Renesas Synergy™ Platform

Wi-Fi Framework

Introduction

This document enables you to effectively use the Wi-Fi framework module in your own design. On completion of this guide, you will be able to add the Wi-Fi framework module to your design, configure it correctly for the target application, and write code using the included application example code as a reference and an efficient starting point. For advanced API descriptions and more complex application projects, see the Renesas Synergy™ Knowledge Base in the References section of this document. These valuable resources demonstrate how to create more complex designs.

The Wi-Fi framework provides generic application interface for Wi-Fi modules with or without on-chip networking capability. Currently, only Qualcomm GT202 is supported. The Wi-Fi framework communicates through SPI with the underlying GT202. This application project provides general guidance on how to integrate a new Wi-Fi module.

Required Resources

To build and run the Wi-Fi framework application example, you need:

- Renesas Synergy™ SK-S7G2 or PK-S5D9 kit
- e^2 studio ISDE v7.5.1 or later, or IAR Embedded Workbench® for Renesas Synergy™ v8.23.3 or later
- Renesas Synergy™ Software Package (SSP) v1.7.0 or later, or Synergy Standalone Configurator (SSC) v7.5.1 or later
- Serial Port Terminal such as Tera Term installed on your PC
- SEGGER J-Link® USB driver
- Qualcomm GT202 PMOD module
- Micro USB cables
- A Wi-Fi router of 2.4 GHz bandwidth, with DHCP server capability to be used as an access point

- All the required Renesas software from the Renesas Synergy Gallery (www.renesas.com/synergy/software).

Prerequisites and Intended Audience

This application project assumes you have some experience with the Renesas e^2 studio ISDE and Synergy Software Package (SSP). Before you perform the procedure in this application note, follow the procedure in the *SSP User Manual* to build and run the Blinky project. Doing so enables you to become familiar with the e^2 studio and the SSP, and to ensure that the debug connection to your board functions properly. In addition, this application project assumes you have some knowledge on Wi-Fi and its communication protocols.

The intended audience are users who want to develop applications with Wi-Fi interface using Synergy S3/S5/S7 MCU Series.

Contents

1. Wi-Fi Framework Module Overview

Wi-Fi is a technology for wireless local area networking with devices based on the IEEE 802.11 standards. Wi-Fi networks are created using radio frequency technology to transfer data between sender and receiver. Wi-Fi most commonly uses the 2.4 GHz (12 cm) UHF and 5 GHz (6 cm) SHF ISM radio bands.

The Synergy Software Package (SSP) includes a Wi-Fi framework that enables the creation of Wi-Fi application using a generic API interface, implemented on a Wi-Fi device driver provided by Wi-Fi vendors. This section introduces the Wi-Fi framework basic blocks and key features and enables you to determine whether the intended Wi-Fi application is supported by the Synergy Wi-Fi framework at a higher level.

Note: In this example application project, the GT202 Pmod module only supports 2.4 GHz bandwidth.

1.1 Wi-Fi Framework Module Architecture

Figure 1 provides an overview of the Synergy Wi-Fi framework layered architecture:

- The Wi-Fi framework includes the enclosed five blocks in the middle of the architecture graph: NSAL, Wi-Fi Framework API, Wi-Fi On-chip Stack API, BSD Socket API, and the Wi-Fi Device Driver Interface.
- The vendor-provided Wi-Fi device drivers are included in the SSP package under SSP_Supplemental.

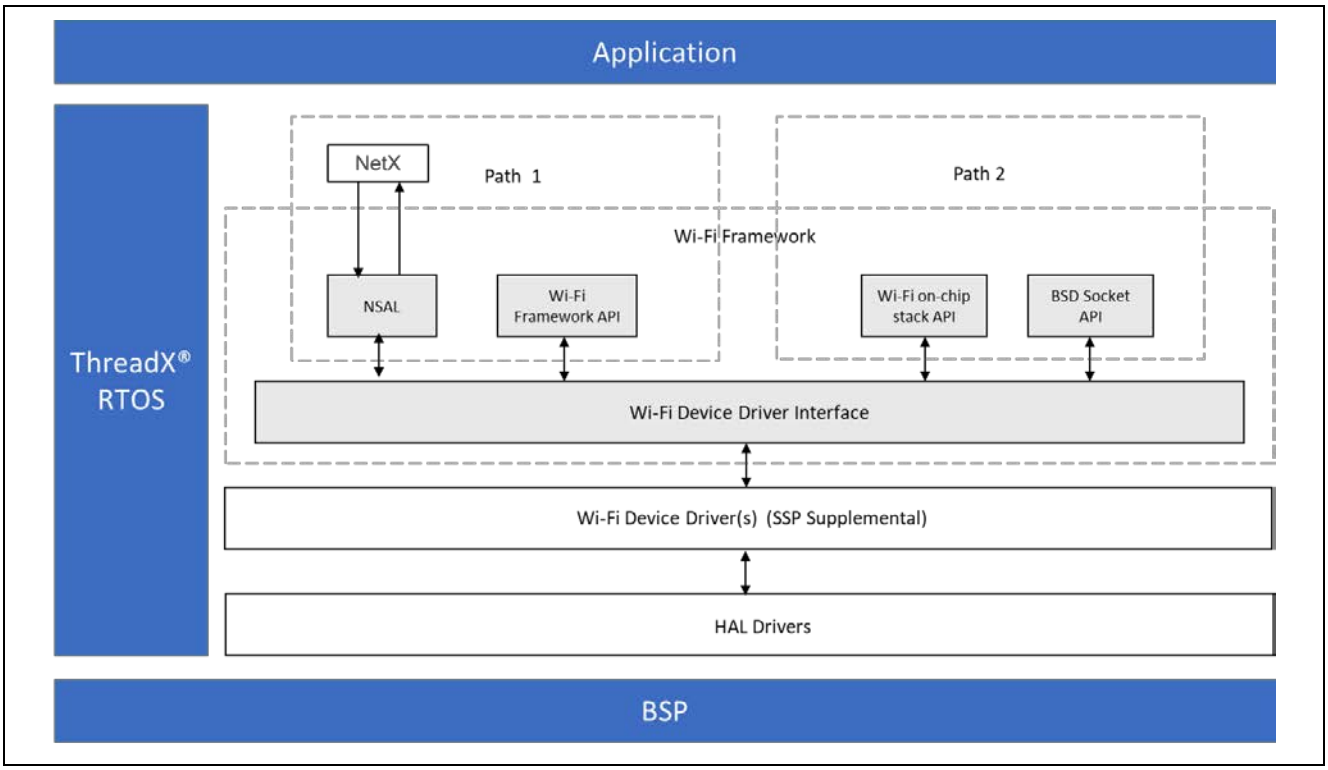

Figure 1. Wi-Fi framework organization, options, and stack implementations

The Wi-Fi framework implementation allows Wi-Fi modules, with or without on-chip networking stack support, to be integrated with the SSP low-level support blocks.

- Path 1: Using the NetX™/NetX Duo™ NSAL, in addition to the Wi-Fi framework APIs blocks (see Figure 1).

Note: For simplicity in this document, NetX refers to both NetX and NetX-Duo when applicable to both.

- Path 2: Using on-chip networking stack support API and the BSD Socket APIs (see Figure 1).

Sections 1.1.1 to 1.1.5 describe the building blocks essential to Wi-Fi application creation.

1.1.1 Synergy Wi-Fi Framework Application Interface

Generic APIs provided by the Wi-Fi framework are used to configure and provision the interface as an Access Point (AP) or as a client, and also perform data transfers. For Wi-Fi framework APIs details, see section 3.1.

1.1.2 Network Stack Abstraction Layer (NSAL)

- The Wi-Fi framework provides a network stack abstraction layer (NSAL), which is used to manipulate data frames of the MAC layer with Wi-Fi framework APIs.
- The NSAL implements the network device driver interface for the NetX IP stack, which allows the physical network interface (that is, the MAC layer implementation) to be retargeted for Wi-Fi, instead of the internal MAC interface for the wired Ethernet port.
- For details on NSAL interface function calls, see section 3.1.3.

1.1.3 SSP HAL Interface

- The HAL interface implements SSP HAL components used by the Wi-Fi module for lower level communication with the Synergy MCU
- This implementation is specific to the Wi-Fi module that uses different HAL components such as the SPI, ICU, IOPORT, SDMMC, and DTC
- For details on the HAL interface APIs, see the *SSP User's Manual*. With the Wi-Fi framework, access to these HAL drivers from the application is not required.

1.1.4 Socket Interface

- Socket APIs provide an interface to the application used with BSD Socket APIs
- Socket APIs require the Wi-Fi module/driver to support the on-chip networking stack and BSD Socket APIs. When the application uses these APIs, it is using the on-chip networking stack present on the Wi-Fi chipset, and is not using NSAL or the networking stack running on the Synergy MCU.
- NetX applications, such as DHCP and DNS, cannot be used with the Wi-Fi on-chip networking stack.
- For details on Socket APIs, see section 3.3.

1.1.5 On-chip Stack Interface

- On-chip stack APIs are an interface to the application configuring the IP address of the module and start/stop DHCP server, when configured in AP mode.
- On-chip stack APIs use the networking stack running on the Wi-Fi chipset. Similar to Socket APIs, the use of on-chip stack APIs and NSAL are mutually exclusive.
- The on-chip stack's capabilities depend on the Wi-Fi module used and may not be supported across different vendor modules.
- For on-chip stack API information, see section 3.2.

1.2 Wi-Fi Framework Module Features

The following features are provided in the Path 1 and Path 2 implementation.

1.2.1 Path 1: Wi-Fi Framework Features using NetX and NASL

Path 1 uses the NetX for networking support. This path uses the network stack abstraction layer (NSAL) to implement the generic MAC layer, making use of Wi-Fi framework APIs. This path enables you to develop application code without getting into details of the Wi-Fi module device driver.

Using NetX and NASL:

- Allows the same application code to be used across different Wi-Fi modules that support Path 1
- Allows easy migration of the Ethernet-based application to a Wi-Fi based application. Once the NetX device driver and the Wi-Fi framework are swapped, existing NetX applications should work as they did previously. Add a call from the application to set the Wi-Fi provisioning.
- Includes the flexibility to debug and fine-tune the application and TCP/IP stack as required by the application.

The current NSAL implementation only provides NetX NSAL. Adding support for a new network stack requires implementing the appropriate NSAL.

1.2.2 Path 2: Wi-Fi Framework Features using the On-Chip Networking Stack Support

Using the on-chip networking stack provides the following:

- It is beneficial when using MCUs with a small memory footprint
- Offers BSD sockets that interface to create socket-based applications with the on-chip TCP/UDP
- Offers an option to integrate third-party application protocols on top of the TCP/IP, such as MQTT and COAP, without using the NetX stack.

1.2.3 Additional Features if Supported by the Wi-Fi Module or Wi-Fi Module Driver

The following features are supported by the framework only if they are supported by the Wi-Fi module or Wi-Fi module driver:

Access Control List Management — Allows application to control which devices can be connected to an access point

Multicast Filter List Management — Allows application to join or leave multicast group

2. Wi-Fi Framework Module Operational Overview

The following operational overview describes a typical Wi-Fi application used to initialize the Wi-Fi module, and Wi-Fi packet transmission/reception using NetX and on-chip network stack support.

2.1 Wi-Fi Module Operational Overview using NSAL and NetX (Path 1)

In Path 1 operation, the application uses NSAL and NetX to establish the Wi-Fi module network application.

2.1.1 Initialization using NetX

The Figure 2 flow chart and following steps illustrate the Wi-Fi module initialization:

1. Create a NetX IP instance in the Synergy configurator.
2. Add the NetX Port using the Wi-Fi framework in the Synergy configurator, and then provide any low-level dependencies and configuration.
3. Generate the Project Content and build the project.
4. While running, the SSP generated code calls `nx_ip_create`.
5. `nx_ip_create` calls the NetX NSAL driver entry point.
6. The NSAL driver entry point calls the Wi-Fi framework `open()` function.
7. Wi-Fi framework `open()` function calls the Wi-Fi device driver `open()` function to initialize and enable the Wi-Fi module.
8. `nx_ip_interface_status_check` API is called and waits for the `NX_IP_LINK_ENABLED` status to be set.
9. On successful completion, the Wi-Fi module is ready for scan and provision.

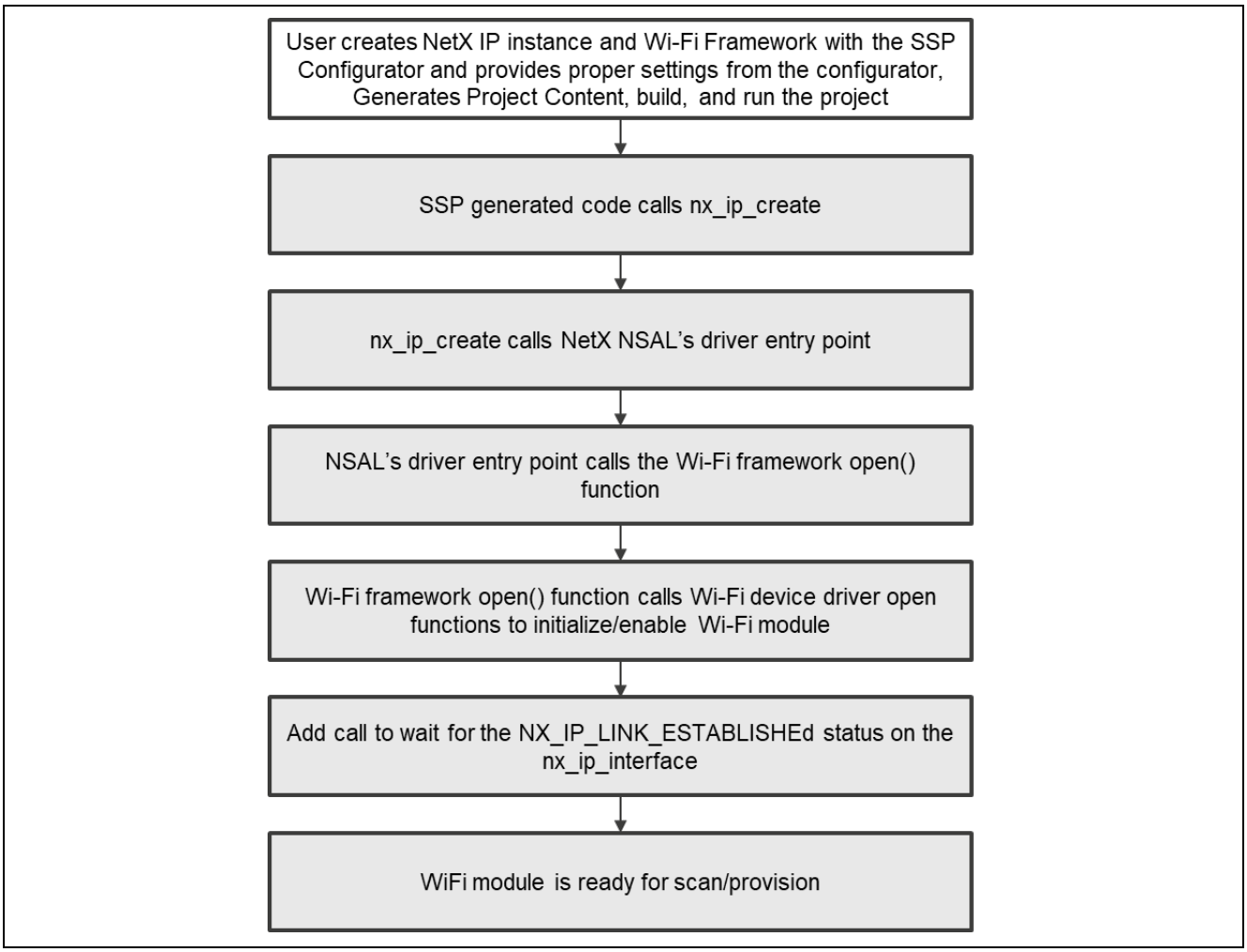

Figure 2. Wi-Fi module initialization with NetX path

2.1.2 Wi-Fi Packet Transmission using NetX

The following general steps and the Figure 3 flow chart illustrate the Wi-Fi module packet transmission using NetX:

1. SSP-generated code initializes the NetX packet pool.
2. Wi-Fi module is initialized and enabled.
3. User application code provisions the Wi-Fi module.
4. User application code calls the NetXs TCP/UDP Socket Send API.
5. NetX Send API calls the NSAL driver entry point for package transmission.
6. NSAL driver entry point calls the Synergy Wi-Fi Framework Transmit API function.
7. Synergy Wi-Fi framework Transmit API function calls the Wi-Fi device driver Transmit API function.
8. Wi-Fi device driver transmits the user data.

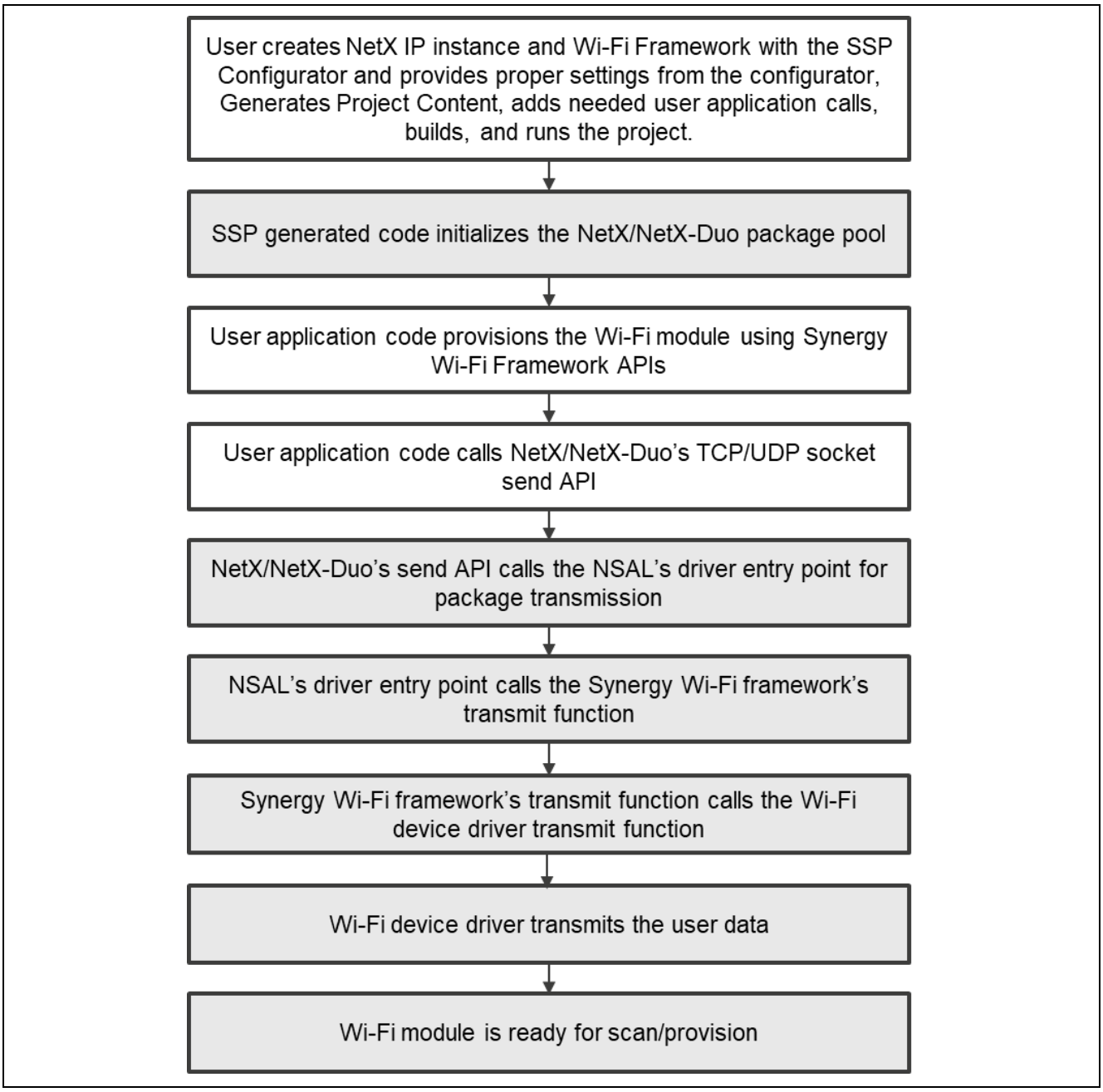

Figure 3. Wi-Fi module data transmission with NetX path

2.1.3 Wi-Fi Packet Reception using NetX

The following general steps and the Figure 4 flow chart illustrate the Wi-Fi module packet reception using NetX:

1. Wi-Fi packet reception starts from the Wi-Fi device driver interrupt service routine.
2. Once the packet is received, the receive callback function of the Wi-Fi device driver transfers the receive data to buffer and initiates the Wi-Fi framework receive callback function.
3. The Wi-Fi framework receive callback function calls the NSAL receive callback function.
4. The NSAL receive callback function calls the NetX deferred receive processing callback.

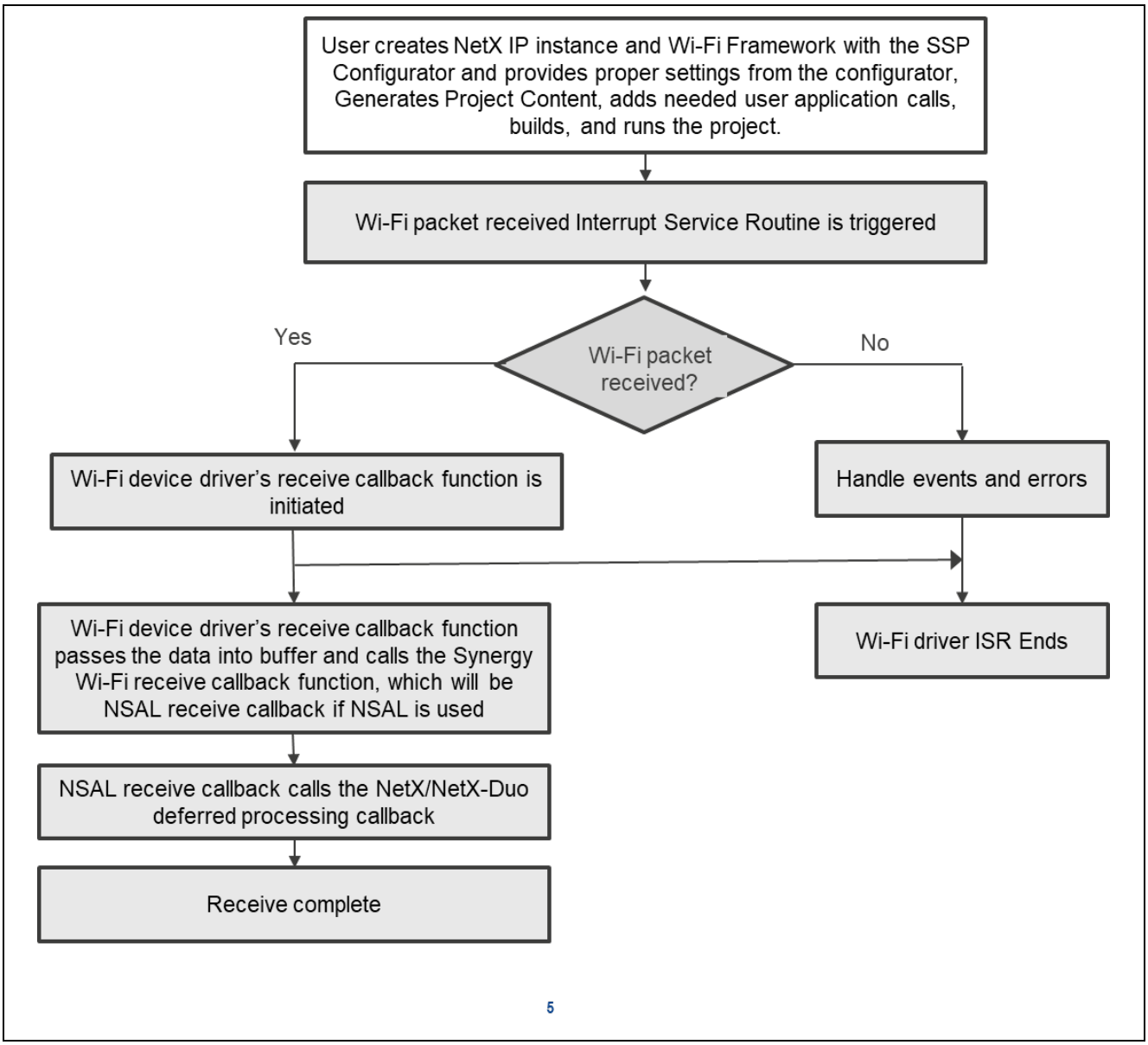

Figure 4. Wi-Fi module packet reception with NetX path

2.2 Wi-Fi Application Operation using On-Chip Networking Stack (Path 2)

The following general steps and the Figure 5 flow chart illustrate the Wi-Fi module operation using On-chip networking stack (Path 2):

1. User application creates the Wi-Fi provision structure.
2. User application calls the BSD Socket interface API to open the on-chip networking stack.
3. The BSD Socket interface open API internally calls the on-chip stack support open API function.
4. The On-chip stack support API calls the Wi-Fi framework open function to enable the Wi-Fi module.
5. User application scans and provisions the Wi-Fi module.
6. User application calls the on-chip network stack APIs to configure the IP address.
7. User application calls the standard BSD Socket APIs to communicate with the Wi-Fi module.
8. On completion, the user application codes call the BSD Socket interface close function to close the Wi-Fi module.

