PSoC® Creator Component for RS-CY8C001-220X

Component Datasheet

Version 1.1

June 2011

Redpine Signals, Inc.
2107 N. First Street, #680
San Jose, CA 95131.
Tel: (408) 748-3385
Fax: (408) 705-2019
Email: info@redpinesignals.com
Website: www.redpinesignals.com

Disclaimer:

The information in this document pertains to information related to Redpine Signals, Inc. products. This information is provided as a service to our customers, and may be used for information purposes only.
Redpine assumes no liabilities or responsibilities for errors or omissions in this document. This document may be changed at any time at Redpine's sole discretion without any prior notice to anyone. Redpine is not committed to updating this document in the future.
Copyright © 2011 Redpine Signals, Inc. All rights reserved.

About this Document

This document is a PSoC® Creator Component Datasheet for the RS-CY8C001-220X Wi-Fi Exanpansion Board Kit. The RS-CY8C001-220X is design to work with the Cypress PSoC3® and PSoC5® development kits that have an expansion connector – CY8CKIT-001 (PSoC3/5), CY8CKIT-030 (PSoC3) and CY8CKIT-050 (PSoC5).

Table Of Contents

1 Features of RS-CY8C001-220X 9
1.1 Overview 9
1.2 Applications 10
1.3 RS9110-N-11-22 Module Features 10
2 Input/Output Connections 12
2.1 Input/Output Connections for SPI 12
2.2 Input/Output Connections for UART 13
3 Schematic Macro Information 14
4 Parameters and Setup 15
4.1 Hardware vs. Software Options 15
4.2 Basic Tab 15
4.3 Advanced Tab 19
5 Application Programming Interface 23
6 API Library for Wi-Fi over SPI Interface 24
6.1 API data structure 24
6.1.1 Scan input parameter structure: 24
6.1.2 Join input parameter structure: 26
6.1.3 IP configuration input parameter structure: 27
6.1.4 Socket create input parameter structure: 28
6.1.5 DNS query input parameter structure: 29
6.1.6 Response scan information data structure: 30
6.1.7 Scan response information with BSSID and Network type data structure 31
6.1.8 Read Packet data structure (From module): 31
6.2 SPI API Library 39
6.2.1 rsi_spi_bootloader.c 39
6.2.2 rsi_spi_band.c 40
6.2.3 rsi_spi_init.c 40
6.2.4 rsi_spi_scan.c 41
6.2.5 rsi_spi_join.c 41
6.2.6 rsi_spi_ipparam.c 41
6.2.7 rsi_spi_socket.c 42
6.2.8 rsi_spi_send_data.c 42
6.2.9 rsi_spi_read_packet.c 43
6.2.10 rsi_spi_socket_close.c 44
6.2.11 rsi_spi_disconnect.c 44
6.2.12 rsi_spi_power_mode.c 45
6.2.13 rsi_spi_interrupt_handler.c 46
6.2.14 rsi_spi_query_conn_status.c 47
6.2.15 rsi_spi_query_dhcp_params.c 47
6.2.16 rsi_spi_query_fwversion.c 47
6.2.17 rsi_spi_query_net_parms.c 48
6.2.18 rsi_spi_query_rssi.c 48
6.2.19 rsi_spi_fwupgrade.c 49
6.2.20 rsi_spi_set_listen_interval.c 49

6.2.21 rsi_spi_set_mac_addr.c 50
6.2.22 rsi_spi_query_dns.c 50
6.2.23 rsi_spi_query_bssid_nwtype.c 50
6.2.24 rsi_api_sysinit.c 51
6.2.25 rsi_interrupt.c 51
6.3 Hardware Abstraction Layer (HAL) Files 54
7 API Library for Wi-Fi over UART Interface 59
7.1.1 Set Band 59
7.1.2 Init 59
7.1.3 Set number of scan results 59
7.1.4 Passive scan 60
7.1.5 61
7.1.6 61
7.1.7 Scan 61
7.1.8 63
7.1.9 Query number of scan results 63
7.1.10 Scan next WiFi networks 63
7.1.11 Query BSSIDs of scanned WiFi networks 63
7.1.12 Set Network type 64
7.1.13 Set Pre Shared key 64
7.1.14 Set Authentication Mode 64
7.1.15 Join 65
7.1.16 Disassociate 66
7.1.17 Power Modes and commands 67
7.1.17.1 Power mode 0 67
7.1.17.2 Power mode 1 67
7.1.17.3 Power mode 2 69
7.1.17.4 Set power mode 70
7.1.17.5 Set sleep timer 70
7.1.17.6 Send ACK 71
7.1.18 Set IP Parameters 71
7.1.19 Listen UDP 72
7.1.20 Listen TCP 73
7.1.21 Query Listen TCP socket status 74
7.1.22 Open UDP socket 74
7.1.23 Open TCP socket 75
7.1.24 Socket close 75
7.1.25 Send data 75
7.1.26 Query RSSI 76
7.1.27 Query Network parameters 76
7.1.28 Query MAC address of Wi-Fi module 76
7.1.29 Query Network type 76
7.1.30 Query FW version 77
7.1.31 Read data 77
7.1.32 Read command response 82
7.2 HAL Files 83
7.2.1 HAL API Requirements 83
7.2.1.1 HAL Macros/APIs 83
7.2.2 Memory requirements for the HAL layer of MCU 85
7.3 Miscellaneous Files 85

8 DC and AC Electrical Characteristics 86

Table of Figures

Figure 1: PSoC Creator Component with SPI as Communication Interface. 10
Figure 2: Component with UART as Communication Interface 10
Figure 3: Schematic Macro Information with SPI 14
Figure 4: Schematic Macro Information with UART 14
Figure 5: Basic Tab of Configure Dialog... 16
Figure 5: Advanced Tab of Configure Dialog.. 19

Table of Tables

Table 1: Input/Output Connections for SPI **13**
Table 2: Input/Output Connections for UART **13**
Table 3: Configuration Options for Basic Tab **18**
Table 4: Configuration Options for Advanced Tab **22**

1 Features of RS-CY8C001-220X

1.1 Overview

The Wi-Fi Expansion Board Kit (EBK) for the Cypress PSoC® 3/5 platforms has Redpine Signals' Connect-io-n™ module, RS9110-N-11-22, mounted. It is a complete IEEE 802.11bgn based wireless device server that directly provides a wireless interface to any equipment with a serial or SPI interface for data transfer. It integrates a MAC, baseband processor, RF transceiver with power amplifier, a frequency reference, and an antenna1 in hardware; and all WLAN protocol and configuration functionality, networking stack in embedded firmware to make a fully self-contained 802.11n WLAN solution for a variety of applications.

The PSoC Creator Component provided as part of this EBK serves as a drop-and-drop component and offers an extensive set of configurable features like the interface to the PSoC SoC (UART or SPI), SSID of the Access Point, Pre-shared Key of the WLAN, IP Configuration, Infrastructure/IBSS network configuration, etc. The same component is usable for PSoC3 and PSoC5. The component can also be used to interface with and configure the following modules from Redpine Signals' Connect-io-n family:

- RS9110-N-11-22 – Fully integrated, single-band (2.4 GHz) 802.11n module
- RS9110-N-11-24 – Small size, single-band (2.4 GHz) 802.11n module
- RS9110-N-11-26 – Fully integrated, dual-band (2.4 and 5 GHz) 802.11n module
- RS9110-N-11-28 – Small size, dual-band (2.4 and 5 GHz) 802.11n module

The Component also offers a comprehensive set of simple APIs which can be called to configure the Wireless LAN connection and exchange data over the network. It can be configured to use UART or SPI as the interface of communication between the PSoC SoC and the RS9110-N-11-22 and depending on this configuration, the input/output connections of the component vary.

[1] Option for external antenna available using a u.FL connector on the module. Please refer to the module's datasheet for more details.

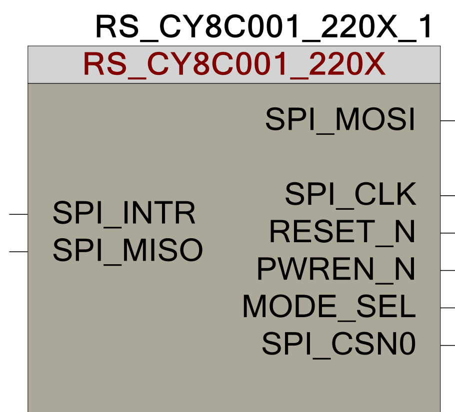


Figure 1: PSoC Creator Component with SPI as Communication Interface

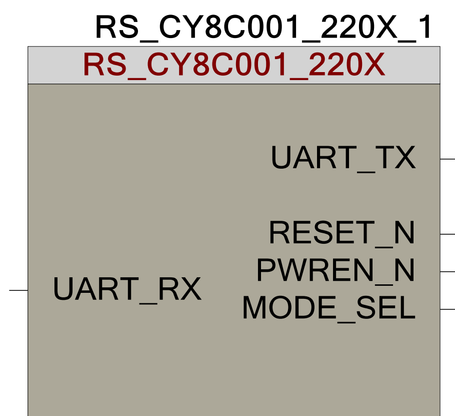


Figure 2: Component with UART as Communication Interface

1.2 Applications

- Lighting Controls
- Metering (Parking Meters, Utility Meters, Power Meters, etc.)
- Industrial M2M communications
- Point of Sale Terminals
- Security Cameras and Surveillance Equipment
- Logistics and Freight Management
- Warehousing
- Digital Picture Frames
- Several medical applications including Patient Monitoring, Remote Diagnostics

1.3 RS9110-N-11-22 Module Features

- Compliant to 802.11b/g and single stream 802.11n
- Fully self-contained serial-to-wireless functionality

- Includes all the protocol and configurations functions for WLAN connectivity in Open and Secure modes of operation
- Payload data through Serial Interface and SPI
- Terminates TCP and UDP connections, and offers transparent serial modem functionality
- Integrated antenna[1], frequency reference and low-frequency clock
- Ultra-low-power operation with power-save modes
- Ad-hoc and infrastructure modes for maximum deployment flexibility
- Single supply – 3.1 to 3.6V operation

2 Input/Output Connections

This sections describes the various input and output connections for the RS-CY8C001-220X Component.

