RS9113 ZigBee Sample App

User Guide

Version 1.7.9

May 2020

Table of Contents

Table of Figures

1 ZigBee Sample Application Overview

1.1 Architecture

The ZigBee host mode APIs create a virtual layer of API functions which are actually available in RS9113 ZigBee stack. The high level architecture has been given in the following diagram.

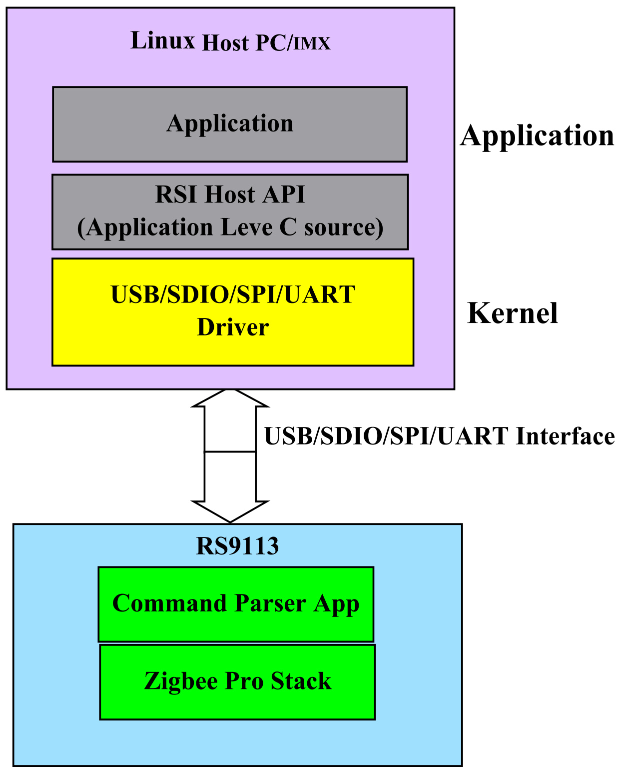


Figure 1: RZSP architecture

When the user connects the RS9113 ZigBee device to the host machine, 9113 exposes itself as USB/SDIO device. The command parser application which is running in 9113 accepts the packets from the host and identifies the appropriate command based on the respective argument values. After identification of command and argument values, parser calls the respective API in the stack.

1.2 Application source code contents

The host side application can found in the following path.
"RS9113.xxZ.WC.GEN.OSI.x.x.x/host/binary/reference_projects/LINUX/Application/zb/src/"

" x.x.x" represents the version number of the release

ZigBee APIs are organized in the following directory structure

	Path(Within *RS9113.xxZ.WC.GENR.x.x.x* folder)
ZIGBEE APIs	host/binary/apis/zb/core/
Interface Specific APIs	host/binary/apis/intf/
HAL APIs	host/binary/apis/hal/
ZIGBEE Reference Applications	host/binary/apis/zb/ref_apps/
ZIGBEE Linux Application	RS9113.WC.GENR.xxx/host/binary/reference_projects/LINUX/Application/zb/src
Linux USB Driver	host/binary/reference_projects/LINUX/Driver/usb/src
Linux SPI Driver	host/binary/reference_projects/LINUX/Driver/spi/src
Linux UART & USB-CDC Driver	host/binary/reference_projects/LINUX/Driver/uart/src
ZigBee Alone UART APP	host/binary/reference_projects/LINUX_WINDOWS/APPLICATION/src
ZigBee Configuration File	host/binary/apis/zb/ref_apps/include/rsi_zigb_config.h

1.2.1 Reference Project

A sample Linux based reference project is provided for achieving platform specific requirements. The purpose of each file is specified below:

1) host/binary/reference_projects/LINUX/Application/zb/src/main.c - Linux platform specific initializations are handled in this main file, later ZigBee main file is called
2) host/binary/reference_projects/LINUX/Application/zb/src/Makefile – Makefile for compiling the reference project
3) host/binary/reference_projects/LINUX/Application/zb/src/rsi_zigb_linux_apis.c – Linux platform specific api's are implemented in this file
4) host/binary/reference_projects/LINUX/Application/src/rsi_nl_app.c – Netlink sockets are used to establish communication between driver and application and vice-versa. Netlink socket related API's are handled here

1.2.2 Reference API's

Operating System independent API's used in the project are listed in the below specified file:

1) host/binary/apis/zb/ref_apps/src/zigb_main.c – ZigBee main file where ZigBee stack initialization and ZigBee state machine is handled
2) host/binary/apis/zb/ref_apps/src/rsi_zigb_app_cb_handler.c – ZigBee Interface specific callbacks are handled e.g., scan_complete, stack_status, network_found, data confirm handling etc.
3) host/binary/apis/zb/core/src/rsi_zigb_api.c - Source APIs in 'C' which the sample application uses to make calles to RS9113
4) host/binary/apis/zb/core/src/rsi_zigb_app_frame_process.c – Processing the received pkt from the device to exctract the command type from it
5) host/binary/apis/zb/core/src/rsi_zigb_build_frame_descriptor.c – Preparing the ZigBee frame descriptor
6) host/binary/apis/zb/core/src/rsi_zigb_delay.c – Add addtional delay in execution
7) host/binary/apis/zb/core/src/rsi_zigb_execute_cmd.c – To transfer the prepared pkt to RS9113
8) host/binary/apis/zb/core/src/rsi_zigb_utility.c – Generic and ZigBee utilities are listed in this file
9) host/binary/apis/zb/ref_apps/include/rsi_zigb_config.h – ZigBee protocol configuration file having parameters setting channel(mask) to scan, scan duration. And as well as default startup attribute set (SAS) and default ZigBee Device Object (ZDO) parameters
10) host/binary/apis/zb/ref_apps/src/rsi_zigb_config.c – File to update default SAS parameters, ZDO , profile, cluster, and simple descriptor.

1.2.3 Reference Driver

Sample driver is written for communicating with the device over different interfaces (SPI/USB/UART/USB-CDC).

For more information refer PRM document.

1.3 Prerequisites to run the sample application

1) RS9113 EVB having ZigBee firmware running inside.
2) Linux board with SPI/USB/USB-CDC/UART interface.
3) Application expects boot loading to be bypassed.
4) Gcc toolchain for building the host sample application for the given platform.
5) Mini USB cable to power RS9113 EVB
6) A 802.15.4, ZigBee coordinator
7) A 802.15.4, ZigBee Packet sniffer like TI CC2531 USB packet sniffer (http://www.ti.com/tool/cc2531emk , http://www.ti.com/tool/packet-sniffer)

1.4 Operating Mode

The operating (oper) mode command frame describes which protocols (WLAN/BT/BTLE/ZigBee) need to be activated in the device. After the device gets powered up, by default WLAN card ready will be received. Once WLAN card ready is received oper mode is the first command to be sent to device to enable various protocols based on the oper mode input.