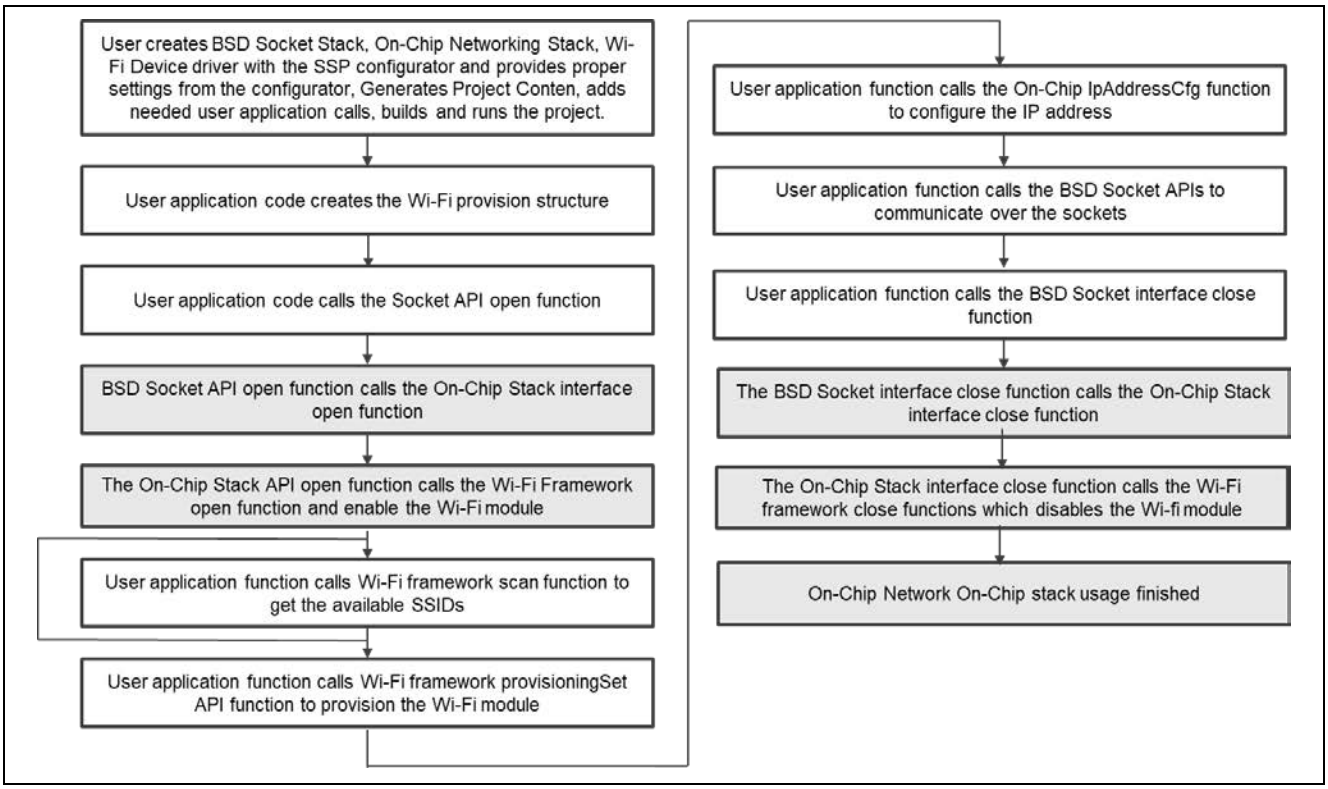

Figure 5. Wi-Fi operational flow with On-chip networking stack support

2.3 Wi-Fi Framework Module Important Operational Notes and Limitations

2.3.1 Wi-Fi Framework Module Operational Notes

- The Wi-Fi module has various parameters as specified by 802.11 standards. It is possible that individual device drivers and Wi-Fi chipsets might not support the configuration of all the functions.
- For the Wi-Fi interface to become active, at minimum configure the channel, Service Set Identifier (SSID), security scheme, and security credentials.
- Current NSAL implementation includes support for NetX (IPv4) and NetX-Duo (IPv6). NetX and NetX Duo support IPv4; however, NetX Duo also supports IPv6. Adding support for a new network stack requires implementing the appropriate NSAL.
- For the security setting, WEP keys can be entered in either ASCII or Hex format and configured to use either 40-or 104-bit keys. WEP key has a 24-bit initialization vector, in addition to the secret key. The key depends on the vendor; 64-bit WEP keys can be referred to as 40-bit keys, and 128-bit WEP keys can be referred to as 104-bit keys. The Wi-Fi framework accepts 1 to 4 WEP keys of a specific format and type. In the provisioning structure, you must fill the security type as **`SF_WIFI_SECURITY_TYPE_WEP`**, and at least one (maximum is four) WEP key in the key buffer.

2.3.2 Wi-Fi Framework Module Limitations

- The Wi-Fi framework does not support the Synergy S1 MCU Series due to memory constraints.
- There is a bug in this version of the Wi-Fi framework. When the NetX and NSAL path is selected, the Synergy configurator disables the on-chip networking stack setting by default. Currently, the Synergy configurator still allows the enabling of the On-chip Networking Stack support. For more information on this issue, see the *SSP v1.4.0 Release Note*.
- For Wi-Fi modules using RSPI, the DTC components are auto-filled as the dependencies for RSPI. When DTC is used with RSPI, 32-bit transfers are required. However, your Wi-Fi module (such as the GT202) vendor code might support 8-bit or 16-bit transfers only. In this case, the DTC component must be removed.

3. Wi-Fi Framework Module APIs Overview

The Wi-Fi framework module APIs can be used in Path 1 and Path 2, and in the Wi-Fi On-chip stack support APIs; in addition to the Wi-Fi Socket APIs.

3.1 Wi-Fi Framework APIs calling through the NetX (Path 1)

The Wi-Fi Framework APIs that apply to the NetX implementation in Path 1 are covered in the following subsections. The `sf_wifi_api.h` header file includes all the structures and APIs that are defined in the *SSP User's Manual* API References for the associated module. These data structures are normally generated by the Synergy Configurator using the properties sheet filled in by the user.

3.1.1 Synergy Wi-Fi Framework Instance

The application must define the Synergy Wi-Fi framework instance before using it. The Synergy Wi-Fi framework instance refers to the Wi-Fi module specific control data, configuration data, and APIs. The application uses this instance to perform operation on Wi-Fi module.

Figure 6 shows members of the Synergy Wi-Fi framework instance (`sf_wifi_instance_t`). The instance is generated when user provides the Name property in the Synergy configurator for the Wi-Fi module.

```
/** This structure encompasses everything that is needed to use an instance of this
interface. */
typedef struct st_sf_wifi_instance
{
    sf_wifi_ctrl_t      * p_ctrl;       ///< Pointer to the control structure for
this instance
    sf_wifi_cfg_t const * p_cfg;        ///< Pointer to the configuration structure
for this instance
    sf_wifi_api_t const * p_api;        ///< Pointer to the API structure for this
instance
} sf_wifi_instance_t;
```

Figure 6. Wi-Fi framework instance

3.1.2 Synergy Wi-Fi Framework APIs

The Wi-Fi framework module defines APIs such as open, close, provision, transmit, and scan related to the Wi-Fi operation.

3.1.2.1 Structures used in Wi-Fi Framework APIs

From a higher level, these are the structures used in the Wi-Fi framework APIs and these structures have their own elements defined as structures in many cases. Some basic data structures used in the Wi-Fi framework are given. All structures and APIs described here are defined in the header file `sf_wifi_api.h`.

- Wi-Fi control parameter structure `sf_wifi_ctrl_t`
  This control parameter structure is the pointer to the user-provided storage for the Wi-Fi module control structure and can be an input or output of the Wi-Fi framework API, depending on how the API uses it.

```
/** WiFi Framework control structure */
typedef struct st_sf_wifi_ctrl
{
    /** Storage for information needed for each WiFi device driver in the system.
*/
    void * p_driver_handle;
} sf_wifi_ctrl_t;
```

Figure 7. Wi-Fi framework control structure

- Wi-Fi configuration parameter structure `sf_wifi_cfg_t`
  This configuration parameter structure is the pointer to the user-defined Wi-Fi module configuration structure and is an input to several of the APIs as described in the section that follows.

```
/** Define the WiFi configuration parameters */
typedef struct st_sf_wifi_cfg
{
    uint8_t                             mac_addr[6];                ///< MAC
address of WiFi Device
    sf_wifi_interface_hw_mode_t         hw_mode;                    ///<
Modulation type: 11a/b/g/n
    uint8_t                             tx_power;                   ///< Sets
transmit power in dBm
    sf_wifi_rts_t                       rts;                        ///<
RTS/CTS handshake flag
    uint16_t                            fragmentation;              ///<
Fragmentation threshold
    uint8_t                             dtim;                       ///<
Delivery traffic indication message interval
    sf_wifi_high_throughput_t           high_throughput;            ///< High-
throughput mode. Only valid for 802.11n
    sf_wifi_preamble_t                  preamble;                   ///<
Preamble type
    sf_wifi_wmm_t                       wmm;                        ///< WiFi
Multimedia Mode flag. If enabled, also requires
    uint8_t                             max_stations;               ///<
Maximum permitted stations. Valid in AP mode only.
    sf_wifi_ssid_broadcast_t            ssid_broadcast;             ///< SSID
broadcast flag. Valid in AP mode only.
    sf_wifi_access_control_t            access_control;             ///< Mode
of access control MAC list
    uint32_t                            beacon;                     ///< Beacon
interval. Valid in AP mode only
    uint32_t                            station_inactivity_timeout; ///<
Station inactivity timeout value. Valid in AP mode only.
    sf_wifi_wds_t                       wds;                        ///< WDS
flag. Valid in AP mode only.
    void                              * p_buffer_pool_rx;           ///<
Pointer to Network stack Rx buffer pool
    sf_wifi_mandatory_high_throughput_t req_high_throughput;        ///< Only
allow HT mode. Valid in AP mode only
    void (*p_callback)(sf_wifi_callback_args_t * p_args);          ///< Pointer
to callback function
    void const                        * p_context;                  ///< User
defined context passed into callback function
    void const                        * p_extend;                   ///<
Instance specific configuration
} sf_wifi_cfg_t;
```

Figure 8. Wi-Fi framework configuration structure

- Wi-Fi Framework API structure `sf_wifi_api_t`
  This structure includes all the Wi-Fi framework API function pointers.

```
/** Framework API structure.  Implementations will use the following API. */
typedef struct st_sf_wifi_api
{
    ssp_err_t (*open)(sf_wifi_ctrl_t * p_ctrl, sf_wifi_cfg_t const * const
    p_cfg);
    ssp_err_t (*close)(sf_wifi_ctrl_t * const p_ctrl);
    ssp_err_t (*multicastListAdd)(sf_wifi_ctrl_t * const p_ctrl, uint8_t const *
    const p_mac_addr);
    ssp_err_t (*multicastListDelete)(sf_wifi_ctrl_t * const p_ctrl, uint8_t
    const * const p_mac_addr);
    ssp_err_t (*statisticsGet)(sf_wifi_ctrl_t * const p_ctrl, sf_wifi_stats_t *
    const p_wifi_device_stats);
    ssp_err_t (*transmit)(sf_wifi_ctrl_t * const p_ctrl, uint8_t * const p_buf,
    uint32_t length);
    ssp_err_t (*provisioningSet)(sf_wifi_ctrl_t * const p_ctrl,
                sf_wifi_provisioning_t const * const p_wifi_provisioning);
    ssp_err_t (*provisioningGet)(sf_wifi_ctrl_t * const p_ctrl,
    sf_wifi_provisioning_t * const p_wifi_provisioning);
    ssp_err_t (*infoGet)(sf_wifi_ctrl_t * const p_ctrl, sf_wifi_info_t * const
    p_wifi_info);
    ssp_err_t (*scan)(sf_wifi_ctrl_t * const p_ctrl, sf_wifi_scan_t * const
    p_scan, uint8_t * const p_cnt);
    ssp_err_t (*ACLAdd)(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const
    p_mac);
    ssp_err_t (*ACLDelete)(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const
    p_mac);
    ssp_err_t (*macAddressGet)(sf_wifi_ctrl_t * const p_ctrl, uint8_t * const
    p_mac);
    ssp_err_t (*macAddressSet)(sf_wifi_ctrl_t * const p_ctrl, uint8_t const *
    const p_mac);
    ssp_err_t (*versionGet)(ssp_version_t * const p_version);
    } sf_wifi_api_t;
```

Figure 9. Wi-Fi framework API structure

Refer to the Wi-Fi framework module in the *SSP User's Manual*. The API References section describes operations and definitions for function data structures, typedefs, defines, API data, API structures, and function variables, including:

- Wi-Fi Statistic and error counters structure for this IP instance `sf_wifi_stats_t`
- Wi-Fi Framework scan structure `sf_wifi_scan_t`
- Wi-Fi Framework access control mode structure `sf_wifi_access_control_t`
- Wi-Fi IP address structure `sf_wifi_ip_addr_t`
- Wi-Fi module device level information structure `sf_wifi_info_t`
- Wi-Fi provisioning parameter structure `sf_wifi_provisioning_t`

3.1.2.2 Wi-Fi Framework APIs

Table 1 shows a complete list of the available APIs, an example API call, and a brief description of API inputs/outputs. The examples assume the name of the Wi-Fi module is `g_sf_wifi0` that is user-provided in the Synergy configurator.

Table 1. Wi-Fi framework module API summary table

Function Name	API Prototype and Description

.open	`ssp_err_t (*open)(sf_wifi_ctrl_t * p_ctrl, sf_wifi_cfg_t const * const p_cfg);`
	The open API initializes the Wi-Fi driver configuration, enables the driver link, enables interrupts, and makes the device ready for data transfer.
	[in,out]
	`p_ctrl see sf_wifi_ctrl_t`
	[in]
	`p_cfg see sf_wifi_cfg_t`

.close	`ssp_err_t (*close)(sf_wifi_ctrl_t * const p_ctrl);`
	The close API de-initializes the network interface and can put it in low power mode or power it off. This API closes the Wi-Fi device driver, disables the driver link, and disables the interrupts.
	[in, out]
	`p_ctrl see sf_wifi_ctrl_t`

.infoGet	`ssp_err_t (*infoGet)(sf_wifi_ctrl_t * const p_ctrl, sf_wifi_info_t * const p_wifi_info);`
	The infoGet API acquires the Wi-Fi module information.
	[in]
	`p_ctrl see sf_wifi_ctrl_t`
	[in, out]
	p_wifi_info pointer to the user-provided storage for the Wi-Fi module configuration structure.

.statisticsGet	`ssp_err_t (*statisticsGet)(sf_wifi_ctrl_t * const p_ctrl, sf_wifi_stats_t * const p_wifi_device_stats);`
	The statisticsGet API gets the interface statistics.
	[in]
	`p_ctrl see sf_wifi_ctrl_t`
	[in, out]
	p_wifi_stats pointer to the user-provided storage for the Wi-Fi module statistics structure.

.transmit	`ssp_err_t (*transmit)(sf_wifi_ctrl_t * const p_ctrl, uint8_t * const p_buf, uint32_t length);`
	The transmit API passes the packet buffer to the Wi-Fi driver for transmission.
	[in]
	`p_ctrl see sf_wifi_ctrl_t`
	p_buf pointer to the network packet buffer
	length is length of network packet

.provisioningGet	`ssp_err_t (*provisioningGet)(sf_wifi_ctrl_t * const p_ctrl, sf_wifi_provisioning_t * const p_wifi_provisioning);`
	The provisioningGet API gets the Wi-Fi module provisioning.
	[in]
	`p_ctrl see sf_wifi_ctrl_t`
	[in, out]
	p_wifi_provisioning pointer to the user-provided storage of the Wi-Fi module provision structure.

Function Name	API Prototype and Description

.provisioningSet	`ssp_err_t (*provisioningSet)(sf_wifi_ctrl_t * const p_ctrl, sf_wifi_provisioning_t const * const p_wifi_provisioning);`
	The provisioningSet API sets the Wi-Fi module provisioning which configures the module in AP or client mode.
	[in]
	`p_ctrl` see `sf_wifi_ctrl_t`
	p_wifi_provisioning pointer to the Wi-Fi module provision structure.
	Note: After the Wi-Fi device is provisioned in any mode, to switch to another mode, close the application and open it again. For example, if the Wi-Fi device is first provisioned in AP mode, to switch to the station mode, the application code must call the `close()` function to de-initialize it and call the `open()` function again to initialize it. To set to station mode, call the `provisioningSet()` function. The same applies for switching from station to AP mode.

.scan	`ssp_err_t (*scan)(sf_wifi_ctrl_t * const p_ctrl, sf_wifi_scan_t * const p_scan, uint8_t * const p_cnt);`
	The scan API scans the available SSIDs, that is, the access points in range.
	[in]
	`p_ctrl` see `sf_wifi_ctrl_t`
	[in, out]
	`p_scan` pointer to the caller-provided Wi-Fi module scan structure that holds the scan result. The caller must ensure that adequate space is available to hold the scan results.
	`p_cnt` pointer to the variable, specifying the maximum number of SSIDs to scan; it is updated to the number of actual SSIDs scanned by the device.

.ACLAdd	`ssp_err_t (*ACLAdd)(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const p_mac);`
	The ACLAdd API adds the given MAC address to the access control list.
	[in]
	`p_ctrl` see `sf_wifi_ctrl_t`
	`p_mac` pointer to the Wi-Fi module MAC address structure.

.ACLDelete	`ssp_err_t (*ACLDelete)(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const p_mac);`
	The ACLDelete API deletes the given MAC address from the access control list.
	[in]
	`p_ctrl` see `sf_wifi_ctrl_t`
	`p_mac` pointer to the Wi-Fi module MAC address structure.

.multicastListAdd	`ssp_err_t (*multicastListAdd)(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const p_mac_addr);`
	The multicastListAdd API adds the given MAC address to the multicast filter list.
	[in]
	`p_ctrl` see `sf_wifi_ctrl_t`
	`p_mac_addr` pointer to the Wi-Fi module MAC address structure.

.multicastListDelete	`ssp_err_t (*multicastListDelete)(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const p_mac_addr);`
	The multicastListDelete API deletes the given MAC address from the multicast filter list.
	[in]
	`p_ctrl` see `sf_wifi_ctrl_t`
	`p_mac_addr` pointer to the Wi-Fi module MAC address structure.

Function Name	API Prototype and Description

.macAddressGet	`ssp_err_t (*macAddressGet)(sf_wifi_ctrl_t * const p_ctrl, uint8_t * const p_mac);`
	The macAddressGet API acquires the MAC address of the Wi-Fi module.
	[in]
	`p_ctrl` see `sf_wifi_ctrl_t`
	[in,out]
	`p_mac` user-provided pointer to the Wi-Fi module MAC address structure.

.macAddressSet	`ssp_err_t (*macAddressSet)(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const p_mac);`
	The macAddressSet API sets the MAC address of the Wi-Fi module.
	[in]
	`p_ctrl` see `sf_wifi_ctrl_t`
	`p_mac` pointer to the Wi-Fi module MAC address structure.

.versionGet	`ssp_err_t (*versionGet)(ssp_version_t * const p_version);` The versionGet API gets the version and stores it in the user-provided pointer. [in, out] `p_version` user-defined pointer to the storage location.

Note: For details on operations and definitions for the function data structures, typedefs, defines, API data, API structures, and function variables, see the API References section for the associated module in the *SSP User's Manual*.

Status Return Values can be found in Error codes in the *SSP User's Manual* API references. Table 2 shows all the possible Wi-Fi framework API error calls.

Table 2. Status Return Values

Name	Description
SSP_ERR_WIFI_CONFIG_FAILED	Wi-Fi module configuration failed
SSP_ERR_WIFI_INIT_FAILED	Wi-Fi module initialization failed
SSP_ERR_WIFI_TRANSMIT_FAILED	Wi-Fi module transmission failed
SSP_ERR_WIFI_INVALID_MODE	Wi-Fi module AP mode API called when provisioned in client mode
SSP_ERR_WIFI_FAILED	Wi-Fi module failed

3.1.3 Wi-Fi NSAL API

The Synergy Wi-Fi framework supports NetX NSAL. This implementation includes the NetX driver and the packet transmit and receive callback functions. These functions are not directly called from the user application if the Wi-Fi module is used when it is already integrated in the Wi-Fi framework, but these functions are called and implemented when a new Wi-Fi module is integrated.

3.1.3.1 Structures used in NSAL APIs

- NetX IP link structure NX_IP_DRIVER
  This structure defines the driver interface structure typically allocated from the local stack and passed to the IP Link Driver.

```
typedef struct NX_IP_DRIVER_STRUCT
{

    /* Define the driver command.  */
    UINT        nx_ip_driver_command;

    /* Define the driver return status.  */
    UINT        nx_ip_driver_status;

    /* Define the physical address that maps to the destination IP address.
*/
    ULONG       nx_ip_driver_physical_address_msw;
    ULONG       nx_ip_driver_physical_address_lsw;

    /* Define the datagram packet (if any) for the driver to send.  */
    NX_PACKET   *nx_ip_driver_packet;

    /* Define the return pointer for raw driver command requests.  */
    ULONG       *nx_ip_driver_return_ptr;

    /* Define the IP pointer associated with the request.  */
    struct NX_IP_STRUCT
                *nx_ip_driver_ptr;

    NX_INTERFACE *nx_ip_driver_interface;

} NX_IP_DRIVER;
```

Figure 10. NetX IP driver structure

- NSAL configuration structure `sf_wifi_nsal_cfg_t`
  - This structure defines the NSAL configuration parameters
  - This structure includes flags that indicate whether zero-copy support is enabled or disabled in the transmit path and the receive path.

```
typedef struct st_sf_wifi_nsal_cfg
{
    sf_wifi_nsal_zero_copy_t    tx_zero_copy;       ///< Transmit path zero copy
support
    sf_wifi_nsal_zero_copy_t    rx_zero_copy;       ///< Receive path zero copy
support
    uint8_t                   * p_tx_packet_buffer; ///< Pointer to Tx buffer
used to consolidate data from chained NetX packets
} sf_wifi_nsal_cfg_t;
```

Figure 11. NSAL configuration structure

The following table lists the NSAL functions.

Table 3. NSAL interface APIs

Function Name	API Prototype and Description

nsal_netx_driver	`void nsal_netx_driver(NX_IP_DRIVER * p_driver, sf_wifi_instance_t const * p_wifi_instance, sf_wifi_nsal_cfg_t * p_wifi_nsal_cfg);`
	The nsal_netx_driver API implements various IP driver commands used by NetX by calling the corresponding Wi-Fi framework APIs.
	[in]
	`p_driver` see NX_IP_DRIVER
	`p_wifi_instance` see `sf_wifi_instance_t`
	`p_wifi_nsal_cfg` see `sf_wifi_nsal_cfg_t`

Function Name	API Prototype and Description

nsal_netx_send_packet	`Static uint32_t nsal_netx_send_packet(NX_IP * p_ip, NX_PACKET *p_packet, sf_wifi_instance_t const * p_wifi_instance, sf_wifi_nsal_cfg_t * p_nsal_cfg);`
	The `nsal_netx_send_packet` API calls the `nsal_netx_transmit` and the Wi-Fi transmit API functions to transmit packets. If zero-copy support is enabled, the same NetX packet is transferred from NetX to Wi-Fi driver. If zero-copy is disabled, the API copies data from NetX packet to the driver buffer.
	[in]
	`p_ip` NetX IP structure, see the *SSP User's Manual* for details
	`p_packet` NetX packet structure, see the *SSP User's Manual* for details
	`p_wifi_instance` see `sf_wifi_instance_t`
	`p_nsal_cfg` see `sf_wifi_nsal_cfg_t`

nsal_netx_receive	`Static uint32_t nsal_netx_receive(void * p_nsal_interface, uint8_t p_buffer, uint32_t length, sf_wifi_nsal_cfg_t * p_wifi_nsal_cfg);`
	The nsal_netx_receive API is called from the Wi-Fi device driver callback function `wifi_driver_callback`. If zero-copy support is enabled, the same NetX packet is transferred from Wi-Fi driver to NetX. If zero-copy is disabled, the API copies data from the driver buffer to NetX packet and then passes the NetX stack for additional processing.
	[in]
	`p_ip` NetX IP interface pointer
	`length` of data buffer
	`p_wifi_nsal_cfg` see `sf_wifi_nsal_cfg_t`
	[in, out]
	`pdata` user provided data buffer for data reception

Note: For details on operations and definitions for the function data structures, typedefs, defines, API data, API structures, and function variables, see the API References section for the associated module in the *SSP User's Manual*.

Status Return Values Error codes are listed in the *NetX User Guide* available from the Synergy Software Package page (www.renesas.com/synergy/ssp), where you can download the *X-Ware™ Component Documents for Renesas Synergy* zip file. A Renesas.com login may be required.

3.2 On-chip Networking Stack Support APIs (Path 2)

The On-chip networking stack support APIs can be used to configure the Wi-Fi module in Path 2, where an on-chip networking stack is called. The API helps to configure the IP address for the interface, and start/stop DHCP server when configured in the AP mode.

3.2.1 On-chip Networking Stack Instance

The application must define the Synergy Wi-Fi framework On-chip networking instance before using it. The Synergy Wi-Fi framework instance refers to the Wi-Fi module specific control data, configuration data, and APIs. The application uses this instance to perform operation on the Wi-Fi module.

The following structures are members of the Synergy Wi-Fi On-chip networking stack support instance (`sf_wifi_onchip_stack_instance_t`).

```
/** SF WiFi On Chip Stack Instance structure */
typedef struct st_sf_wifi_onchip_stack_instance
{
    sf_wifi_onchip_stack_ctrl_t         * p_ctrl;       ///< Pointer to the control
structure for this instance
    sf_wifi_onchip_stack_cfg_t const    * p_cfg;        ///< Pointer to the configuration
structure for this instance
    sf_wifi_onchip_stack_api_t const    * p_api;        ///< Pointer to the API structure
for this instance
      } sf_wifi_onchip_stack_instance_t;
```

Figure 12. On-Chip Networking Stack Support instance

The control structure, configuration structure, and the API structure of the on-chip networking stack support follow.

3.2.1.1 On-chip Network Stack Support Structures

From a higher level, these are the structures used in the Wi-Fi framework BSD Socket interface APIs. These structures have their own elements defined as structures in many cases. All structures and APIs described are defined in the `sf_wifi_onchip_stack_api.h` header file.

- On-chip stack API control structure `sf_wifi_onchip_stack_ctrl_t`
  This control parameter structure is the pointer to the user-provided storage for the Wi-Fi module On-chip stack control structure and can be an input or output depending on how the API uses it.

```
   /** WiFi Framework control structure */
typedef struct st_sf_wifi_onchip_stack_ctrl
{
    /** Storage for information needed for each WiFi device driver in the system. */
    sf_wifi_instance_t * p_lower_lvl_wifi;    ///< Pointer to SF WiFi instance
} sf_wifi_onchip_stack_ctrl_t;
```

Figure 13. On-chip networking stack support control structure

- On-chip stack API configuration structure `sf_wifi_onchip_stack_cfg_t`
  This configuration parameter structure is the pointer to the user-defined Wi-Fi module configuration structure and is an input to several of the APIs described in the following section.

```
/** Define the WiFi configuration parameters */
      typedef struct st_sf_wifi_onchip_stack_cfg
      {
          sf_wifi_instance_t const    * p_lower_lvl_wifi;    ///< Pointer to SF WiFi
      instance
          void                        * p_extend;            ///< Extended configuration
      } sf_wifi_onchip_stack_cfg_t;
```

Figure 14. On-Chip Networking Stack support configuration structure

- On-chip stack API structure `sf_wifi_onchip_stack_api_t`
  This structure includes all the Wi-Fi framework On-chip stack API function pointers.

```
/** Framework API structure.  Implementations will use the following API. */
typedef struct st_sf_onchip_stack_wifi_api
{
   ssp_err_t (*open)(sf_wifi_onchip_stack_ctrl_t * p_ctrl,
sf_wifi_onchip_stack_cfg_t const * const p_cfg);
   ssp_err_t (*close)(sf_wifi_onchip_stack_ctrl_t * const p_ctrl);
   ssp_err_t (*dhcpServerStart)(sf_wifi_onchip_stack_ctrl_t * const p_ctrl,
   sf_wifi_ip_addr_t const * const p_start_ip,
               sf_wifi_ip_addr_t const * const p_end_ip);
   ssp_err_t (*dhcpServerStop)(sf_wifi_onchip_stack_ctrl_t * const p_ctrl);
   ssp_err_t (*versionGet)(ssp_version_t * const p_version);
} sf_wifi_onchip_stack_api_t;
```

Figure 15. On-chip networking stack support API structure

3.2.2 On-chip Network Stack Support APIs

Table 4 shows a complete list of the available APIs, an example API call, and a short description of the input/output of the APIs.