2.1 Input/Output Connections for SPI

The following table lists the input/output connections for the RS-CY8C001-220X component when SPI is selected as the communication interface between the PSoC and the RS9110-N-11-22 module.

S.No.	Connection Name	Direction	Description
1.	MODE_SEL	Output	SPI and UART select signal. '0' – UART is selected. '1' – SPI is selected.
2.	PWREN_N	Output	Active low, power-enable signal. This signal controls the power to the RS9110-N-11-22 module by controlling a power gate on the expansion board. '0' – Enable power to the module. '1' – Disable power to the module.
3.	RESET_N	Ouput	Active low, reset signal. This module resets the RS9110-N-11-22 module. It needs to be active for at least 10ms after power on to properly reset the module. '0' – Active (module is under reset) '1' – Inactive (module is out of reset)
4.	SPI_CLK	Ouput	SPI Clock
5.	SPI_CSN0	Ouput	Active low, SPI Select signal. This is the select signal for the RS9110-N-11-22 module's SPI Slave.
6.	SPI_MOSI	Ouput	SPI Master-Out-Slave-In Data signal.
7.	SPI_MISO	Input	SPI Master-In-Slave-Out Data signal.
8.	SPI_INTR	Input	Active high, level sensitive interrupt to the PSoC. This signal is asserted by the module when: 1. The module has to transmit data to the PSoC through SPI 2. The module wakes up from sleep mode.

			'0' – No interrupt is active '1' – There is an interrupt pending for the PSoC.

Table 1: Input/Output Connections for SPI

2.2 Input/Output Connections for UART

The following table lists the input/output connections for the RS-CY8C001-220X component when UART is selected as the communication interface between the PSoC and the RS9110-N-11-22 module.

S.No.	Connection Name	Direction	Description
1.	MODE_SEL	Output	SPI and UART select signal. '0' – UART is selected. '1' – SPI is selected.
2.	PWREN_N	Output	Active low, power-enable signal. This signal controls the power to the RS9110-N-11-22 module by controlling a power gate on the expansion board. '0' – Enable power to the module. '1' – Disable power to the module.
3.	RESET_N	Ouput	Active low, reset signal. This module resets the RS9110-N-11-22 module. It needs to be active for at least 10ms after power on to properly reset the module. '0' – Active (module is under reset) '1' – Inactive (module is out of reset)
4.	UART_TX	Output	UART Transmit Data signal.
5.	UART_RX	Input	UART Receive Data signal.

Table 2: Input/Output Connections for UART

3 Schematic Macro Information

The default RS-CY8C001-220X Component in the Component Catalog is a schematic macro for the PSoC3 using UART as the communication interface. The component's configuration can be modified to use the same component for PSoC3 with UART or PSoC5 with UART or SPI.

The other default settings of the component are explained in more detail in Section 4. The component is connected to the digital input and output Pins components.

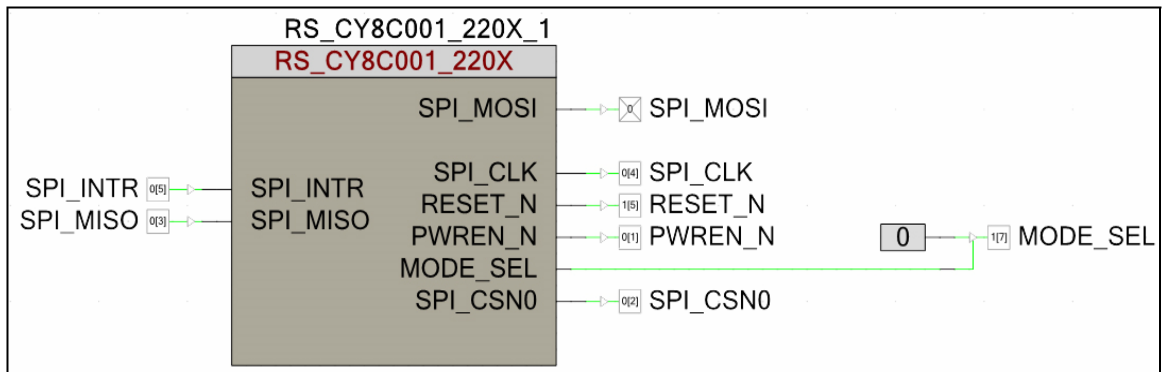


Figure 3: Schematic Macro Information with SPI

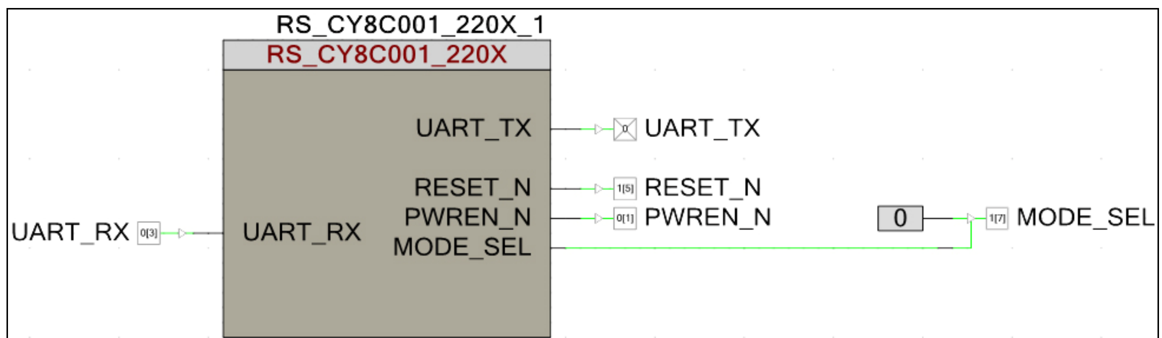


Figure 4: Schematic Macro Information with UART

4 Parameters and Setup

Drag an RS-CY8C001-220X component onto your design and double-click it to open the Configure dialog. The dialog has two tabs – Basic and Advanced – which are explained in the sub-sections below. The configurable parameters are prefixed with "RSI_XY_" (XY = 2-digit number) to maintain a logical order for the parameters and make them easier to read and understand for the user.

4.1 Hardware vs. Software Options

Hardware configuration options change the way the project is synthesized and placed in the hardware. You must rebuild the hardware if you make changes to any of these options. Software configuration options do not affect synthesis or placement. When setting these parameters before build time you are setting their initial value which may be modified at any time with the API provided.

The following sections describe the RS-CY8C001-220X component's parameters, and how they are configured using the dialog.

4.2 Basic Tab

The image below shows the configuration options for the Basic tab.