To enable ZigBee, oper mode command should be sent first after receiving WLAN CARD READY, only then ZIGBEE CARD READY will be received. Unless we receive ZigBee card ready we can't proceed any further.

Incase of USB-CDC and UART interface, instead of WLAN CARD READY "Loading done" will be received indicating device ready status.

For more details about oper mode usage and command format, refer WLAN Documentation.

Note:-

Co-ex is supported only with End-Device image (i.e., RS9113.xxZ.WC.GEN.OSI.x.x.x.rps). For Coordinator and router, respective images has to be loaded.

1.5 Building the ZigBee API's along with sample application

1.5.1 Compiling Application

1) Go to "host/binary/reference_projects/LINUX/Application/zb/src/"
2) To build Home automation app issue the following command

   # make
3) If you are using different tool chain update the gcc tool chain with the platform specific tool chain in the Makefile present in "host/binary/reference_projects/LINUX/Application/zb/src/" and "host/binary/reference_projects/LINUX/Application/wlan/src/". Finally issue "make" command after modification.

1.5.2 Compiling Driver

1) Go to directory "host/binary/reference_projects/LINUX/Driver/"

2) Update the kernel path in the Makefile present in the corresponding interface specific "src/" directory

   For e.g., if you are using SPI interface then modify kernel path in "spi/src/Makefile"

3) Issue the following command to compile driver

   # make

4) Generated "ko" driver module should be used to install driver.

1.6 Execute ZigBee sample application

1) Insert the card and check device status(detection) in case of USB

   #dmesg –c

   USB device detection status should be as shown below

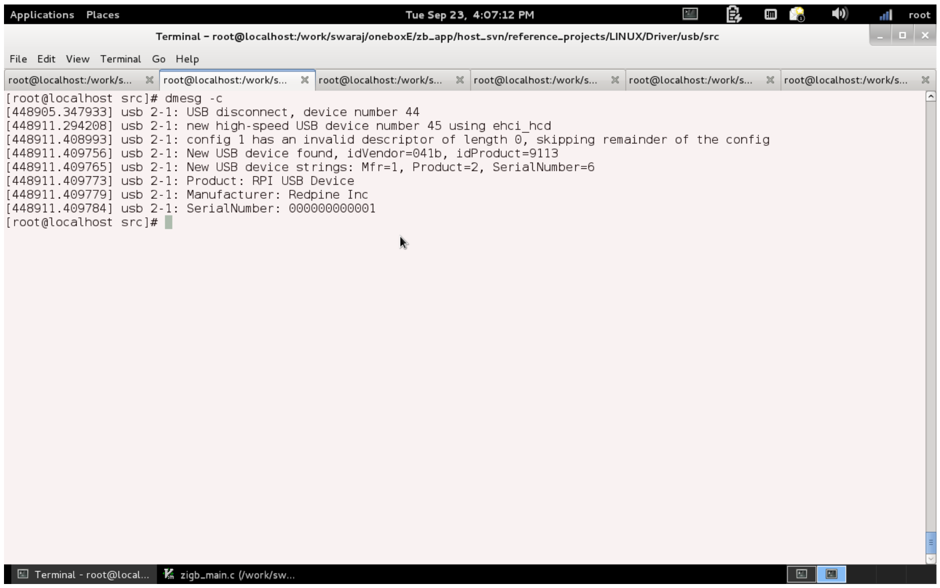


Figure 2: RS9113 USB mode card detection

In case of USB-CDC it should be as shown below

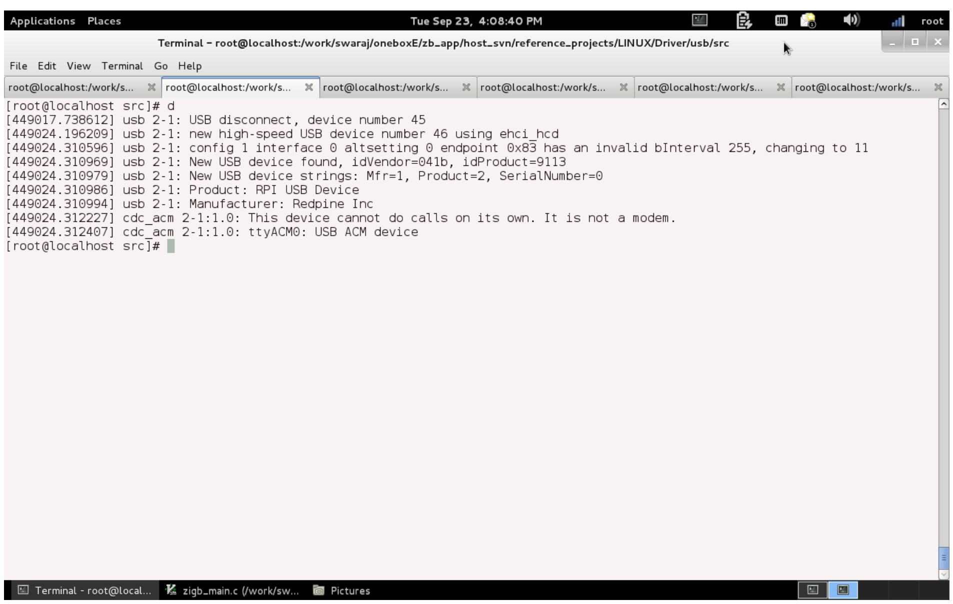


Figure 3: RS9113 USB-CDC mode card detection

1.6.1 Installing Driver

1) Go to driver directory "host/binary/reference_projects/LINUX/Driver/<*interface*>/src"
2) <interface> can be usb/spi/uart
   (uart and usb-cdc will use same uart driver)
3) Insert generated "ko" module, using the following command
   # insmod rps<interface>.ko
   e.g., insmod rpsusb.ko

1.6.2 Running Application

1.6.2.1 For USB/SPI interface

1) Go to "host/binary/reference_projects/LINUX/Application/zb/src/"
2) Run ZigBee app by issuing the following command
   # ./rsi_wsc_zigb_app

3) ZigBee app will wait for card ready to proceed further, once wifi app is started then card ready will be received by ZigBee app too.
   So, open a new terminal and run wifi application with the following cmd
   # ./rsi_wsc_wifi_app

4) In case if you want to restart ZigBee app issue the following command, which will skip card ready
   # ./rsi_wsc_zigb_app 1

1.6.2.2 For UART/USB-CDC

1) Go to "host/binary/reference_projects/LINUX/Application/wlan/src/"
2) Run the wifi app by issuing the following command
   # ./rsi_wsc_app

3) Open a new terminal and Go to "host/binary/reference_projects/LINUX/Application/uart/src/" .
4) If the interface is UART , run the serial application by issuing following command
   # ./rsi_serial

5) If the interface is USB-CDC ,run the serial application by issuing following command
   # ./rsi_serial 1
6) Open a new terminal , go to "host/binary/reference_projects/LINUX/Application/zb/src/"
7) Run ZigBee app by issuing the following command
   # ./rsi_wsc_zigb_app 1

Note:

- Before running the above applications, please ensure that the coordinator is permitting the join requests from child devices

1.7 Application State machine states

The sample application architecture is based on the state machine.
The state machine will have different states. Please refer to the rsi_zigb_app_sm.h file for the defined states and the events.