Table 4. On-chip Network Stack Support APIs

Function Name	API Prototype and Description

.open	`ssp_err_t (*open)(sf_wifi_onchip_stack_ctrl_t * p_ctrl, sf_wifi_onchip_stack_cfg_t const * const p_cfg);`
	The open API initializes the driver configuration, enables the driver link, enables interrupts, and makes the device ready for data transfer.
	[in]
	`p_ctrl` see `sf_wifi_onchip_stack_ctrl_t`
	[in, out]
	`p_cfg` see `sf_wifi_onchip_stack_cfg_t`

.close	`ssp_err_t (*close)(sf_wifi_onchip_stack_ctrl_t * const p_ctrl);`
	The close API de-initializes the network interface and can put it in low power mode or power it off. This API closes the Wi-Fi device driver, disables the driver link, and disables the interrupts.
	[in,out]
	`p_ctrl` see `sf_wifi_onchip_stack_ctrl_t`

.IpAddressCfg	`ssp_err_t (*ipAddressCfg)(sf_wifi_onchip_stack_ctrl_t * const p_ctrl, sf_wifi_onchip_stack_ip_cfg_t * const p_cfg);`
	The IpAddressCfg API configures the IP address of the interface.
	[in]
	`p_ctrl` see `sf_wifi_onchip_stack_ctrl_t`
	`p_cfg` `sf_wifi_onchip_stack_cfg_t`

Function Name	API Prototype and Description

.dhcpServerStart	`ssp_err_t (*dhcpServerStart)(sf_wifi_onchip_stack_ctrl_t * const p_ctrl, sf_wifi_ip_addr_t const * const p_start_ip, sf_wifi_ip_addr_t const * const p_end_ip);`
	
	The `dhcpServerStart` API starts the DHCP server on the interface (when configured in AP mode) using the on-chip networking stack.
	[in]
	`p_ctrl` see `sf_wifi_onchip_stack_ctrl_t`
	`p_start_ip` pointer to the start IP address structure
	`p_end_ip` pointer to the end IP address structure

.dhcpServerStop	`ssp_err_t (*dhcpServerStop)(sf_wifi_onchip_stack_ctrl_t * const p_ctrl);`
	The `dhcpServerStop` API stops the DHCP server on the interface (when configured in AP mode) using the on-chip networking stack.
	[in]
	`p_ctrl` see `sf_wifi_onchip_stack_ctrl_t`

.versionGet	`ssp_err_t (*versionGet)(ssp_version_t * const p_version);` The `versionGet` API gets the version and stores it in the user-provided pointer. [in,out] `p_version` User-provided buffer to hold the version information.

Note: For details on operations and definitions for the function data structures, typedefs, defines, API data, API structures, and function variables, see the module API References in the *SSP User's Manual*.

For the On-chip Networking Stack support API return status, see Table 2 in this document.

3.3 BSD Socket APIs (Path 2)

This section introduces the BSD Socket API interface provided by the Wi-Fi framework in the Path 2. This API interface requires the Wi-Fi module/driver to provide support for on-chip networking stack and BSD socket APIs. When the application uses these APIs, it is using the on-chip networking stack present on the Wi-Fi chipset, and not the NSAL (the networking stack running on the Synergy MCU).

3.3.1 BSD Socket Instance

You must define the Synergy Wi-Fi framework BSD Socket instance before using it. The Synergy Wi-Fi framework BSD socket instance refers to the Wi-Fi module specific control data, configuration data, and APIs. The application uses this instance to perform operation on the Wi-Fi module.

Following are members of the Synergy Wi-Fi framework instance (`sf_socket_instance_t`).

```
/** This structure encompasses everything that is needed to use an instance
of this interface. */
typedef struct st_sf_socket_instance
{
    sf_socket_ctrl_t       * p_ctrl;    ///< Pointer to the control structure
for this instance
    sf_socket_cfg_t  const * p_cfg;     ///< Pointer to the configuration
structure for this instance
    sf_socket_api_t  const * p_api;     ///< Pointer to the API structure for
this instance
} sf_socket_instance_t;
```

Figure 16. BSD Socket Interface instance

3.3.2 Structures used in the BSD Socket APIs

- BSD Socket interface control structure `sf_socket_ctrl_t`

```
/** Socket Interface control structure */
typedef struct st_sf_socket_ctrl
{
    sf_wifi_onchip_stack_instance_t * p_lower_lvl_onchip_wifi; ///< low level wifi
interface
    } sf_socket_ctrl_t;
```

Figure 17. BSD Socket interface control structure

- BSD Socket interface configuration information structure `sf_socket_cfg_t`

```
/** Socket Interface configuration structure */
typedef struct st_sf_socket_cfg
{
    sf_wifi_onchip_stack_instance_t * p_lower_lvl_onchip_wifi; ///< Pointer
to SF on-chip stack instance
    void                            * p_extend;                ///< Extended
configuration
} sf_socket_cfg_t;
```

Figure 18. BSD Socket interface configuration structure

- BSD Socket interface API structure `sf_socket_api_t`

```
///< This API will not be used as part of the Socket's API but it will be called
beforehand to ensure stack is setup
/** Socket Interface API */
typedef struct st_sf_socket_api{
    ssp_err_t (*open)(sf_socket_ctrl_t * p_ctrl, sf_socket_cfg_t const * const
    p_cfg);
    ssp_err_t (*close)(sf_socket_ctrl_t * const p_ctrl);
    ssp_err_t (*versionGet)(ssp_version_t * const p_version);
    } sf_socket_api_t;
```

Figure 19. BSD Socket interface API structure

3.3.2.1 Wi-Fi Framework BSD Socket Interface APIs

Table 5 shows a complete list of the available APIs, an example API call, and a short description of the input/output of the APIs. The BSD socket APIs included in this table are created by the Wi-Fi framework.

Table 5. Wi-Fi framework BSD Socket interface APIs

Function Name	Example API Call and Description

.open	`ssp_err_t (*open)(sf_socket_ctrl_t * p_ctrl, sf_socket_cfg_t const * const p_cfg);`
	The open API initializes the driver configuration, enables the driver link, enables interrupts, and makes the device ready for data transfer.
	[in]
	`p_cfg` see `sf_socket_cfg_t`
	[in,out]
	`p_ctrl` see `sf_socket_ctrl_t`

Function Name	Example API Call and Description

.close	`ssp_err_t (*close)(sf_socket_ctrl_t * const p_ctrl);`
	The close API de-initializes the network interface and can put it in low power mode or power it off. This API closes the Wi-Fi device driver, disables the driver link, and disables the interrupts.
	[in,out]
	`p_ctrl` see `sf_socket_ctrl_t`

.versionGet	`ssp_err_t (*versionGet)(ssp_version_t * const p_version);` The `versionGet` API gets the version and stores it in the user-provided pointer. [in,out] `p_version` User-provided buffer to hold the version information.

Note: For details on operations and definitions for the function data structures, typedefs, defines, API data, API structures, and function variables, see the module API References section in the *SSP User's Manual*.

For the On-chip Networking Stack support API return status, see Table 2 in this document.

Standard Socket APIs

Application can use the following APIs to perform data transfer using sockets. This includes socket APIs that are compliant with BSD APIs.

- socket
- close
- bind
- listen
- accept
- connect
- send
- recv
- recvfrom
- sendto
- setsockopt
- getsockopt
- select.

These APIs are standard BSD APIs that can be referenced from many open sources including those specified in the API References section for the associated module in the *SSP User's Manual*. Not all Wi-Fi chipsets provide extra on-chip support.

4. Adding a Wi-Fi Framework Module in an Application

This section describes how to add a Wi-Fi framework module in an application using the Synergy configurator.

Note: This section assumes you are familiar with creating a project, adding threads, adding a stack to a thread, and configuring a block within the stack. If you are unfamiliar with any of these items, refer to the first few chapters of the *SSP User's Manual* to learn how to perform these important steps when creating SSP-based applications.

Figure 20 shows that when creating a Wi-Fi framework-based application, start by adding a new thread to project.

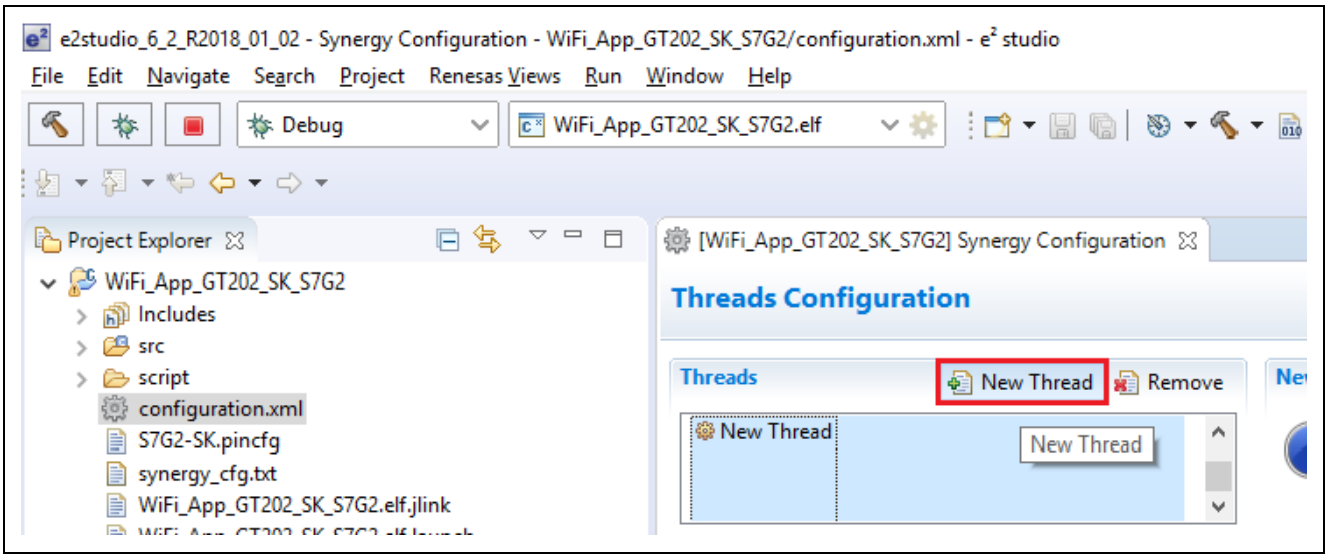

Figure 20. Adding a new Thread

Follow section 4.1 and section 4.2 to add the NetX path and on-chip networking support path.

4.1 Adding a Wi-Fi Framework Module using NetX (Path 1)

To add a Wi-Fi framework module to an application based on NetX, add a NetX IP Instance to the new thread.

4.1.1 Add the NetX IP instance

Table 6 lists the Add the NetX IP Instance.

Table 6. Wi-Fi framework module selection sequence using NetX in an application

Resource	ISDE Tab	Stacks Selection Sequence
g_ip0 NetX IP Instance	Threads	New Stack> X-Wave> NetX> NetX IP Instance

Figure 21 shows how to click the Thread pane from the Synergy configurator and select the NetX IP Instance stack.

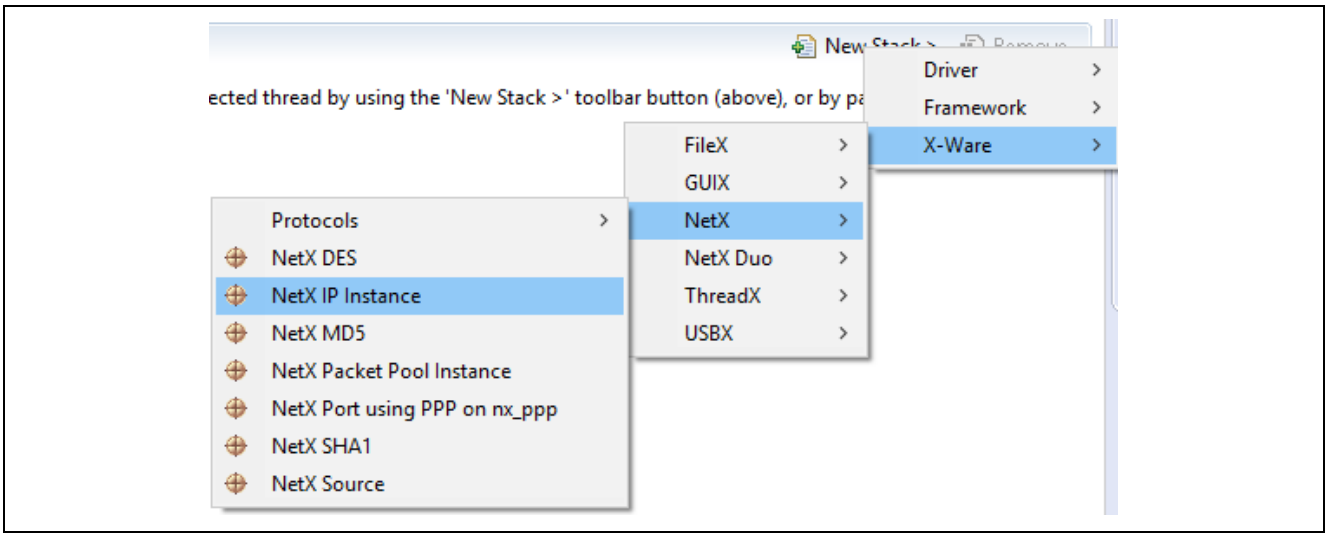

Figure 21. Adding NetX IP Instance

4.1.2 Adding the NetX Port using Wi-Fi Framework on sf_wifi_nsal_nx

Figure 22 shows when the NetX IP instances are added to the thread stack, the Synergy configurator automatically adds the NetX common and NetX Packet Pool instance. The Synergy configurator also added a module with a Pink band “Add NetX Network Driver.” This is where the Wi-Fi framework on NetX is pulled in. This adds the TCP/IP stack to the user application.

Table 7. Adding NetX port using Wi-Fi-framework on sf_wifi_nsal_nx

Resource	ISDE Tab	Stacks Selection Sequence
NetX Port using Wi-Fi framework on `sf_wifi_nsal_nx`	Thread	Add NetX Network Driver>NetX Port using Wi-Fi framework on `sf_wifi_nsal_nx`

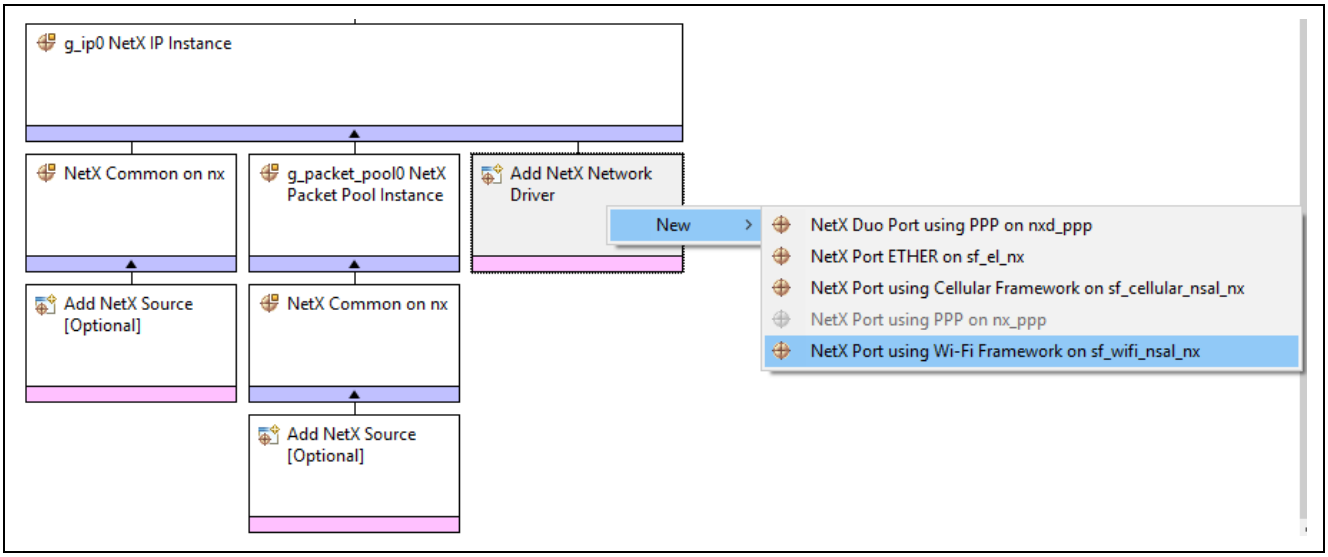

Figure 22. Adding NetX Port using Wi-Fi Framework on sf_wifi_nsal_nx

Adding the Wi-Fi framework on `sf_wifi_nsal_nx` adds the GT202 Wi-Fi Device Driver on `sf_wifi_gt202`. In addition Figure 23 shows the Synergy configurator has also added a module with a pink band. “Add SPI Driver.” It adds this module because the GT202 uses the SPI port to communicate with the MCU.

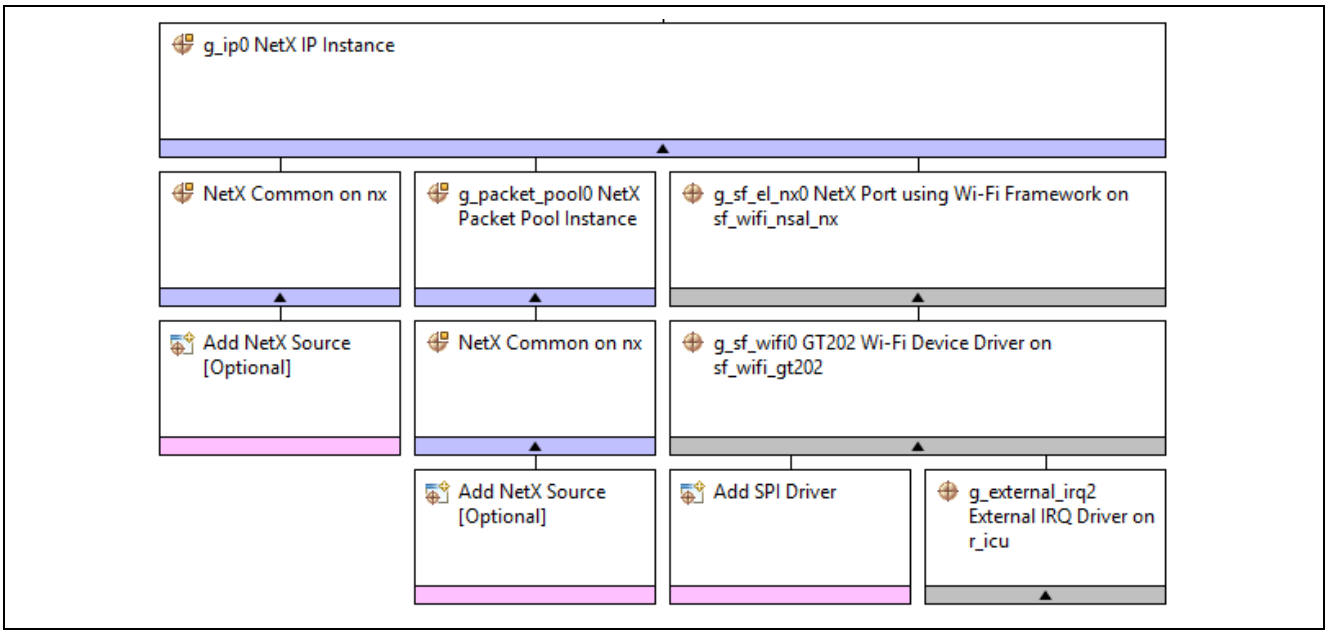

Figure 23. Wi-Fi Framework Configurator with NetX

4.2 Adding the Wi-Fi Framework Module using On-chip Wi-Fi Stack (Path 2)

Figure 20 shows that to add the Wi-Fi framework module using the On-chip Wi-Fi Stack in an application, the user can add a new thread, and then add the BSD Socket using GT202 On-chip Stack to the new thread as listed in Table 8.

Table 8. Wi-Fi framework module selection using On-Chip networking stack in an application

Resource	ISDE Tab	Stacks Selection Sequence
g_sf_socket0 BSD Socket using GT202 On-Chip Stack on GT202 Wi-Fi Framework	Threads	New Stack> Framework> Networking> Wi-Fi> BSD Socket using GT202 On-Chip Stack on GT202 Wi-Fi Framework

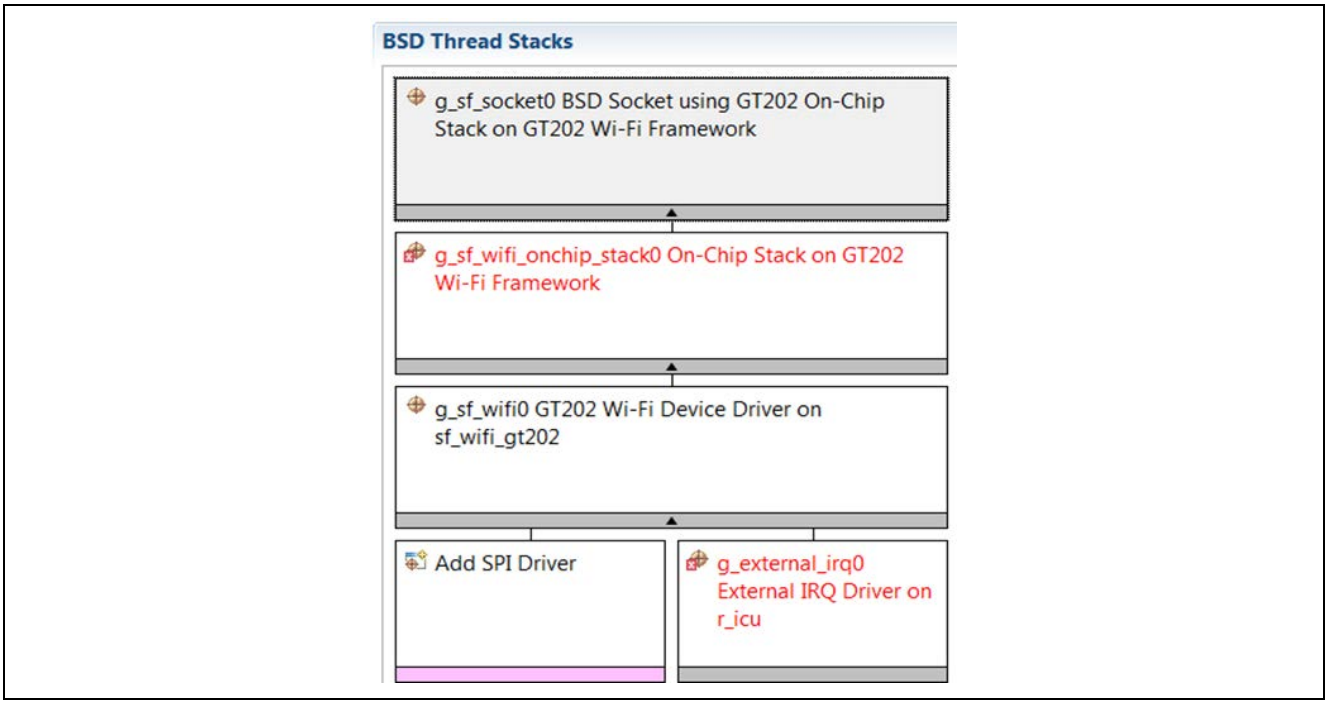

Figure 24. Including the On-chip Networking Stack Support

5. Configuring the Wi-Fi Framework Module

Figure 25 shows when the Wi-Fi application is added to a thread, the Wi-Fi thread properties need to be properly configured as indicated in Table 9.

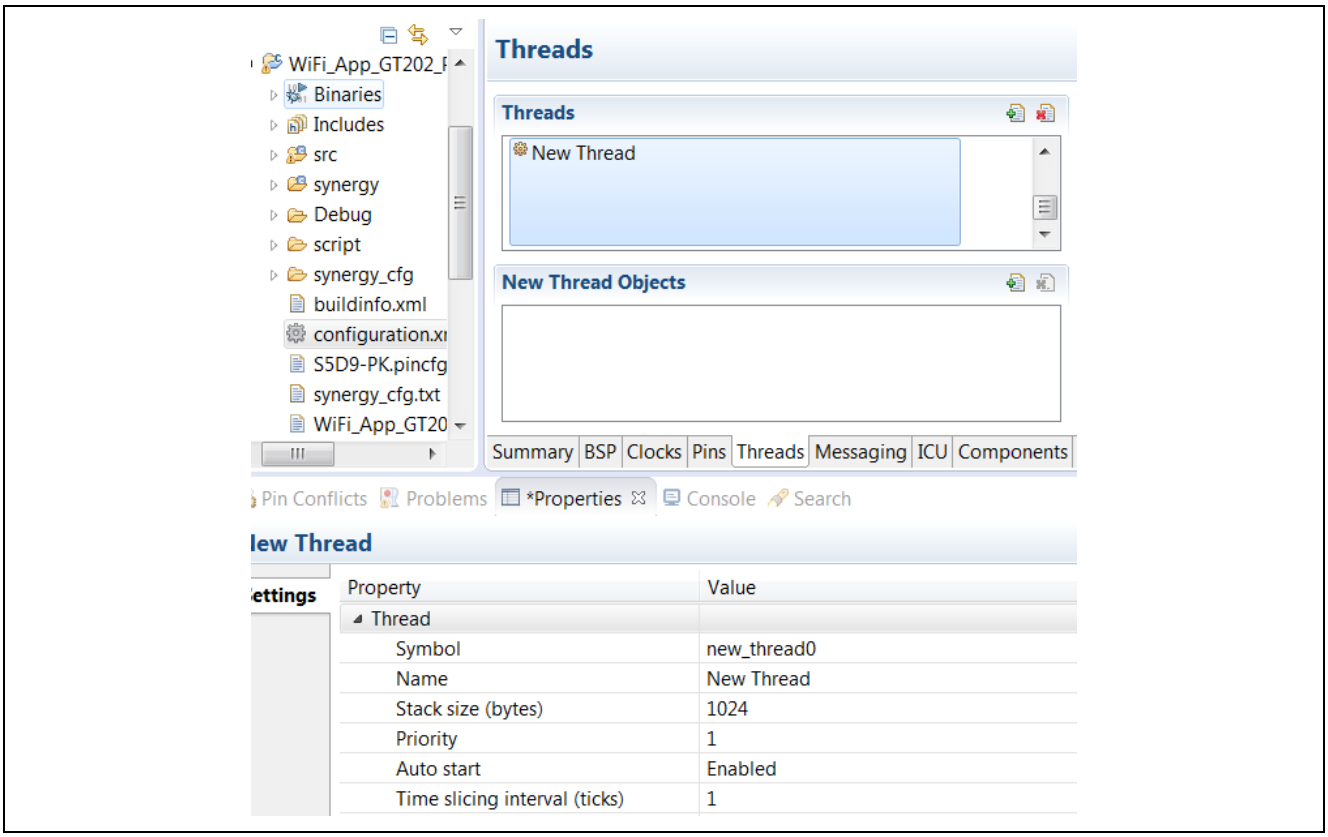

Figure 25. New thread configuration

The following table has the available configurations for the new thread.

