit Power level of the module. This value can be set as follows: 0 – Low power (6-9 dBm) 1 – Medium power (10-14 dBm) 2 – High power (15-17dBm)
psk[RSI_PSK_LEN]	uint8	Pre-shared key (Only in Security mode). It is an unused input in open mode
ssid[RSI_SSID_LEN]	uint8	SSID of the access point to join or to create (in ad-hoc mode)
ibssMode	uint8	IBSS Mode 0 – Joiner

<sup>1</sup> In the case of IBSS (ad-hoc mode), 4 joiners to a network created by the module, is supported.

<sup>2</sup> Please check the Release Notes of the individual modules' software/firmware releases to check whether this feature is supported.

Structure Member Name	Structure Member Type	Description
		1 – Creator Unused in infrastructure mode.
ibssChannel	uint8	Channel number for IBSS Creator Mode. Should be '0' in IBSS joiner mode. Unused in infrastructure mode.

Table 4:Join input parameter data structure

### 2.1.3 IP configuration input parameter structure:

```
typedef union {
    struct {
        uint8 ipparamCmd[2];
        uint8 dhcpMode;
        uint8 ipaddr[4];
        uint8 netmask[4];
        uint8 gateway[4];
        uint8 dnsip[4];
        uint8 padding;
    } ipparamFrameSnd;
    uint8 uIpparamBuf[20];
} rsi uIpparam;
```

Structure Member Name	Structure Member type	Description
ipparamCmd[2]	uint8	IP configuration command (0x01) filled by <code>rsi_ipparam_set(rsi_uIpparam *uIpparamFrame)</code> API internally.
dhcpMode	uint8	The mode with which the TCP/IP stack has to be configured. 0 – Manual 1 – DHCP
ipaddr[4]	uint8	IP address of the TCP/IP stack (valid only in Manual mode)

Structure Member Name	Structure Member type	Description
netmask[4]	uint8	Subnet mask of the TCP/IP stack (valid only in Manual mode)
gateway[4]	uint8	Default gateway of the TCP/IP stack (valid only in Manual mode)
dnsip[4]	uint8	It is the DNS server's IP address. Optional input. Should be used when DNS feature is used in DHCP disabled mode.

Table 5:IP configuration input parameter data structure

#### 2.1.4 Socket create input parameter structure:

```
typedef union {
    struct {
        uint8 socketCmd[2];
        uint8 socketType[2];
        uint8 moduleSocket[2];
        uint8 destSocket[2];
        uint8 destIpaddr[4];
    } socketFrameSnd;
    uint8 uSocketBuf[12];
} rsi\_uSocket;
```

Structure Member Name	Structure Member Type	Description
socketCmd[2]	uint8	Socket create command (0x02) to be filled by the <code>rsi_socket(rsi_uSocket *uSocketFrame)</code> API internally.
socketType[2]	uint8	Type of the socket 0 – TCP Client 1 – UDP Client 2 – TCP Server (Listening TCP) 3 – Multicast socket 4- Listening UDP. This

Structure Member Name	Structure Member Type	Description
		type of socket is used to receive/send from any remote UDP socket with any remote IP and port number.
moduleSocket[2]	uint8	Local port on which the socket has to be bound.
destSocket[2]	uint8	The destination's port. This port number is not valid for a listening socket.
destIpaddr[4]	uint8	The destination's IP address. This IP address is not valid for a listening socket.

Table 6:Socket create input parameter data structure

### 2.1.5 DNS query input parameter structure:

```
typedef union {
    struct {
        uint8 DnsGetCmd[2];
        uint8 DomainName[RSI_MAX_DOMAIN_NAME_LEN];
    } dnsFrameSnd;
    uint8 uDnsBuf[RSI_MAX_DOMAIN_NAME_LEN+2];
} rsi uDns;
```

Structure Member Name	Structure Member Type	Description
DnsGetCmd[2]	uint8	Dns query command (0x11) filled by calling API internally.
DomainName[RSI_MAX_DOMAIN_NAME_LEN]	uint8	Domain name, example: www.website.com

Table 7:DNS query input parameter data structure

### 2.1.6 Response scan information data structure:

This data structure contain important fields related scanned access points returned by the modules.

```
typedef struct {
```

```
uint8 rfChannel;
uint8 securityMode;
uint8 rssiVal;
uint8 ssid[RSI_SSID_LEN];
} rsi\_scanInfo;
```

Structure Member Name	Structure Member Type	Description
rfChannel	uint8	Channel Number of the Access Point. This value can be one of many values, as listed.
securityMode	uint8	Security Mode of the scanned Access Point. 0 – Open (No Security) 1 – WPA 2 – WPA2 3 – WEP
rssiVal	uint8	Absolute value of the RSSI information. For example, if the RSSI is -20dBm, the value returned is 20. RSSI information indicates the signal strength of the Access Point.
ssid[RSI_SSID_LEN]	uint8	SSID of the Access Point

**Table 8: Scan response data structure**

### 2.1.7 Scan response with BSSID and Network type data structure

```
typedef struct {
uint8 rfChannel;
uint8 securityMode;
uint8 rssiVal;
uint8 ssid[RSI_SSID_LEN];
uint8 uNetworkType;
uint8 BSSID[6];
} rsi\_bssid\_nwtypeInfo;
```

Structure Member Name	Structure Member Type	Description
rfChannel	uint8	Channel Number of the Access Point. This value can be one of many values, as listed.

Structure Member Name	Structure Member Type	Description
securityMode	uint8	Security Mode of the scanned Access Point. 0 – Open (No Security) 1 – WPA 2 – WPA2 3 – WEP
rssiVar	uint8	Absolute value of the RSSI information. For example, if the RSSI is -20dBm, the value returned is 20. RSSI information indicates the signal strength of the Access Point.
ssid[RSI_SSID_LEN]	uint8	SSID of the Access Point
uNetworkType	uint8	Whether the Station detected is in 0-IBSS Mode 1-Infrastructure Mode
BSSID[6]	uint8	The MAC addresses for the scanned access points.

**Table 9:Scan Response information with BSSID & network type data structure**

### **2.1.8 Read Packet data structure (From module):**

This is important out data structure called `rsi_uCmdRsp`, is used by the library to pass the values received from the Wi-Fi module to the application. This structure is updated for each call of the `rsi_spi_read_packet` API with the appropriate information. The `rsi_uCmdRsp` structure is a union of multiple structures and is explained below.

```
typedef union {
    uint8 rspCode[2];
    rsi_scanResponse scanResponse;
    rsi_mgmtResponse mgmtResponse;
    rsi_rssiFrameRcv rssiFrameRcv;
    rsi_socketFrameRcv socketFrameRcv;
    rsi_socketCloseFrameRcv socketCloseFrameRcv;
    rsi_ipparamFrameRcv ipparamFrameRcv;
    rsi_conStatusFrameRcv conStatusFrameRcv;
```

```

rsi_qryDhcpInfoFrameRcv  qryDhcpInfoFrameRcv;
rsi_qryNetParmsFrameRcv  qryNetParmsFrameRcv;
rsi_qryFwversionFrameRcv  qryFwversionFrameRcv;
rsi_disconnectFrameRcv  disconnectFrameRcv;
rsi_setMacAddrFrameRcv  setMacAddrFrameRcv;
rsi_recvFrameUdp  recvFrameUdp;
rsi_recvFrameTcp  recvFrameTcp;
rsi_recvRemTerm  recvRemTerm;
rsi_recvLtcpEst  recvLtcpEst;
rsi_dnsFrameRcv  dnsFrameRcv;
rsi_bssid_nwtypeFrameRecv  bssid_nwtypeFrameRecv;
uint8
uCmdRspBuf[RSI_FRAME_CMD_RSP_LEN +
RSI_MAX_PAYLOAD_SIZE];
} rsi uCmdRsp;