Various sates in the state machine are:

1) **FSM_CARD_NOT_READY** – RS9113 device is not initialized
2) **FSM_CARD_READY** – Device ready indication received, as Stack is not initialized yet request for stack initialization is sent
3) **FSM_INIT_STACK** - Once stack ready confirm is received reset stack request is sent which will reset all states and variables
4) **FSM_RESET_STACK** – As reset stack is done, request for dev type is sent to know what type of device is it i.e either End device or router or coordinator
5) **FSM_GET_DEV_TYPE** – once device type is confirmed that it is either End device or Router then scan is initialized
6) **FSM_INIT_COORDINATOR –** if the device type is Coordinator then coordinator specific initializations are done in this state and finally energy scan request is issued
7) **FSM_INIT_ENDDEV_ROUTER -** if the device type is EndDevice/Router then device specific initializations are done in this state and finally init scan request is issued
8) **FSM_INIT_SCAN** – all scan related packet will exchanged in this state
9) **FSM_SCAN_DONE** - Once scan is done either join request or form network request is sent based on the device type. If device is coordinator form network request will be issued else join request will be sent to device if any network is already present.
10) **FSM_FORM_NET** – Once stack status success confirm is received state machine state will be changed to FSM_ZB_FORMED.
11) **FSM_JOIN_NETWORK** – In this state , the device waits for the stack status , if it is success ,then

    - If devicetype= *EndDevice*

    it sends match descriptor request() if **APITEST** flag is disabled, else it sends rsi_zigb_network_state() then changes the state to FSM_API_TEST.

    - If devicetype= *Router.*

    if **APITEST** flag is disabled it sends rsi_zigb_permit_join () then changes the state to FSM_ZB_HANDLE_ROUTER, else it sends rsi_zigb_permit_join () then changes the state to FSM_API_TEST.

12) **FSM_ZB_FORMED** – In this state allows the Application to enable join permit on the device for the specified duration in seconds and handle the
    the callbacks.

13) **FSM_ZB_CONNECTED** – All data packets will be handled in this state.
14) **FSM_API_TEST** – All the apis are excited from this state and the application prints the number of apis executed, passed and failed.
15) **FSM_HANDLE_ROUTER** – The router checks the permit join response and handles the stack callbacks in this state.

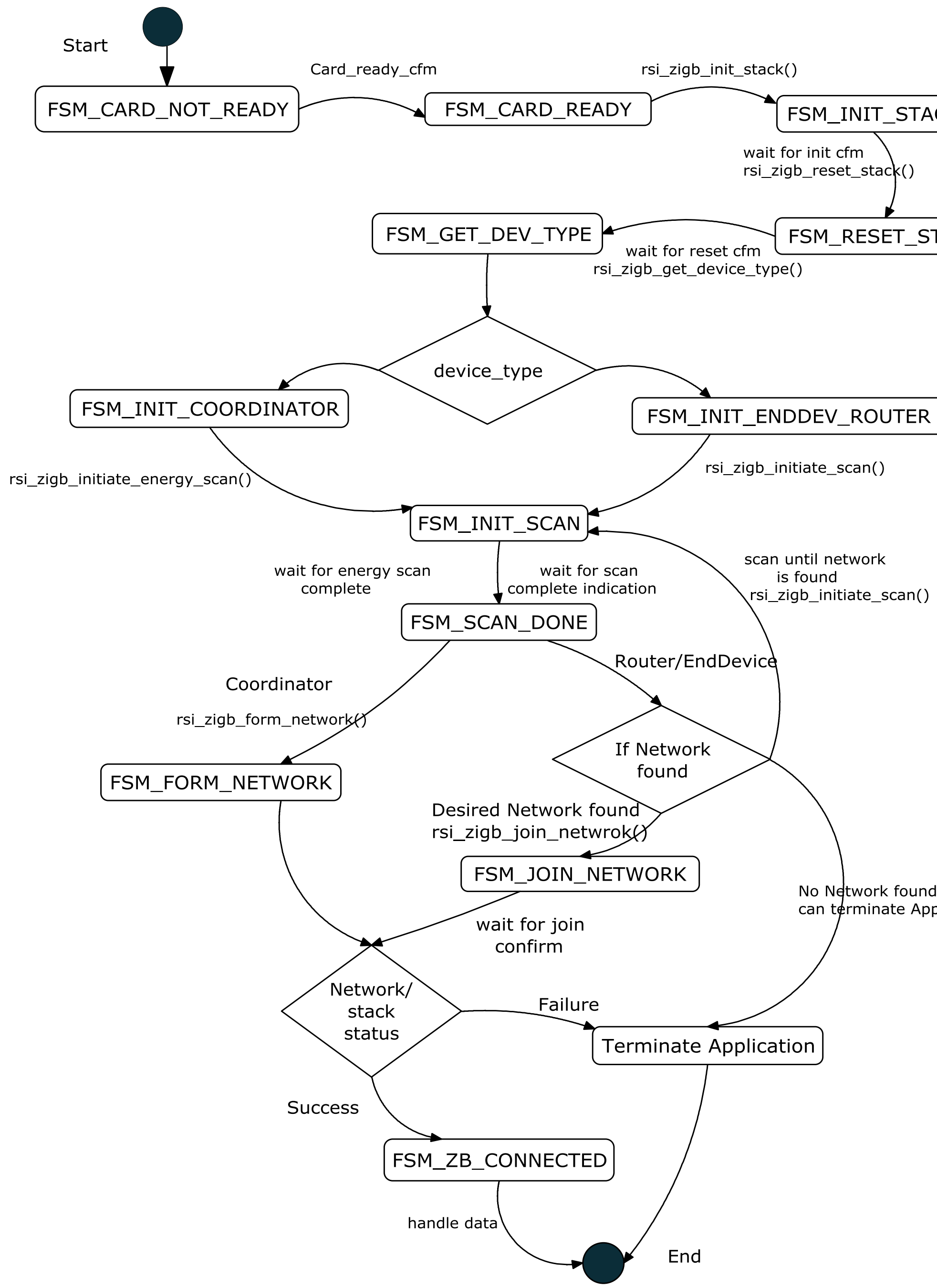


Figure 4: ZigBee Application State Machine diagram

1.8 Event callbacks

The application callbacks that are being invoked by stack are **asynchronous calls** which are provided in the file rsi_zigb_app_cb_handler.c present in “host/binary/apis/zb/ref_apps/src”
The callback functions that are available are given bellow.