Table 9. Configurations for the new thread

ISDE Property	Value	Description
Symbol	new_thread0 (default)	User can specify different name.
Name	New Thread (default)	User can specify different name.
Stack size (bytes)	1024 (default)	Application dependent.
Priority	1 (default)	User can adjust this priority based on specific application.
Auto start	Enabled (default)	User can adjust this setting based on the application implementation
Time slicing interval (ticks)	1 (default)	User can adjust this priority based on specific application.

5.1 Wi-Fi Framework Configurations using NetX (Path 1)

Available configurations for the Wi-Fi application using NetX are described in the following subsections.

5.1.1 Configurations for the NetX IP Instance

Refer to Figure 26 for configurations to Adding the NetX IP Instance in this document.

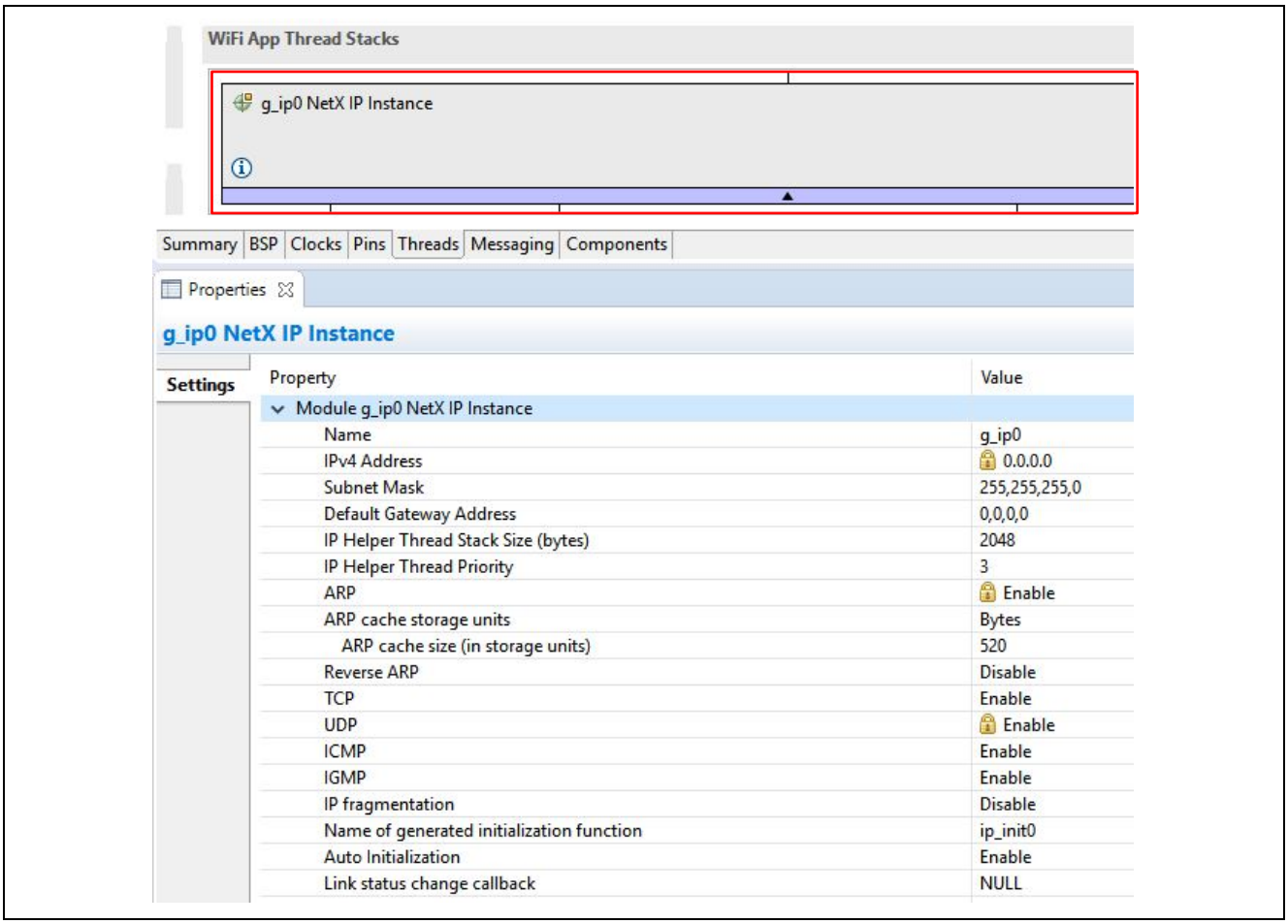

Figure 26. Configurations for the NetX IP Instance

Table 10 lists detailed descriptions for the various configuration properties.

Table 10. Configuration settings for Wi-Fi framework module on NetX

ISDE Property	Value	Description
Name	Default: `g_ip0`	NetX IP instance name
Ipv4 Address	Can be static or dynamic Default: 192,168,0,2	IP address for the NetX stack. Lookback can be done by either using the same address or by using 127.*.*.*
Subnet Mask	Default: 255,255,255,0	
IP Helper Thread Stack Size	Default: 2048	Each IP instance has a helper thread. The first processing in the IP helper thread is to finish the network driver initialization associated with the IP create service. After the network driver initialization is complete, the helper thread starts an endless loop to process packet and periodic requests. If unexpected behavior occurs within the IP helper thread, the first debugging step is to increase its stack size during the IP create service. If the stack is too small, the IP helper thread might overwrite memory, which can cause unexpected errors.
IP Helper Thread Priority	Default: 3	User can adjust this priority based on specific application.

ISDE Property	Value	Description
ARP	Default: Enable	Address Resolution Protocol. ARP should always be enabled.
ARP Cache Size in Bytes	Default: 512	The ARP cache can be viewed as an array of internal ARP mapping data structures. Each internal structure is capable of maintaining the association between an IP address and a physical hardware address. In addition, each data structure contains link pointers to be part of multiple linked lists.
Reverse ARP	Default: Disable	Selection is based on the application requirement.
TCP	Default: Enable	Selection is based on the application requirement.
UDP	Default: Enable	Selection is based on the application requirement.
ICMP	Default: Enable	Before ICMP messages can be processed by NetX, the application must call the ***nx_icmp_enable*** service to enable ICMP processing. Thereafter, the application can issue ping requests and field incoming ping packets.
IGMP	Default: Enable	Before any multicasting activity can take place in NetX, the application must call the ***nx_igmp_enable*** service. This service performs basic IGMP initialization in preparation for multicast requests.
IP fragmentation	Default: Disable	IP fragmentation should be avoided if possible, especially for reliable protocols such as TCP.
Name of generated initialization function	Default: ip_init0	User can specify different name.
Auto Initialization	Default: Enable	Select automatically to initialize the IP instance.

5.1.2 Configuration for Wi-Fi framework module on NetX

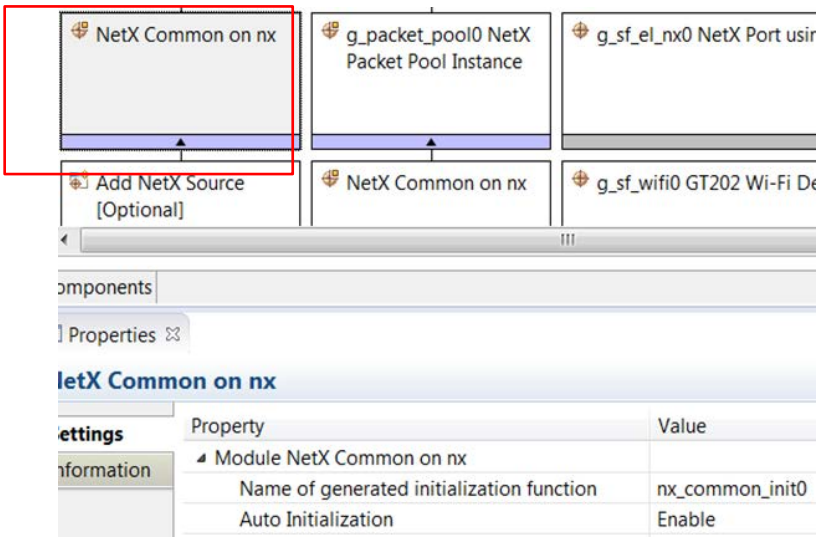

Figure 27. Configurations for the NetX common block

Table 11 lists detailed descriptions for the various configuration properties.

Table 11. Configurations for NetX common module

ISDE Property	Value	Description
Name of generated initialization function	Default: nx_common_init0	User selection for the name of the initialization function.
Auto Initialization	Default: Enable	Select automatically to initialize the NetX common block.

5.1.3 Configurations for NetX Packet Pool Instance

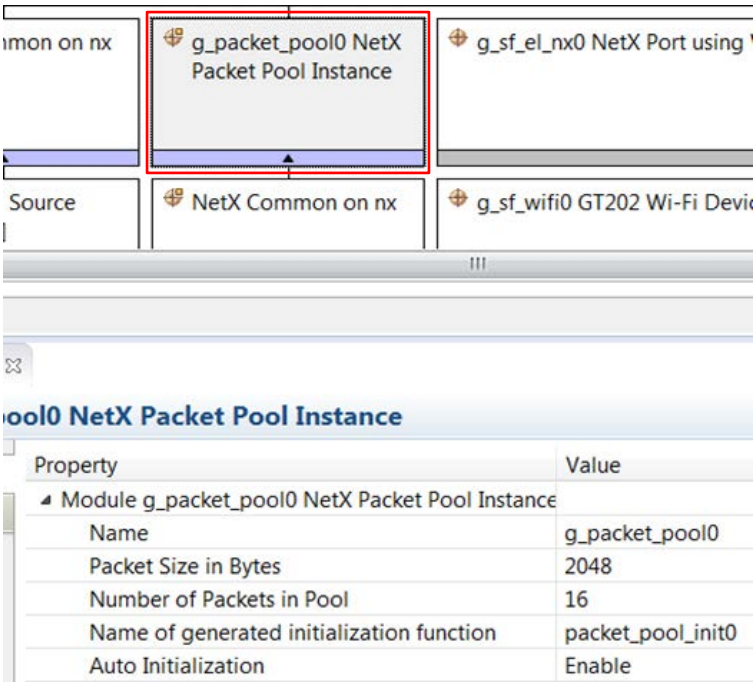

Figure 28. Configurations for NetX packet pool instance

Table 12 lists detailed description for the various configuration properties.

Table 12. NetX pool instance properties

ISDE Property	Value	Description
Name	Default: `g_packet_pool0`	User selection for the name of the packet pool block.
Packet Size in Bytes	Default: 2048	Packet size and number of packets in pool determines the packet pool memory size.
Number of Packets in Pool	Default: 16	Packet pools are created either during initialization or during run time by application threads.
Name of generated initialization function	Default: `packet_pool_init0`	User selection for the name of the initialization function.
Auto Initialization	Default: Enable	Select automatic packet pool creation.

5.1.4 Configurations for the NetX Port

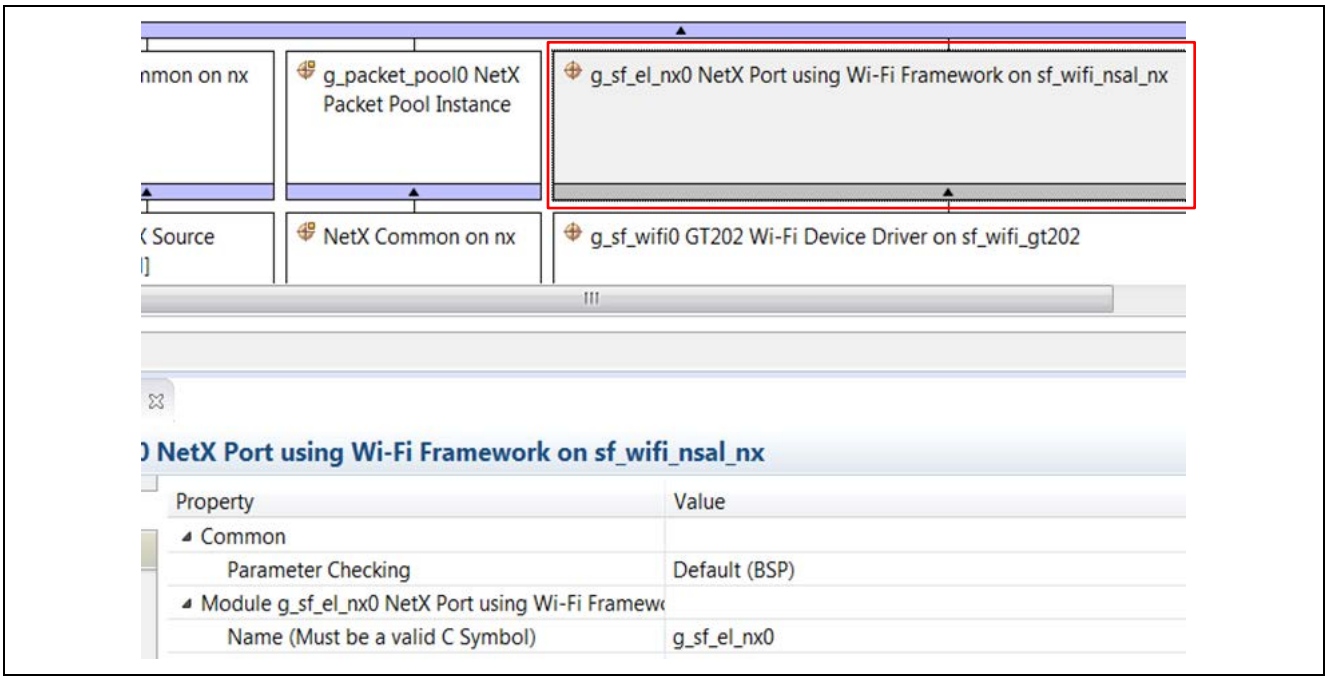

Figure 29. Configurations for the NetX port using Wi-Fi framework on sf_wifi_nsal_nx

Table 13 lists detailed descriptions for the various configuration properties.

Table 13. Properties for NetX port using Wi-Fi framework on sf_wifi_nsal_nx

ISDE Property	Value	Description
Parameter Checking	Default: Default (BSP)	This feature can be disabled when testing is complete to save code space and to speed up execution.
Name	Default: `g_sf_el_nx0`	User selection for the name of the NetX port.

5.1.5 Configurations for the Wi-Fi Module Device Driver

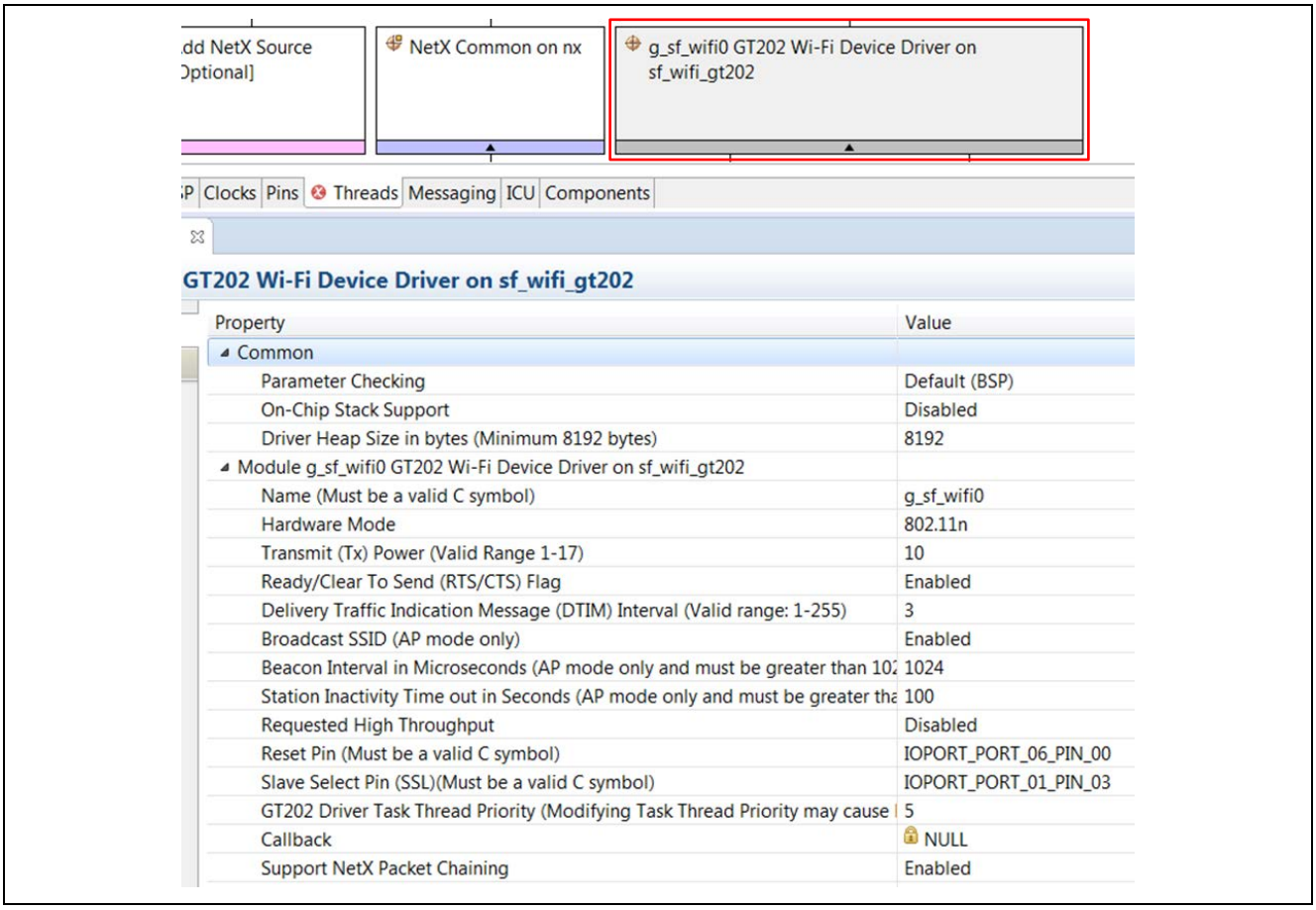

Figure 30. Configurations for the Wi-Fi module device driver

Table 14 lists detailed descriptions for the various configuration properties.

Table 14. Wi-Fi Device Driver configurations

ISDE Property	Value	Description
Parameter Checking	BSP (Default), Enabled, Disabled	This feature can be disabled when testing is complete to save code space and to speed up execution.
On-Chip Stack Support	Enabled, Disabled (Default) Disabled	Enable the On-chip stack support when using On-chip networking stack.
Driver Heap Size in bytes	Default: 8192	This setting depends on the Wi-Fi driver implementation.
Name	Default: `g_sf_wifi0`	User selection for the name of the Wi-Fi framework instance.
Hardware Mode	Option: 802.11a/b/g/n Default: 802.11n	Hardware mode of the Wi-Fi module. 802.11ac is not supported.
Transmit (Tx) Power (Valid Range 1-17)	Default: 10	Transmit power in dBm.
Ready/Clear To Send	Enabled (Default), Disabled	RTS/CTS enable
Delivery Traffic Indication Message (DTIM)	3	Enable DTIM if network statistics is required. Valid range is 1-255.
Broadcast SSID	Enabled (Default), Disabled	SSID broadcast flag. Valid in AP-mode only.

ISDE Property	Value	Description
Beacon Interval in Microseconds	Default: 1024	Beacon interval. Valid in AP-mode only.
Station Inactivity Time out in Seconds	Default: 100	Station inactivity timeout value. Valid in AP-mode only.
Requested High Throughput	Enabled, Disabled (Default)	High-throughput mode. Only valid for 802.11n.
Reset Pin	Default: IOPORT_PORT_06_PIN_00	This setting depends on the hardware configuration.
Slave Select Pin	Default: IOPORT_PORT_01_PIN_03	This setting depends on the hardware configuration.
GT202 Driver Task Thread Priority	Default: 5	User selection is based on application status.
Callback	Default: NULL	When NSAL is used, this callback is locked.
Support NetX Packet Chaining	Enabled (Default), Disabled	NetX packet chaining allows more flexibility for packet transmission.

5.2 Wi-Fi Framework Configurations using On-chip Networking Stack Support (Path 2)

5.2.1 Configurations for the BSD Socket

When the BSD Socket stack is created, the Synergy configurator provides the following default configurations.

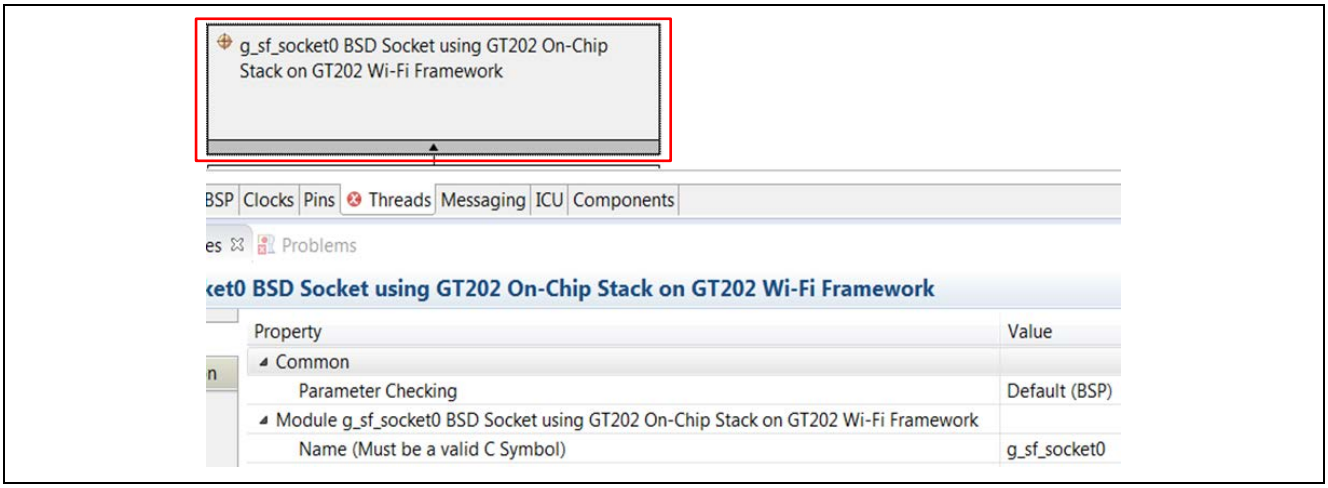

Figure 31. Configurations for the BSD Socket

Table 15 lists detailed descriptions for the various configuration properties.

Table 15. BSD socket configurations

ISDE Property	Value	Description
Parameter Checking	BSP (Default), Enabled, Disabled	This feature can be disabled when testing is complete to save code space and to speed up execution.
Name	Default: `g_sf_socket0`	User selection for the name of the BSD socket interface.

5.2.2 Configurations for the On-chip Stack

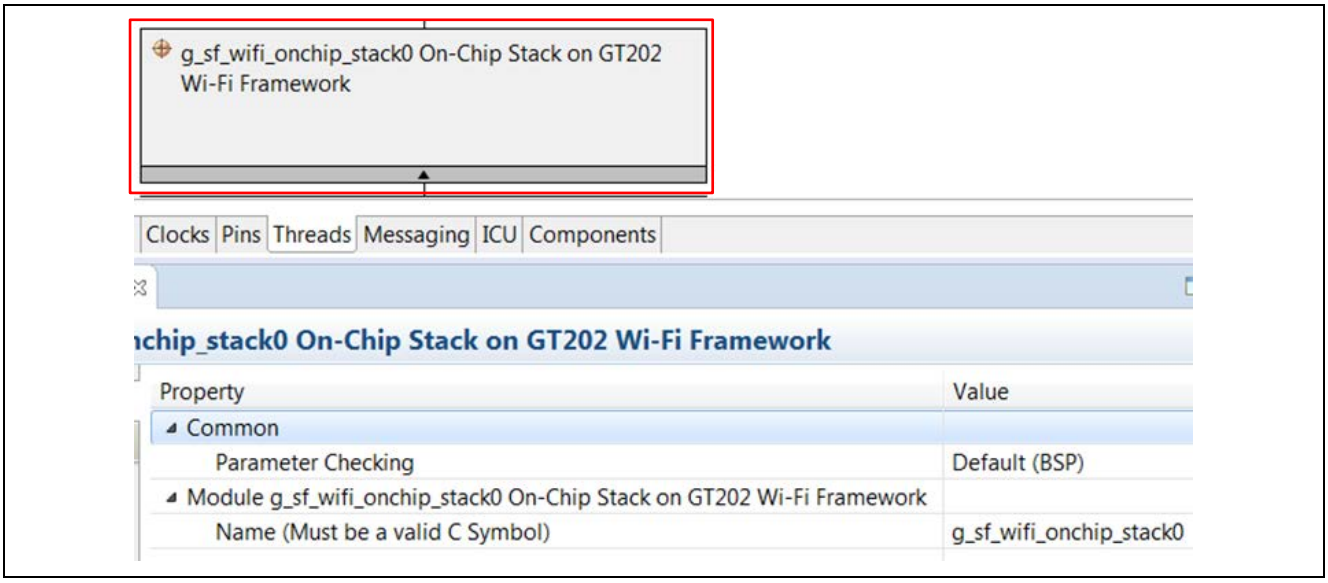

Figure 32. Configurations for the On-chip stack on GT202

Table 16 lists detailed descriptions for the various configuration properties.

Table 16. On-chip stack support configurations

ISDE Property	Value	Description
Parameter Checking	BSP (Default), Enabled, Disabled	This feature can be disabled when testing is complete to save code space and to speed up execution.
Name	Default: `g_sf_wifi_onchip_stack0`	User selection for the on-chip networking stack.

5.2.3 Configurations for the Wi-Fi Module Device Driver

On-chip networking stack Support uses the same Wi-Fi module device driver as the NetX solution. For the available configurations, refer to section 0.