```

Structure Name	Structure Member name	Structure Member Type	Description
<a href="#">rsi_scanRespo nse</a>			Structure for scan responses.
	rspCode[4]	uint8	Response code for scan (0x95 in rspCode[0]).
	scanCount[4] ]	uint8	Number of access points found.
	strScanInfo [RSI_AP_SCA NNED_MAX]	rsi_scanIn fo	Scanned Access point information <refer table>
	status	uint8	Status of scan command 0000 Success 0002 Already associated 0003 No Access Point found 000A Invalid channel
<a href="#">rsi_mgmtRespo nse</a>			Structure for management packet response

	rspCode[2]	uint8	Response code in rspCode[0] 0x89 Card Ready 0x97 Band 0x94 Init 0x96 Join 0x91 ffinst1 upgrade done 0x92 ffinst2 upgrade done 0x93 ffddata upgrade done
	status	uint8	Status of the management command issued. 0000 success Failure code for join command 0002 Already associated 0004 PSK not configured / Incorrect PSK 0008 Fail to join in security mode 0014 Authentication failure
<a href="#">rsi_rssiFrameRcv</a>			Structure for rssi query response
	rspCode[2]	uint8	Response code (0x08 in rspCode[0])
	rssiVal[2]	uint8	Rssi value
	errCode[4]	uint8	Error code (0- on success & Non zero on failure)
<a href="#">rsi_socketFrameRcv</a>			Structure for socket create response
	rspCode[2]	uint8	Response code (0x02 in rspCode[0])
	socketType[2]	uint8	Type of the socket created. 0 – TCP Client 1 – UDP Client 2 – TCP Server (Listening TCP) 3 – Multicast socket 4- Listening UDP.
	socketDescriptor[2]	uint8	Created socket descriptor (or handle).Need to use this number while sending data through this



			socket using <code>rsi_send_data</code> and close this socket using <code>rsi_socket_close</code> API's.
	<code>moduleSocket[2]</code>	<code>uint8</code>	Local port number
	<code>moduleIpaddr[4]</code>	<code>uint8</code>	Local ipaddress
	<code>errCode[4]</code>	<code>uint8</code>	<p>Error code</p> <p>0 – Success</p> <p>-2: Socket not available. A maximum of 8 sockets can be operational at a time. If creation of more than 8 sockets is attempted, then this error is issued</p> <p>-95: ARP request failed</p> <p>-121 : Error issued when trying to connect to non-existent TCP server socket in remote terminal</p> <p>-123: Invalid socket parameters (if invalid parameters are given like, source port number 0, destination IP starts with 224 etc.)</p> <p>-124: TCP socket open failure</p> <p>-127: Socket already exists</p>
<a href="#"><u>rsi_socketCloseFrameRcv</u></a>			Structure for socket close response
	<code>rspCode[2]</code>	<code>uint8</code>	Response code (0x06 in <code>rspCode[0]</code> )
	<code>socketDsc[2]</code>	<code>uint8</code>	Descriptor of the socket closed
	<code>errorCode[4]</code>	<code>uint8</code>	<p>Error codes:</p> <p>0 – Success</p> <p>-2 : Socket not available</p> <p>-91: IGMP Error</p> <p>-95: ARP request failed</p>
<a href="#"><u>rsi_ipparamFrameRcv</u></a>			Structure for ipconfiguration response
	<code>rspCode[2]</code>	<code>uint8</code>	Response Code (0x01 in <code>rspCode[0]</code> )
	<code>macAddr[6]</code>	<code>uint8</code>	Mac address of WiFi module

	ipaddr[4]	uint8	IP address of Wi-Fi module
	netmask[4]	uint8	Network mask configured
	gateway[4]	uint8	Gateway configured
	errCode[4]	uint8	Error codes: 0 – Success -100: DHCP handshake failure -4 : IP configuration failed
<a href="#"><u>rsi_conStatusFrameRcv</u></a>			Structure for Connection query status response
	rspCode[2]	uint8	Response code (0x0A in rspCode[0])
	state[2]	uint8	This indicates whether the module is connected to an Access Point or not. 0 – Not connected 1 – Connected
	errorCode[4]	uint8	Error code =0 –on success !=0 – on failure
<a href="#"><u>rsi_gryDhcpInfoFrameRcv</u></a>			Structure for query dhcp information response.
	rspCode[2]	uint8	Response code(0x0B in rspCode[0])
	leaseTime[4]	uint8	The total lease time for the DHCP connection.
	leaseTimeLeft[4]	uint8	The lease time left for the DHCP connection.
	renewTime[4]	uint8	The lease time left for the DHCP connection.
	Rebind_time[4]	uint8	The time left for rebind of the IP address acquired through DHCP.
	serverIpAddress[4]	uint8	The IP address of the DHCP server.
	errorCode[4]	uint8	Error code =0 –on success !=0 – on failure

<a href="#"><u>rsi_gryNetParamsFrameRcv</u></a>			Structure for query network param response.
	rspCode[2]	uint8	Response code (0x09 in rspCode[0])
	wlanState[2]	uint8	This indicates whether the module is connected to an Access Point or not. 0 – Not connected 1 – Connected
	ssid[32]	uint8	This value is the SSID of the Access Point to which the module is connected.
	ipaddr[4]	uint8	This is the IP Address configured to Wi-Fi module.
	subnetMask[4]	uint8	This is the Subnet Mask configured to Wi-Fi module.
	gateway[4]	uint8	This is the gateway configured to WiFi module
	dhcpMode[2]	uint8	This value indicates whether the module is configured for DHCP or Manual IP configuration. 0 – Manual IP configuration 1 – DHCP
	connType[2]	uint8	This value indicates whether the module is operational in Infrastructure mode or AdHoc mode <sup>1</sup> . 0 – AdHoc mode 1 – Infrastructure mode
	errorCode[4]	uint8	Error code: =0 on success !=0 on failure
<a href="#"><u>rsi_gryFwversionFrameRcv</u></a>			Structure for query firmware version response
	rspCode[2]	uint8	Response code (0x0F in rspCode[0])
	fwversion[20]	uint8	Version of the firmware loaded in the module. This is given in string format. The firmware version format

<sup>1</sup> Please check the release notes of the firmware to see if AdHoc mode is supported.

			is x.y.z (e.g., 1.3.0).
<a href="#"><u>rsi_disconnectFrameRcv</u></a>			Structure for disconnect to Wi-Fi network response
	rspCode[2]	uint8	Response code (0x0C in rspCode[0])
	errorCode[4]	uint8	Error code: =0 on success !=0 on failure
<a href="#"><u>rsi_setMacAddrFrameRcv</u></a>			Structure for set mac address command response.
	rspCode[2]	uint8	Response code (0x10 in rspCode[0])
	errorCode[4]	uint8	Error code 0: Success. 1: This error code is returned if the command is given after WLAN configuration (after "Join" command).
	padding[2]	uint8	Padding purpose
<a href="#"><u>rsi_recvFrameUdp</u></a>			Structure for receive upd packet
	rspCode[2]	uint8	Response code (0x07 in rspCode[0])
	recvSocket[2]	uint8	Socket descriptor on which data received
	recvBufLen[4]	uint8	Receive packet length
	recvDataOffsetSize[2]	uint8	Offset where the actual payload data start in the buffer.
	fromPortNum[2]	uint8	Port number of remote machine (from where this packet received)
	fromIpaddr[4]	uint8	IP address of remote machine (from where this packet received)
	recvDataOffsetBuf[RSI_RXDATA_OFFSET_UDP]	uint8	Dummy data before actual payload start, need to ignore this content.
	recvDataBuf[RSI_MAX_PA	uint8	Actual payload data.

	YLOAD_SIZE]		
	padding[2]	uint8	padding
<a href="#"><u>rsi_recvFrameTcp</u></a>			Structure for receive TCP packet.
	rspCode[2]	uint8	Response code (0x07 in rspCode[0])
	recvSocket[2]	uint8	Socket descriptor on which data received
	recvBufLen[4]	uint8	Receive packet length
	recvDataOffsetSize[2]	uint8	Offset where the actual payload data start in the buffer.
	fromPortNum[2]	uint8	Port number of remote machine (from where this packet received)
	fromIpaddr[4]	uint8	IP address of remote machine (from where this packet received)
	recvDataOffsetBuf[RSI_RXDATA_OFFSET_TCP]	uint8	Dummy data before actual payload start, need to ignore this content.
	recvDataBuf[RSI_MAX_PAYLOAD_SIZE]	uint8	Actual payload data.
	padding[2]	uint8	padding
<a href="#"><u>rsi_recvRemote</u></a>			Structure for remote terminate response.
	rspCode[2]	uint8	Response code (0x05 in rspCode[0])
	socket[2]	uint8	Socket descriptor for which the Remote termination has happened.
	errCode[4]	uint8	Error code: 0 – Success, -121: Socket creation failed.
<a href="#"><u>rsi_recvLtcpest</u></a>			Structure for Listening socket establishment response.
	rspCode[2]	uint8	Response code (0x04 in rspCode[0])
	socket[2]	uint8	Socket descriptor for which the

			connection has happened.
	fromPortNum [2]	uint8	Port number of remote client.
	fromIpaddr[ 4]	uint8	IP address of remote client.
	errCode[4]	uint8	Error code: =0 on success !=0 on failure
<a href="#">rsi_dnsFrameRcv</a>			Structure for DNS query response.
	rspCode[2]	uint8	Response code (0x14 in rspCode[0])
	uipcount[2]	uint8	This indicates number of IPs resolved for the given domain name
	ipaddr[RSI_MAX_DNS_REPLY][4]	uint8	This indicates number of IPs resolved for the given domain name
	errCode[4]	uint8	Error code. 0 :Success failure code: -190 DNS_RESPONSE_TIME_OUT -75 DNS_ID_ERROR -74 DNS_OPCODE_ERROR -73 DNS_RCODE_ERROR -72 DNS_COUNT_ERROR -85 INVALID_VALUE -70 DNS_CLASS_ERROR -69 DNS_NOT_FOUND
<a href="#">rsi_bssid_nwtypeFrameRecv</a>			Structure for query BSSID and type of network response.
	rspCode[4]	uint8	Response code (0xA1 in rspCode[0])
	scanCount[4]	uint8	Number of access points scanned.
	strBssid_NwtypeInfo[RSI_AP_SCANNE	rsi_bssid_nwtypeInfo	Scanned access point information (along with BSSID & network type)

	D_MAX];		
	status	uint8	Status of scan command. 0000 Success 0002 Already associated 0003 No Access Point found 000A Invalid channel

Table 10: Common Response data structure

## 2.2 API Library

The API Library provides APIs which are called by the Application of the MCU in order to configure the Wi-Fi module and also exchange data over the network.