1) rsi_zigb_app_scan_complete_handler() – called when scan is completed in the all the requested channels

   refer section AppScanCompleteResp.

2) rsi_zigb_app_energy_scan_result__handler() – called when scan is completed in all the channels

   refer section AppEnergyScanResultResp.

3) rsi_zigb_app_network_found_handler() – invoked whenever a network is found while scanning

   refer section AppNetworkFoundResp.

4) rsi_zigb_app_stack_status_handler() – stack/network status information is informed with this callback

   refer section AppZigBeeStackStatusResp.

5) rsi_zigb_app_incoming_many_to_one_route_request_handler()

   refer section AppIncomingManyToOneRouteRequestResp.

6) rsi_zigb_app_handle_data_indication() – if stack receives any data request those will be indicated using this event callback unless the destination has a valid endpoint and profile/cluster

   refer section AppHandleDataIndicationResp.

7) rsi_zigb_app_handle_data_confirmation() – this will be triggered by stack to indicate the status of the data request sent from us

   refer section AppHandleDataConfirmationResp.

8) rsi_zigb_app_child_join_handler() – this event callback is called to inform the status of the child joining/leaving the network

   refer section AppChildJoinResp.

Important Note:

1) The application developer should not call any of the API call that sends command/data to the device from any of these callbacks.
2) For complete information about ZigBee API's and callbacks refer Section ZigBee Host API description

1.9 Functionality of sample application for End device

The sample application provided was ZigBee Home Automation supported Switch.
The device after stack and simple descriptor initialization, scans the channels(as per CHANNEL_MASK) , and then joins the network(first network which is found) .
Then it sends match descriptor request to the parent, if the parent responds with the desired match descriptor response, then the application sends On/Off toggle Command continuously. Detailed description is given below.

1.9.1 Simple Descriptor

1. User defined cluster, profile can be prepared and that information should be set in simple descriptor on an endpoint. This simple descriptor is sent to stack using

rsi_zigb_set_simple_descriptor(). Later on for data communication this information will be used.

2. Sample switch cluster and simple descriptor information is stored in rsi_zigb_config.c and that will be used by default

1.9.2 Scan

1) This app will initiate Scan in the given channel (rsi_zigb_initiate_scan) for the available ZigBee networks.

2) To modify the operating channels(mask) open RS9113.xxZ.x.x.x.LNX/host/binary/apis/*ref_apps/include/rsi_zigb_config.h* and modify the definition of CHANNEL_MASK_c to desired channels

3) Once scan is completed rsi_zigb_app_scan_complete_handler() will be invoked indicating status of scan. If any beacons are found status will be return with beacon found.

1.9.3 Network Information

1) Please make sure that at least one PAN coordinator is available in the selected channel. If there are no networks in the selected channels then stack status handler will return network status fail and application will be stopped.

2) The available ZigBee networks will be given to the application using the event callback rsi_zigb_app_network_found_handler() which is available in rsi_zigb_app_cb_handler.c file.

3) If multiple networks are available rsi_zigb_app_network_found_handler() will be invoked once for each network and information about each network is sent as part of the payload.

4) The sample application selects the first received network information for association.

5) Sample application will issue the rsi_zigb_join_network() to join the chosen network.

6) Application developers can make their own logic to store the received network information and select the desired parent from the stored information of the available networks.

1.9.4 Match descriptor

1) After successful association to its chosen parent, sample application issues the match descriptor request with our profile/cluster information to the coordinator.

2) If the response for match descriptor request is received, then it issues the ZigBee Home automation supported TOGGLE command to the coordinator/router from where the match descriptor response is received. Match descriptor will return success response if the coordinator/router has the ZigBee Home Automation supported Light cluster.

3) The snapshot of the sniffer log (TI CC2531 packet sniffer) for the above given sample application functionality has been given here.

1.9.5 Data

1) If we receive any valid data supporting our profile, then rsi_zigb_app_handle_data_indication() will be called to indicate that there is pending data.
2) This handler will be invoked asynchronously.

1.10 Functionality of sample application for Router

The sample application provided is for ZigBee Router as a range extender.
The router joins the coordinator and enables permit join. It then only handles the stack callbacks.

1.10.1 Scan

1) This app will initiate Scan in the given channel (rsi_zigb_initiate_scan) for the available ZigBee networks.

2) To modify the operating channels(mask) open RS9113.xxZ.x.x.x.LNX/host/binary/apis/*ref_apps/include/rsi_zigb_config.h* and modify the definition of CHANNEL_MASK_c to desired channels.

3) Once scan is completed rsi_zigb_app_scan_complete_handler() will be called indicating status of scan. If any beacons are found status will be return with beacon found.

1.10.2 Network Information

1) Please make sure that at least one PAN coordinator is available in the selected channel. If there are no networks in the selected channels then stack status Resp will return network status fail and application will be stopped.

2) The available ZigBee networks will be given to the application using the callback rsi_zigb_app_network_found_handler() which is available in rsi_zigb_app_cb_handler.c file.

3) If multiple networks are available rsi_zigb_app_network_found_handler() will be called once for each network and information about each network is sent as part of the payload.

4) The sample application selects the first received network information for association.

5) Sample application will issue the rsi_zigb_join_network() to join the chosen network.

6) Application developers can make their own logic to store the received network information and select the desired parent from the stored information of the available networks.

1.10.3 Permit Join

1) The router after successfully joining the network, issues Permit join command which allows the other devices to join to the router.

1.10.4 Event Callbacks

1) If the device receives the event callbacks from stack , example child_join_handler if any child joins/leaves, data indication, stack status, etc will be handled by the router.
2) The callbacks will be invoked asynchronously.

1.11 Test Setup for Router

The below figure describes the test setup for the router. The RPI device is a simple range extender, which joins the Coordinator. The switch (End Device) joins the RPI router, while the Light (Router) joins the Coordinator.