5.3 Configuration for the Wi-Fi Framework Module Low Level Drivers

This implementation is specific to the Wi-Fi module for use with different HAL components such as the SPI, ICU, IOPORT, SDMMC, and DTC. For the GT202 Wi-Fi module, the SSP supports communication with GT202 via SPI interface on the r_rspi or r_sci_spi. See the appropriate module guides for these blocks for relevant hardware configuration settings.

6. Using the Wi-Fi Framework Module in an Application

After the Wi-Fi module and Synergy MCU are selected, refer to the flow chart in Figure 33 to start application development. Follow the general steps in this section.

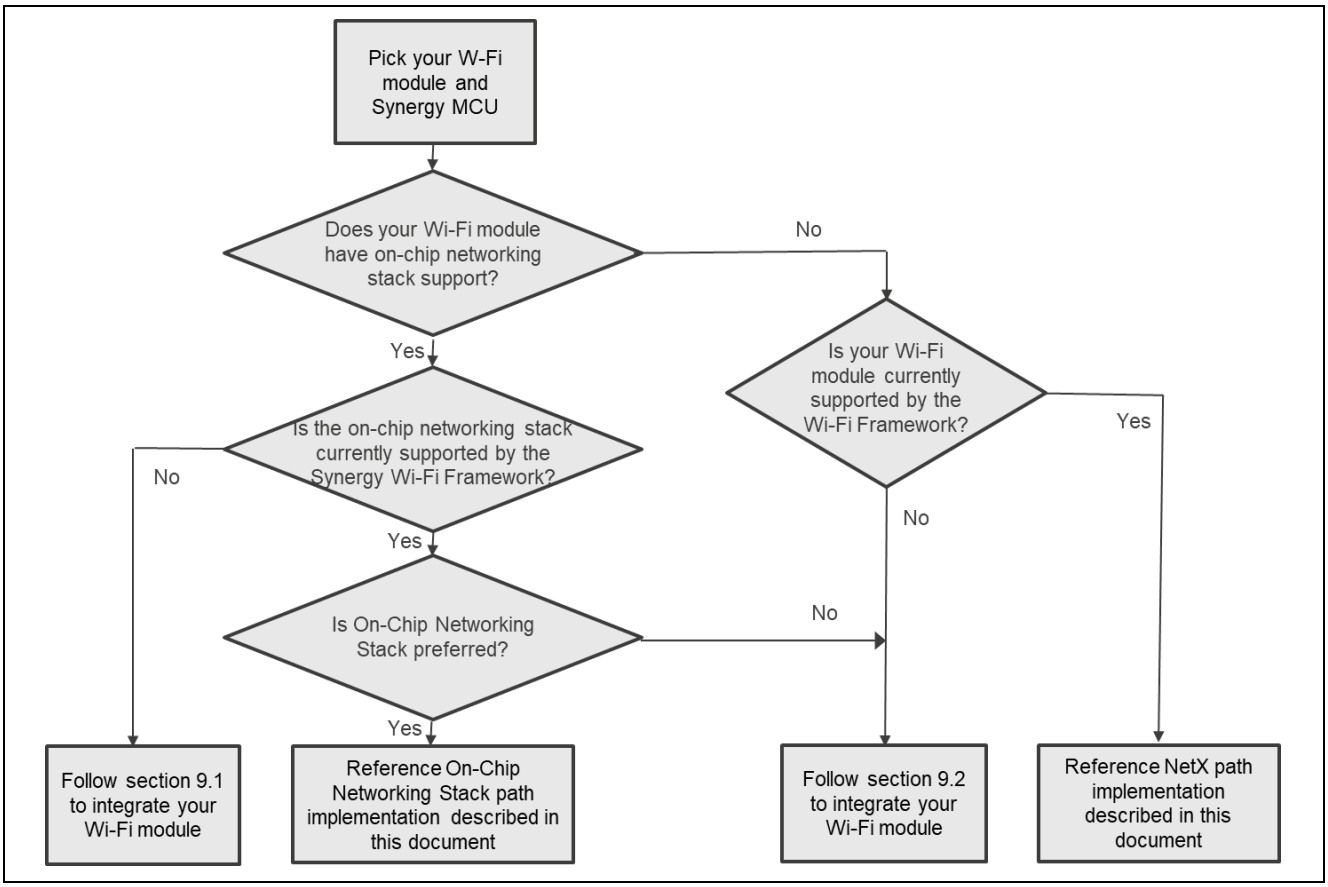

Figure 33. Flowchart for Wi-Fi application path

6.1 Steps when using the Wi-Fi Framework Module with NetX (Path 1)

If your application implements a higher-level application protocols such as the protocols described in the configurator view that follows, start by adding those blocks in the configurator.

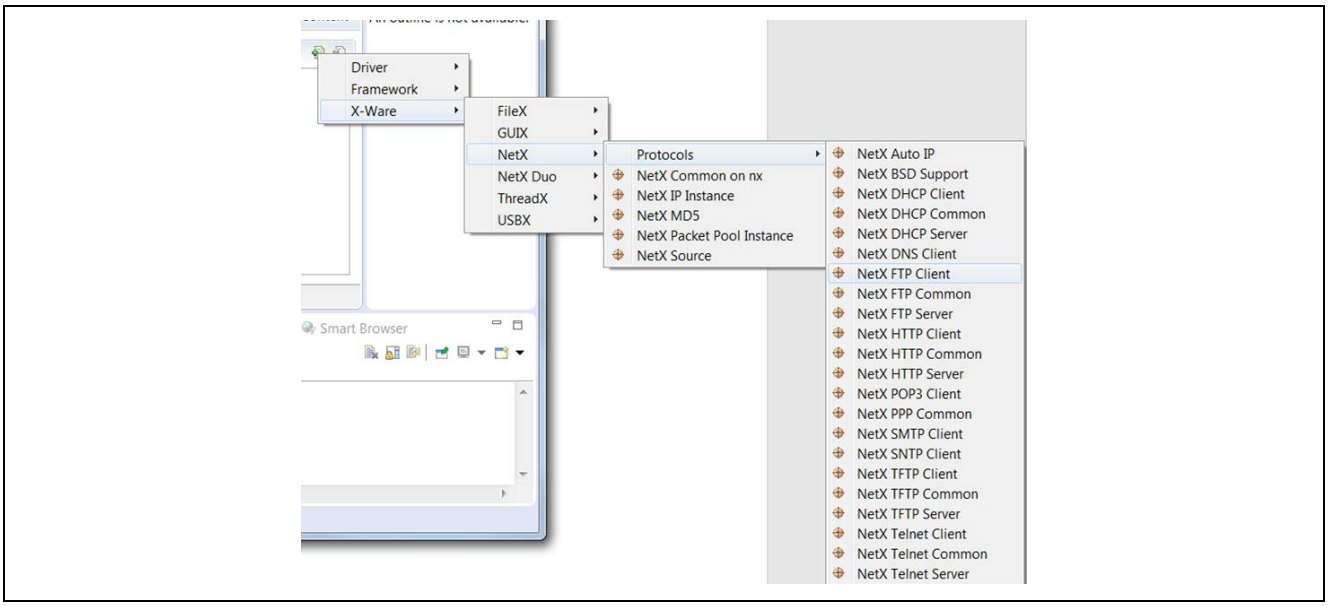

Figure 34. NetX based application protocols

These higher-level protocols pull in the dependency module such as the NetX IP instance, to the project. Section 5.1.1 shows you how to configure the NetX IP instance.

Figure 35 shows the steps in a typical operational flow when using the Wi-Fi framework module with NetX in an application.

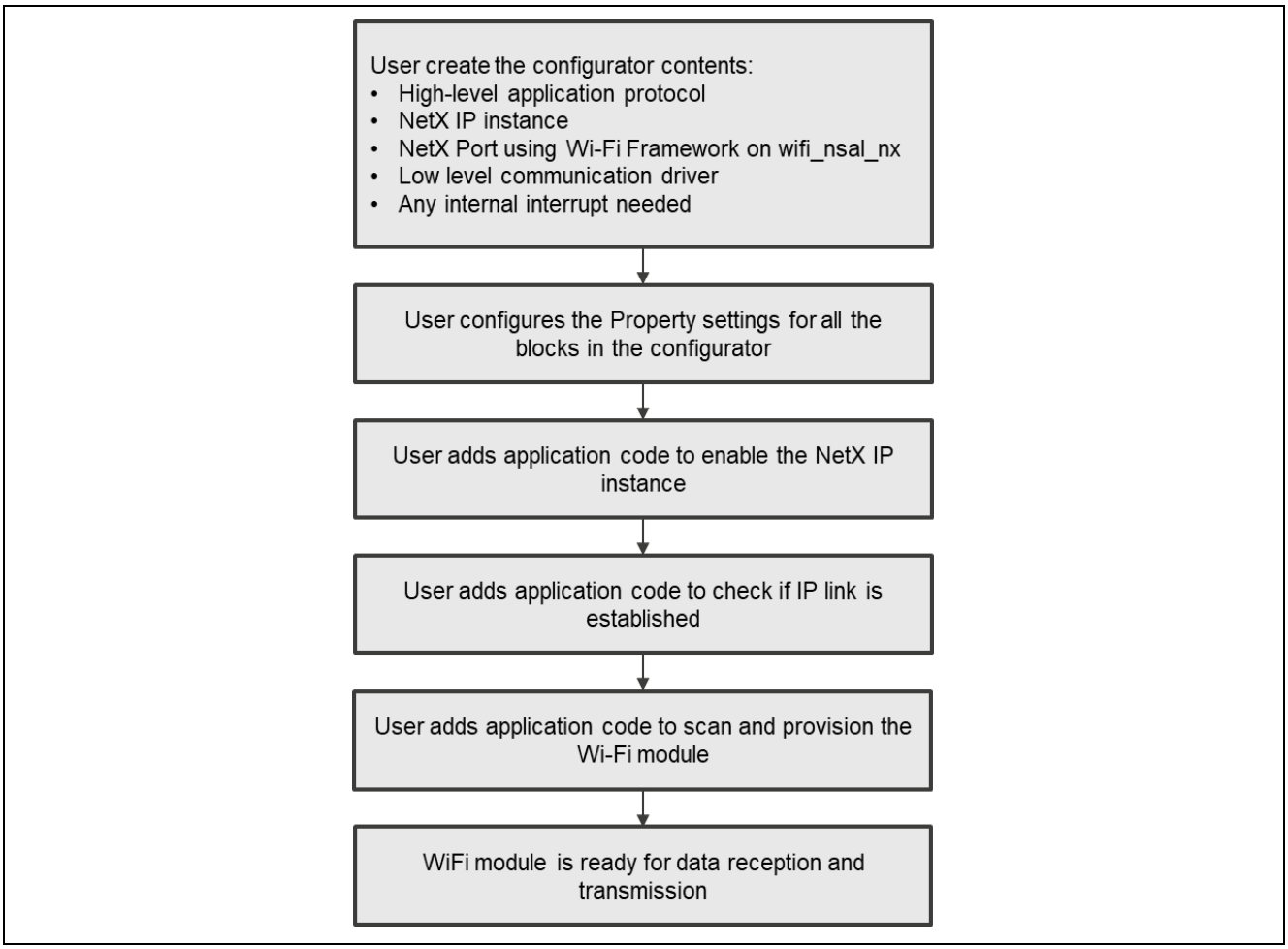

Figure 35. Typical operational flow for Wi-Fi application using NetX

6.2 Using the Wi-Fi Framework Module with On-chip Stack (Path 2)

Figure 36 shows the steps in a typical operational flow when using the Wi-Fi framework module with On-chip stack in an application.

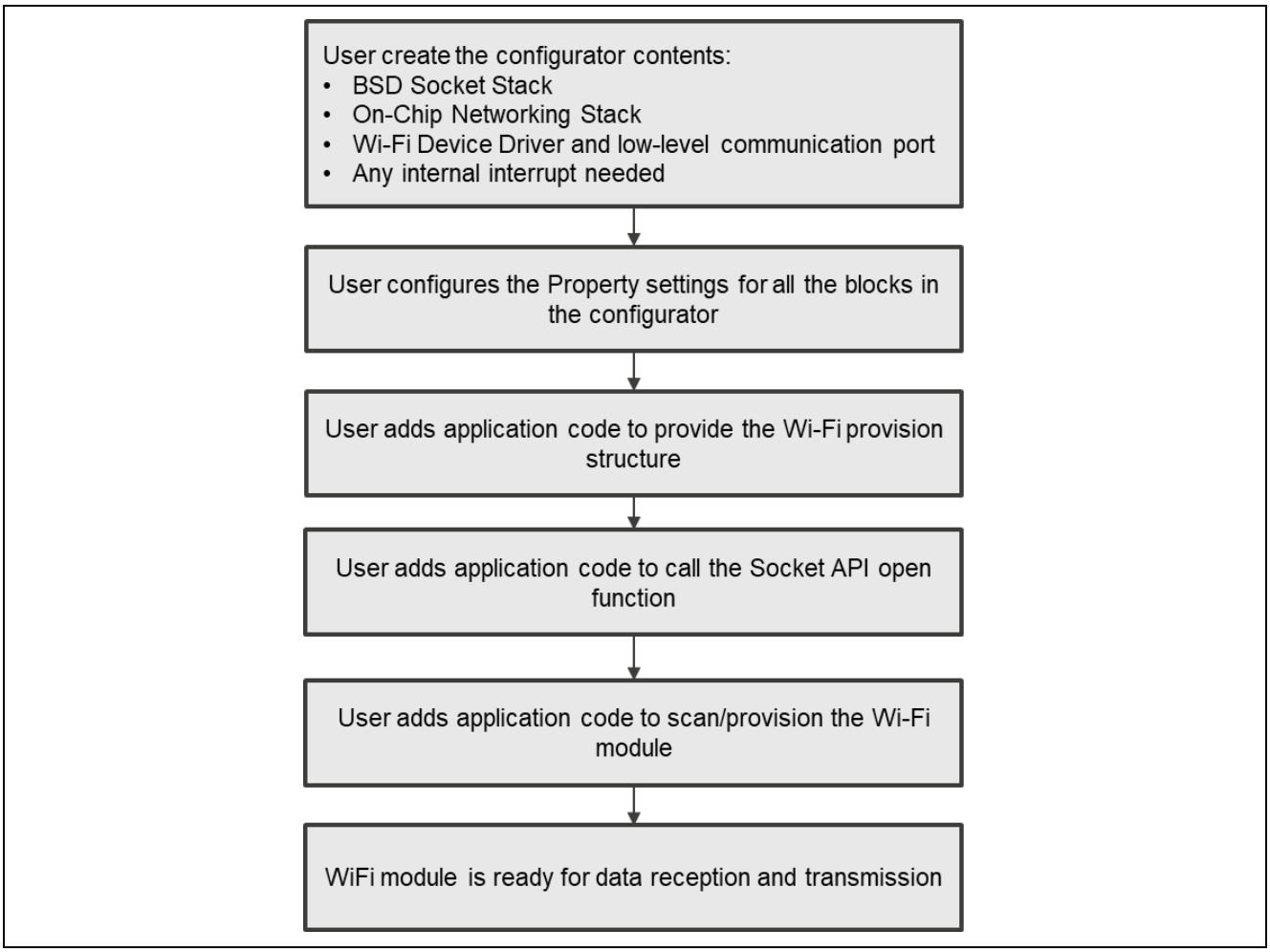

Figure 36. Typical operational flow for Wi-Fi application using On-chip networking stack support

7. The Wi-Fi Framework Module Application Project

The application project associated with this guide demonstrates the typical steps for using the Wi-Fi framework module in an example application. You may want to import and open the application project within ISDE and view the configuration settings for the Wi-Fi framework module. You can also read the code in the `wifi_app_thread_entry.c` file, used to illustrate the Wi-Fi framework module APIs in a complete design.

7.1 Overview of the Application Project

7.1.1 NetX and NSAL Interface using Path 1

This application project is implemented on the SK-S7G2 and PK-S5D9 boards with Longsys Wi-Fi module GT202 using the NetX networking stack. Communication with the GT-202 device driver is through the NSAL interface.

7.1.2 Console Framework User Interface

This application project allows interactions with hardware through the Console framework on USB CDC. A COM port terminal on the PC such as Tera Term, is required to review the Wi-Fi scan result in addition to providing the Wi-Fi module provision information.

7.1.3 DHCP Client Application

This application project initializes the NetX TCP/IP, scans, and provisions the Wi-Fi module. DHCP client is added on top of the NetX interface. The DHCP client layer acquires the IP address from the DHCP server.

7.1.4 Using the Ping Application to Confirm the Connection

To confirm the networking connectivity of the SK-S7G2 and PK-S5D9, ping the acquired IP address.

Note: In this example application project, the On-chip networking stack example is not implemented.

Figure 37 describes the Wi-Fi activity in the example application project.

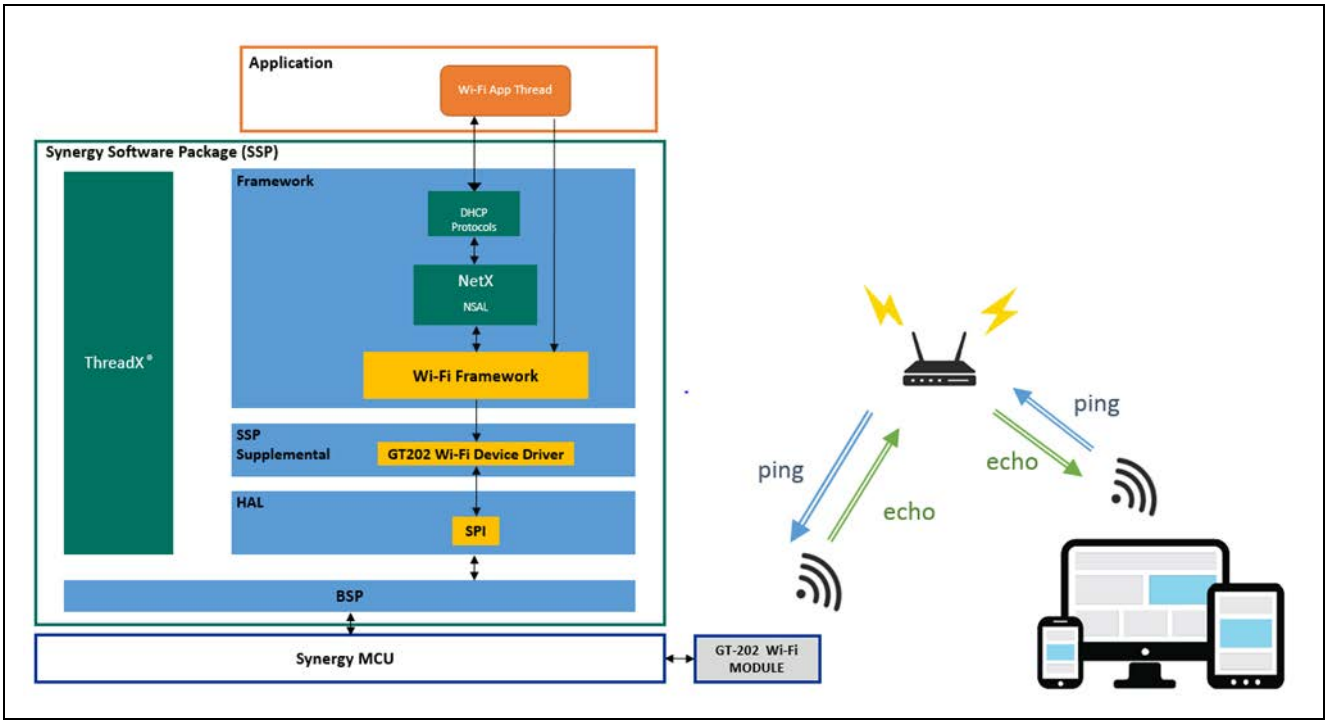

Figure 37. Wi-Fi sample application setup

7.2 Software Architecture

The application project has two threads:

- The Wi-Fi application thread
- The Console framework thread based on the USB CDC device communication framework.

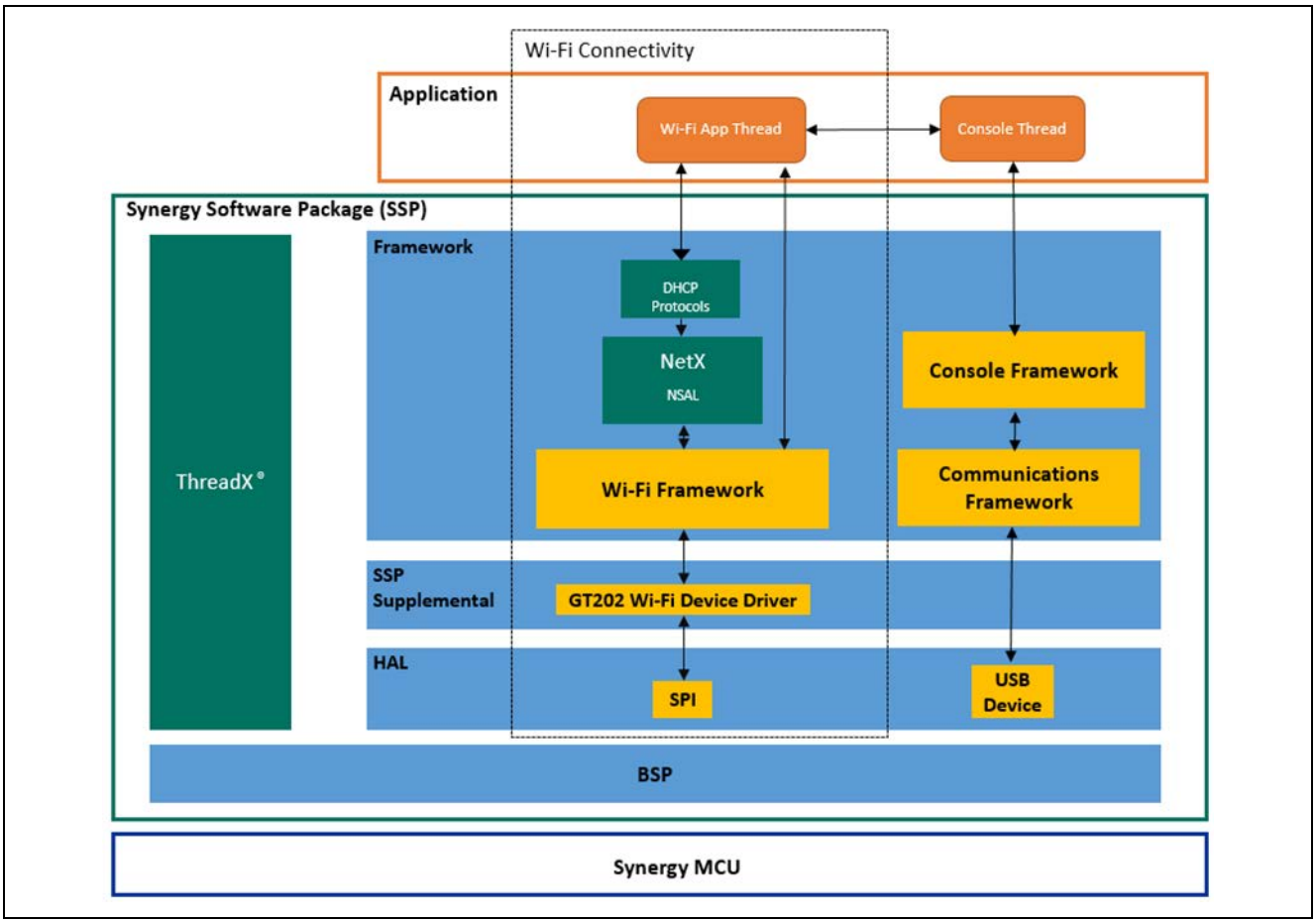

Figure 38. Wi-Fi sample application software architecture

The following sections describe the functionality of the two application threads and their interaction.

7.2.1 Console Application Thread

The Console application thread is a user interface where you can:

- Select the Wi-Fi access point to connect
- Provide the Wi-Fi access point password
- Receive message for confirmation of successful provision
- Receive message for the resolved IP address.

See the *Console Framework Module Guide* for details on designing a Console framework.

Note: With the current USB CDC driver, when testing with Windows 10, you must change the USBX Device Configuration Class Code from **Communications** to **Miscellaneous**. To do this, go to the **Console Thread** in the **Threads** tab and change the **Class Code** property of the USBX Device Configuration.

7.2.2 Wi-Fi Application Thread

The Wi-Fi application thread along with the created configurator code, includes the DHCP client application, with the TCP/IP core stack. In addition, it also includes the Wi-Fi framework, NSAL, and Wi-Fi device driver. Essentially, this thread is responsible for DHCP client, Wi-Fi connectivity, and provisioning of the user code.

7.3 Wi-Fi Framework Module Code Overview

Figure 39 shows the directory structure for the Wi-Fi framework application source code.