The files included in the library are listed in the sections below – each file includes a specific API or a list of APIs. Please refer to the HTML documentation for more details.

### 2.2.1 [rsi\\_spi\\_bootloader.c](#)

This file contains the API for loading the software bootloader.

#### API Prototype:

```
int16 rsi_bootloader(void)
```

#### Parameters:

None

#### Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

#### Description:

This API is used to load the bootloader code into Wi-Fi module at specified locations and bring the module out from soft reset. The bootloader in turn loads the functional firmware from the module's non-volatile memory and gives control to functional firmware. This API has to be called only after the `rsi_spi_iface_init` API.

Note: Inside this file, the user should replace "Firmware/sbinst1", "Firmware/sbinst2", "Firmware/sbdata1", "Firmware/sbdata2" with proper paths depending on the user's setup, so that these files can be referenced properly.

### 2.2.2 [rsi\\_spi\\_band.c](#)

This file contains the API for the Band command.

API Prototype:

```
int16 rsi_band(uint8 band)
```

Parameters:

uint8 band 0-for 2.4GHz and 1-for 5GHz.

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to select the 2.4 GHz mode or 5GHz mode. This API is required only for the RS9110-N-11-28 module which has an option of operating in either the 2.4GHz or 5GHz modes. It is optional for the RS9110-N-11-22 and RS9110-N-11-24 modules which can operate in the 2.4GHz band only. By default, all modules operate in the 2.4GHz mode. This API has to be called only after the `rsi_bootloader` API.

### 2.2.3 [rsi\\_spi\\_init.c](#)

This file contains the API for the Init command, which initializes the module's Baseband and RF components.

API Prototype:

```
int16 rsi_init(void)
```

Parameters:

None

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:



This API initializes the Wi-Fi module's Baseband and RF components. It has to be called only after the `rsi_bootloader` and `rsi_band` APIs.

#### **2.2.4**     [rsi\\_spi\\_scan.c](#)

This file contains the API for the Scan command.

##### API Prototype:

```
int16 rsi_scan(rsi_uScan *uScanFrame)
```

##### Parameters:

`rsi_uScan *uScanFrame` – Pointer scan parameter structure.

##### Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

##### Description:

This API is used to scan for Access Points. This API should be called only after the `rsi_init` API.

#### **2.2.5**     [rsi\\_spi\\_join.c](#)

This file contains the API for the Join command.

##### API Prototype:

```
int16 rsi_join(rsi_uJoin *uJoinFrame)
```

##### Parameters:

`rsi_uJoin *uJoinFrame` – Pointer to join parameter structure.

##### Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

##### Description:

This API is used to connect the Wi-Fi module to an Access Point. This API should be called only after `rsi_scan` API.

#### **2.2.6**     [rsi\\_spi\\_ipparam.c](#)

This file contains the API for IP configuration.

##### API Prototype:

```
int16 rsi_ipparam_set(rsi_uIpparam *uIpparamFrame)
```

Parameters:

`rsi_uIpparam *uIpparamFrame` – Pointer to the ip configuration parameter structure.

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to configure the IP address, Subnet Mask and Gateway IP address for the module in manual mode or DHCP mode. The Wi-Fi module should be successfully connected to an Access point (using the `rsi_join` API) before calling this API. If DHCP mode is enabled, then it has to be ensured that a DHCP server is present in the network.

### **2.2.7**     [rsi\\_spi\\_socket.c](#)

This file contains the API to open a TCP/UDP Sever/Client socket inside the Wi-Fi module.

API Prototype:

```
int16 rsi_socket(rsi_uSocket *uSocketFrame)
```

Parameters:

`rsi_uSocket *uSocketFrame` – Pointer to socket create parameter structure.

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to open a TCP/UDP Server/Client socket in the Wi-Fi module. It has to be called only after the module has been assigned an IP address using the `rsi_ipparam_set` API.

### **2.2.8**     [rsi\\_spi\\_send\\_data.c](#)

This file contains the API to send application data payloads to the Wi-Fi module, which then transmits them over the Wi-Fi network.

API Prototype:

```
int16 rsi_send_data(
```

```
uint16 socketDescriptor,  
uint8 *payload,  
uint32 payloadLen,  
uint8 protocol  
)
```

#### Parameters:

uint16 socketDescriptor – Socket descriptor, used to identify the socket on which data has to be transmitted. The socket descriptor is returned by the module at the time of socket creation – it is updated by the library in the `uCmdRspFrame` structure after the call to the `rsi_spi_read_packet` API, which follows the call to the `rsi_socket` API.

uint8 \*payload – Pointer to data payload buffer which has to be transmitted.

uint32 payloadLen – Length of the data payload.

uint8 protocol – Type of the protocol (TCP/UDP).

0 –for UDP

1- for TCP

#### Returns:

0 on success

-1 on Timeout

-2 on SPI interface level failure

-3 on receiving a “buffer full” response from the module.

-4 if the module is in sleep mode

#### Description:

This API used to send TCP/UDP data using an already opened socket. This function should be called after successfully opening a socket using the `rsi_socket` API. If this API return error codes like -1,-3,-4, Application need to retry this function until successfully send the packet over WiFi module.

### **2.2.9 [rsi\\_spi\\_read\\_packet.c](#)**

This file contains the API to receive responses from the Wi-Fi module for the commands (e.g., `rsi_band`, `rsi_init`, `rsi_scan`, etc.) that are sent to it.

#### API Prototype:

```
int16 rsi_read_packet(  
    rsi_uCmdRsp *uCmdRspFrame  
)
```

#### Parameters:

`rsi_uCmdRsp *uCmdRspFrame` – This is an output parameter to hold the response frame from the module.

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to read packets from the Wi-Fi module. The packets may be responses to commands sent to it or asynchronous packets like received data, remote socket termination, etc. The application has to check the Packet IRQ Status by calling the `rsi_checkPktIrq` API regularly and then call this API if it finds that a packet is pending to be read from the Wi-Fi module.

This API serves as a common API to read the responses from the Wi-Fi module for all the command (`rsi_bootloader`, `rsi_band`, `rsi_init`, `rsi_scan`, etc.) packets sent to it and for the data packets that the module receives over the Wi-Fi network. This API has to be called after each command API and also regularly to check if any data is pending from the module (which it has received over the network).

### **2.2.10 [rsi\\_spi\\_socket\\_close.c](#)**

This file contains the API to close a socket.

API Prototype:

```
int16 rsi_socket_close(  
    uint16 socketDescriptor  
)
```

Parameters:

`uint16 socketDescriptor` – Socket number to close. The socket descriptor is returned by the module at the time of socket creation.

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to close an already open socket.

### 2.2.11 [rsi\\_spi\\_disconnect.c](#)

This file contains the API for the Disconnect command.

API Prototype:

```
int16 rsi_disconnect(void)
```

Parameters:

None

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This function is used to disconnect the module's Wi-Fi connection.

### 2.2.12 [rsi\\_spi\\_power\\_mode.c](#)

This file contains the API for setting the power mode of the Wi-Fi module. It has a list of functions used for power save implementation.

#### 1. Set Power Mode

API Prototype:

```
int16 rsi_power_mode(uint8 powerMode)
```

Parameters:

powerMode – powersave mode value

- 0 - No powersave
- 1 - Powermode1
- 2 - powermode2

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to set different power save modes of the module. Please refer to the Software Programming Reference Manual for more information on these modes. This API should be called only after `rsi_init` API.