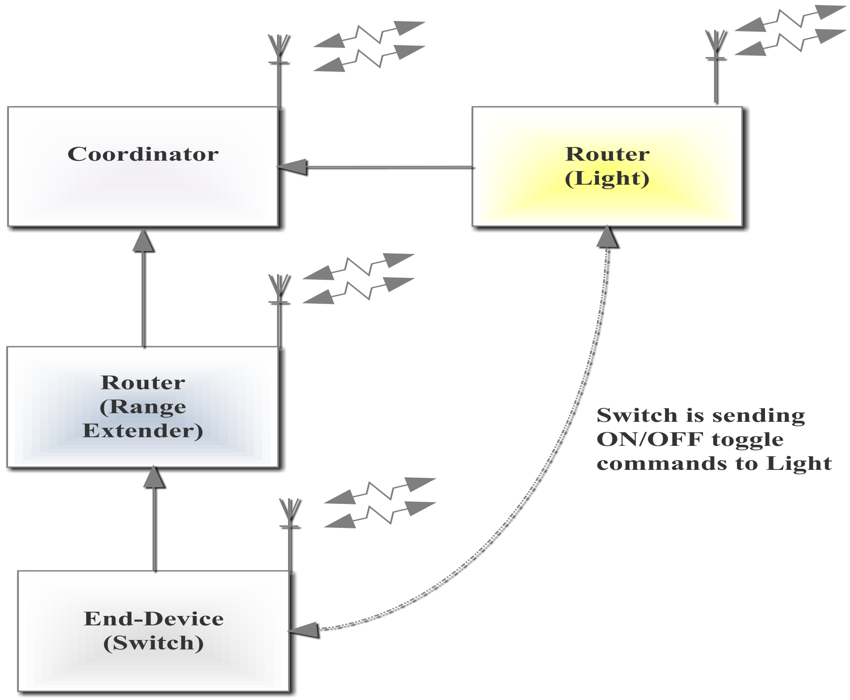


Figure 5 : Router Test Setup

1.12 Functionality of sample application for coordinator

The sample application provided is for ZigBee Coordinator as a simple coordinator. The coordinator forms the network and enables permit join. It then only handles the stack callbacks.

1.12.1 Form Network

1) This function allows the Application to establish the Network in the specified channel with the specified Extended PAN Id. The channel is updated from the CHANNEL_MASK_c (in *rsi_zigb_config.h*).If multiple channels are set in the mask, then the lowest channel value is taken as default channel.
2) The formation procedure is an asynchronous call. The stack event callback AppZigBeeStackStatusResp to indicate the status of formation to the Application.

3) This AppZigBeeStackStatusResp is invoked by the ZigBee Stack to indicate any kind of Network status to the application. For example: upon establishing the network, this function shall be called by the stack to indicate status ZigBeeNetworkIsUp. If the device doesn't form the network, a status of ZigBeeNWkisDown status is indicated via this function call.

1.12.2 Permit Join

1) This function allows the Application to enable join permit on the device for the specified duration in seconds.

1.12.3 Energy scan

1) The energy is an additional feature to get the energies(RSSI values) in the channels. This function allows the Application to initiate Scan of specified type in the specified channel mask for the specified duration. The Scan procedure is an asynchronous call. To perform energy scan , scan type field must be *g_MAC_ED_SCAN_TYPE_c*.
2) To modify the operating channels(mask) open RS9113.xxZ.x.x.x.LNX/host/binary/apis/*ref_apps/include/rsi_zigb_config.h* and modify the definition of CHANNEL_MASK_c to desired channels
3) Once scan is completed *AppenergyscanresultResp* will be invoked indicating status of scan result.

1.12.4 AppEnergyScanResultResp

1) This API *rsi_zigb_app_energy_scan_result_handler* is invoked to report RSSI value measured on the required channel to the application. Here,the developer can take decision to select a channel with low RSSI value.

1.13 Test Setup for Coordinator

The below figure describes the test setup for the Coordinator.