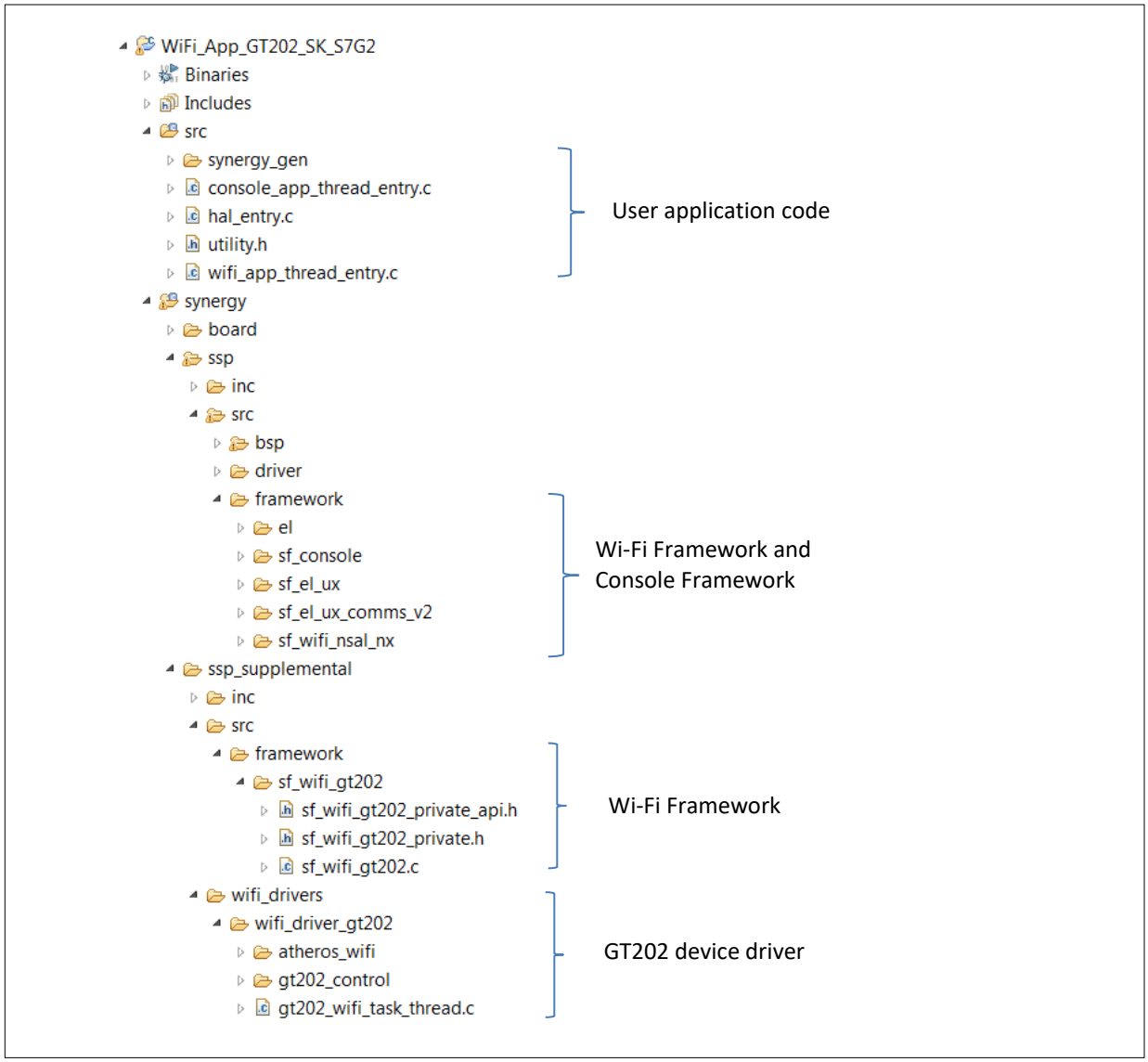

Figure 39. Wi-Fi framework application source code

7.3.1 Configurator generated code (src/synergy_gen folder)

The configurator generated code is part of the `common_data.c/h` and `wifi_app_thread.c/h` in the src/synergy_gen folder. The code in these files, are common code specific to the selected module stack components of the thread. In this case, the common code associated with NetX is available in the `g_common_init()` function. Additionally, in the common code, the NetX driver entry function, packet pool creation for the Wi-Fi framework is available as part of the configurator created code.

7.3.2 User application code (src/wifi_app_thread_entry.c)

In the user application code (`src/wifi_app_thread_entry.c`), the code for checking the status of the IP link creation, provisioning the Wi-Fi with user credentials, starting the DHCP client, and acquiring the IP address from the DHCP server are added. The step sequence is as follows:

1. Check if the IP link is enabled with
   `nx_ip_interface_status_check (&g_ip0, 0, NX_IP_LINK_ENABLED, &ip_status, NX_WAIT_FOREVER)`
2. Provision the Wi-Fi module with
   `g_sf_wifi0.p_api->provisioningSet (g_sf_wifi0.p_ctrl, &g_provision_info);`

Note: This sample application has fixed settings for the following provision configurations:

- Mode: Client
- Security: WPA2
- Encryption: Auto

Slight adjustment of the following code can allow provision for Wi-Fi modules that have different security settings.

```
    /** Set other provisioning configuration */
        g_provision_info.mode = SF_WIFI_INTERFACE_MODE_CLIENT;
        g_provision_info.security = SF_WIFI_SECURITY_TYPE_WPA2;
        g_provision_info.encryption = SF_WIFI_ENCRYPTION_TYPE_AUTO;
```

Figure 40. Example GT202 provisioning settings

3. Start the DHCP client with
   nx_dhcp_start (&g_dhcp_client0);
4. Check the DHCP server to determine if the IP address is resolved with
   nx_ip_status_check (&g_ip0, NX_IP_ADDRESS_RESOLVED, (ULONG *) &status, 10);
5. Acquire the leased IP address with
   nx_ip_interface_address_get (&g_ip0, 0, &ip0_ip_address, &ip0_mask);

The application is then ready for TCP/IP communication with other clients on the Wi-Fi network using the IP address.

7.4 Configurations

The hardware and the application configurations for the property settings in the configurator are described.

For both the SK-S7G2 and PK-S5D9 projects, the following Wi-Fi thread property works well for this application. The user should adjust these settings within their own application.

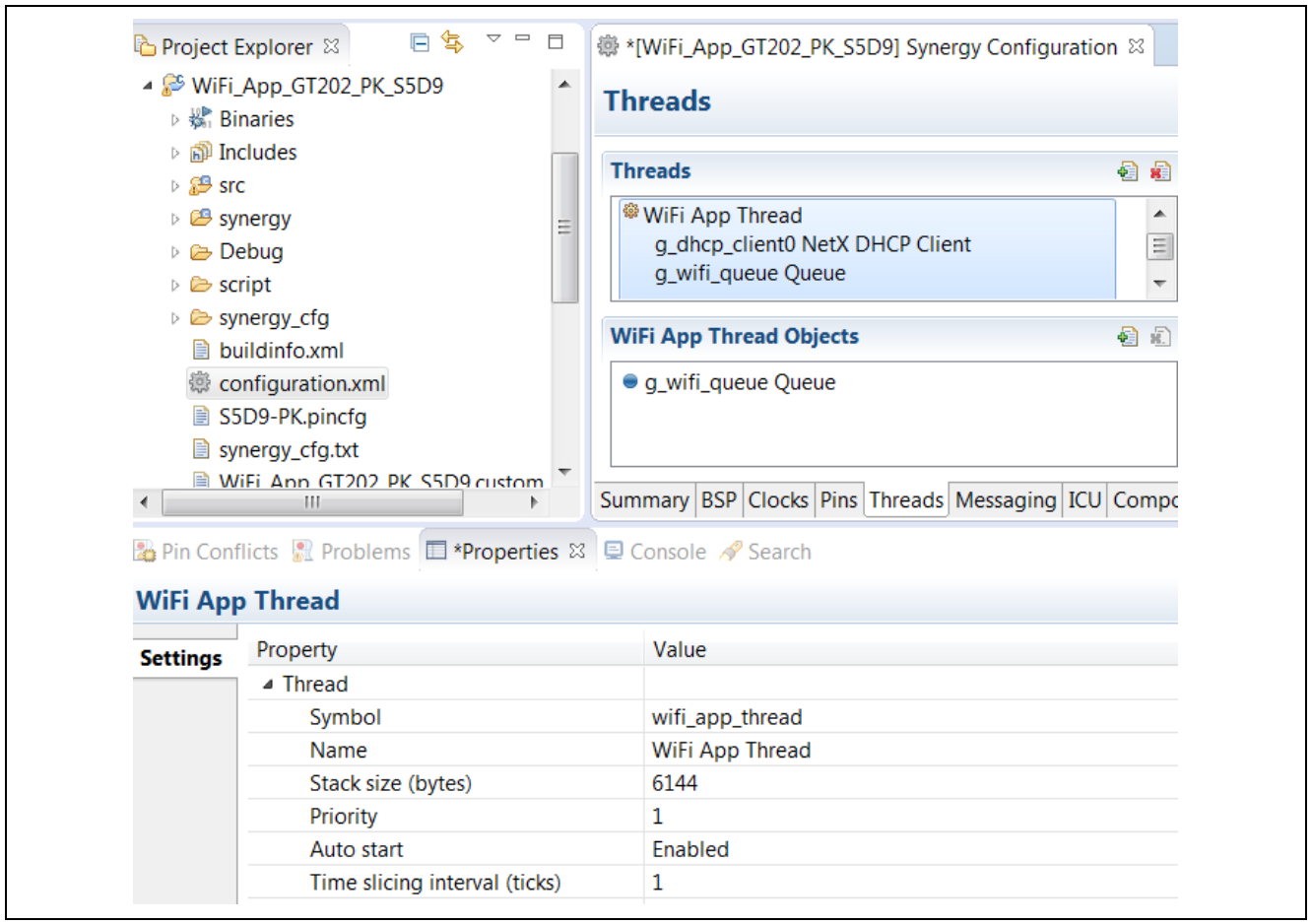

Figure 41. Wi-Fi thread configurations

Notice that the Stack size for the Wi-Fi application project increased from 1024 byte from the default to 6144, this setting provides reliable operation with this application project but may need to be adjust for a new Wi-Fi application.

7.4.1 DHCP Client Configuration

To set up the DHCP client, see Figure 42.

Figure 42. DHCP client configuration

The DHCP Common block is provided automatically by the Synergy configurator when the DHCP client is included. See Figure 43 for the configuration information.

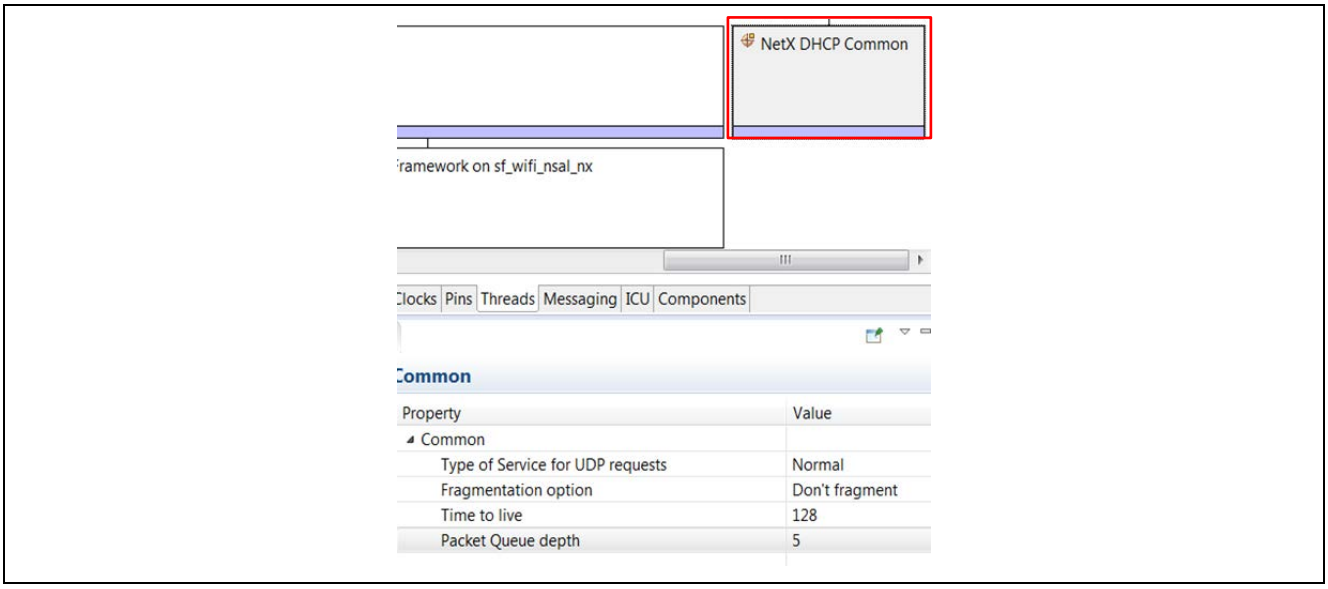

Figure 43. NetX DHCP Common configurations

See the *NetX™ DHCP Client Module Guide* for detailed description of the configuration properties for the DHCP Client block and DHCP Common block. This sample project uses the default settings provided by the configurator for these two blocks.

7.4.2 NetX Related Configurations

See section 5.1 for the available configurations of NetX blocks. In this section, only properties that changed from the default configuration are described in detail. The properties that changed from the default value are marked with the red boxes.

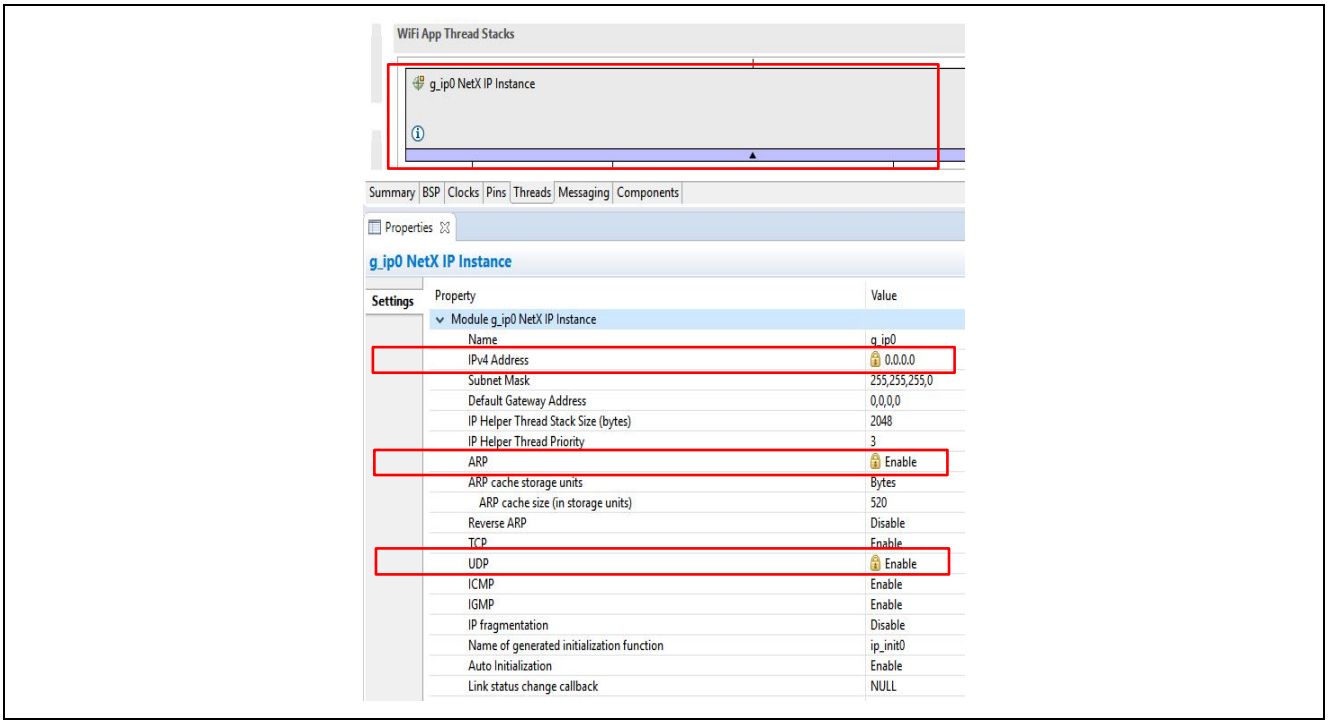

Figure 44. NetX IP instance configuration

Table 17 lists additional information on properties that differ from the default value.

Table 17. NetX IP instance property in sample application

ISDE Property	Value	Description
Ipv4 Address	0, 0, 0, 0	When DHCP client is used, this value must be 0,0,0,0 and modification of this property is locked.
ARP	Enable	When DHCP client is used, this ARP must be enabled, and modification of this property is locked.
UDP	Enable	When DHCP client is used, UDP must be enabled and modification of this property is locked.

There are no changes for the following blocks from the default setting:

- NetX Common on NetX (section 5.1.2
- NetX Packet Pool Instance (section 5.1.3)
- NetX Port using Wi-Fi framework on sf_wifi_nsal_nx (section 5.1.4)

See the specified references of the corresponding sections for the related configurations.

7.4.3 Wi-Fi Device Driver Configuration

See section 5.1.5 for the available configurations of the Wi-Fi device driver block. In this section, only the properties that changed from the default configuration are described in detail.

The properties that change from the default value are marked in the red boxes.

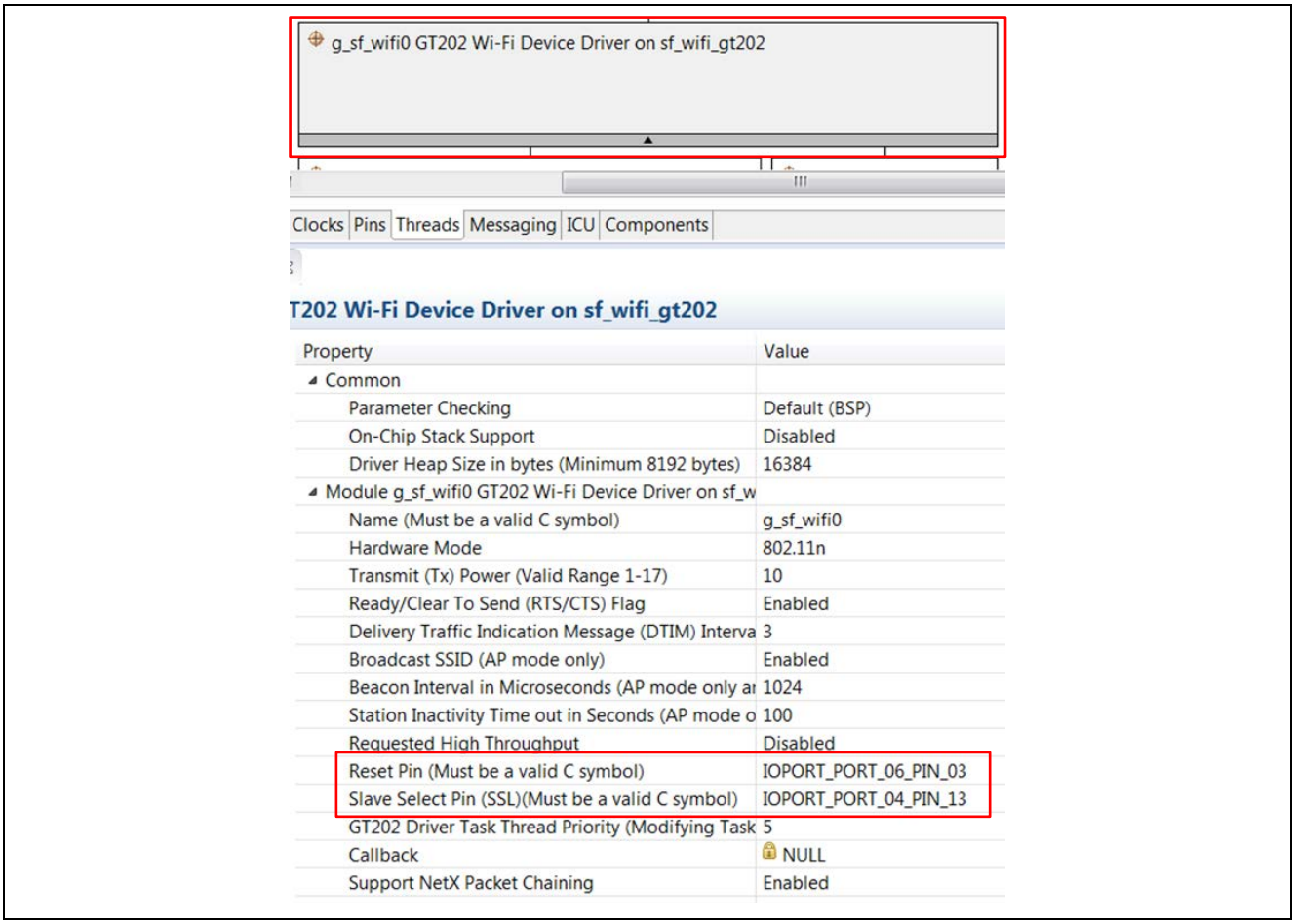

Figure 45. Wi-Fi device driver configuration in the sample project

Table 18 lists additional information on the properties that differ from the default values.

Table 18. Wi-Fi device driver configuration in the sample project

ISDE Property	Value	Description
Reset Pin	IOPORT_PORT_06_PIN_03	Hardware configuration on SK-S7G2 and PK-S5D9
Slave Select Pin	IOPORT_PORT_04_PIN_13	Hardware configuration on SK-S7G2 and PK-S5D9

Figure 45 shows that P6_03 is used as the reset pin for the Wi-Fi module. By default, P6_03 is configured as input, the user needs to reconfigure P6_03 as a low output.

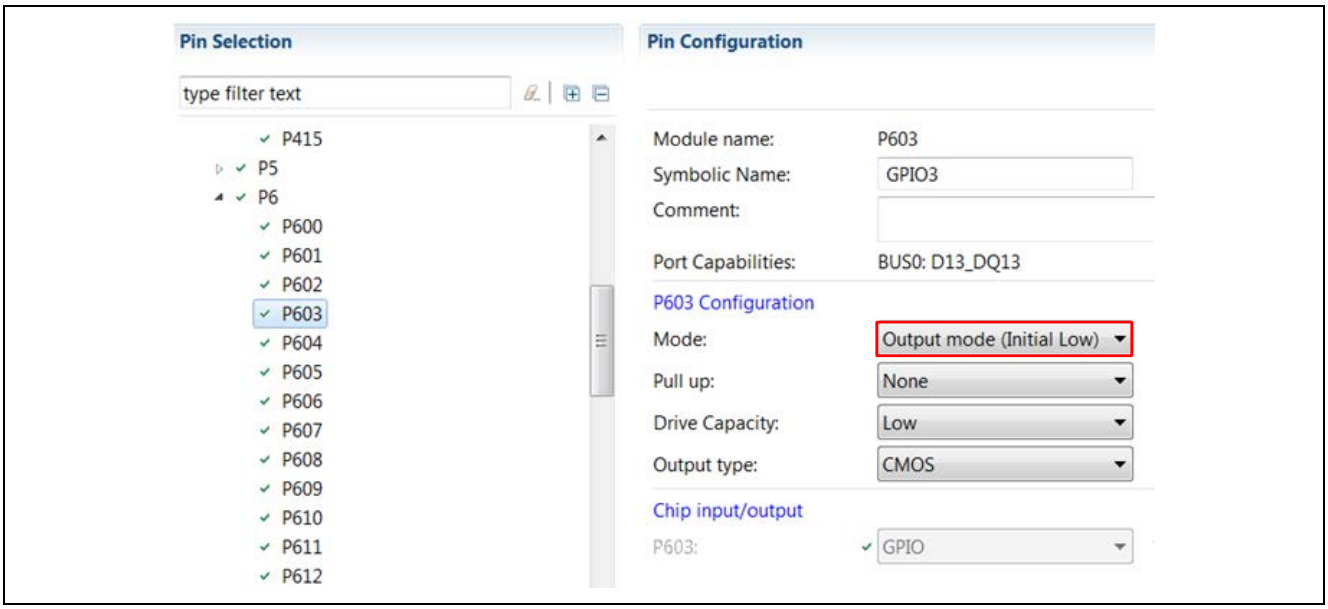

Figure 46. Configure P6_03 to output low

In addition, P4_13 is used as the slave select pin for the Wi-Fi module. By default, P4_13 is configured as input, the user needs to reconfigure P4_13 as a high output.

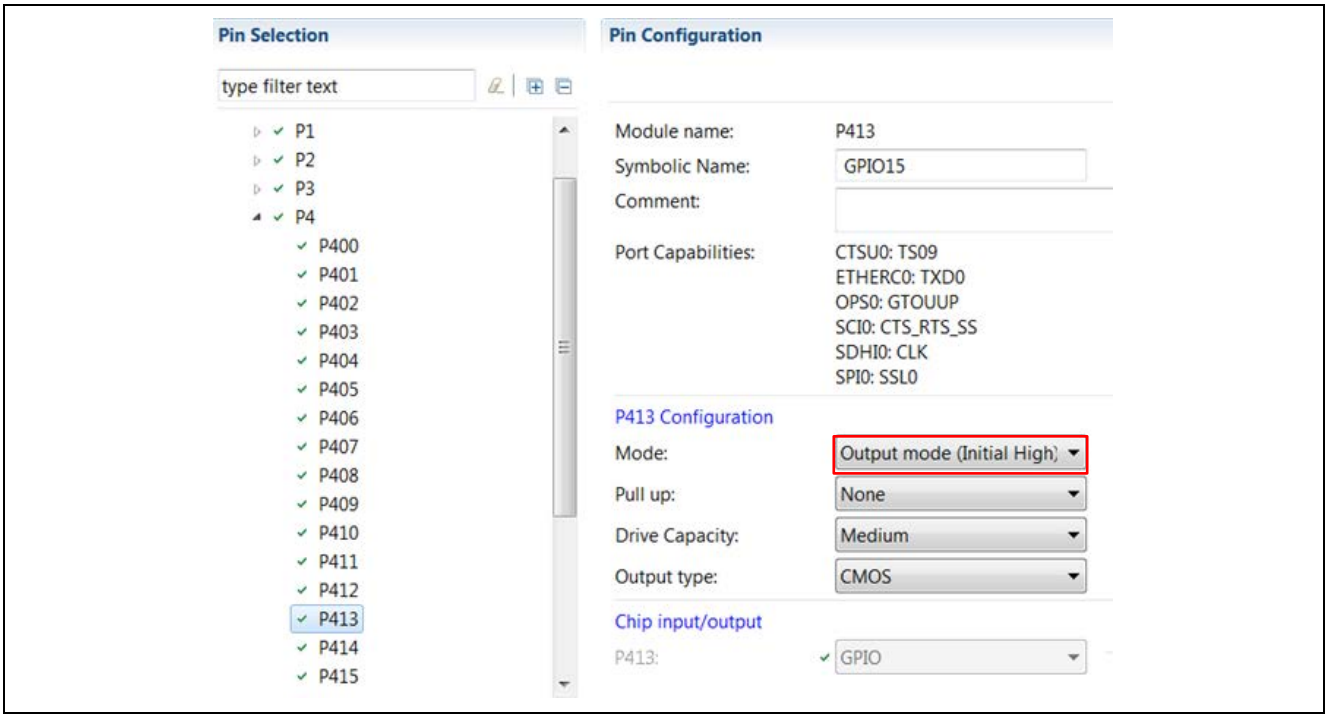

Figure 47. Configure P4_13 to output high

7.4.4 SPI Communication Configuration

The GT202 uses SPI to communicate with the MCU. For details on SPI peripheral settings, see the *SPI Module Guide*.

This application project uses the RSPI interface to communicate with GT202. Table 19 lists the RSPI block added.