#### 2. Hold Power Save

API Prototype:

```
int16 rsi_pwrsave_hold(void)
```

Parameters:

None

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to temporarily hold the module from going into sleep mode, after it wakes up at DTIM, in Power Mode 1 (please refer to the Software Programming Reference Manual for more information on Power Mode 1) – this is done when the application has to send packets or commands to the module. After sending the packets/commands the application should call the `rsi_pwrsave_continue` API to move module back to full power save mode. This function is useful only for Power Mode 1.

3. Continue Power Save

API Prototype:

```
int16 rsi_pwrsave_continue(void)
```

Parameters:

None

Returns:

- 0 on success
- 1 on timeout
- 2 on spi interface level failure.

Description:

This API is used to move the module back to full power save mode after the data/command packets are transmitted by the application, which follows the call to the `rsi_pwrsave_hold` API. This API may be used only in Power Mode 1.

NOTE: Please refer to [Section 4.2](#) for more information on how to use the Power Mode APIs.

**2.2.13 [rsi\\_spi\\_interrupt\\_handler.c](#)**

This file contains the API for handling the interrupt from the Wi-Fi module. The interrupt signal has to be registered as an external interrupt for the MCU. The interrupt signal from the module is an active high signal.

API Prototype:

```
void rsi_intHandler(void)
```

Parameters:

None

Returns:

None

Description:

When the MCU is configured for Interrupt mode. this API should be called in the Interrupt Service Routine service the interrupt from the Wi-Fi module. The interrupt signal from the Wi-Fi module is an active-high level-sensitive interrupt. So it is recommended that the interrupt be disabled first, then this API be called and then the interrupt be enabled.

If polling mode is used instead of interrupts, the application should call this API explicitly to update the events from the module. Please note that if the polling mode is used, bit '3' of the register SPI\_HOST\_INTR should be polled. Please refer to the programming reference manual for more details on the register.

This API reads the content of interrupt status register and updates the events accordingly.

#### **2.2.14 [rsi\\_spi\\_query\\_conn\\_status.c](#)**

This file contains the API for querying the Wi-Fi connection status.

API Prototype:

```
int16 rsi_query_conn_status(void)
```

Parameters:

None

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to query the connection status of the Wi-Fi module.

#### **2.2.15 [rsi\\_spi\\_query\\_dhcp\\_params.c](#)**

This file contains the API for querying DHCP parameters.

API Prototype:

```
int16 rsi_query_dhcp_params(void)
```

Parameters:

None

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to query the DHCP parameters of the Wi-Fi module.

**2.2.16** [rsi\\_spi\\_query\\_fwversion.c](#)

This file contains the API for querying the firmware version of the Wi-Fi module.

API Prototype:

```
int16 rsi_query_fwversion(void)
```

Parameters:

None

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to query the firmware version of the Wi-Fi module.

**2.2.17** [rsi\\_spi\\_query\\_macaddress.c](#)

This file contains the API for querying the MAC address of the module.

API Prototype:

```
int16 rsi_query_macaddress(void)
```

Parameters:

None

Returns:

- 0 on success
- 1 on Time out
- 2 on SPI interface level failure

Description:

This API is used to query the MAC address of the Wi-Fi module.



### 2.2.18 [rsi\\_spi\\_query\\_net\\_parms.c](#)

This file contains the API for querying the network parameters of the Wi-Fi module.

API Prototype:

```
int16 rsi_query_net_parms(void)
```

Parameters:

None

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This function is used to query the network parameters of the Wi-Fi module.

### 2.2.19 [rsi\\_spi\\_query\\_rssi.c](#)

This file contains the API for querying the RSSI of the Access Point to which the Wi-Fi module is connected.

API Prototype:

```
int16 rsi_query_rssi(void)
```

Parameters:

None

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to query the RSSI value of the Access Point to which the Wi-Fi module is connected. It should be called after successfully connecting to an Access Point using the `rsi_join` API.

### 2.2.20 [rsi\\_spi\\_fwupgrade.c](#)

This file contains the API to upgrade the firmware of the Wi-Fi module.

API Prototype:

```
int16 rsi_fwupgrade(void)
```

Parameters:

None

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to upgrade the firmware in the Wi-Fi module. It is enabled only if the `RSI_FIRMWARE_UPGRADE` macro is set to '1' in `rsi_global.h`. This function should be called immediately after spi interface initialization. After successful firmware up gradation application need to reset the Wi-Fi module for normal operation.

Note: Inside this file, the user should replace "Firmware/iuinst1", "Firmware/iuinst2", "Firmware/iudata", "Firmware/ffinst1", "Firmware/ffinst2", "Firmware/ffdata" with proper paths depending on the user's setup, so that these files can be referenced properly.

#### **2.2.21 [rsi\\_spi\\_set\\_listen\\_interval.c](#)**

This file contains the API to set the listen interval for the Wi-Fi module.

API Prototype:

```
int16 rsi_set_listen_interval(uint8 *listeninterval)
```

Parameters:

uint8 \*listeninterval- Pointer to listen interval (pointer to 2byte array).

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to set the listen interval for the module. It should be called before the `rsi_join` API.

#### **2.2.22 [rsi\\_spi\\_set\\_mac\\_addr.c](#)**

This file contains the API to set the MAC address of the Wi-Fi module, overriding the MAC address stored in the module's non-volatile memory.

API Prototype:

```
int16 rsi_set_mac_addr(uint8 *macAddress)
```

Parameters:

uint8 \*macAddress- Pointer to mac address (pointer to 6byte array).

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to override the MAC address provided by the module. It should be called before the `rsi_join` API.

### **2.2.23 [rsi\\_spi\\_query\\_dns.c](#)**

This file contain the API to query DNS for given domain name.

API Prototype:

```
int16 rsi_query_dns(rsi_uDns *uDnsFrame)
```

Parameters:

rsi\_uDns \*uDnsFrame – Pointer to DNS query frame.

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to query the ip addresses for given the Domain name.

### **2.2.24 [rsi\\_spi\\_query\\_bssid\\_nwtype.c](#)**

This file contain the API to query scanned access point information along with BSSID and network type.

API Prototype:

```
int16 rsi_query_bssid_nwtype(void)
```

Parameters:

None.

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to query scanned access point information along with BSSID and network type. This API can be called only after successful scanning.

#### 2.2.25 [rsi\\_api\\_sysinit.c](#)

This file contains the APIs for module initialization.

##### 1. System Initialization

###### API Prototype:

```
int16 rsi_sys_init(void)
```

###### Parameters:

None

###### Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

###### Description:

This API is used to initialize the module and its SPI interface.

##### 2. Module's Power Off/On

###### API Prototype:

```
int16 rsi_module_power_cycle(void)
```

###### Parameters:

None

###### Returns:

- 0 on success
- 1 on Failure

###### Description:

This API is used to power cycle the module. This API is valid only if there is a power gate, external to the module, which is controlling the power to the module using a GPIO signal of the MCU.

#### 2.2.26 [rsi\\_interrupt.c](#)

This file contains the APIs to retrieve the status of events from the module. It is the responsibility of the application to monitor these events regularly and handle them accordingly.

##### 1. Check Packet Interrupt Status

###### API Prototype:

```
uint8 rsi_checkPktIrq(void)
```

Parameters:

None

Returns:

- 1 – if there is a packet event pending to be addressed
- 0 – if there is no packet event pending to be addressed

Description:

This API is used to read the status of the data/command packet pending interrupt event. It is the responsibility of the Application to monitor data pending event regularly by calling this function, If any packet is pending application to call to `rsi_spi_read_packet` API to retrieve the pending packet from module and handle accordingly.

2. Clear Data Interrupt

API Prototype:

```
void rsi_clearPktIrq(void)
```

Parameters:

None

Returns:

None

Description:

This API is used to clear data packet/command pending interrupt event. It should be called by the application after servicing the event when it is detected using the `rsi_checkPktIrq` API.

3. Check Buffer Full Status

API Prototype:

```
uint8 rsi_checkBufferFullIrq(void)
```

Parameters:

None

Returns:

- 1 – if the buffer of the Wi-Fi module is full
- 0 – if the buffer of the Wi-Fi module is not full

Description:

This API is used to read the status of the Buffer Full event. If the buffer full event is set then the application should not send any packet/command to the module until it is cleared.

4. Check All Interrupts

API Prototype:

```
uint8 rsi_checkIrqStatus(void)
```

Parameters:

None

Returns:

Pending events' bitmap.

Bit 0 represent's buffer full event

0- indicate buffers in the module are not full.

1- Indicate buffers in the module are full.