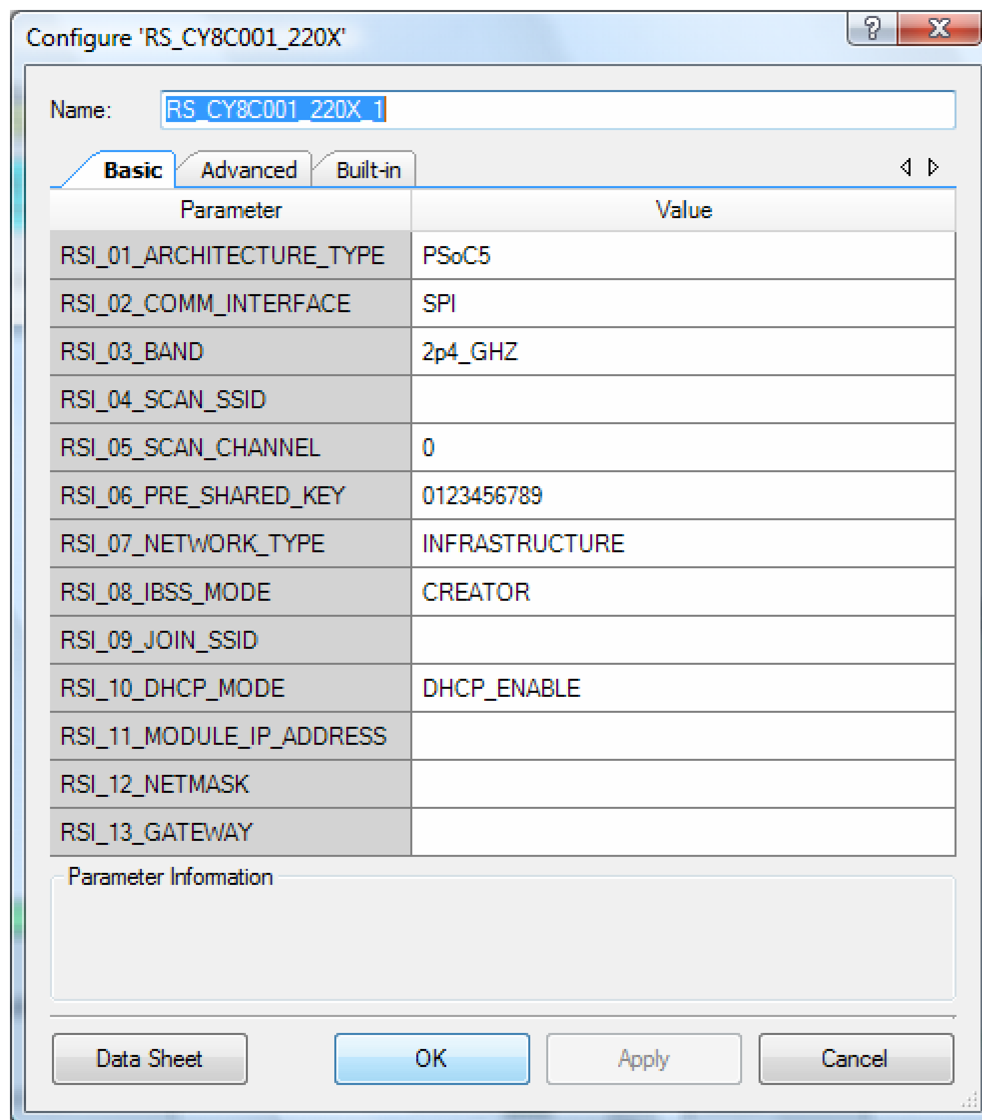


Figure 5: Basic Tab of Configure Dialog

The table below describes each parameter of the Basic tab and the possible options for each of them.

Parameter	Options	Description
RSI_01_ARCHITECTURE_TYPE	PSoC3 PSoC5	This option selects whether the component has to be compiled for PSoC3 or PSoC5.
RSI_02_COMM_INTERFACE	SPI UART	This option selects whether the component has to use UART or SPI to communication with the

Parameter	Options	Description
		RS9110-N-11-22 module.
RSI_03_BAND	2p4_GHZ 5_GHZ	This option selects whether the Wi-Fi module has to operate in the 2.4GHz or 5GHz band. This option is NOT valid for the RS9110-N-11-22 module which is present on the RS-CY8C001-220X EBK since that module operates only in the 2.4GHz band. This option is valid only when the RS9110-N-11-26 and RS9110-N-11-28 modules are used which are dual band modules.
RSI_04_SCAN_SSID	<string of ASCII characters less than 32 characters in length>	This option allows the user to configure the Wi-Fi module to scan for a particular SSID, especially when the SSID is not being broadcasted. This field can be left empty if the module has to be configured to scan all available networks.
RSI_05_SCAN_CHANNEL	0 to 11	This the channel in which the module will scan for Wi-Fi networks. Selecting 0 configures the module to scan in all channels from 1 to 11. Selecting any other number configures the module to scan in that particular channel.
RSI_06_PRE_SHARED_KEY	<string of characters>	This is the pre-shared key or passphrase for connecting to secure networks. The module supports WEP, WPA/WPA2 (AES & TKIP) security modes. For WEP-64bit, the input has to be a 10-digit hexadecimal number, e.g., 098765ABCD. For WEP-128bit, the input has to be a 26-digit hexadecimal number, e.g., 0123456789ABCDEF0123456789. For WPA/WPA2, the input has to be a string of ASCII characters, less than 32 characters in length.

Parameter	Options	Description
RSI_07_NETWORK_TYPE	INFRASTRUCTURE IBSS	This option selects whether the module connects to an Access Point (INFRASTRUCTURE) or, connects to or creates an Adhoc (IBSS) network. If IBSS is selected, the Advanced tab contains more parameters for the channel, security, etc., that need to be configured for the IBSS network.
RSI_08_IBSS_MODE	CREATOR JOINER	This parameter is valid only if "IBSS" is selected for the RSI_07_NETWORK_TYPE parameter. This option decides whether the Wi-Fi module has to create a new IBSS network or join an existing IBSS network.
RSI_09_JOIN_SSID	<string of ASCII characters less than 32 characters in length>	This parameter configures the Wi-Fi module to connect to a Wi-Fi network (Infrastructure or existing IBSS) or create an IBSS network depending on the inputs selected for the other parameters.
RSI_10_DHCP_MODE	DHCP_DISABLE DHCP_ENABLE	This parameter configures the module to use DHCP to acquire an IP address or to use a static IP address. If IBSS is selected for NETWORK_TYPE, then DHCP_MODE has to be set to DHCP_DISABLE.
RSI_11_MODULE_IP_ADDRESS	<4-byte dot-decimal format>	This parameter is the static IP address to be assigned to the module if DHCP is disabled.
RSI_12_NETMASK	<4-byte dot-decimal format>	This parameter is the subnet mask to be assigned to the module if DHCP is disabled.
RSI_13_GATEWAY	<4-byte dot-decimal format>	This parameter is the gateway IP address to be assigned to the module if DHCP is disabled.

Table 3: Configuration Options for Basic Tab

4.3 Advanced Tab

The image below shows a screenshot of some of the configuration options for the Advanced tab.

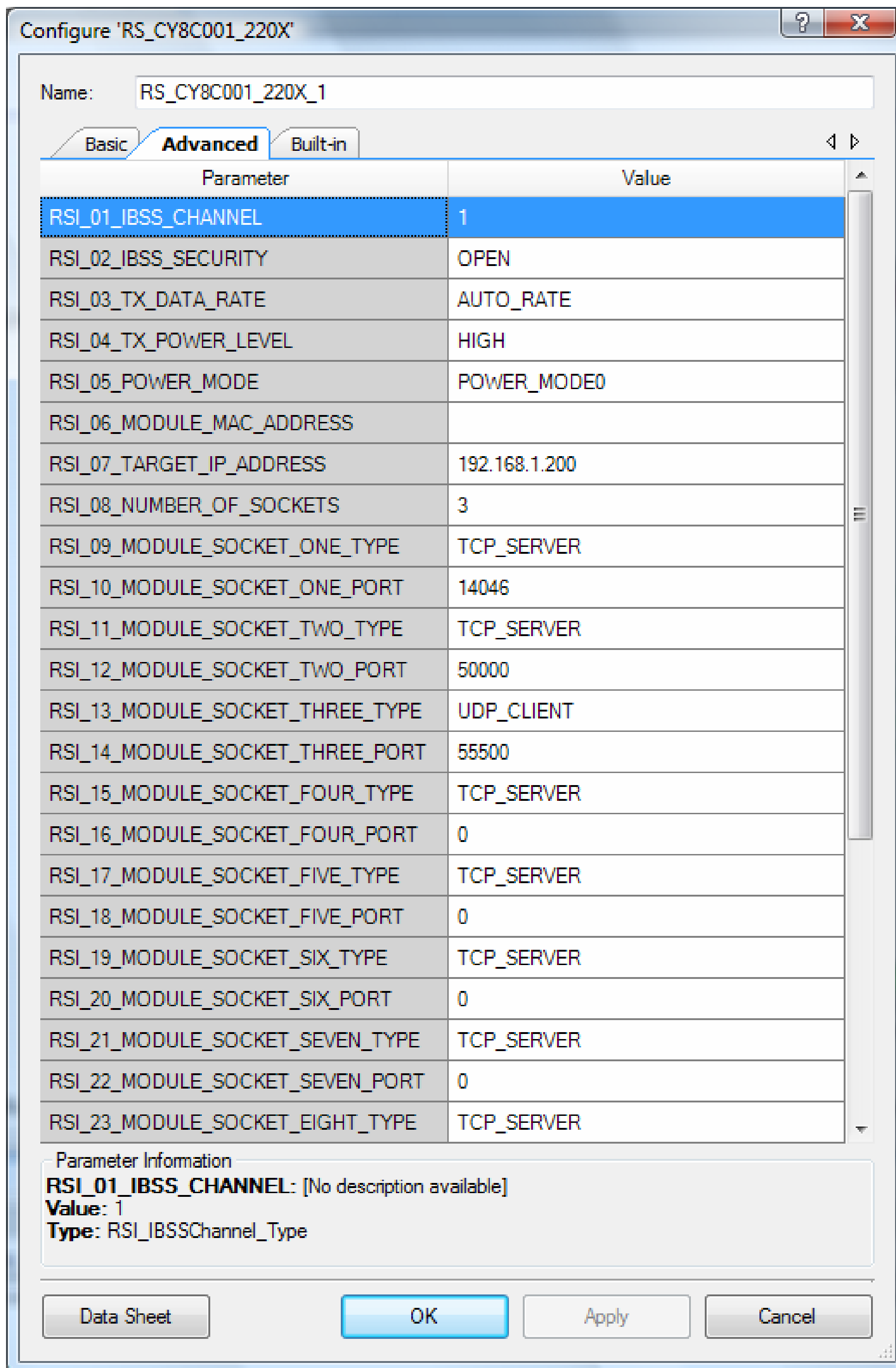


Figure 6: Advanced Tab of Configure Dialog

The table below describes each parameter of the Advanced tab and the possible options for each of them.