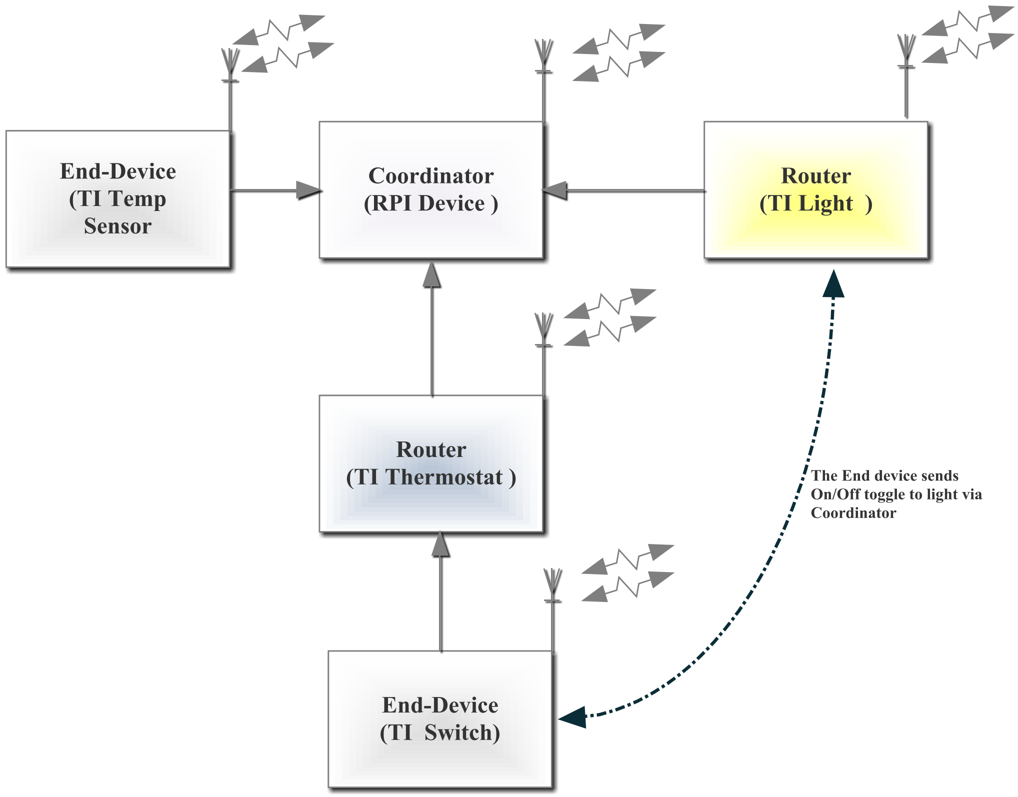
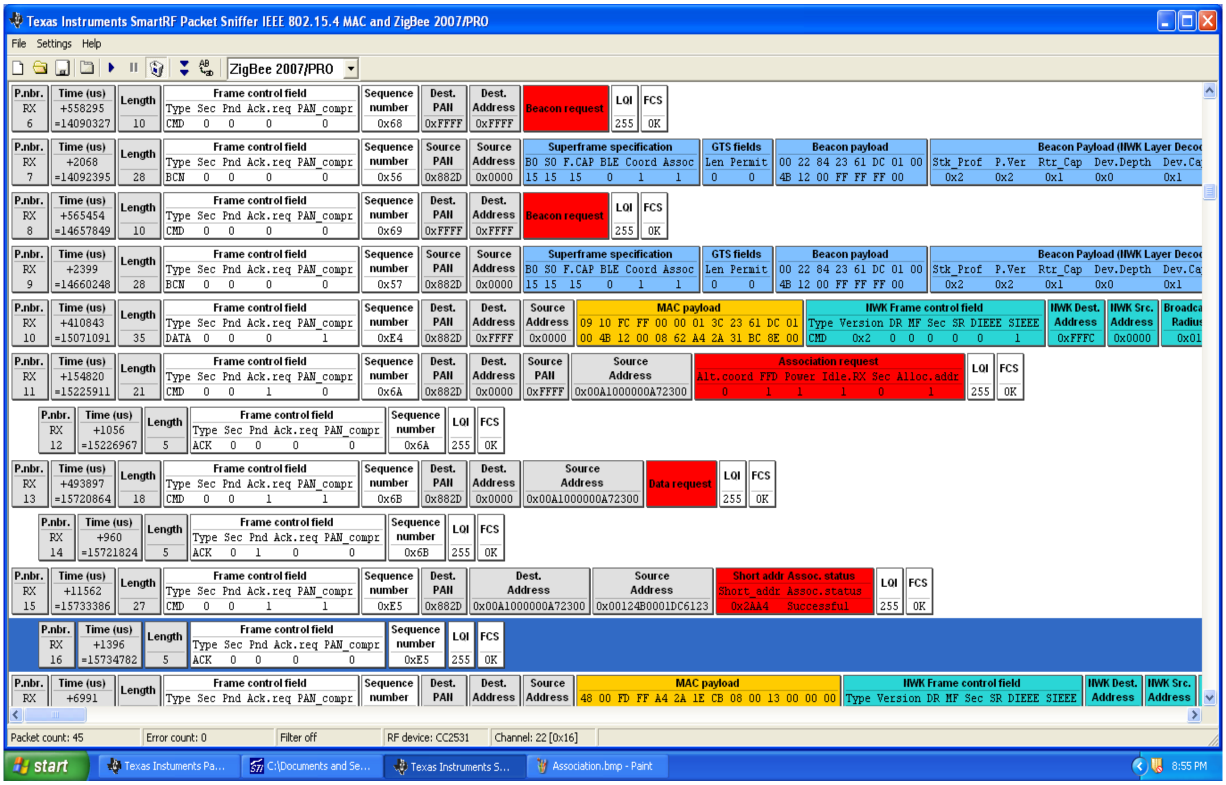
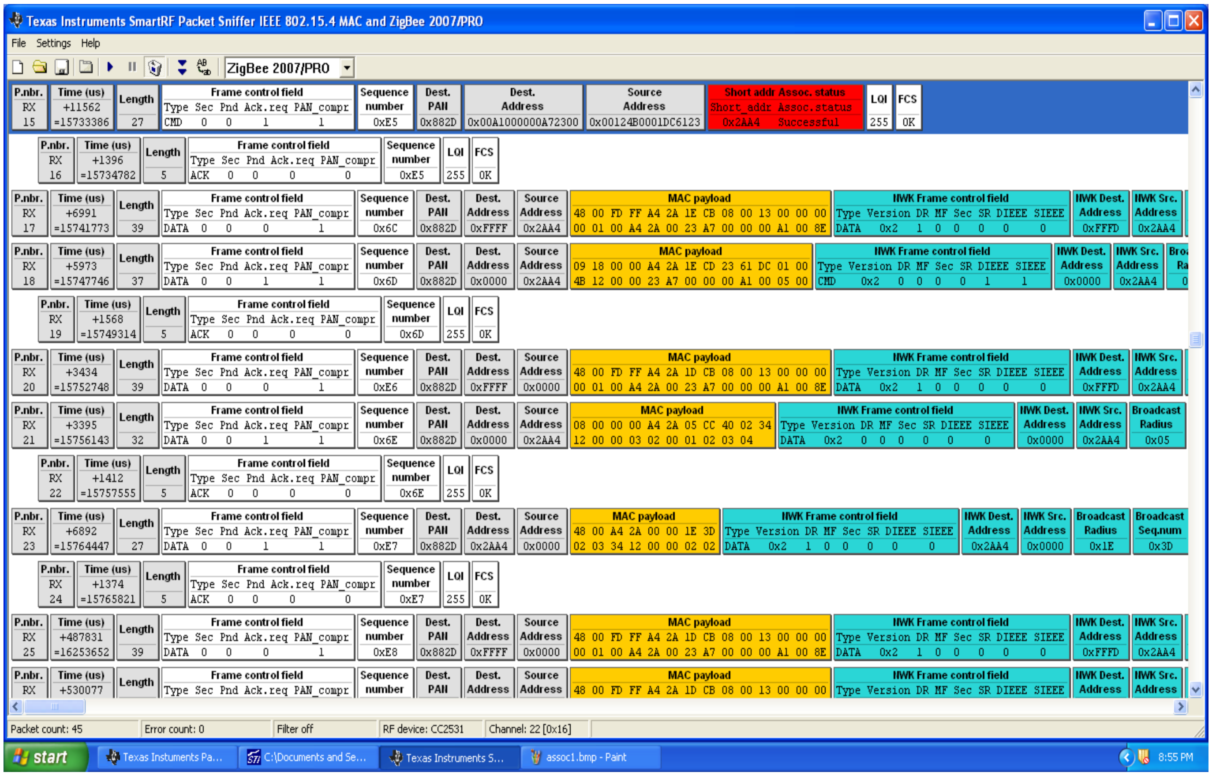


Figure 6 : Coordinator Test Setup

1.13.1 Screenshot for RS9113 association to the coordinator



1.13.2 Screenshot for RS9113 sending Toggle command to the Coordinator



2 HAL Porting Instructions

The following are the general steps required to port the driver on to the target platform.

1) Modify the Hardware abstraction layer (HAL) based on hardware MCU platform and interface selected. For example configuring SPI involves configuring the SPI pins (SPI_MISO /SPI_MOSI/SPI_CLK/SPI_CS), Interrupt signal of module, Clock polarity (CPOL), Clock phase (CPHASE), SPI read/writes API, Data Endianess and optionally timer.
2) Sample HAL specific files are present in the following location in package:

   RS9113.WC.GENR.x.x.x/host/binary/reference_projects/LINUX_WINDOWS/Application/src

   Files for reference:

   - rsi_hal_mcu_uart.c
   - rsi_uart_frame_rd_wr.c
3) Build the APIs along with the application using tool chain provided with the Host MCU.

This following section provides the information on how to customize the HAL layer so that RS9113 ZigBee Pro stack can run on the user's own target platform.

2.1 Frame write

In Linux we are using a sample application for writing the frame to device. The finally prepared frame is sent from Application to device using UART/USB CDC/SPI/USB interface. Frame write is handled in "host/binary/reference_projects/LINUX_WINDOWS/Application/src"

- rsi_frame_write(desc, payload, payload_length) API is called for preparing the frame from the obtained descriptor and payload arguments.
- rsi_alloc_and_init_cmdbuff() API is called to prepare frame from the received descriptor and payload .
- rsi_uart_send() is the last called API to indicate the packet to device. For MCUs port this rsi_uart_send API to interact with the device.