Table 19. Add SPI driver

Resource	ISDE Tab	Stacks Selection Sequence
Add SPI driver	Thread	Add SPI driver>g_spi0 SPI driver on r_rspi

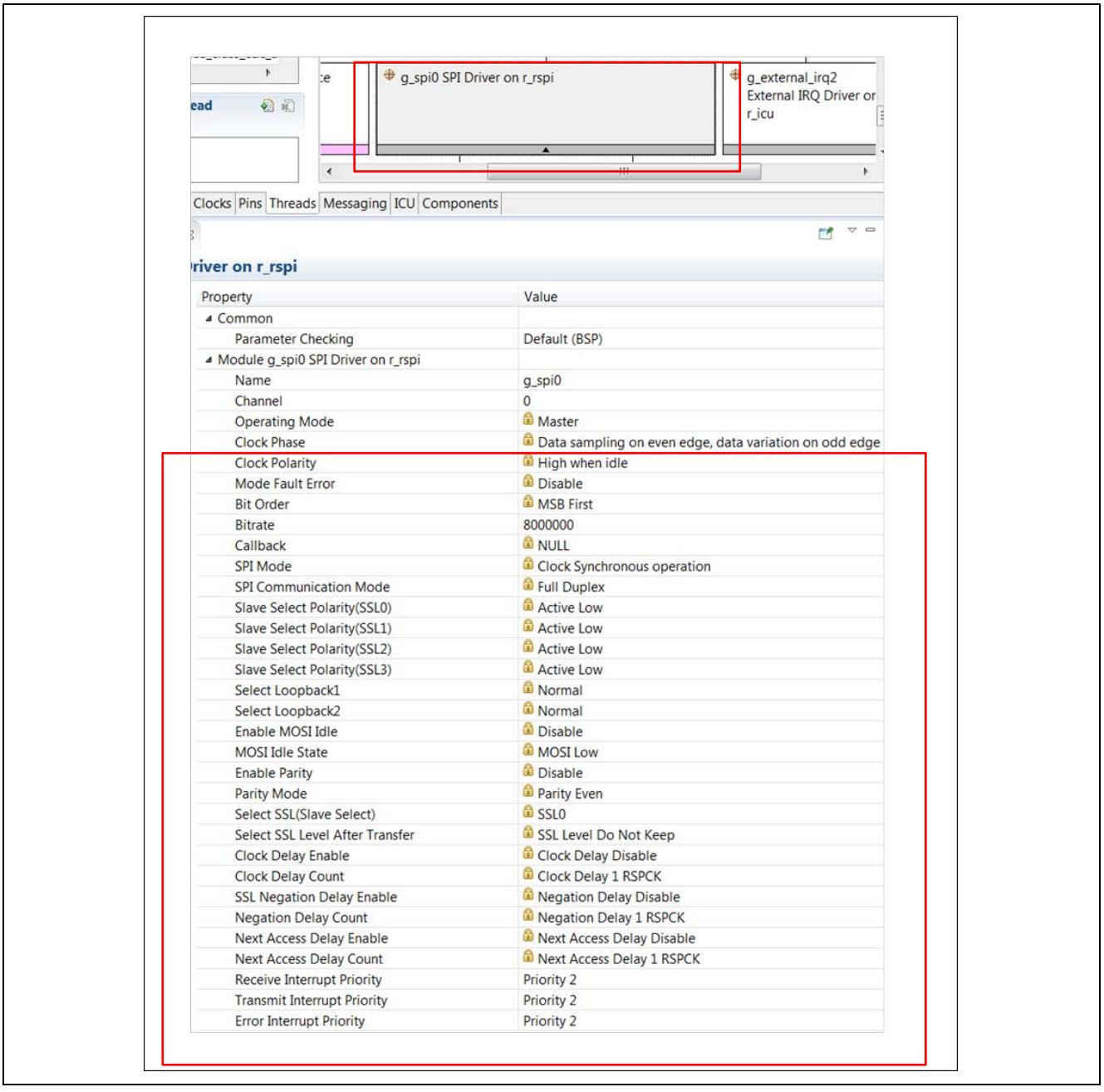

Figure 48. RSPI configuration

See the *RSPI HAL Module Guide* for detailed description of the R_RSPI configuration properties. When the RSPI is pulled in to complete the GT202 device driver, the Synergy configurator automatically adjusted several settings and have most of the settings locked to avoid misconfigurations.

Table 20 shows the updated properties, other properties assume the default value.

Table 20. RSPI configuration for GT202

ISDE Property	Value	Description
Operation Mode	Master	Modification of this property is locked.
Clock Phase	Data sampling on even edge, data variation on odd edge	Modification of this property is locked.
Clock Polarity	High when idle	Modification of this property is locked.
Mode Fault Error	Disabled	Modification of this property is locked.
Bit Order	MSB First	Modification of this property is locked.
Bitrate	8000000	This setting is valid. See the GT202 User Manual for other permitted settings.
Callback	NULL	Modification of this property is locked.
SPI Mode	Clock synchronous operation	Modification of this property is locked.
SPI Communication Mode	Full duplex	Modification of this property is locked.
Slave Select Polarity (SSL0)	Active-low	Modification of this property is locked.
Slave Select Polarity (SSL1)	Active-low	Modification of this property is locked.
Slave Select Polarity (SSL2)	Active-low	the modification of this property is locked
Slave Select Polarity (SSL3)	Active-low	Modification of this property is locked.
Select Loopback1	Normal	Modification of this property is locked.
Select Loopback2	Normal	Modification of this property is locked.
Enable MOSI Idle	Disabled	Modification of this property is locked.
MOSI Idle State	MOSI Low	Modification of this property is locked.
Enable Parity	Disable	Modification of this property is locked.
Parity Mode	Parity Even	Modification of this property is locked.
Select SSL (Slave Select)	SSL0	Modification of this property is locked.
Select SSL Level After Transfer	SSL Level Do Not Keep	Modification of this property is locked.
Clock Delay Enable	Clock Delay Disable	Modification of this property is locked.
Clock Delay Count	Clock Delay 1 RSPCK	Modification of this property is locked.
SSL Negation Delay Enable	Negation Delay Disable	Modification of this property is locked.
Negation Delay Count	Negation Delay 1 RSPCK	Modification of this property is locked.
Next Access Delay Enable	Next Access Delay Disable	Modification of this property is locked.
Next Access Delay Count	Next Access Delay 1 RSPCK	Modification of this property is locked.

7.4.5 SPI Hardware Pin Configuration

The RSPI is selected for the PMODB on the SK-S7G2 and PK-S5D9 kits. The SPI interface on the PMODB uses P410, P411, and P412. On SK-S7G2, these pins are defaulted to SCI0 _B only based on the SK-S7G2 board package file. You must disable the SCI0 or reconfigure the SCI0 to other pins before assigning these pins to PMODB on SK-S7G2. In this application example, the SCI0 is disabled as shown in Figure 49.

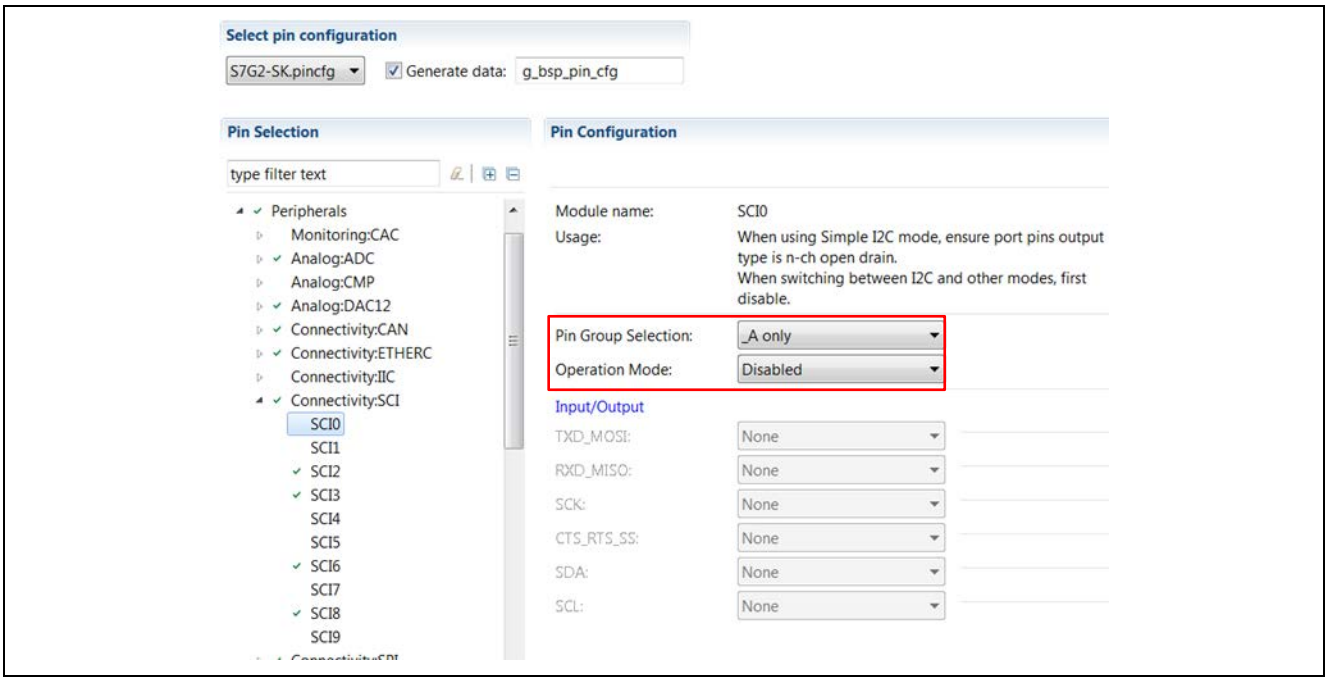

Figure 49. Disable SCI0 on SK-S7G2

Figure 50 shows that after the SCI0 is disabled, assign SPI0 to _B only on SK-S7G2 to enable the RSPI operation on the Wi-Fi driver.

For the PK-S5D9 MCU, pins P4_10, P4_11, and P4_12 are assigned to SPI0 in mixed mode. Assignments are based on the default PK-S5D9 board package. Figure 50 shows how to change the SPI0 to _B only, which must be done to allow proper communication with the Wi-Fi device driver.

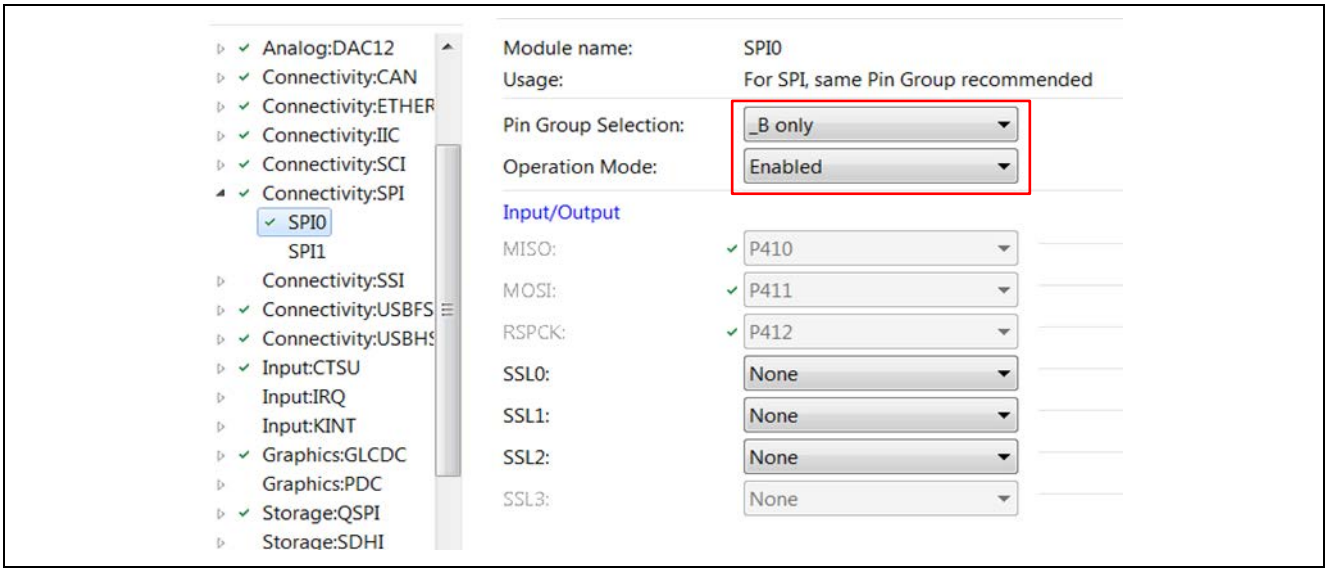

Figure 50. Assigned SPI0 to _B only on SK-S7G2 and PK-S5D9

By default, the DTC transfer module is included when the RSPI interface is included. The DTC driver must be removed because DTC uses 4-bytes (word) transfers, but the GT202 Wi-Fi driver supports 8-bit transfer mode.

7.4.6 PMODB Interrupt Pin

Figure 51 shows the configuration for the PMOD interrupt pin setup.

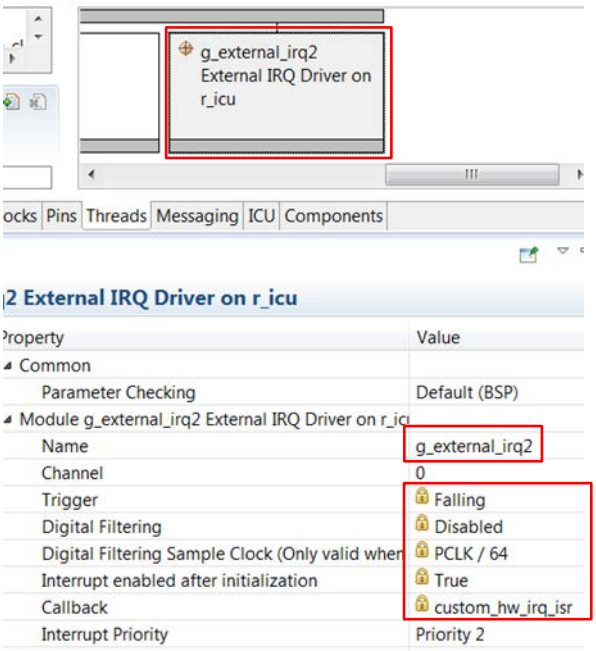

Figure 51. PMODB interrupt configuration

For details on the external IRQ configuration properties, see the *External IRQ HAL Module Guide*. When the external IRQ is pulled in to complete the GT202 device driver, the Synergy configurator automatically adjusts several settings and locks most of the settings to avoid configuration issues.

Table 21 shows the updated properties, other properties assume the default value.

Table 21. PMODB interrupt configuration

ISDE Property	Value	Description
Name	`g_external_irq2`	Name of the IRQ
Trigger	Falling	Determined by GT202 requirement. Modification of this property is locked
Digital Filtering	Disabled	Modification of this property is locked.
Digital Filtering Sample Clock	PCLK/64	Modification of this property is locked.
Interrupt enabled after initialization	True	Modification of this property is locked.
Callback	`custom_hw_irq_isr`	Determined by GT202 device driver implementation. Modification of this property is locked.
Interrupt Priority	Priority 2	This priority is appropriate for the sample application project used. Consider the particular application when selecting the priority for this interrupt.

With all the specified configuration, Figure 52 shows an overview of the finished configurator.

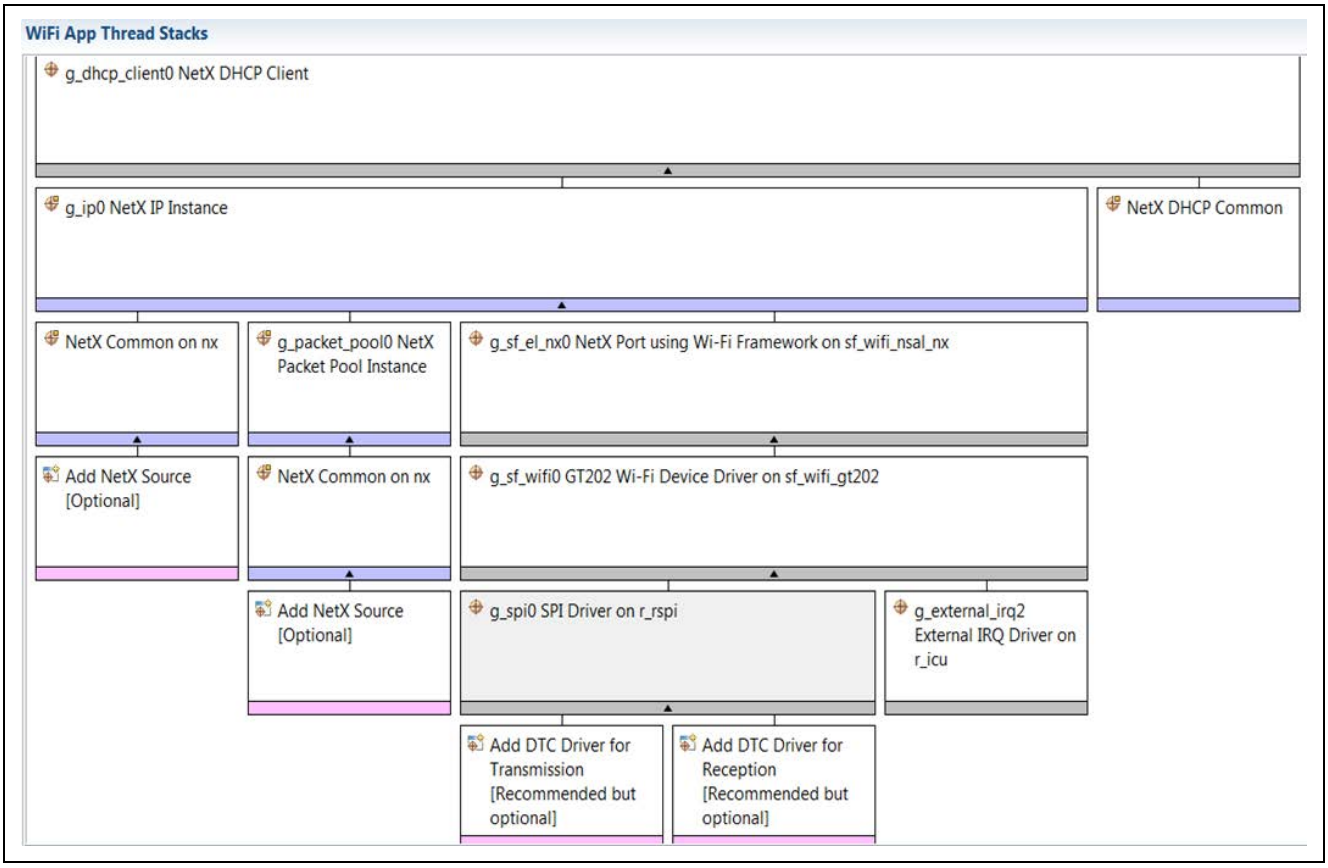

Figure 52. Complete configurator overview

7.4.7 SK-S7G2 PMOD Usage Caveat and Workarounds

The PMOD connector pin numbers on schematics for the SK-S7G2 Starter kit are shown incorrectly. Table 22 lists the actual pins following the PMOD standard.

Table 22. PMOD pin assignment

PMOD Pin	Signal	Direction	Description
1	SS	Out	Slave Select
2	MOSI	Out	Master Out Slave In
3	MISO	In	Master In Slave Out
4	SCK	Out	Serial clock
5	GND		
6	VCC		
7	INT	In	Interrupt signal from slave to master
8	RESET	Out	Reset signal from slave to master
9	N/S	N/S	
10	N/S	N/S	
11	GND		
12	VCC		

N/S: Not Specified. The meaning of these pins depends on the PMOD module, these pins can be unconnected or can be specific inputs or outputs for additional module. These pins are not used in SK-S7G2 and PK-S5D9 applications.

7.5 GT202 Wi-Fi Module and Driver Limitations/Known Issues

- GT202 on-chip stack only accepts single TCP connection. After accepting single connection, the GT-202 vendor driver fails to notify the upper layer for new connection request.
- GT202 vendor driver code has compilation warnings in it.
- GT202 Wi-Fi module driver does not support access control list.
- GT202 Wi-Fi module driver does not provide API for multicast filtering.
- The `recv socket` API implemented by GT202 vendor driver is non-blocking. However, the BSD `Socket` API specification for `recv` is blocking API.
- Workaround: Application should call `select` API before calling `recv` API, which makes the application wait until there is data available on the socket or timeout occurs.
- If the pin configuration is not set up properly, then the driver code within the GT202 module enters an infinite loop, causing the Wi-Fi framework open API to not return.

8. Running the Wi-Fi Framework Module Application Project

Review the following points before proceeding to the operation of this sample project:

- Make sure you have your PC connected to a Wi-Fi access point before running this application project. You must know the password of this access point because it is required for provisioning the GT202.
- This sample project only supports Wi-Fi security type WPA2, if you need to change to a different security type, you can edit the encryption field in the Wi-Fi provisioning structure. See Figure 40 for the related code adjustment.
- When you have your Wi-Fi connection and the password ready, you can import the Wi-Fi framework module application project and see it executing on SK-S7G2 or PK-S5D9.
- Refer to the *Renesas Synergy™ Project Import Guide* (r11an0023eu0121-synergy-ssp-import-guide.pdf), included in this package, for instructions on how to import the project into e^2 studio or IAR EW for Synergy, then build and run the application.
- The vendor provided GT202 device driver has over 500 warnings. This is normal with the version of the device driver.

8.1 SK-S7G2 Board Setup Details

Make sure that 3.3 V is selected for PMOD B using jumper (J15), as shown in Figure 53.

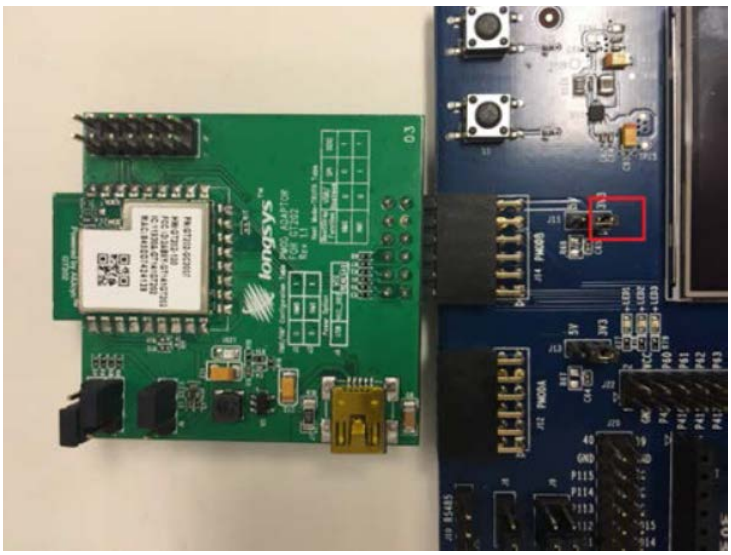

Figure 53. Hardware setup

Note: It is important to select 3.3 V for the modules. Otherwise, the modules might be damaged.

After setting the jumper as suggested:

1. Connect the micro USB cable to the J19 port to power up the board.
2. Connect the USB device from J5 to the PC.

8.2 Install the USB CDC Device Driver

The Console framework in this application project uses the Communication framework on USB CDC Device. This requires the USB CDC device driver being installed on the PC.

For Windows10, it is not necessary to install the USB CDC device driver because the SK-S7G2 and PK-S5D9 can be detected as a USB serial device as shown in Figure 54.

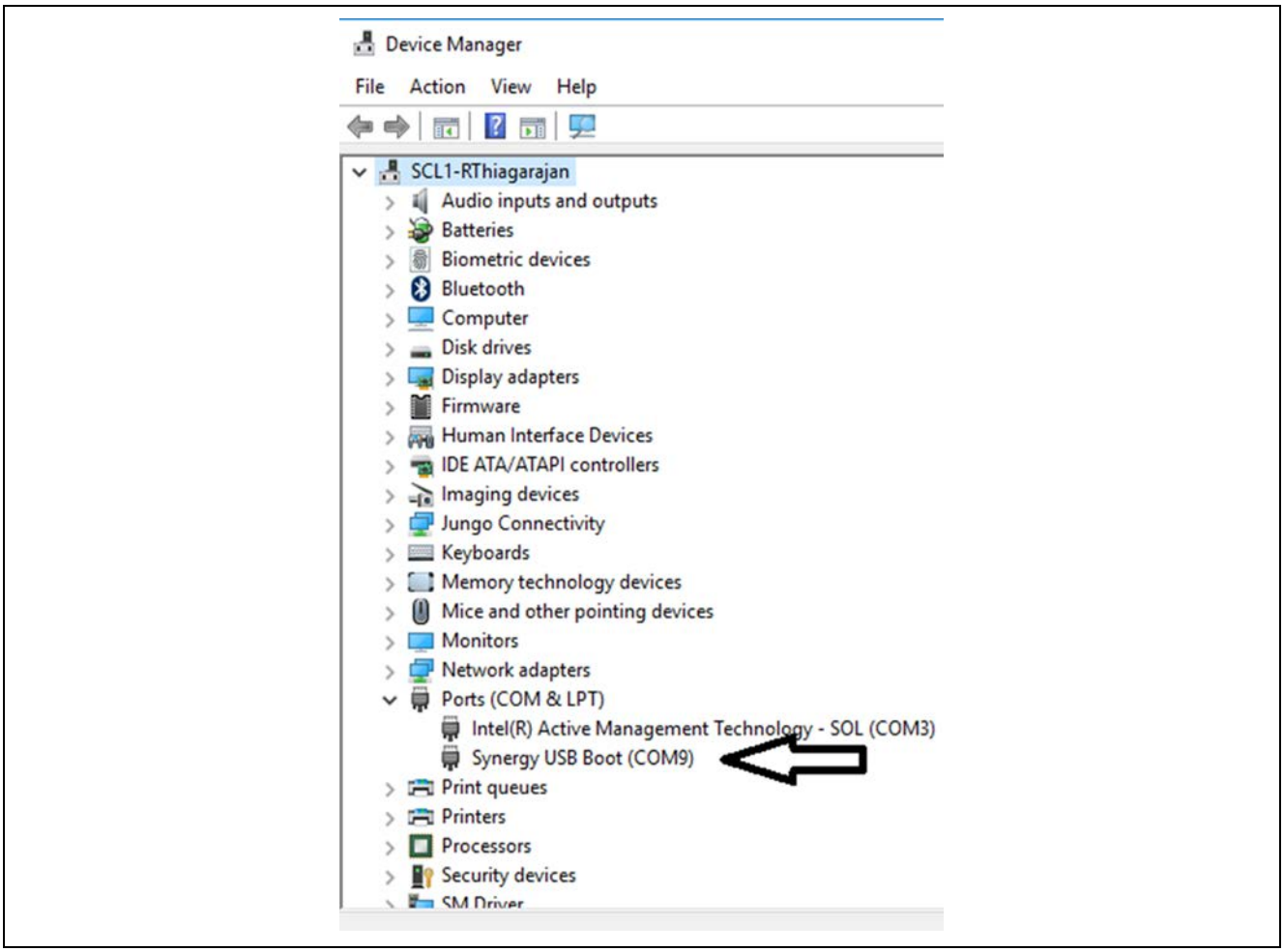

Figure 54. USB CDC port enumeration on Windows10

For Windows7, after the SK-S7G2 USB device port is connected to the PC, it is first detected as **Unknown Device**. You can then right-click on this device and select **Update Driver software**.