Parameter	Options	Description
RSI_01_IBSS_CHANNEL	1 to 11	This parameter sets the channel in which the module will create an IBSS network. This parameter is valid only if the NETWORK_TYPE parameter is set to IBSS and IBSS_MODE is set to CREATOR.
RSI_02_IBSS_SECURITY	OPEN WEP	The module supports WEP security mode (64 and 128-bit) in IBSS. This parameter selects whether the IBSS network is Open or Secure (WEP). This parameter is valid only if NETWORK_TYPE is set to IBSS.
RSI_03_TX_DATA_RATE	Auto Rate 802.11b rates – 1, 2, 5.5 & 11 Mbps 802.11g rates – 6, 9, 12, 18, 24, 36, 48, 54 Mbps 802.11n rates – MCS0 to MCS7	This parameter selects the Transmit Data Rate to be used by the module for data packets.
RSI_04_TX_POWER_LEVEL	HIGH MEDIUM LOW	This parameter selects the transmit power level of the Wi-Fi module. 'HIGH' configures the module to use a transmit power level of greater than 14dB. MEDIUM configures the module to use a transmit power level between 10 and 14dB. LOW configures the module to use a transmit power level between 6 and 10dB.
RSI_05_POWER_MODE	POWER_MODE0 POWER_MODE1	This parameter configures the Power Save mode of the module. Power Mode 0 is for disabling

Parameter	Options	Description
	POWER_MODE2	Power Save. Power Mode 1 and Power Mode 2 enable different forms of Power Save. Please refer to the module's datasheet for more details on each of these power modes.
RSI_06_MODULE_MAC_ADDRESS	<6-byte hexadecimal number, each byte separated by a ':' >	This parameter is used to override the MAC address stored in the module's non-volatile memory. This field can be left empty if module has to use its own MAC address.
RSI_07_TARGET_IP_ADDRESS	<4-byte dot-decimal format>	This parameter sets the IP address of the remote PC/Laptop with which the module tries to establish TCP or UDP connections based on the socket parameters once the Wireless connection is established with an Access Point.
RSI_08_NUMBER_OF_SOCKETS	1 to 8	This parameter sets the number of sockets that the module has to open. The module supports a maximum of 8 sockets – these can be any combination of TCP Server or Client and UDP Server or Client.
RSI_09_MODULE_SOCKET_ONE_TYPE RSI_10_MODULE_SOCKET_ONE_PORT . . . RSI_23_MODULE_SOCKET_EIGHT_TYPE RSI_24_MODULE_SOCKET_EIGHT_PORT	Type: TCP_SERVER TCP_CLIENT UDP_CLIENT Port: 0 to 65535	These parameters allow the user to configure the type of each socket (TCP_SERVER, TCP_CLIENT or UDP_CLIENT) and the port number to be assigned to them. The values are valid only for the number of sockets selected for the NUMBER_OF_SOCKETS parameter.
RSI_25_TARGET_ONE_PORT . . .	0 to 65535	These parameters allow the user to configure the port numbers of the TCP or UDP connections on the remote PC/Laptop whose IP address is configured in the

Parameter	Options	Description
RSI_32_TARGET_EIGHT_PORT		TARGET_IP_ADDRESS parameter.
RSI_33_MAX_PAYLOAD	PSoC3: 1 to 512 PSoC5: 1 to 1400	This parameter configures the maximum size (in bytes) of the data payload that will be transmitted or received. This number depends on the amount of RAM available on the PSoC and hence, is different for PSoC3 and PSoC5. The maximum payload allowed by the Wi-Fi module is 1400 bytes.

Table 4: Configuration Options for Advanced Tab

5 Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software. The following tables list and describe the interface to each function for SPI and UART interfaces. The detailed description for each API along with the parameters is available in HTML format (generated using doxygen) in the Documentation\API_Library\SPI and Documentation\API_Library\UART folders.

By default, PSoC Creator assigns the instance name "RS_CY8C001_220X_1" to the first instance of the component in a given design. You can rename the instance to any unique value that follows the syntactic rules for identifiers.

6 API Library for Wi-Fi over SPI Interface

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

6.1 API data structure

This section contains important data structures 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.

6.1.1 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[2]

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
161	23
165	24

Table 3: Channel Number Parameter (5 GHz)

[2] DFS is not supported in the current release; it is advised to not use channels from 52 to 140 if the environment is expected to co-exist with Radar signals.

6.1.2 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;
                uint8 padding[2];
        } joinFrameSnd;
        uint8 uJoinBuf[RSI_SSID_LEN + RSI_PSK_LEN + 8];
} rsi_uJoin;
```

Structure Member Name	Structure Member Type	Description
nwType	uint8	Network type 0 – IBSS (ad-hoc, open mode)[3] 1 – Infrastructure 2 – IBSS (ad-hoc) with WEP security
securityType	uint8	Security type[4] 0 – OPEN 1 – WPA1 2 – WPA2 3 – WEP
dataRate	uint8	Transmission data rate. Rate at which the data has to be transmitted. For

[3] In the case of IBSS (ad-hoc mode), 4 joinees to a network created by the module, is supported.
[4] 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
		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 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

6.1.3 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 `rsi_ipparam_set(rsi_uIppara *uIpparamFrame)` 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)
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

6.1.4 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 `rsi_socket(rsi_uSocket *uSocketFrame)` 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 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

6.1.5 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

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

6.1.6 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

6.1.7 Scan response information 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.
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
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

6.1.8 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	StructureMember Type	Description
rsi_scanResponse			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_	rsi_scanInfo	Scanned Access point information <refer

	AP_SCANNED_MAX]		table>
	status	uint8	Status of scan command 0000 Success 0002 Already associated 0003 No Access Point found 000A Invalid channel
rsi_mgmtResponse			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 ffdata 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
rsi_rssiFrameRcv			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)
rsi_socketFrameRcv			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 rsi_send_data and close this socket using rsi_socket_close API's.
	moduleSocket[2]	uint8	Local port number
	moduleIpaddr[4]	uint8	Local ipaddress
	errCode[4]	uint8	Error code 0 – Success -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 -95: ARP request failed -121 : Error issued when trying to connect to non-existent TCP server socket in remote terminal -123: Invalid socket parameters (if invalid parameters are given like, source port number 0, destination IP starts with 224 etc.) -124: TCP socket open failure -127: Socket already exists
rsi_socketCloseFrameRcv			Structure for socket close response
	rspCode[2]	uint8	Response code (0x06 in rspCode[0])
	socketDsc[2]	uint8	Descriptor of the socket closed
	errorCode[4]	uint8	Error codes: 0 – Success -2 : Socket not available -91: IGMP Error -95: ARP request failed
rsi_ipparamFrameRcv			Structure for ipconfiguration response
	rspCode[2]	uint8	Response Code (0x01 in rspCode[0])
	macAddr[6]	uint8	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
rsi_conStatusFrameRcv			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
rsi_qryDhcpInfoFrameRcv			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.
	serverIpAddr[4]	uint8	The IP address of the DHCP server.
	errorCode[4]	uint8	Error code =0 –on success !=0 – on failure
rsi_qryNetParmsFrameRcv			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[5]. 0 – AdHoc mode 1 – Infrastructure mode
	errorCode[4]	uint8	Error code: =0 on success !=0 on failure
rsi_qryFwversionFrameRcv			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 is x.y.z (e.g., 1.3.0).
rsi_disconnectFrameRcv			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
rsi_setMacAddrFrameRcv			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

[5] Please check the release notes of the firmware to see if AdHoc mode is supported.

			command is given after WLAN configuration (after "Join" command).
	padding[2]	uint8	Padding purpose
rsi_recvFrameUdp			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_PAYLOAD_SIZE]	uint8	Actual payload data.
	padding[2]	uint8	padding
rsi_recvFrameTcp			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
rsi_recvRemTerm			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.
rsi_recvLtcpEst			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
rsi_dnsFrameRcv			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
rsi_bssid_nwtypeFrameRecv			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_SCANNED_MAX];	rsi_bssid_nwtypeInfo	Scanned access point information (along with BSSID & network type)
	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

6.2 SPI 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.

6.2.1 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: Here this API is expecting that The soft boot load files sbinst1,sbinst2,sbdata1 and if required sbdata2 are placed in a folder with name "Firmware" in API_Lib directory of the release.

6.2.2 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-11-26 and RS9110-N-11-28 modules which have 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.

6.2.3 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.

6.2.4 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.

6.2.5 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.

6.2.6 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.

6.2.7 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.

6.2.8 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.

6.2.9 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).

6.2.10 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.

6.2.11 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.

6.2.12 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 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.

6.2.13 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. Please refer to the Wi-Fi module's datasheet for more hardware-related details on the interrupt 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.

In polling mode the application should call this API explicitly to update the events from the module.

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

6.2.14 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.

6.2.15 rsi_spi_query_dhcp_params.c

This file contains the API for querying DHCP parameters.

API Prototype:

```
int16 rsi_query_dhcp_parms(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.