2.2 Frame Read

On Linux platform sample application will read data from device and finally that data/frame is parsed in the zigb_main().

- Application is using a separate thread for reading frame from device "recvThread" it is created in "host/binary/reference_projects/LINUX_WINDOWS/Application/src/main.c". Once the frame is read it is queued to rcv_queue and pkt_pending flag is being set to indicate that a frame is received.
- The zigb_main() thread will check for pkt_pending flag, if set then rsi_frame_read() will be called to dequeue the frame and process it.

- Instead of separate receive thread user can define ISR for handling receive packets. For non OS platforms recv_pkt_serial() should be called from Interrupt handler/ recvThread for performing read operation and enqueuing it to soft queue.
- recv_pkt_serial() will read 4 bytes pre_desc_buf for getting the payload length and actual data offset of frame. Later on the actual packet is read.
- This total handling of data packet reading is handled in a state machine. Finally read data payload is enqueued to soft queue and raise a pkt_pending.

3 Test Mode

In order to test the APIs, "*api_test*" application has to be invoked. The test app sends a command frame and verifies the status in the response frames. At the end of test it displays the list of APIs executed, passed & failed .

> **Note:**
>
> The test app only checks for the status of the response, it may not handle the payloads of the all the response frames.

3.1 Test Application source code

The host side application can found in the following path.
"RS9113.xxZ.WC.GEN.OSI.x.x.x/host/binary/reference_projects/LINUX/Application/zb/src/"

" x.x.x represents the version number of the release

3.1.1 Reference Project:

For reference project refer Reference Project.

3.1.2 Reference API's

Operating System independent API's used in the project are listed in the below specified file:

1) host/binary/apis/zb/ref_apps/src/rsi_zigb_api_test.c – ZigBee APIs
test file , the APIs specific to the device type are validated.
The remaining files are same as described in Sample Application Reference APIs.

3.1.3 Reference Driver

Sample driver is written for communicating with the device over different interfaces (SPI/USB/UART/USB-CDC).

For more information refer PRM document.

3.2 Building the ZigBee API's along with test application

3.2.1 Compiling Application

1) Go to "host/binary/reference_projects/LINUX/Application/zb/src/"

2) To build api test , issue the following command

   # make apitest

3) If you are using different tool chain update the gcc tool chain with the platform specific tool chain in the Makefile present in "host/binary/reference_projects/LINUX/Application/zb/src/" and

   "host/binary/reference_projects/LINUX/Application/wlan/src/". Finally issue "make" command after modification.

3.2.2 Compiling Driver

1) Go to directory "host/binary/reference_projects/LINUX/Driver/"
2) Update the kernel path in the Makefile present in the corresponding interface specific "src/" directory

   For e.g., if you are using SPI interface then modify kernel path in "spi/src/Makefile"

3) Issue the following command to compile driver

   # make

4) Generated "ko" driver module should be used to install driver.

3.3 Execute ZigBee test application

1) Insert the card and check device status(detection) in case of USB

   #dmesg –c

   USB device detection status should be as shown below

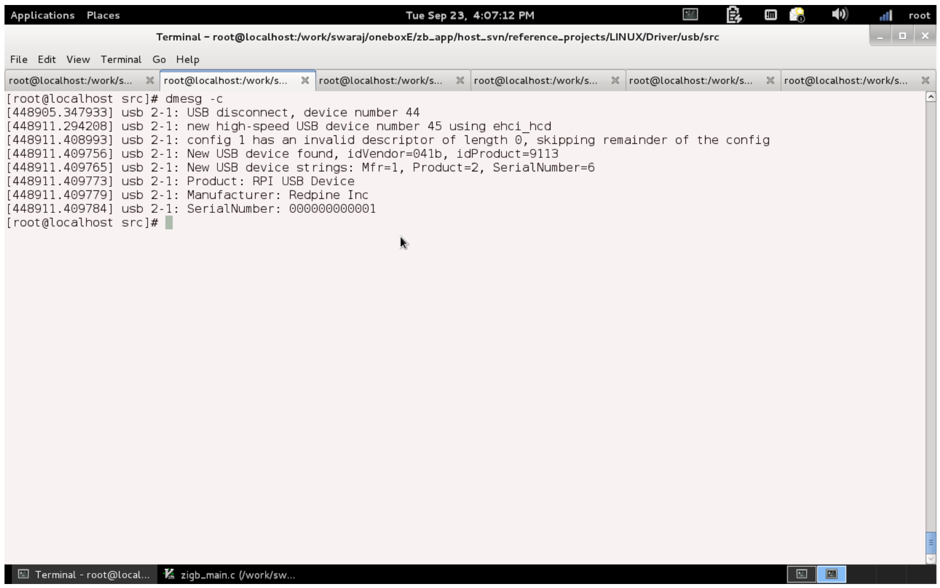


Figure 7: RS9113 USB mode card detection

In case of USB-CDC it should be as shown below

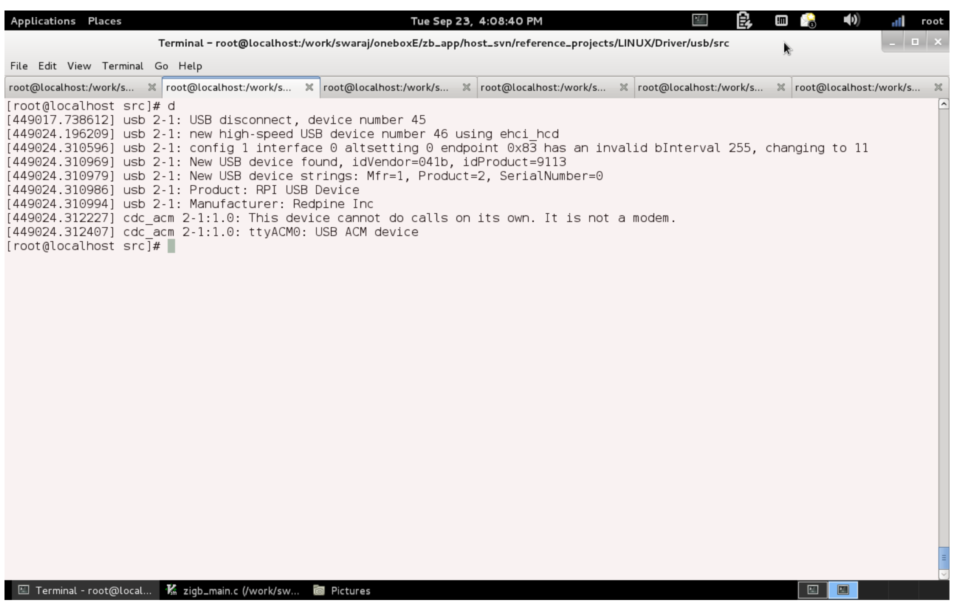


Figure 8: RS9113 USB-CDC mode card detection

3.3.1 Installing Driver

1) Go to driver directory
"host/binary/reference_projects/LINUX/Driver/*<interface>*/src"
2) <interface> can be usb/spi/uart
(uart and usb-cdc will use same uart driver)
3) Insert generated "ko" module, using the following command
insmod rps<interface>.ko
e.g., insmod rpsusb.ko