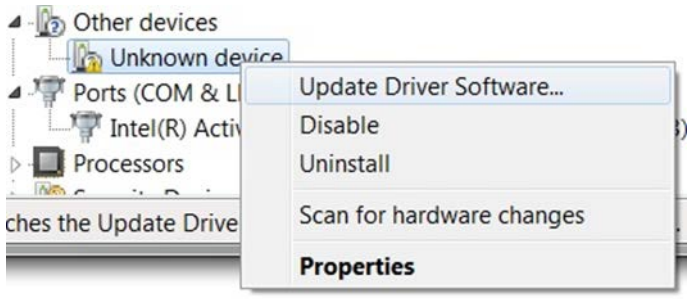

Figure 55. Update USB CDC driver

When prompted for the location of the drivers, browse to the location of the Windows USB serial driver provided as part of this application project. After the driver is updated, a new COM device is displayed in the Device Manager as in Windows10.

8.3 Running the Sample Project

When you run the software and confirm the COM port enumeration, open the Tera Term or other serial terminal, and select the enumerated COM port, then perform the following steps to provision the GT202 Wi-Fi module.

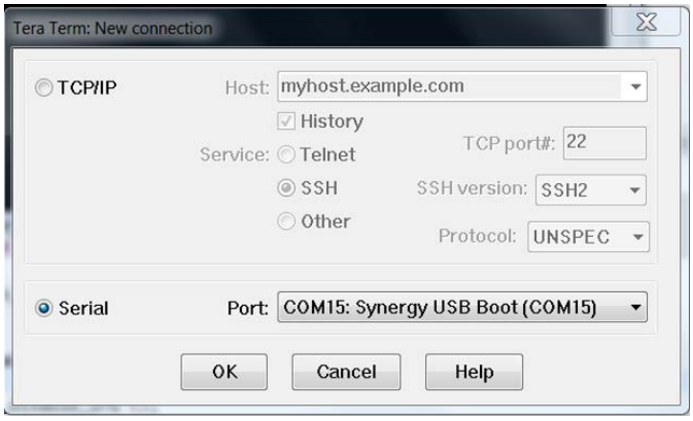

Figure 56. Connect to the COM port

8.3.1 View the Available Commands

There are two commands to use to provide the SSID and password to start the GT202 provisioning. On opening the Tera Term, type **?** and press the **Enter** key to review the available commands.

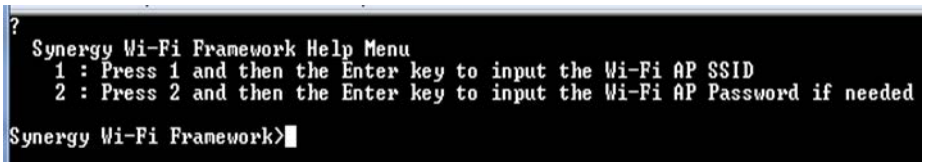

Figure 57. Commands available

8.3.2 Provision the Wi-Fi Module

Press **1** and the **Enter** key to input your Wi-Fi access point SSID. Press **2** and the **Enter** key to input your Wi-Fi access point password as shown in Figure 58. After you provided the AP password, the Wi-Fi module provision starts.

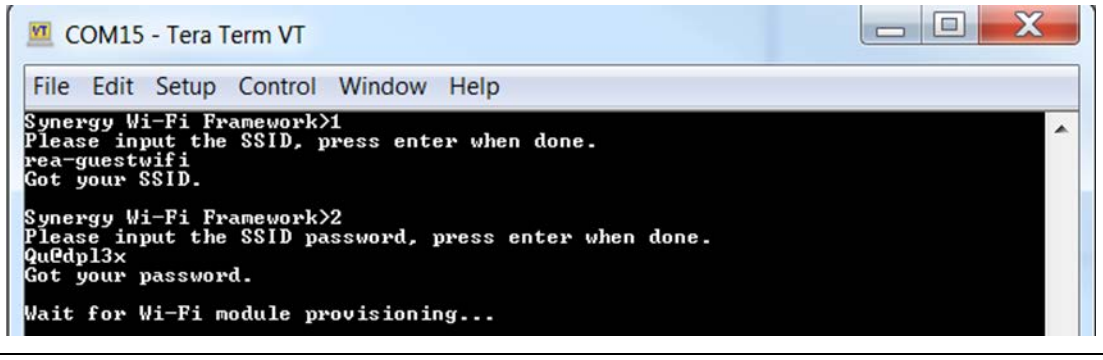

Figure 58. Input SSID and Password

When the provision successfully finishes, the console window updates as shown in the following screen.

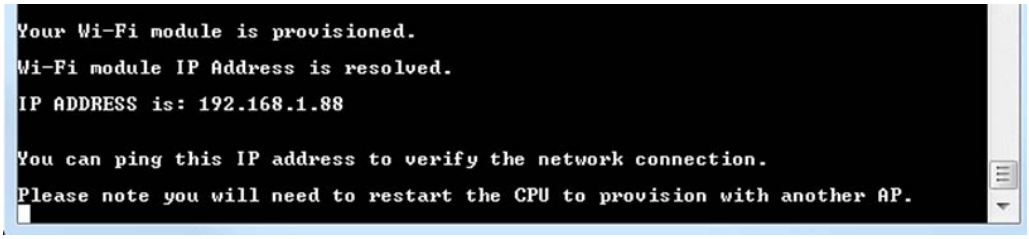

Figure 59. GT202 successful provision

If the provision failed, a message displays on the console screen, as shown in the following screen.

Figure 60. Provision failed

If the provision failed, you can check on the following points for troubleshooting:

- Whether the Wi-Fi router is configured for 2.4 GHz bandwidth.
- Whether your PC is connected to the access point you intended to use
- Whether the SSID and password are correct
- Whether the security type of the access point is WPA2

If the GT202 provision hangs, double check that the GT202 is properly connected to the correct PMODB port (J14). As described in section 6, the GT202 vendor provided driver code hangs if the pin configuration does not match.

8.3.3 Ping the Wi-Fi Module from the PC

After the GT202 is provisioned and you got the IP address from the console window, open a command window and type the following message as an example (you should use the IP address printed on the console from your setup):

```
>ping 192.168.1.37
```

Figure 61 shows an example of a successful ping result. This proves the successful link connection.

```
Pinging 192.168.1.37 with 32 bytes of data:
Reply from 192.168.1.37: bytes=32 time=5ms TTL=128
Reply from 192.168.1.37: bytes=32 time=5ms TTL=128
Reply from 192.168.1.37: bytes=32 time=4ms TTL=128
Reply from 192.168.1.37: bytes=32 time=2ms TTL=128

Ping statistics for 192.168.1.37:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 2ms, Maximum = 5ms, Average = 4ms
```

Figure 61. Ping the Wi-Fi module

9. Customizing the Wi-Fi Framework Module for a Target Application

This section describes how to add a new Wi-Fi module support to the Synergy Wi-Fi framework. It involves creating standard set of structures and APIs around the Wi-Fi device driver that are used by user application. Steps to integrate a new Wi-Fi driver using the Path 1 approach are provided in this section. Future application project will include steps for `sf_wifi_ctrl_t`. This document does not include instructions to integrate a new Wi-Fi module to support on-chip networking stack.

In this section, the new Wi-Fi module is referred to as “myWiFi”. While adding support for the actual module, replace “myWiFi” with the actual module name, for example BCM43362, or GT202.

9.1 Wi-Fi Framework Device Driver Source and Header Files Overviews

The first step is to create the Wi-Fi framework device driver source and header files for the new driver. The Wi-Fi framework source and header files are located in the following directory:

`/synergy/ssp_supplemental/`

To create the Wi-Fi framework device driver, start by creating the directory structure by copying and pasting an existing driver such as the GT202, and then renaming the new module based on the module selected.

Copy, paste, and rename the files and directories highlighted in green in Figure 62. The highlights in red indicate the new directories and file structure created by a developer.

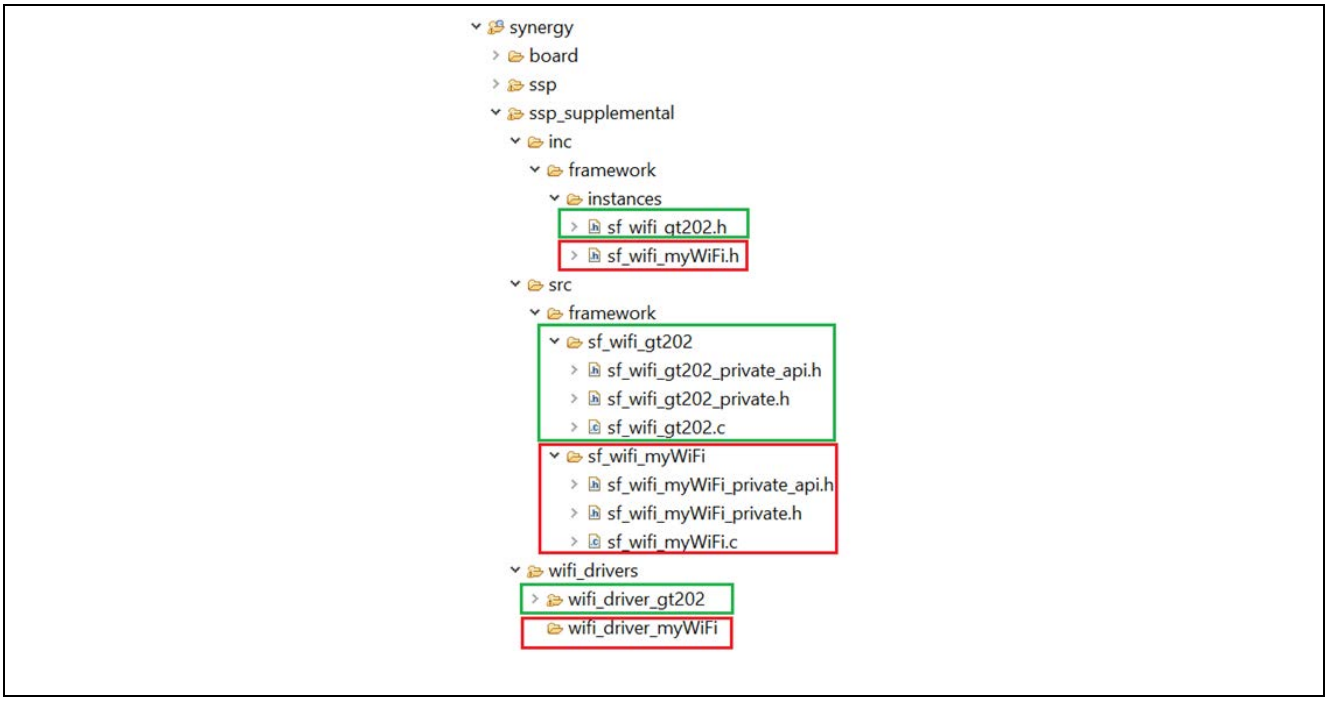

Figure 62. Setting up the Wi-Fi module driver directory file structure

A description of each file and folder is as follows:

- sf_wifi_myWiFi.h = This header file contains structure definition of the extended configuration which includes lower-level communication details. It also includes version information.
- wifi_driver_myWiFi = This directory contains the Wi-Fi module driver source code (3rd party vendor source code).
- sf_wifi_myWiFi _private_api.h = This header file contains prototypes of the Wi-Fi framework template APIs
- sf_wifi_myWiFi _private.h = This header file contains private macro and structure definitions used by the Wi-Fi framework module template.
- sf_wifi_myWiFi.c = This source file contains implementation of the Wi-Fi framework APIs template.

9.2 Instance Header File

The instance header file sf_wifi_myWiFi.h has the structure definition for the extended configuration. The extended configuration includes pointer to lower-level communication interface instances, for example: r_spi; r_sdmmc; the I/O Port pins used by the Wi-Fi module, such as the reset pin; the slave select pin; and, the driver task thread priority (the priority of the thread created internally by the device driver code of the Wi-Fi module). This structure may also contain additional configurable fields specific to the Wi-Fi module.

To adapt the new instance header file to the new driver, perform the following steps to update this file:

1. Change all GT202 references to MYWIFI.
2. Change all GT202 references to myWiFi.
3. Review the include files.

   Most Wi-Fi modules have interrupt pins and I/O that must be monitored or controlled by the Wi-Fi framework. These include files are probably sufficient. However, different Wi-Fi modules might use different communication peripherals, therefore proper peripheral include file should be considered. For example, if the Wi-Fi module uses SPI communication, `r_spi_api.h` must be included, or if IIC is used, `r_iic_api.h` is required. Make sure that the correct API communication driver is included.

4. Update the instance version number defined by:
   SF_WIFI_MYWIFI_CODE_VERSION_MAJOR
   SF_WIFI_MYWIFI_CODE_VERSION_MINOR

9.3 Framework APIs

The sf_wifi_myWiFi_private_api.h file contains all the API prototypes that are supported by myWiFi. To update the APIs for myWiFi, find and replace GT202 and gt202 with MYWIFI and myWiFi, respectively.

The result is displayed in the following figure.

```
#ifndef SF_WIFI_myWiFi_PRIVATE_API_H
#define SF_WIFI_myWiFi_PRIVATE_API_H

/* * Private Instance API Functions. DO NOT USE! Use functions through Interface API structure instead.*/
ssp_err_t SF_WIFI_myWiFi_Open(sf_wifi_ctrl_t * p_ctrl, sf_wifi_cfg_t const * const p_cfg);
ssp_err_t SF_WIFI_myWiFi_Close(sf_wifi_ctrl_t * const p_ctrl);
ssp_err_t SF_WIFI_myWiFi_MulticastListAdd(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const p_mac_addr);
ssp_err_t SF_WIFI_myWiFi_MulticastListDelete(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const p_mac_addr);
ssp_err_t SF_WIFI_myWiFi_StatisticsGet(sf_wifi_ctrl_t * const p_ctrl, sf_wifi_stats_t * const p_wifi_device_stats);
ssp_err_t SF_WIFI_myWiFi_Transmit(sf_wifi_ctrl_t * const p_ctrl, uint8_t * const p_buf, uint32_t length);
ssp_err_t SF_WIFI_myWiFi_ProvisioningSet(sf_wifi_ctrl_t * const p_ctrl,
        sf_wifi_provisioning_t const * const p_wifi_provisioning);
ssp_err_t SF_WIFI_myWiFi_ProvisioningGet(sf_wifi_ctrl_t * const p_ctrl,
        sf_wifi_provisioning_t * const p_wifi_provisioning);
ssp_err_t SF_WIFI_myWiFi_InfoGet(sf_wifi_ctrl_t * const p_ctrl, sf_wifi_info_t * const p_wifi_info);
ssp_err_t SF_WIFI_myWiFi_Scan(sf_wifi_ctrl_t * const p_ctrl, sf_wifi_scan_t * const p_scan, uint8_t * const p_cnt);
ssp_err_t SF_WIFI_myWiFi_ACLAdd(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const p_mac);
ssp_err_t SF_WIFI_myWiFi_ACLDelete(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const p_mac);
ssp_err_t SF_WIFI_myWiFi_MACAddressGet(sf_wifi_ctrl_t * const p_ctrl, uint8_t * const p_mac);
ssp_err_t SF_WIFI_myWiFi_MACAddressSet(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const p_mac);
ssp_err_t SF_WIFI_myWiFi_VersionGet(ssp_version_t * const p_version);

#endif /** SF_WIFI_myWiFi_PRIVATE_API_H */
```

Figure 63. Custom Wi-Fi module API header

This file contains prototypes of the Wi-Fi framework APIs which are described in section 3.1.

9.4 Private Structure/Macro Definitions

The sf_wifi_myWi_private.h contains the private structures and macros definitions used by myWiFi framework. This file should not contain the private structures, data types, and macros definitions of the device driver. It contains structure definitions that are used inside the Wi-Fi framework control structure and handles communication with the Wi-Fi device driver.

Find and replace GT202 and gt202 with MYWIFI and myWiFi, respectively.

9.5 Framework API Implementation

The `sf_wifi_myWiFi.c` file contains the Synergy Wi-Fi framework API implementation. The implementation in this file is generic but it does make calls to the myWiFi driver and control structures. Find and replace GT202 and gt202 with MYWIFI and myWiFi, respectively.

From these framework APIs, some APIs are used by NSAL such as open, close, transmit, and the receive callback.

The **NSAL NetX driver** functions implement various IP driver commands used by NetX by calling the corresponding Wi-Fi framework APIs. When implementing the Wi-Fi device driver for the new Wi-Fi module, ensure these commands are handled properly by the Wi-Fi device driver:

- NX_LINK_INTERFACE_ATTACH
  - Handle this request by executing the open API to initialize the custom Wi-Fi module
- NX_LINK_INITIALIZE
  - Handle this request by executing the macAddressGet API
  - Use driver mtu = 1.5k
- NX_LINK_UNINITIALIZE
  - Handle this request by executing the close API.

9.5.1 NSAL Transmit API Interface

The NSAL Transmit API is called from the `nsal_netx_send_packet` function (see section 3.1.3).

- If the Wi-Fi driver supports zero copy, it can send the fragmental packets and release the packet buffer after the final packet is sent
- If the Wi-Fi driver does not support zero copy, it needs to chain the fragmental packets to a single transmit buffer and pass the pointer to the transmit buffer and data length to the `nsal_netx_send_packet` function call.

9.5.2 NSAL Receive Callback Interface

The `nsal_netx_receive` function is called from the `wifi_driver_callback` function (see section 3.1.3). The wifi_driver_callback is called by the Wi-Fi module device driver. This callback fills in the Synergy Wi-Fi callback arguments and passes it as an argument to the callback function. When the driver receives a data frame, it fills in the Synergy Wi-Fi callback arguments.

```
/** Wi-Fi framework callback parameter definition */
typedef struct st_sf_wifi_callback_arg
{
    sf_wifi_event_t         event;                  ///< Event code
    uint8_t               * p_data;                 ///< Packet data
    uint32_t                length;                 ///< Packet Data length
    void const            * p_context;              ///< Context provided to user during callback
} sf_wifi_callback_args_t;
```

Figure 64. Construct the Wi-Fi driver callback argument

After the argument is established, the Wi-Fi driver can call the Wi-Fi `driver_callback` function. See the sample code that follows for the implementation. To interface the `nsal_netx_receive` function, if the Wi-Fi driver supports zero copy, the driver passes the NetX packet directly to the `nsal_netx_receive` function. If the Wi-Fi driver does not support zero copy, it passes the received data to the NetX stack without allocating the NetX packet.

```
sf_wifi_cfg_t * p_wifi_cfg = NULL;
sf_wifi_callback_args_t callback_args;

p_wifi_cfg = &g_template_driver.wifi_cfg;
if (p_wifi_cfg->p_callback)
{
    callback_args.event = SF_WIFI_EVENT_RX;
    callback_args.p_data = (uint8_t *)buffer;
    callback_args.length = length;
    callback_args.p_context = p_wifi_cfg->p_context;
    p_wifi_cfg->p_callback(&callback_args);
}
```

Figure 65. Implement the custom receive callback

9.6 Updating the Driver Source Code

This folder contains the Wi-Fi module device driver source code provided by vendor.

The `/wifi_drivers/wifi_driver_my/` folder contains the Wi-Fi module device driver. This is where most of the changes must be made to get the new module to work. Each module is different, but there is a simple process that can be followed to update the drivers. An example process is shown in Figure 66 but might not apply to your module.

1. Download the Wi-Fi module vendor driver code. This code must be integrated into the device driver.
2. In the folder `/wifi_drivers/wifi_driver_myWiFi/`, rename the **gt202_ctrl** folder to **myWiFi_ctrl** and then rename each file so that it is **myWiFi_filename**. The **wifi_driver_myWiFi** directory is displayed as shown in Figure 66.

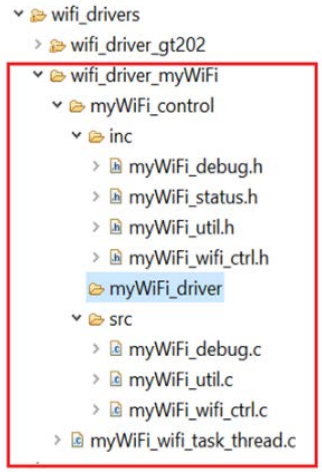

Figure 66. Custom Wi-Fi driver directory structure

3. Review each file in the `myWiFi_control` folder, find and replace GT202 and gt202 with **MYWIFI** and **myWiFi**, respectively.
4. Review the `myWiFi_wifi_ctrl.c` source file. Verify that each implemented function has the correct code required to operate the Wi-Fi module. Where necessary, make any adjustments needed to make it compatible with the new module.
5. Make sure that in the project properties, the header file paths are added so that the compiler can locate the files.

9.7 Updating the Wi-Fi Driver Configuration Header File

The new myWiFi requires a configuration file similar to other modules. The `sf_wifi_gt202_cfg.h` file serves as the template for `sf_wifi_myWiFi_cfg.h`.

Perform the following steps to create the configuration header file.

1. Copy `sf_wifi_gt202_cfg.h` from the `synergy_cfg/ssp_cfg/framework/` directory. Paste it and rename it as `sf_wifi_myWiFi_cfg.h` as shown in Figure 67. The red highlighted file is the newly created file.
2. Open the new file, `sf_wifi_myWiFiModule_cfg.h`, find and replace GT202 and gt202 with **MYWIFI** and **myWiFi** respectively.
3. Build the driver. If there are any outstanding issues in the build, resolve them.

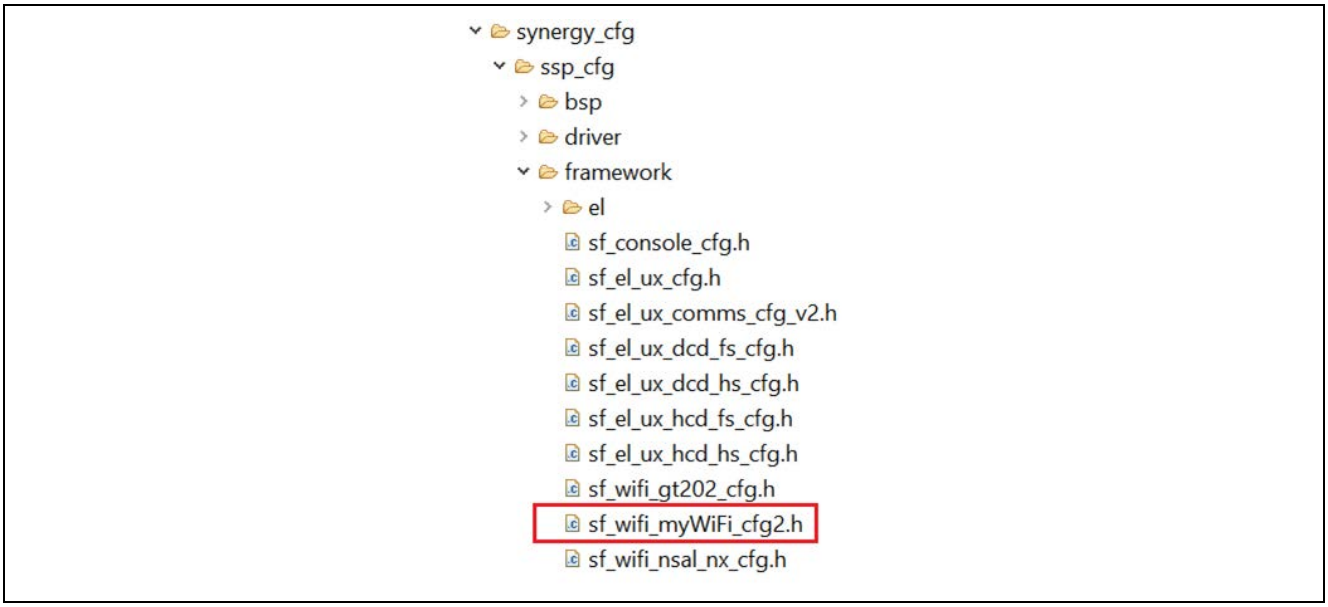

Figure 67. Wi-Fi module configuration

10. Wi-Fi Framework Module Conclusion

This document provided all the necessary information required to select, add, configure, and use the module in an example project. Many of these steps are time-consuming and error-prone activities from previous generations of embedded systems. The Renesas Synergy Platform makes these steps less time-consuming and removes the common errors like conflicting configuration settings or incorrect selection of low-level modules. The use of the high-level APIs demonstrated in this application project illustrates the additional development-time savings achieved by allowing work to begin at a high level and avoiding the time required in older development environments when using or creating low-level drivers.

11. Wi-Fi Framework Module Next Steps

After you have mastered a simple Wi-Fi framework project, you might want to review a more complex example. Other application projects and application notes that demonstrate Wi-Fi framework can be found in the References section at the end of this document.

12. Wi-Fi Module Resource Information

Renesas modules have Knowledge Base articles that provide helpful resources and related links. The following metadata section includes suggested modules and links.

12.1 SSP User Manual

The SSP distribution is available in html and PDF format from www.renesas.com/synergy/ssp.

12.2 Knowledge Base

To find up-to-date Wi-Fi resources and related links, visit the Renesas Knowledgebase (knowledgebase.renesas.com), enter “sf_wifi” for the Wi-Fi module name, and include “module guide” in the search.

In the Knowledgebase, use these searches to view the RSPI, Console Framework, and External Interrupt articles:

- Search on “r_rspi module guide” to view RSPI module guide resources.
- Search on “sf_console module guide” to view Console Framework module guide resources
- Search on “r_icu module guide” to view the external IRQ module guide resources

12.3 Longsys GT202 Module reference link

https://www.arrow.com/en/products/gt202kits-b/~/media/4e4847f9d2ba448d89e1a68526328364.ashx

Website and Support

Visit the following vanity URLs to learn about key elements of the Synergy Platform, download components and related documentation, and get support.

Synergy Software	www.renesas.com/synergy/software
Synergy Software Package	www.renesas.com/synergy/ssp
Software add-ons	www.renesas.com/synergy/addons
Software glossary	www.renesas.com/synergy/softwareglossary
Development tools	www.renesas.com/synergy/tools
Synergy Hardware	www.renesas.com/synergy/hardware
Microcontrollers	www.renesas.com/synergy/mcus
MCU glossary	www.renesas.com/synergy/mcuglossary
Parametric search	www.renesas.com/synergy/parametric
Kits	www.renesas.com/synergy/kits
Synergy Solutions Gallery	www.renesas.com/synergy/solutionsgallery
Partner projects	www.renesas.com/synergy/partnerprojects
Application projects	www.renesas.com/synergy/applicationprojects
Self-service support resources:	
Documentation	www.renesas.com/synergy/docs
Knowledgebase	www.renesas.com/synergy/knowledgebase
Forums	www.renesas.com/synergy/forum
Training	www.renesas.com/synergy/training
Videos	www.renesas.com/synergy/videos
Chat and web ticket	www.renesas.com/synergy/resourcelibrary

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Aug.29.17	—	Initial Release
1.01	Mar.26.18	—	Updated to 1.4.0
1.02	Sep.21.18	—	Document title changed
1.03	Mar.06.19	—	Updated with SSP 1.6.0
1.0.4	Oct.11.19	—	Updated with SSP 1.7.0

.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.
(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/.