6.2.16 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.

6.2.17 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.

6.2.18 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.

6.2.19rsi_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_config.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: Here this API is expecting that the image upgrade files (iunst1,iust2,iudata) and functional firmware files (ffinst1,ffinst2,ffdata) are placed in a folder with name "Firmware" in API_Lib directory of the release.

6.2.20rsi_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.

6.2.21 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.

6.2.22 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.

6.2.23 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.

6.2.24 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.

6.2.25 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.

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.

6.3 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.

7 API Library for Wi-Fi over UART Interface

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 `API_Lib\rsi_lib_api.c` file contains these APIs. The rest of the files with 'rsi_lib_' prefix are related to the implementation of these APIs.

The APIs included in the library are listed below. Please refer to the HTML documentation for more details on each API like the parameters, return values, etc.

All the parameters to the APIs are INPUT parameters unless otherwise mentioned explicitly as OUTPUT parameter. All the APIs are non-blocking – the application has to call APIs to issue commands to the Wi-Fi module and then call the rsi_read_cmd_rsp API to read the response for the command.

7.1.1 Set Band

API Prototype:

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

Description:

Configures the band for the Wi-Fi module.

Parameters:

Band: 0 – 2.4GHz, 1 – 5GHz

Return value:

0 for success, 0xff for failure

Prerequisites:

This is the first API to be called, to configure the band.

7.1.2 Init

API Prototype:

```
int16 rsi_init_baseband (void)
```

Description:

Initializes the baseband and RF components of the Wi-Fi module.

Prerequisites:

rsi_band should have been executed before calling this API.

7.1.3 Set number of scan results

API Prototype:

```
int16 rsi_scan_num (uint8 num)
```

Description:

Configures the number of scan results returned by the module each time the rsi_scan_next API is called. This function is useful if the MCU has limited memory and cannot accommodate the complete list of scanned Access Points at one go.

This API can be skipped if the host has enough memory to store the scan results. Scan response structure is explained later in this document.

Parameters:

num: This parameter configures the number of scan results the module returns for the Scan and NextScan commands.

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband should have been executed

7.1.4 Passive scan

API Prototype:

```
int16 rsi_scan_passive (uint16 *bitmap)
```

Description:

Scans for Wi-Fi networks in Passive mode.

Parameters:

bit_map: Parameter to configure for which channels passive scan is to be done.

For example, if only channel 1 and channel 4 are required to be scanned passively, then the value for bit_map is calculated as

Channel[n]	Channel[4]	Channel[3]	Channel[2]	Channel[1]
0	1	0	0	1

Decimal for <000..001001> is 9. Hence, bitmap value is 9.

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband should have been executed.

7.1.5

7.1.6

7.1.7 Scan

API Prototype:

```
int16 rsi_scan (struct rsi_scanFrameSnd_s *scan_frame)
```

Description:

Scans for Wi-Fi networks in Active mode.

Parameters:

scan_frame : A pointer to the rsi_scanFrameSnd_s structure, which is explained below

```
typedef struct rsi_scanFrameSnd_s
{
  uint16 channel;
  uint8  ssid[SSID_LEN]; /* SSID_LEN is 32 bytes */
} rsi_scanFrameSnd_t;
```

Structure description

channel: 0- All channel scan, or a particular channel

Channel Number on which the scan has to be done.

Parameters for 2.4 GHz

Channel Number	chan_num 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 5: Channel Parameters for 2.4 Ghz

Parameters for 5 GHz

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
161	23
165	24

Table 6: Channel Parameters for 5 Ghz

ssid : SSID of the AP to be scanned if it is in hidden mode.

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband should have been executed.

7.1.8

7.1.9 Query number of scan results

API Prototype:

```
int16 rsi_query_scan_num (void)
```

Description:

Requests the number of Access Points scanned by the module.

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband and rsi_scan should have been executed.

7.1.10Scan next WiFi networks

API Prototype:

```
int16 rsi_scan_next (void)
```

Description:

Requests the next set of scanned Access Points. This API is valid only if the rsi_scan_num API has been used to configure the number of scan results to be returned by the module each time this API is called.

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband, rsi_scan_num and rsi_scan should have been executed.

7.1.11Query BSSIDs of scanned WiFi networks

API Prototype:

```
int16 rsi_query_bssid (void)
```

Description:

Requests for the BSSIDs of the Access Points scanned by the module.

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband and rsi_scan should have been executed.

7.1.12Set Network type

API Prototype:

```
int16 rsi_setNetworkType (uint8 nwType, uint16
ibss_type, uint16 channel_num)
```

Description:

Sets the network type to Infrastructure, IBSS or IBSS Security

Parameters:

nwType: 0-IBSS (Ah-hoc), 1- Infrastructure, 2- IBSS Security (WEP)

ibss_type: If nwType is '0', then it is applicable. 0 indicates IBSS Joiner and 1 indicates IBSS Creator

channel_num: This is valid only in Creator Mode and indicates the channel in which the IBSS has to be created. For Joiner mode, this parameter should be set to 0.

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband and rsi_scan should have been executed.

7.1.13 Set Pre Shared key

API Prototype:

```
int16 rsi_set_psk (uint8 *psk)
```

Description:

Sets the pre-shared key which is used while connecting to a secure Access Point.

Parameters:

psk: Passphrase string (ASCII) .The maximum length of the PSK is 32 characters

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband and rsi_scan should have been executed

7.1.14Set Authentication Mode

API Prototype:

```
int16 rsi_set_auth_mode (uint8 auth_mode)
```

Description:

Sets the authentication mode to Open or Shared Key Authentication. This command is required if the Access Point supports only Shared Key mode of authentication. This command should be issued before the Join command. This command is relevant only when the mode of encryption is WEP.

Paremeters:

auth_mode: 0 – Open, 1 – Shared Key

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband and rsi_scan should have been executed

7.1.15 Join

API Prototype:

```
int16 rsi_join (struct rsi_joinFrameSnd_s *jf)
```

Description:

Connects to an Access Point.

Parameters:

Jf: A pointer to the join frame structure, which is explained below

```
typedef struct rsi_joinFrameSnd_s
{
  uint8  txdataRate;
  uint8  txpowerLevel;
  uint8  ssid[SSID_LEN];
} rsi_joinFrameSnd_t;
```

Structure description:

ssid: The SSID name of the network. The maximum length of the SSID name is 32 characters. This can be the SSID of the Access Point or Client to which the module has to connect to in Infrastructure or IBSS Joiner modes respectively. It can also be the SSID of the IBSS that is to be created by the module, according to the inputs to the `rsi_setNetworkType` command.

txpowerLevel: This fixes the Transmit Power level of the module. This value can be set as follows:

0 – Low power (7dBm)

1 – Medium power (10dBm)

2 – High power (16 to 17dBm)

TxdataRate: Rate at which the data has to be transmitted. Refer to the table below for the various data rates and the corresponding values. For Channel 14, only 11b rates (1, 2, 5.5 and 11 Mbps) are allowed.

Data Rate (Mbps)	Value of uTxDataRate
Auto-rate	0
1	1
2	2
5.5	3
11	4
6	5
9	6
12	7
18	8
24	9
36	10
48	11
54	12
MCS0	13
MCS1	14
MCS2	15
MCS3	16
MCS4	17
MCS5	18
MCS6	19
MCS7	20

Table 3: Data Rate Parameter

This structure in the code may have other fields related to join, but these are the parameters that are used for rsi_join command.

Return value: 0 for success 0xff for failure

Prerequisites: rsi_band, rsi_init_baseband and rsi_scan should have been executed successfully.

7.1.16 Disassociate

API Prototype:

```
int16 rsi_disconnect (void)
```

Description:

Disconnects the Wi-Fi connection.

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband, rsi_scan and rsi_join should have been executed successfully.

7.1.17 Power Modes and commands

The RS9110-N-11-2X module supports three power modes with the UART interface. The Host can switch among the power modes using the rsi_ powermode commands depending on the Wi-Fi connection status as defined in this section.

The power modes supported by the RS9110-N-11-2X module for UART interface are classified based on the Host's capability to negotiate with RS9110-N-11-2X and the Wi-Fi connection status.

7.1.17.1 Power mode 0

In this mode, power save is disabled. The module will be in Power Mode 0 by default.