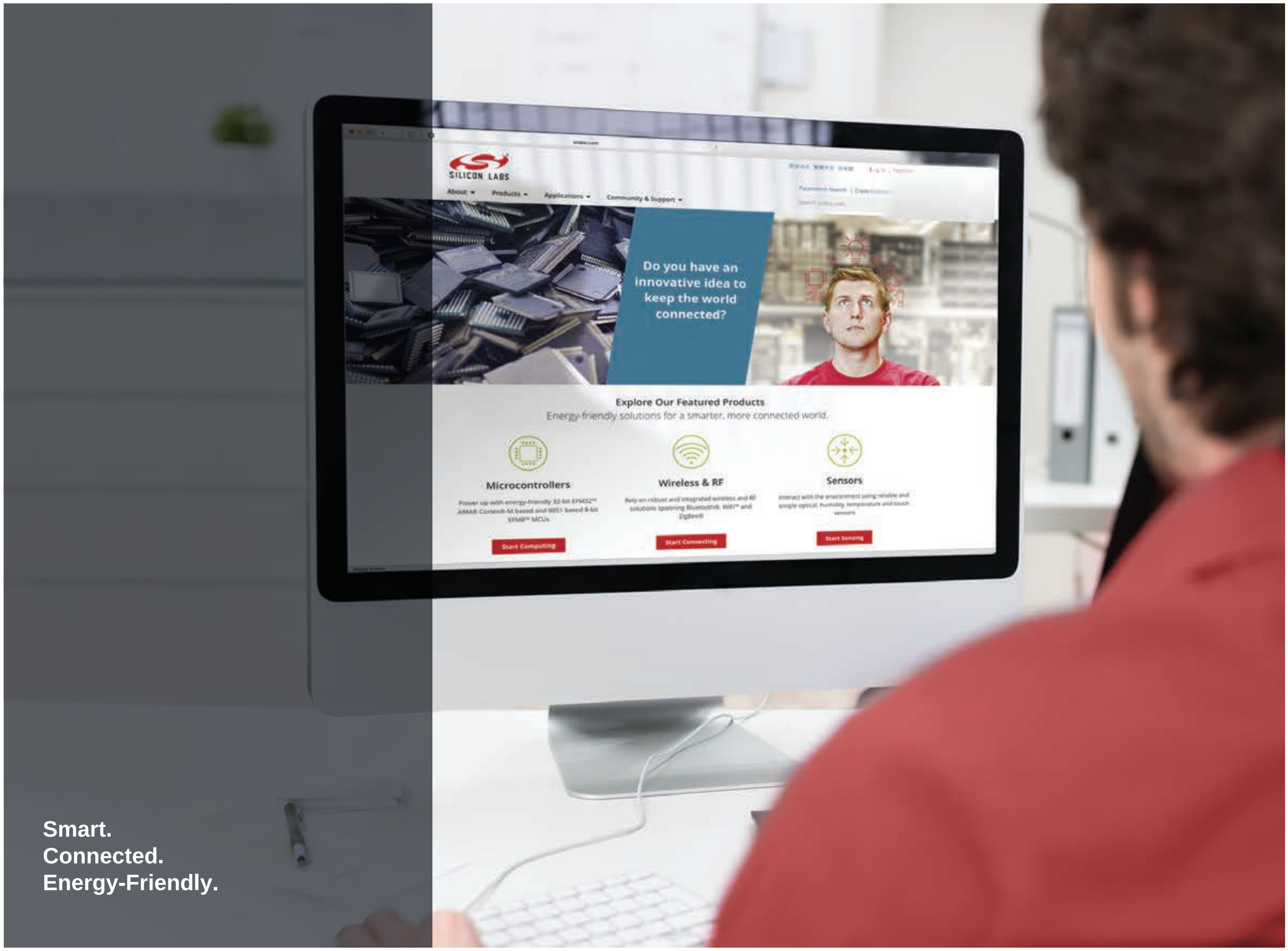
3.3.2 Running Application

3.3.2.1 For USB/SPI interface

1) Go to "host//binary/reference_projects/LINUX/Application/zb/src/"
2) Run ZigBee app by issuing the following command
./rsi_wsc_zigb_app
3) ZigBee app will wait for card ready to proceed further, once wifi app is started then card ready will be received by ZigBee app too.
So, open a new terminal and run wifi application with the following cmd
./rsi_wsc_wifi_app
4) In case if you want to restart ZigBee app issue the following command, which will skip card ready
./rsi_wsc_zigb_app 1

3.3.2.2 For UART/USB-CDC

1) Go to "host/binary/reference_projects/LINUX/Application/wlan/src/"
2) Run the wifi app by issuing the following command
./rsi_wsc_app
3) Go to "host//binary/reference_projects/LINUX/Application/uart/src/" .
4) If the interface is UART ,run the serial application by issuing following command
./rsi_serial
5) If the interface is USB-CDC ,run the serial application by issuing following command
./rsi_serial 1
6) Go to "host//binary/reference_projects/LINUX/Application/zb/src/"
7) Run ZigBee app by issuing the following command
./rsi_wsc_zigb_app 1



Products
www.silabs.com/products

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

Disclaimer

Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice to the product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Without prior notification, Silicon Labs may update product firmware during the manufacturing process for security or reliability reasons. Such changes will not alter the specifications or the performance of the product. Silicon Labs shall have no liability for the consequences of use of the information supplied in this document. This document does not imply or expressly grant any license to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any FDA Class III devices, applications for which FDA premarket approval is required, or Life Support Systems without the specific written consent of Silicon Labs. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons. Silicon Labs disclaims all express and implied warranties and shall not be responsible or liable for any injuries or damages related to use of a Silicon Labs product in such unauthorized applications.

Trademark Information

Silicon Laboratories Inc.®, Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, ClockBuilder®, CMEMS®, DSPLL®, EFM®, EFM32®, EFR, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Ember®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, Gecko OS, Gecko OS Studio, ISOmodem®, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress® , Zentri, the Zentri logo and Zentri DMS, Z-Wave®, and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. Wi-Fi is a registered trademark of the Wi-Fi Alliance. All other products or brand names mentioned herein are trademarks of their respective holders.

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com