Bit 1 represent's buffer empty event

0- indicate buffers in the module are not empty

1- indicate buffers in the module are empty.

Bit 3 represent's data pending event

0- indicate no data/command response pending from module.

1- Indicate data/command response pending from module.

Bit 5 represent power save event in power save mode 1.

0- indicate module in sleep.

1- Indicate module in wakeup state, waiting for data/ack from application.

Description:

This API is used to read the bitmap of all the pending events (value of the interrupt status register) from the Wi-Fi module. This application can use this API to monitor all the events at a time.

## 5. Check Power Mode Interrupt Status

API Prototype:

```
uint8 rsi_checkPowerModeIrq(void)
```

Parameters:

None

Returns:

1 – if there is a power mode event pending to be addressed

0 – if there is no power mode event pending to be addressed

Description:

This API is used to read the status of the power save interrupt event. It is used only in Power Mode 1 (please refer to the Software Programming Reference Manual for more information on Power Modes). When this event is raised then only application can send command/data to module in power save mode1.

### 2.2.27 [rsi\\_spi\\_feat\\_select.c](#)

This file contains the API to enable particular features based on the bitmap value passed to the function.

API Prototype:

```
int16 rsi_spi_feat_sel(uint8 bitmap)
```

Parameters:

uint8 bitmap –Bitmap value to be sent.

Returns:

- 0 on success
- 1 on Time out
- 2 on spi interface level failure

Description:

This API is used to enable some features for example 63 byte PSK WPA/WPA2 security can be enabled by passing a bitmap value of 8.

For more details on this, Please look into software PRM. This API should be called immediately after rsi\_band() API called.

NOTE: The API Library contains other files like `rsi_spi_framerdwr.c`, `rsi_spi_regrdwr.c`, `rsi_spi_memrdwr.c`, `rsi_lib_util.c` which are for the library's internal usage. The user may ignore these files and their functionality.

## 2.3 Hardware Abstraction Layer (HAL) Files

The HAL files included in the API Library have placeholders for HAL APIs which need to be provided by the MCU's BSP. These can be filled with the MCU's HAL APIs directly or some more code might be needed to be written as wrappers if the MCU's HAL APIs are not directly compatible with them.

The HAL files are listed below.

1. [rsi\\_hal.h](#) – This is the header file for the HAL layer.
2. [rsi\\_hal\\_mcu\\_interrupt.c](#) – This file contains the list of functions for configuring the microcontroller interrupts. Following are list of API's which need to be defined in this file.
  - a. Intialize the Interrupts

API Prototype:

```
void rsi_spiIrqStart(void)
```

Parameters:

None

Returns:

None

Description:

This HAL API should contain the code to initialize the register related to interrupts.

b. Enable the Interrupts

API Prototype:

```
void rsi_spiIrqEnable(void)
```

Parameters:

None

Returns:

None

Description:

This HAL API should contain the code to enable interrupts.

c. Disable the Interrupts

API Prototype:

```
void rsi_spiIrqDisable(void)
```

Parameters:

None

Returns:

None

Description:

This HAL API should contain the code to disable interrupts.

d. Clear Pending Interrupts

API Prototype:

```
void rsi_spiIrqClearPending(void)
```

Parameters:

None

Returns:

None

Description:

This HAL API should contain the code to clear the handled interrupts.



3. [rsi\\_hal\\_mcu\\_timers.c](#) – The file contains the functions for implementing timers and delays.

a. Millisecond delay

API Prototype:

```
void rsi_delayMs (uint16 delay)
```

Parameters:

uint16 delay- Number of milliseconds

Returns:

None

Description:

This HAL API should contain the code to introduce a delay in milliseconds.

b. Microseconds delay

API Prototype:

```
void rsi_delayUs (uint16 delay)
```

Parameters:

uint16 delay- Number of microseconds

Returns:

None

Description:

This HAL API should contain the code to introduce a delay in microseconds.

4. [rsi\\_hal\\_mcu\\_spi.c](#) – This file contains the functions needed to transact data between the MCU and the Wi-Fi module, through the SPI interface.

a. Sending data through SPI interface

API Prototype:

```
int16 rsi_spiSend(  
    uint8 *ptrBuf,  
    uint16 bufLen,  
    uint8 *valBuf,  
    uint8 mode)
```

Parameters:

uint8 \*ptrBuf – Pointer to the buffer containing the data to be sent through SPI interface.

uint16 bufLen – Length of the data to be sent through SPI interface.

uint8 \*valBuf – Pointer to a four byte buffer to hold first two bytes of data received from the module while sending data through SPI interface.

uint8 mode – Specifies the mode (8-bit/32-bit mode) of SPI transfers.

0 for 8-bit mode

1 for 32-bit mode

Returns:

0 on success

-1 on Failure

Description:

This API is used to send data to the Wi-Fi module through the SPI interface.

b. Receive data through SPI interface

API Prototype:

```
int16 rsi_spiRecv(  
    uint8 *ptrBuf,  
    uint16 bufLen,  
    uint8 mode)
```

Parameters:

uint8 \*ptrBuf – Pointer the buffer to hold the received data from module through SPI interface.

uint16 bufLen – Number of bytes to read from the module.

uint8 mode – Specifies the mode (8-bit/32-bit mode) of SPI transfers.

0 for 8-bit mode

1 for 32-bit mode

Returns:

0 on success

-1 on Failure

Description:

This API is used to receive data from Wi-Fi module through the SPI interface.

NOTE: The [rsi\\_hal\\_mcu\\_spi.c](#) file contains calls to macros for sending and receiving 8-bit and 32-bit data. These macros have been named as RSI\_SPI\_SEND\_BYTE, RSI\_SPI\_SEND\_4BYTE, RSI\_SPI\_READ\_BYTE and RSI\_SPI\_READ\_4BYTE. The calls to these macros depend on the value assigned to the mode parameter in the `rsi_spiSend` and `rsi_spiRecv` APIs.

The user has two options on the usage of the `rsi_spiSend` and `rsi_spiRecv` APIs:

- 1) Use the code inside these APIs and define the 4 macros in the `rsi_hal.h` file, according to the APIs provided by the MCU's BSP. The user has to ensure that the byte order is not changed for 32-bit read/write macros because of endianness. The transfers should always be little endian. For example, if `RSI_SPI_SEND_4BYTE(0x01020304)` is called, then 0x01 is transmitted first followed by 0x02, 0x03 and 0x04 in that order.
- 2) Rewrite the code inside these APIs according to the APIs provided by the MCU's BSP. In this case, the user has to take care that he follows the process exactly as shown in the pseudo code in `rsi_hal_mcu_spi.c`.

5. [rsi\\_hal\\_mcu\\_ioports.c](#) – This file contains API to control different pins of the microcontroller which interface with the module and other components related to the module.

NOTE: `rsi_modulePower()` function may not be applicable to all platforms. It corresponds to a platform that can turn off/on power to the module through a pin driven by the microcontroller.

a. Reset Wi-Fi module

API Prototype:

```
void rsi_moduleReset(uint8 tf)
```

Parameters:

uint8 tf- To set or clear reset pin of the Wi-Fi module

Returns:

None

Description:

This HAL API is used to set or clear the active-low reset pin of the Wi-Fi module.

b. Power Wi-Fi module

API Prototype:

```
void rsi_modulePower(uint8 tf);
```

Parameters:

uint8 tf- To on or off power to Wi-Fi module

Returns:

None

Description:

This HAL API is used to turn on or off the power to the Wi-Fi module.