7.1.17.2 Power mode 1

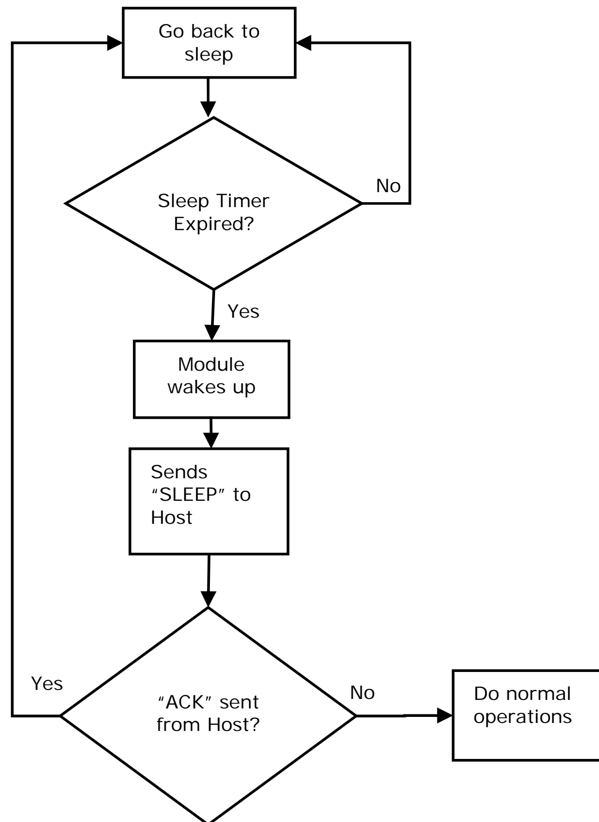
The RS9110-N-11-2X module can put the Baseband, RF and also the Core Control block to sleep in this mode.

The average amount of power consumed in this mode would depend on the sleep period and also on the time taken to exchange the SLEEP and ACK messages.

The functioning of the module in this mode depends on the connection status as explained below

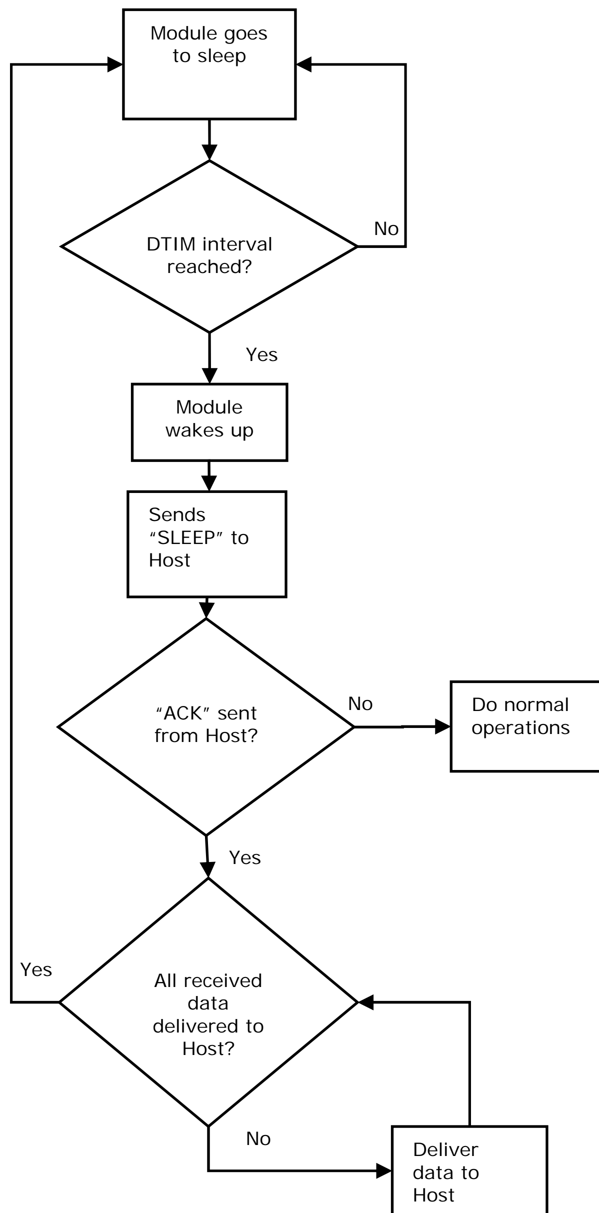
Before Wi-Fi connection

In this state, the module is configured with a sleep timer through the `rsi_set_sleep_timer` command. The input to this command indicates the amount of time the module puts the Core Control Block to sleep. Once the timer expires, the module wakes up the Core Control block and sends the "SLEEP" , in upper case ASCII (0x53 0x4C 0x45 0x45 0x50) message to the Host. If the Host sends "ACK" , in upper case ASCII (0x41 0x43 0x4B), the module will put to sleep the Core Control block for another interval of the sleep period. But, if the host wants to perform any Wi-Fi related activity viz., SCAN, JOIN, etc., it has to issue these commands accordingly to the module.



After Wi-Fi Connection

In this state, the RS9110-N-11-2X module periodically wakes up to receive DTIM (Delivery Traffic Indication Message) from the Access Point (AP). Once it wakes up, it sends "SLEEP", in upper case ASCII (0x53 0x4C 0x45 0x45 0x50) message to host. If the host has data to be transmitted, it sends the data packet instead of "ACK" to the module through the UART interface. Once host finishes sending all packets, it can send "ACK" in upper case ASCII (0x41 0x43 0x4B) to allow the module to go to sleep. If the AP has sent data to the module after the module wakes up, then all these packets will be given to the host before the module checks "ACK" message to go to sleep. Hence, even if "ACK" has already come from host, the module will not go to sleep if there are packets to be sent to Host.



7.1.17.3 Power mode 2

The RS9110-N-11-2X module puts to sleep the Baseband and RF components in this mode. The Core Control block interacting with the Host is always functional. Hence, the module can receive commands from the host at any time and there is no SLEEP-ACK message exchange with the Host.

NOTE: The average current consumption in Power Mode 2 is higher than Power Mode 1.

The functioning of the module in this mode depends on the connection status as explained below

Before Wi-Fi connection

If this power mode is enabled before the Wi-Fi connection is established, the module powers off the Baseband and RF components until the Host reconfigures the module to Power Mode 0.

If the host wants to perform any Wi-Fi related activity viz., SCAN, JOIN, etc., it has to switch to Power Mode 0. This can be done at any time during the operation. The Host can switch back to Power Modes 1 or 2 after the Wi-Fi connection is established.

After Wi-Fi Connection:

In this state, the RS9110-N-11-2X module gets information from the Access Point to which it is connected for any buffered data at every beacon. If there is no data from the host to be transmitted or received from the remote terminal or AP, it puts to sleep the Baseband and RF components.

7.1.17.4 Set power mode

API Prototype:

```
int16 rsi_powermode (uint8 pwr_mode)
```

Description:

Sets the power save mode 1, power mode 2 or power mode 0

pr_mode:

0- power mode 0,

1 – power mode1,

2 – power mode2

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband should have been executed successfully. The command rsi_set_sleep_timer should also have been executed if rsi_powermode is used before WiFi connection is established.

7.1.17.5 Set sleep timer

API Prototype:

```
int16 rsi_set_sleep_timer (uint16 sleep_time)
```

Description:

This command configures the sleep timer which is used in Power Mode 1 when the module has not established the Wi-Fi connection.

sleep_time : The value of the timer in milliseconds. The maximum value is 10000 milliseconds

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband and scan should have been executed successfully.

7.1.17.6 Send ACK

API Prototype:

```
int16 rsi_send_ack(void)
```

Description:

Gives ACK to the SLEEP indication from the module.

The command rsi_send_ack is used after detecting the RSI_EVENT_SLEEP event using rsi_read_data command.

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband and scan should have been executed successfully. And the power mode 1 is configured using rsi_powermode command.

7.1.18 Set IP Parameters

API Prototype:

```
int16 rsi_ipparam_set (rsi_ipparamFrameSnd_t *ip_param)
```

Description:

This command configures the IP address, subnet mask and default gateway of the TCP/IP stack in the RS9110-N-11-2X module.

Parameters:

ip_param: A pointer to the rsi_ipparamFrameSnd_t structure, which is explained below

```
typedef struct rsi_ipparamFrameSnd_s
{
  uint8    dhcpMode
  uint8    ipaddr[IP_ADDR_STR_LEN]; /* IP_ADDR_STR_LEN is 15 */
  uint8    netmask[IP_ADDR_STR_LEN];
  uint8    gateway[IP_ADDR_STR_LEN];
```

```
} rsi_ipparamFrameSnd_t;
```

Structure description:

dhcpMode: Used to configure TCP/IP stack in manual or DHCP modes.

0 – Manual

1 – DHCP

2 – Auto-IP[6] - Like DHCP, AUTOIP is a module that enables dynamic IPv4 addresses to be assigned to a device on startup. However, DHCP requires a DHCP server. AUTOIP is a server-less method of choosing an IP address. If the module is configured with AUTOIP, it will get an address with a 169.254/16 prefix (that is, 169.254.xxx.xxx).

ipaddr: IP address in dotted decimal format. This can be 0's in the case of DHCP.

netmask: Subnet mask in dotted decimal format. This can be 0's in the case of DHCP.

gateway: Gateway in the dotted decimal format. This can be 0's in the case of DHCP.

Here IP addresses are in ASCII dotted decimal format. Ex: "192.168.40.60"

Return value: 0 for success 0xff for failure

Prerequisites: rsi_band, rsi_init_baseband, rsi_scan and rsi_join should have been executed successfully.

7.1.19 Listen UDP

API Prototype:

```
int16 rsi_uart_socket_ludp_open (rsi_socketFrame_t *sf)
```

Description:

Opens a UDP server socket.

Parameters:

Sf: A pointer to the rsi_socketFrame_t structure, which is explained below

```
typedef struct rsi_socketFrame_s
{
  int8    handle;
  uint8   remote_ip[IP_ADDR_STR_LEN];
  uint16  rport;
  uint16  lport;
  uint8   *buf;
  uint16  buf_len;
}rsi_socketFrame_t;
```

[6] Not supported in RS9110-N-11-24

Structure description:

This structure is used in all TCP/UDP related APIs to open socket or to transmit the data. The parameters applicable for each API are explained.

lport – Local port on the RS9110-N-11-2X module

All other fields are not applicable for this API.

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband, rsi_scan, rsi_join and rsi_ipparam_set should have been executed successfully

7.1.20Listen TCP

API Prototype:

```
int16 rsi_uart_socket_ltcp_open (rsi_socketFrame_t *sf)
```

Description:

Opens a TCP server socket.

This commands opens a TCP listening socket on the local IP address and the specified "port". Once the listening socket is open, it automatically accepts remote *connect* requests. The status of the connection on a listen socket can be queried by the A `rsi_uart_query_ltcp_status` command. Only one connection can be established on a single invocation of this command.

If multiple connections on a port have to be established, then the same command has to be invoked another time

For multiple connections on the same port:

a. Open the first LTCP socket in module (for example port no. 8001)
b. Socket handle returned for this socket would be 1.
c. Connect this socket to the remote peer socket
d. You can now open the second socket in module with the same port no. 8001
e. Socket handle returned for the new socket would be 2
f. Connect this socket to another remote peer socket

Parameters:

lport – Local port on the RS9110-N-11-2X module

All other fields are not applicable for this API.

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband, rsi_scan, rsi_join and rsi_ipparam_set should have been executed successfully

7.1.21 Query Listen TCP socket status

API Prototype:

int16 rsi_uart_query_ltcp_status (uint16 handle)

Description:

Gets the connection status of a TCP server socket.

Parameters :

Handle: TCP/UDP socket handle of an already open socket using `rsi_uart_socket_ltcp_open`.

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband, rsi_scan, rsi_join, rsi_ipparam_set and `rsi_uart_socket_ltcp_open` should have been executed successfully

7.1.22 Open UDP socket

API Prototype:

```
int16 rsi_uart_socket_udp_open (rsi_socketFrame_t *sf)
```

Description:

Opens a UDP socket.

This command opens a User Datagram Protocol (UDP) socket and sets the remote system's host : port address. The UDP socket is virtually connected to the peer specified by the IP and the port. The destination IP and port address could be changed on the fly for sending the data to a different peer

Parameters:

remote_ip – IP Address of the Target server

rport – Target port (0 to 65535)

lport – Local port on the RS9110-N-11-2X module

All other fields are not applicable for this API.

Return value:

0 for success 0xff for failure

Prerequisites:

rsi_band, rsi_init_baseband, rsi_scan, rsi_join and rsi_ipparam_set should have been executed successfully

7.1.23 Open TCP socket

API Prototype:

```
int16 rsi_uart_socket_tcp_open (rsi_socketFrame_t *sf)
```

Description: Opens a TCP socket.

This command opens a Transmission Control Protocol (TCP) client socket and attempts to connect it to the specified "port" on a server defined by "host". A listening socket should be created in the server before issuing this command, for connection to go through successfully.

Parameters:

remote_ip – IP Address of the Target server

rport – The Target Port (0 to 65535)

lport – Local Port on the RS9110-N-11-2X module

All other fields are not applicable for this API.

Return value: 0 for success 0xff for failure

Prerequisites: rsi_band, rsi_init_baseband, rsi_scan, rsi_join and rsi_ipparam_set should have been executed successfully

7.1.24 Socket close

API Prototype:

```
int16 rsi_socket_close (int16 sid)
```

Description: This command closes a TCP/UDP socket in the module.

Paramters:

Sid : TCP/UDP socket handle of an already open socket

Return value: 0 for success 0xff for failure

Prerequisites: rsi_band, rsi_init_baseband, rsi_scan, rsi_join and rsi_ipparam_set should have been executed successfully. And a socket was opened successfully in other words sid should be a valid socket handle.

7.1.25 Send data

API Prototype:

```
int16 rsi_send (rsi_socketFrame_t *sf)
```

Description: This command sends a byte stream of a certain size to the socket specified by the socket handle.

handle - TCP/UDP socket handle of an already open socket.

Buf: A pointer to the buffer which has to be sent

Buf_len - The exact size of the byte stream that is to be sent, limited to a maximum of 1400 bytes. This includes byte stuffing to be done by the Host before sending the data to the module.

remot_ip – Destination IP Address. Should be '0' if transacting on a TCP socket

rport – Destination Port. Should be '0' if transacting on a TCP socket

All other fields are not applicable for this API.

Return value: 0 for success 0xff for failure

Prerequisites: rsi_band, rsi_init_baseband, rsi_scan, rsi_join and rsi_ipparam_set should have been executed successfully. . And a socket was opened successfully in other words handle should be a valid socket handle

7.1.26 Query RSSI

API Prototype:

```
int16 rsi_query_rssi (void)
```

Description: Requests the RSSI of the Access Point to which the module is connected.

Return value: 0 for success 0xff for failure

Prerequisites: rsi_band, rsi_init_baseband, rsi_scan and rsi_join should have been executed successfully

7.1.27 Query Network parameters

API Prototype:

```
int16 rsi_query_net_parms (void)
```

Description: Requests the network parameters for the Wi-Fi connection.

Return value: 0 for success 0xff for failure

Prerequisites: rsi_band, rsi_init_baseband, rsi_scan, rsi_join and rsi_ipparam_set should have been executed successfully.

7.1.28 Query MAC address of Wi-Fi module

API Prototype:

```
int16 rsi_query_mac_addr (void)
```

Description: Requests the MAC address of the Wi-Fi module.

Return value: 0 for success 0xff for failure

Prerequisites: This command can be used at any point of time after module boots up.

7.1.29 Query Network type

API Prototype:

```
int16 rsi_query_nwtype (void)
```

Description: Requests the network type of the scanned networks.

Return value: 0 for success 0xff for failure

Prerequisites: rsi_band, rsi_init_baseband, rsi_scan should have been executed successfully

7.1.30 Query FW version

API Prototype:

```
int16 rsi_query_fwversion (void)
```

Description: Requests the Firmware version present in the module.

Return value: 0 for success 0xff for failure

Prerequisites: This command can be used at any point of time after module boots up.

7.1.31 Read data

API Prototype:

```
int16 rsi_read_data(void *rsp, uint16 *event)
```

Description: Reads the data/ any other asynchronous packet

Parameters:

rsp [OUT] : A generic pointer to store the data/async packet

A pointer to a union of all responses is passed as "rsp" parameter.

event[OUT]: A pointer to uint16 used to store events like

0x02- RSI_EVENT_RX_DATA if Rx data is received

0x04- RSI_EVENT_SOCKET_CLOSE if socket close indication is received

0x08- RSI_EVENT_SLEEP if SLEEP indication is received

0x01- RSI_EVENT_CMD_RESPONSE if a command response is received for an already sent command.

Return value:

0- for success

1- RSI_ERROR_NO_RX_PENDING if there is no packet.