## 3 Applications

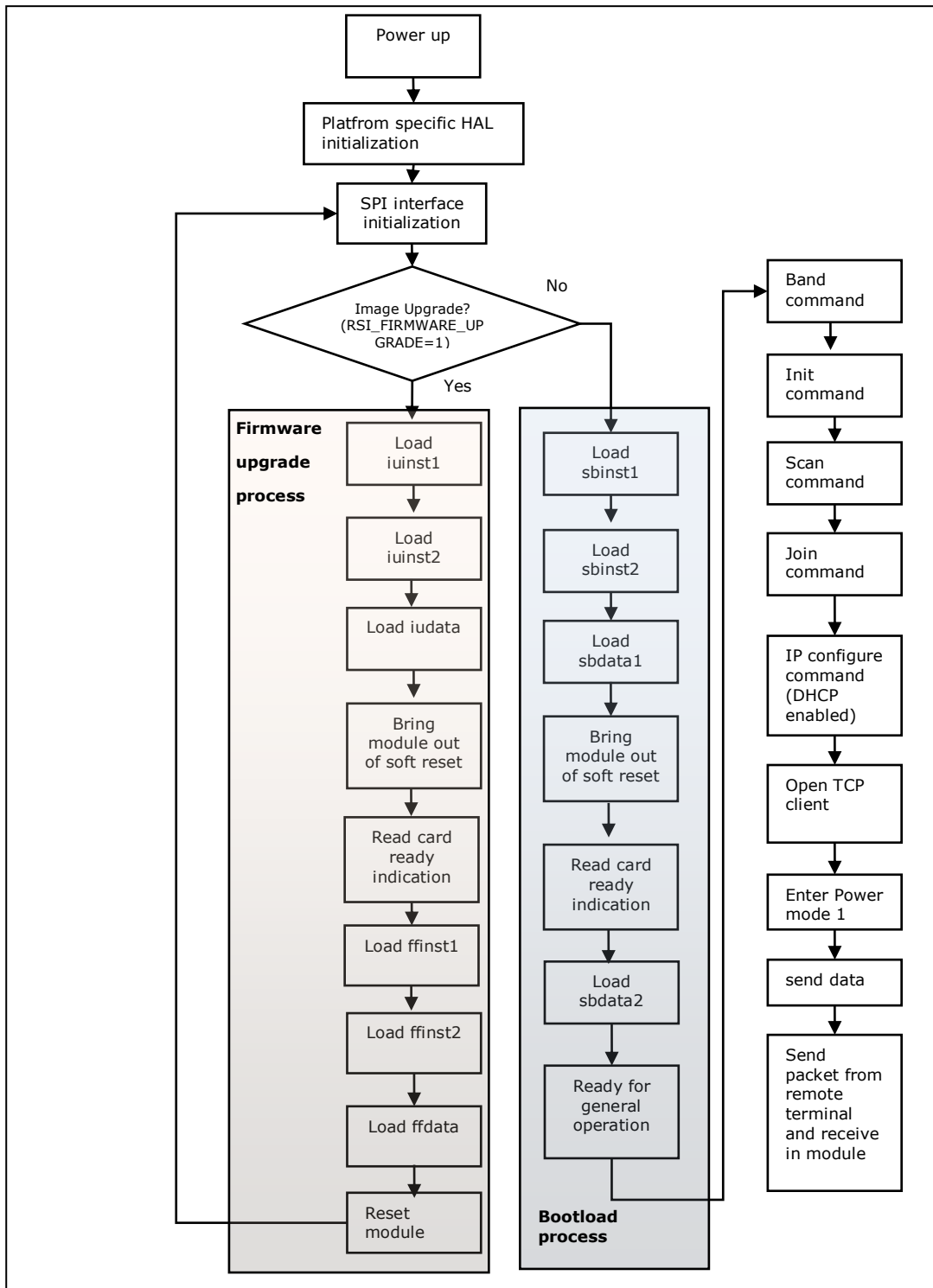
The files in the Applications folders are of two types. The MCU folder contains files for the application layer of the MCU. These have to be modified to setup the application for the system which the user wants to realize. The user has to call the APIs provided in the API library to setup the wireless connection and exchange data over the network.

The PC folder contains files to verify the application written on the MCU. They include applications for starting TCP and UDP server and client port on Windows XP SP2 and Linux PCs.

### 3.1 MCU Applications

These files are the top level application files used to configure the Wi-Fi modules.

1. [main.c](#) – This file is a sample application to demonstrate the general flow and some important commands. It is for illustrative purposes only and the developer should develop his applications corresponding to his requirements. The file contains the entry point for the application. It also has the initialization of parameters of the global structure and the operations to control & configure the module, like scanning networks, joining to an Access Point etc.



**Figure 1: Application Command Sequence**

2. [rsi\\_app\\_util.h](#) and [rsi\\_app\\_util.c](#) – These files contain list of utility functions which are used by `rsi_config_init` API and debug prints.
3. [rsi\\_config.h](#) and [rsi\\_config\\_init.c](#) – These files contain all the parameters to configure the module. Some example parameters are SSID of the Access Point to which the module should connect, IP address to be configured in the module, etc.

To facilitate Application development we have defined a data structure named [rsi\\_api](#) as described below. This structure is initialized by the application using `rsi_init_struct` API of the `rsi_config_init.c` file (application layer file) and then use this data structure to pass the parameter to the library API calls. The user may change the values assigned to the macros without worrying about understanding the contents of the structure.

The contents of this structure are explained in brief below, using the declaration of the structure in [rsi\\_global.h](#) file (which is also an application layer file).

```
typedef struct {  
    uint8 band;  
    uint8 powerMode[2];  
    uint8 macAddress[6];  
    rsi_uScan uScanFrame;  
    rsi_uJoin uJoinFrame;  
    rsi_uIpparam uIpparamFrame;  
    rsi_uDns uDnsFrame;  
    uint8 listeninterval[2];  
} rsi\_api;
```

4. [rsi\\_global.h](#) – Following are list of macro's needed to be defined in `rsi_global.h` based on application requirement.
  - a. `#define RSI_POLLED` – To select poll mode.
  - b. `#define RSI_INTERRUPTS` – To select interrupt mode.
  - c. `#define RSI_HWTIMER` – To use hardware timer for time out implementation. If this macro is defined then the `rsi_spiTimer1`, `rsi_spiTimer2`, `rsi_spiTimer3` variables should be incremented in timer ISR.
  - d. `#define RSI_TICKS_PER_SECOND 10000` – To set number of ticks per second (it is platform dependent).

- e. `# define RSI_DEBUG_PRINT` To enable debug prints.
- f. `#define DEBUG_LVL PL3|PL2` – To enable debug prints at different levels.

Note: The MCU's BSP needs to provide the print API.

- g. `#define RSI_MAX_PAYLOAD_SIZE` – To configure maximum packet size.
- h. `#define RSI_LITTLE_ENDIAN` – define this macro for little endian MCU's, If not defined by default big endian is enabled.
- i. `#define BIT_32_SUPPORT` - define this macro if MCU can support 8bit/32bit modes simultaneously ( where spi command sent in 8bit mode and payload sent in 32bit mode).
- j. `#define RSI_FIRMWARE_UPGRADE` - define with 0 to disable firmware upgrade, define with 1 to enable firmware upgrade.
- k. `#define RSI_AP_SCANNED_MAX 15` – Maximum number of access points can scan, please refer PRM for maximum number access point module can return .
- l. `#define RSI_MAX_SOCKETS 8` –Maximum number of sockets module support, refer PRM for more information.
- m. `#define RSI_PSK_LEN 32` – PSK length for more information refer PRM.
- n. `#define RSI_SSID_LEN 32` –SSID length module supports for more information refer PRM .
- o. `#define RSI_63BYTE_PSK_SUPPORT` – define with 1 to enable 63 byte PSK for WPA/WPA2-PSK security mode. Otherwise comment out or set to 0.
- p. `#define RSI_MODULE_23_24` – define with '1' if the module part number is RS9110-N-11-24-(xx). Otherwise define to '0'

Following are list of macro's need to define in `rsi_config.h` based application requirement. This Macro's used by `rsi_init_struct` function to initialize `rsi_api` data structure.

- a. `#define RSI_MODULE_IP_ADDRESS` To configure IP address to module.
- b. `#define RSI_GATEWAY` To configure gateway IP address to module.



- c. `#define RSI_TARGET_IP_ADDRESS` To configure target IP address.
- d. `#define RSI_NETMASK` To configure network mask to the module.
- e. `#define RSI_SECURITY_TYPE` To configure `RSI_SECURITY_OPEN` or `RSI_SECURITY_WPA1` or `RSI_SECURITY_WPA2` or `RSI_SECURITY_WEP` security type.
- f. `#define RSI_PSK` To configure PSK, if security mode is enabled.
- g. `#define RSI_SCAN_SSID` To scan only particular access point configure this macro.
- h. `#define RSI_SCAN_CHANNEL` To scan only particular channel configure this macro, if 0 module will scan all the channels.
- i. `#define RSI_JOIN_SSID` To configure SSID to join.
- j. `#define RSI_MAC_ADDRESS` To configure MAC address to the module.
- k. `#define RSI_DOMAIN_NAME` To configure server name to check DNS query.
- l. `#define RSI_DNS_SERVER` To configure DNS server if used manual IP configuration mode.
- m. `#define RSI_IP_CFG_MODE` To enable or disable DHCP while IP configuration (`RSI_DHCP_MODE_DISABLE` or `RSI_DHCP_MODE_ENABLE` or `RSI_AUTO_IP_CFG` ).
- n. `#define RSI_NETWORK_TYPE` To select network type (`RSI_INFRASTRUCTURE_MODE` or `RSI_IBSS_OPEN_MODE` or `RSI_IBSS_SEC_MODE`)
- o. `#define RSI_IBSS_MODE` To select IBSS mode.
- p. `#define RSI_IBSS_CHANNEL` To select IBSS channel.
- q. `#define RSI_BAND` To select BAND (`RSI_BAND_2P5GHZ` or `RSI_BAND_5GHZ` ).
- r. `#define RSI_DATA_RATE` To select data rate auto or fixed data rate (`RSI_DATA_RATE_AUTO` or `RSI_DATA_RATE_(1, 2, 5P5, 11, 6, 9, 12)`).
- s. `#define RSI_POWER_LEVEL` To select power level (`RSI_POWER_LEVEL_LOW` or `RSI_POWER_LEVEL_MEDIUM` or `RSI_POWER_LEVEL_HIGH`)
- t. `#define RSI_POWER_MODE` To select power save mode (0-powermode1,2-powemode2,1-powermode1)

- u. `#define RSI_LISTEN_INTERVAL` To set the listen interval for the Wi-Fi module in Powersave.

Note: The above way of configuring global structure is optional. If the user wants to use global `rsi_api` and initialize using `rsi_init_struct`, then only define the above MACROS in `rsi_config.h`

## 3.2 PC Applications (Remote Applications)

These applications are run on the remote PC/Laptop to receive data sent from the module, or to send data to the module.

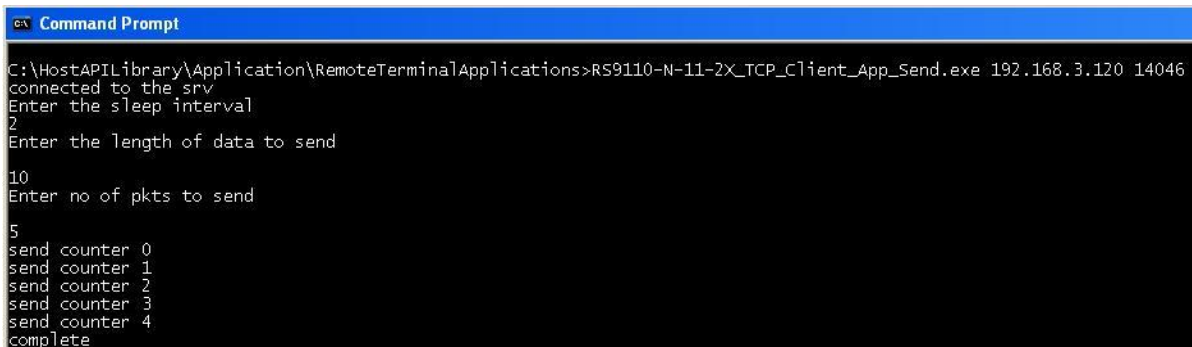
They can be used to verify the application written by the user on the MCU. The following sections explain the steps to be followed to use them. They are to be executed once the Wi-Fi module is connected to an Access Point and has been assigned an IP address. If the TCP/UDP client applications are to be used on the PC, then the MCU has to program the Wi-Fi module to open a server socket before these applications are executed.

The applications are provided for Windows XP SP2 and Linux environments, but the sections below discuss only the Windows variants. Similar steps can be followed for the Linux variants.

### 3.2.1 Steps to Transmit TCP Data to the Wi-Fi Module

Execute the `Applications\PC\Windows\RS9110-N-11-2X-TCP_Client_App_Send.exe` application in the Command Prompt window in Windows, to open a TCP client port on the PC and transmit TCP data to the Wi-Fi module. The sample parameters entered in the example below are 192.168.3.120 (IP address of the Wi-Fi module) and 14046 (port number assigned to Wi-Fi module's TCP server socket). The user can display the data received by the module by using debug prints in his specific platform.

NOTE: The application should be executed after the MCU successfully opens the TCP server socket in the Wi-Fi module.

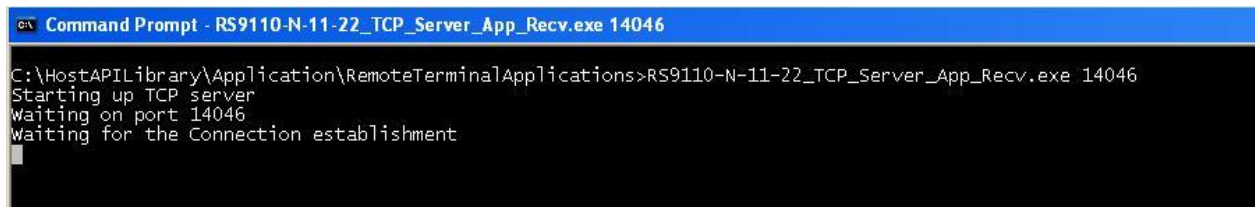


```
Command Prompt
C:\HostAPILibrary\Application\RemoteTerminalApplications>RS9110-N-11-2X-TCP_Client_App_Send.exe 192.168.3.120 14046
connected to the srv
Enter the sleep interval
2
Enter the length of data to send
10
Enter no of pkts to send
5
send counter 0
send counter 1
send counter 2
send counter 3
send counter 4
complete
```

**Figure 2: Transmit TCP Data to the Wi-Fi Module**

### 3.2.1 Steps to Receive TCP data from the Wi-Fi Module

Execute the Applications\PC\Windows\RS9110-N-11-2X-TCP\_Server\_App\_Recv.exe application in the Command Prompt window in Windows, to open a TCP server port on the PC and receive TCP data from the Wi-Fi module. The parameter entered in the example below is 14046 (port number assigned to Wi-Fi module's TCP client socket).

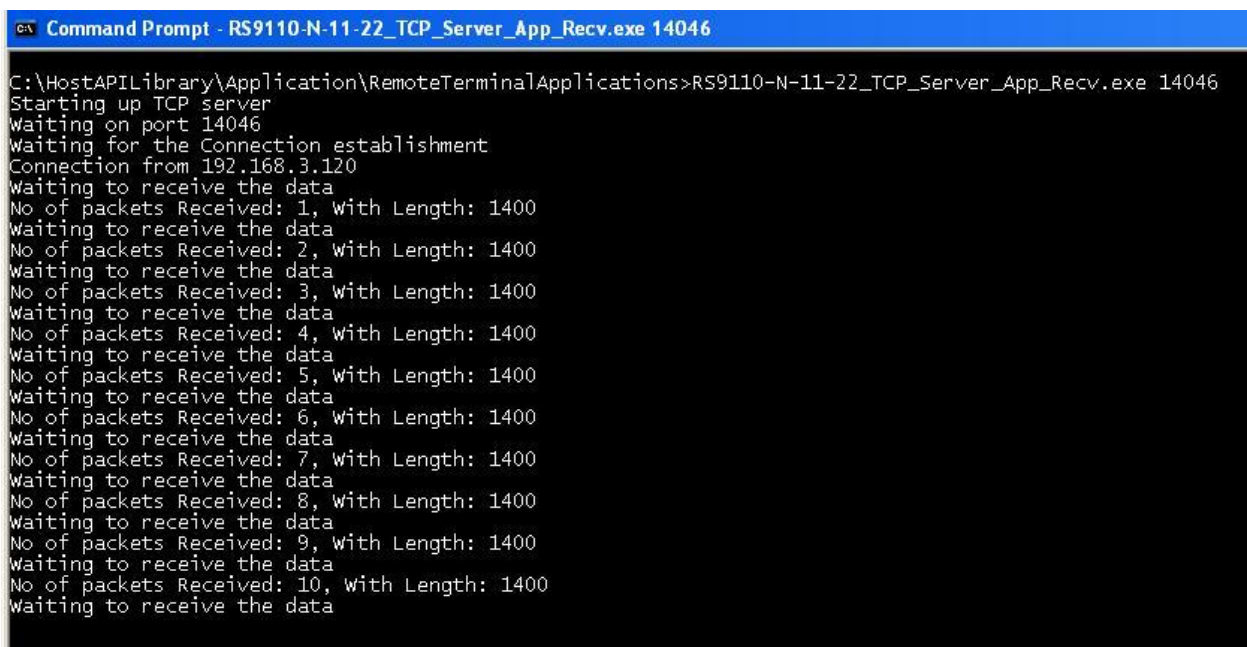


```
Command Prompt - RS9110-N-11-22_TCP_Server_App_Recv.exe 14046

C:\HostAPILibrary\Application\RemoteTerminalApplications>RS9110-N-11-22_TCP_Server_App_Recv.exe 14046
Starting up TCP server
Waiting on port 14046
Waiting for the Connection establishment
```

**Figure 3: Receive TCP Data from the Wi-Fi Module – 1**

NOTE: The Server application on the PC has to be executed before the Wi-Fi module is configured by the MCU to connect to the TCP server by opening a TCP client socket. Once the TCP connection is successful, the Server application on the PC waits for data to be received from the MCU+Wi-Fi Module, as shown in the image below.