3- RSI_ERROR_INVALID_PKT_RECVD if invalid packet is received

Union of all responses:

Response for rsi_ipparam_set command

```
struct rsi_IpparamRsp_s
```

```
{
  uint8  macAddr[6];
  uint8  ipaddr[4];
  uint8  subnetMask[4];
  uint8  gatewayAddr[4];
};
Description:
macAddr:  uint8[6], hex, MAC Address
ipaddr[4];    uint8[4], hex, IP Address
subnetMask[4];  uint8[4], hex, Subnet Mask
gatewayAddr[4]; uint8[4], hex, Gateway Address

Response for rsi_uart_query_ltcp_status command
struct rsi_SocketLtcpQueryRsp_s
{
  int8 socketHandle; /* uint8, hex, socket handle */
  uint8 ipAddr[4];  /* uint8[4], hex, IP Address */
  uint16 port; /* hex, port the socket is bound to */
};
Description:
socketHandle:  int8, hex, socket handle, -1 indicates that the
connection has not been established yet.
ipAddr[4];   uint8[4], hex, IP Address
port:  hex, port the socket is bound to

Response for rsi_query_net_parms comamnd
struct rsi_QueryNetworkParmsRsp_s
{
  uint8    ssid[32];
  uint8    securityType;
  uint8    psk[32];
  uint8    channel;
  uint8    macAddr[6];
  uint8    dhcpMode;
  uint8    ipAddr[4];
  uint8    subnetMask[4];
```

```
uint8    gatewayAddr[4];
uint8    nOpenSockets;
};
 Description:
ssid[32]: SSID of the connected AP
securityType: AP's security type;
               0=Open, 1=WPA, 2=WPA2, 3=WEP
psk[32]:   ascii, pre-shared key
channel:   channel number of the AP
macAddr[6]; Hex mac address of the module
dhcpMode:   ascii, '0'=Manual, '1'=DHCP,
ipAddr[4]:  hex, IP Address of the module
subnetMask[4]: hex, Subnet Mask of the module
gatewayAddr[4]: hex, Gateway Address of the module
nOpenSockets;  hex, Number of open  sockets,
               range from 0 to 8
```

Response for rsi_query_bssid command

```
/**
 * Query BSSID
 */
  Struct rsi_queryBssidRsp_s
  {
    uint8  ssid[32];
    uint8  bssid[6];
  };
Description:
ssid[32]: SSID of AP
bssid[6]: BSSID of AP
```

Response for rsi_query_nwtype command

```
  /**
   * Query NwType
   */
  Struct rsi_queryNwTypeRsp_s
  {
```

```
    uint8  ssid[32];
    uint8  nwType;
  };
Description:
ssid[32]: SSID of AP
nwType: Network Type of the AP
```

Response for rsi_query_fwversion command

```
/**
  * Query FW Version
  */
  struct rsi_queryFWVersionRsp_s
  {
    uint8 fwVersion[10];
  };
Description:
fwVersion[10]: Firmware version, ASCII. Ex: "4.3.0"
```

Response for rsi_query_mac_addr command

```
  /**
   * Query MAC Address
   */
  struct rsi_queryMACRsp_s
  {
    uint8 mac_addr[10];
  };
Description:
mac_addr[10]: MAC address of the module in hex, 6 bytes.
```

Response for rsi_scan, rsi_scan_next commands

```
struct rsi_scanInfo_s
{
   uint8  ssid[SSID_LEN];
    uint8  securityMode;
    uint8  rssiVal;
};
```

```
Description:
  ssid[SSID_LEN]:  ssid of scanned access point
  securityMode:  0=open, 1=wpa1, 2=wpa2, 3=wep
  rssiVal:  absolute value of RSSI
```

Structure for receive data

```
typedef struct rsi_recvSocketFrame_s
{
  /*
   * Info to be filled by the library from received rx packet
   */
  uint8   *buffer;
  uint16  buf_len;
  uint8   handle;
  uint8   remote_ip[4];
  uint16  rport;
} rsi_recvSocketFrame_t
```

Description:

buffer: Pointer to a character buffer. Library will assign the address of the buffer where the rx packet is filled up. This buffer may be overwritten with the new incoming packet. In order to avoid this application has to process the received buffer quickly or sufficient no.of rx buffers should be supplied to the library during initialization stage using rsi_set_rx_buffer API

buf_len: Length of the buffer received

handle: socket handle of a socket on which the data is received

remote_ip[4]: IP address of the remote host.

rport: Port number in the remote host.

remote_ip and rport are applicable for UDP sockets.

```
typedef union
{
struct rsi_IpparamRsp_s IpparamRsp;
struct rsi_SocketLtcpQueryRsp_s SocketLtcpQueryRsp;
/* Response for all socket open APIs */
uint8 socketHandle; /* uint8, hex, socket handle */
/* Response for RSSI Query */
uint16 rssi_val;
```

```
struct rsi_QueryNetworkParmsRsp_s QueryNetworkParmsRsp;
Struct rsi_queryBssidRsp_s queryBssidRsp;
Struct rsi_queryNwTypeRsp_s queryNwTypeRsp;
struct rsi_queryFWVersionRsp_s queryFWVersionRsp;
struct rsi_queryMACRsp_s queryMACRsp;

   /* AP_SCANNED_MAX is 15 */
struct rsi_scanInfo_s scanInfo[AP_SCANNED_MAX];
rsi_recvSocketFrame_t  rcv_data;
}rsi_uUartRsp;
```

7.1.32 Read command response

API Prototype:

```
int16 rsi_read_cmd_rsp(void *rsp);
```

Description:

This function is used to read the response for a command

Parameters:

rsp : A generic pointer. A union, which has all the possible response types is passed to this function. (Explained above)

Return value:

0 for success and Error code for failure. The following are the all possible error codes when the application reads the response for any command given above except rsi_read_data API. These error codes should be interpreted depending upon the command issued before calling the rsi_read_cmd_rsp function.

Error Codes

1 RSI_ERROR_NO_RX_PENDING if there is no RX packet from module

2 RSI_ERROR_NO_CMD_RSP_PENDING if the received packet is not the command response type

-1 Waiting for the connection from peer

-2 Unable to allocate socket

-3 Deauthentication from the Access Point

-4 Illegal IP/Port Parameter

-5 TCP/IP Configuration Failure

-6 Invalid Socket

-7 Association not done

-8 Error in command

-9 Error with byte stuffing for escape characters

-10 IP Lease expired

-11 TCP Connection Closed

-12 Pre-Shared Key not sent for connection to a secure access point

-13 No Access Points Scanned

-14 INIT command already issued.

-15 JOIN command already issued.

-16 DHCP Failure

-17 Baud Rate Not Supported

-18 Encryption mode not supported

-124 Connection establishment not supported

-127 Socket already exists

7.2 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 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_api.h` – This header file contains all the APIs which the library expects from the target platform.
3. `rsi_hal_timers.c` – The file contains the functions for implementing timers and delays.
4. `rsi_hal_init.c` – This file contains functions for initializing the HAL layer of the MCU.

7.2.1 HAL API Requirements

The following are the APIs and macros that the API library expects from the HAL layer of the MCU. For complete documentation of the APIs and macros, please read the HTML documentation included with the release package.

7.2.1.1 HAL Macros/APIs

During porting to a particular platform some of the macros may be implemented as inline code or may be replaced with a function (If it is function, prototype of the function has to be mentioned in the rsi_hal_api.h file)

1. `HAL_MODULE_POWER_ON()`

Description: Power on the Wi-Fi module by controlling a GPIO connected to a power gate which is controlling the power to the module (optional).

2. HAL_MODULE_POWER_OFF()

   Description: Power off the Wi-Fi module by controlling a GPIO connected to a power gate which is controlling the power to the module (optional).

3. HAL_MODULE_RESET()

   Description: Place the Wi-Fi module under reset by driving the Wi-Fi module's Reset signal low for at least 20ms.

4. HAL_MODULE_CLEAR_RESET()

   Description: Bring the Wi-Fi module out of reset by driving the Wi-Fi module's Reset signal high

5. DISABLE_RX_UART_INTERRUPT()

   Description: Disables the RX UART interrupt

6. ENABLE_RX_UART_INTERRUPT()

   Description: Enables the RX UART interrupt

7. void UART_Start(void)

   Description: Initialize and Enable the UART interface.

8. int16 UART_GetChar(uint8 *dataByte)

   Description: Return the next available byte of data from the receive FIFO.

   Parameters:

   dataByte [OUT] The read byte is stored in this.

   Return value:

   0 – success, -1- failure

9. int16 UART_PutChar(uint8 txDataByte)

   Description: Transmit a byte on the UART interface. This function should wait to send the byte until the TX register or buffer has room.

   Parameters:

   txDataByte: Byte to be transmitted over UART interface.

   It should return 0 for success and 0xff for the failure.

10. void delayMs(uint16 delay)

    Description: Delays by an integer number of milliseconds.

    Parameters:

    Delay: Number of milliseconds.

11. `void delayUs(uint16 delay)`

    `Description`: Delays by an integer number of microseconds.

    Parameters:

    Delay: Number of microseconds.

```
Apart from this API Library expects the following from the HAL

- Calling rsi_receive function in RX Interrupt Handler of UART
HAL. Include rsi_hal.h for the prototype.
```

7.2.2 Memory requirements for the HAL layer of MCU

Application has to allocate RX buffers and give them to the API Library so that the library copies the received bytes in to the buffers. API to set an Rx buffer is explained below.

void rsi_set_rx_buffer (uint8 *buffer, uint16 buf_size)

Description: Sets the Rx buffer

Parameters:

buffer : Address of the buffer

buf_size : MAX Size of the buffer

Application can set any number of rx buffers. And first 34 bytes of the buffer are used by the library. Hence, MAX buffer size that is used for storing the received data is buf_size – 34.

7.3 Miscellaneous Files

The API library contains two other files, which are explained below.

- **rsi_uart_api.h** – This file contains all the required API prototypes and the macros related to UART API library.
- **rsi_data_types.h –** This file contains the data types defined for the specific MCU. This has to be included in all the source files and might need to be modified according to the variable types supported on the MCU.

8 DC and AC Electrical Characteristics

TBD.

*****

Component Revision History

Version No.	Date	Changes
1.0	June 2011	Initial Version