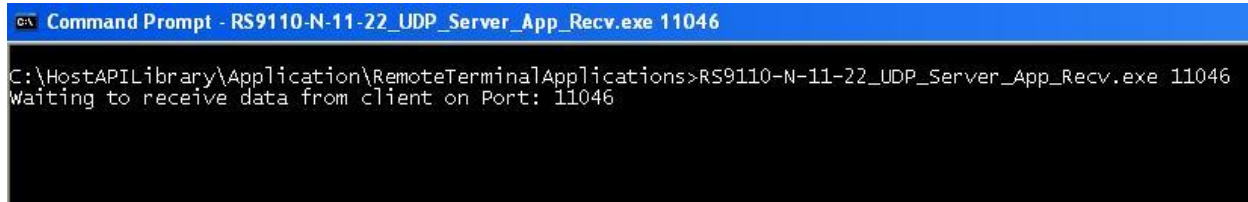
```
Command Prompt - RS9110-N-11-22_TCP_Server_App_Recv.exe 14046

C:\HostAPILibrary\Application\RemoteTerminalApplications>RS9110-N-11-22_TCP_Server_App_Recv.exe 14046
Starting up TCP server
Waiting on port 14046
Waiting for the Connection establishment
Connection from 192.168.3.120
Waiting to receive the data
No. of packets Received: 1, With Length: 1400
Waiting to receive the data
No. of packets Received: 2, With Length: 1400
Waiting to receive the data
No. of packets Received: 3, With Length: 1400
Waiting to receive the data
No. of packets Received: 4, With Length: 1400
Waiting to receive the data
No. of packets Received: 5, With Length: 1400
Waiting to receive the data
No. of packets Received: 6, With Length: 1400
Waiting to receive the data
No. of packets Received: 7, With Length: 1400
Waiting to receive the data
No. of packets Received: 8, With Length: 1400
Waiting to receive the data
No. of packets Received: 9, With Length: 1400
Waiting to receive the data
No. of packets Received: 10, With Length: 1400
Waiting to receive the data
```

**Figure 4: Receive TCP Data from the Wi-Fi Module – 2**

### 3.2.2 Steps to Receive UDP Data from the Wi-Fi Module

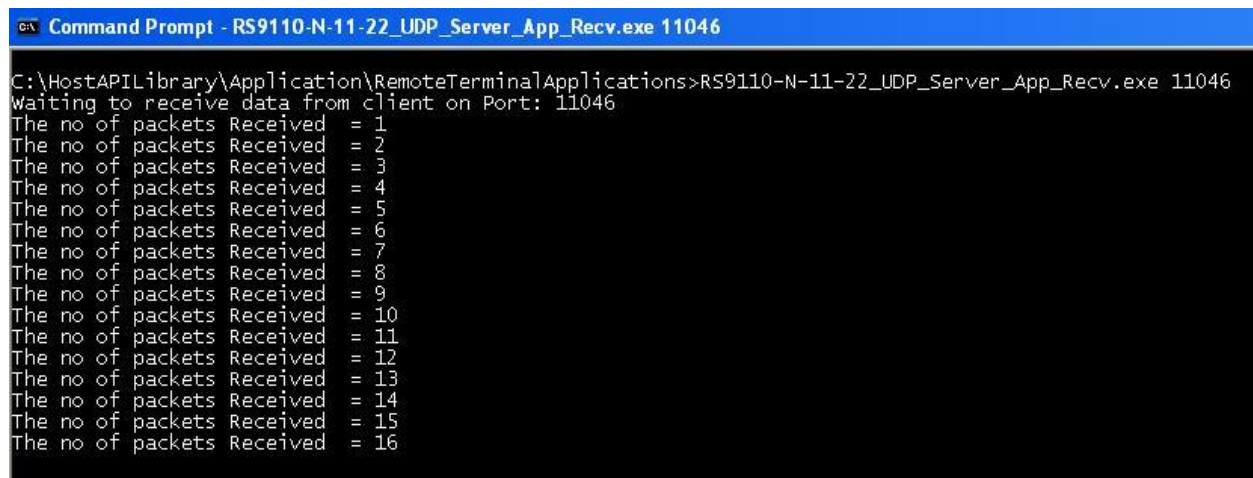
Execute the Applications\PC\Windows\RS9110-N-11-2X-UDP\_App\_Recv.exe application in the Command Prompt window in Windows, to open a UDP port on the PC and receive UDP data from the Wi-Fi module. The parameter entered in the example below is 11046 (port number assigned to Wi-Fi module's UDP socket).



```
C:\ Command Prompt - RS9110-N-11-22_UDP_Server_App_Recv.exe 11046
C:\HostAPILibrary\Application\RemoteTerminalApplications>RS9110-N-11-22_UDP_Server_App_Recv.exe 11046
Waiting to receive data from client on Port: 11046
```

**Figure 5: Receive UDP Data from the Wi-Fi Module – 1**

NOTE: The application on the PC has to be executed before the Wi-Fi module is configured by the MCU to connect to the UDP application by opening a UDP client socket. Once the UDP connection is successful, the Server application on the PC waits for data to be received from the MCU+Wi-Fi Module, as shown in the image below.



```
C:\ Command Prompt - RS9110-N-11-22_UDP_Server_App_Recv.exe 11046
C:\HostAPILibrary\Application\RemoteTerminalApplications>RS9110-N-11-22_UDP_Server_App_Recv.exe 11046
Waiting to receive data from client on Port: 11046
The no of packets Received = 1
The no of packets Received = 2
The no of packets Received = 3
The no of packets Received = 4
The no of packets Received = 5
The no of packets Received = 6
The no of packets Received = 7
The no of packets Received = 8
The no of packets Received = 9
The no of packets Received = 10
The no of packets Received = 11
The no of packets Received = 12
The no of packets Received = 13
The no of packets Received = 14
The no of packets Received = 15
The no of packets Received = 16
```

**Figure 6: Receive UDP Data from the Wi-Fi Module – 2**

## 4 Application Notes

### 4.1 Typical Usage of APIs

This application note describes a typical sequence to call the APIs of the library in the application. The application has to first call the API for a command, then wait for a "Data Pending Event" to occur and then service it. The application can perform other tasks during this wait period. Once the data pending event is received, the application has to call the `rsi_spi_read_packet` API to read the response from the module and parse the response and handle it appropriately.

```
retval = rsi_init();
if(retval!=0)
{
    return -1;
}
else
{
    RSI_RESPONSE_TIMEOUT(INITTIMEOUT);
    rsi_spi_read_packet(&uCmdRspFrame);
    rsi_clearPktIrq();
    /*
     * handle the response based on response code
     * and error code
     */
}
```

**Note:** `RSI_RESPONSE_TIMEOUT(TIMEOUT)` is defined in `rsi_global.h`. This macro return from the loop either on timeout or data pending event raise.

### 4.2 Power Mode 1

This application note explains the procedure to send packets in Power Mode 1 using the APIs in the library. The applications needs to first call the `rsi_power_mode` API to program the module for Power Mode 1. Next, if any data has to be transmitted, it has to call the `rsi_pwrsave_hold` API to request the module not to go to sleep mode the next time it wakes up – the module wakes up every DTIM interval. The application has to call the `rsi_send_data` API till it is successful – it returns a success when the module wakes up at the next DTIM. Once all the pending data is sent to the module, the application has to call the `rsi_pwrsave_continue` API to program the module to go into sleep mode till the next DTIM interval.

```
rsi_power_mode(1);
rsi_pwrsave_hold();
retval = 1;
while(retval != 0)
{
    retval = rsi_send_data(sock_no,&send_buffer,
        sizeof(send_buffer), PROTOCOL_TCP);
}
rsi_pwrsave_continue();
```

---

NOTE: Power save mode 1 can be configure by assigning a value of '1' to the `RSI_POWER_MODE` macro in `rsi_config.h` file.

---

## 5 Compiling the Driver Source Code

Makefiles are provided to compile the driver in a generic C compiler.

API\_Lib\Makefile – Compiles all the C-files inside API\_Lib to generate their object files.

MCU\Makefile – Compiles the [MCU Applications](#) (main.c), referencing the API object files.

## 6 Documentation

This folder contains the complete documentation, including this document and also HTML files, for all the C source code files, functions and macros defined in the API Library, including descriptions, parameters, return values, error codes, etc. The [index.html](#) is the starting point for browsing the HTML documentation.

For information on Error Codes and Response Codes for the commands described in this API manual, please refer to the programming reference manual.