NXP Semiconductors
Application Note

Document Number: AN12566
Rev. 0, 09/2019

Host SDK on Linux OS

1. Introduction

This document provides a detailed description for Kinetis Host Application Programming Interface (Host API) implementing the Framework Serial Connectivity Interface (FSCI).

The Host API can be deployed from a PC tool (ie. Test Tool for Connectivity Products) or from a host processor to perform control and monitoring of a wireless protocol stack running on the Kinetis microcontroller. The software modules and libraries implementing the Host API are included in the Kinetis Wireless Host Software Development Kit (SDK).

This document describes how to install all software requirements to develop a Thread, Bluetooth® Low Energy (BLE) and Zigbee examples; running on Kinetis KW41Z devices, which are interfaced from a Linux® OS by the Host API and the Host SDK.

Contents

2. Get the MCUXpresso SDK

The MCUXpresso Software Development Kit (SDK) includes full source code under a permissive open-source license for all hardware abstraction and peripheral driver software.

Click below to download the FRDM-KW41Z SDK.

https://mcuxpresso.nxp.com/en/select?device=FRDM-KW41Z

Note:

This application note is developed on SDK version 2.2.1 from August 2019.

3. Host SDK

The Kinetis Wireless Host SDK consist in a set of cross-platform C language libraries which can be integrated into a variety of user defined applications for interacting with Kinetis Wireless Microcontrollers.

The HSDK software is designed to help developers to interact with Host SDK from Python and C programming languages.

The Kinetis Wireless Host SDK runs on Windows OS, Linux OS, Apple OS X ® and OpenWrt. This document describes a subset of functionalities related to interfacing with Thread, Zigbee and BLE stacks from a Linux OS with focus on C language bindings.

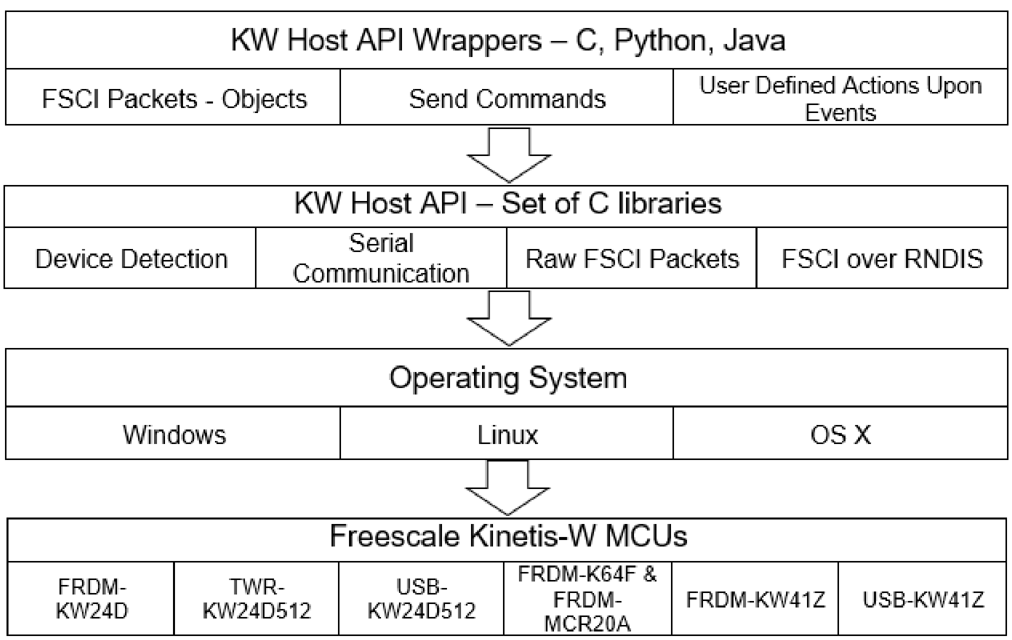


Figure 1. Kinetis Wireless Host Software System Block diagram.

4. Framework Serial Communication Interface (FSCI)

The FSCI module allows interfacing the Kinetis protocol stack with a host system or PC tool using a serial communication interface.

FSCI can be implemented using the set of Linux OS libraries exposing the Host API described in this document and the NXP Test Tool for Connectivity Products PC application (Running on Windows OS).

The FSCI module sends and receives messages as shown in figure 2. This structure is not specific to a serial interface and is designed to offer the best communication reliability. The device is expecting messages in **little-endian** format and responds with messages in **little-endian** format.

An FSCI frame has the following fields:

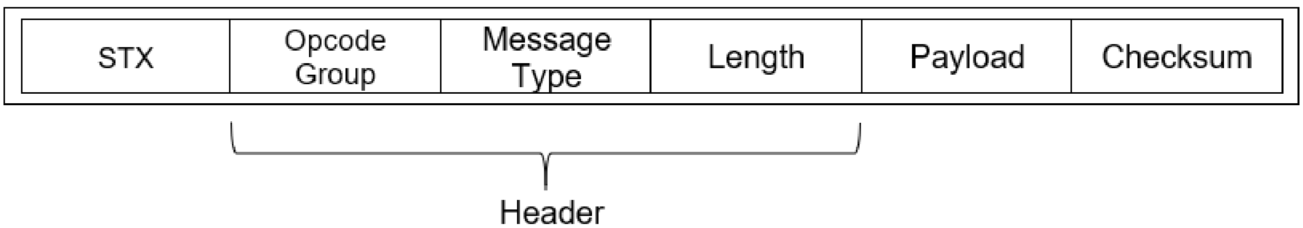


Figure 2. FSCI frame fields.

Table 1.

FSCI Frame Format STX	1 byte	Used for synchronization over the serial interface. The value is always 0x02.
Opcode Group	1 byte	Distinguishes between different Service Access Primitives (MLME, MCPS, GAP, GATT).
Message Type	1 byte	Specifies the exact message opcode that is contained in the packet.
Length	2 bytes	The length of the packet payload, excluding the header and the checksum. The length field content shall be provided in little endian format.
Payload	Variable	Payload of the actual message.
Checksum	1 / 2 bytes	Field used to check the data integrity of the packet. When virtual interfaces are used to distinguish between the BLE and Thread stacks when both running concurrently on the same device, this field expands to two bytes to embed the virtual interface number.

The FSCI messages acts as a Remote Procedure Call (RPC) mechanism in which a message triggers a remote procedure/callback.

5. Linux OS Host Software Installation Guide

Prerequisites

5.1.1. SDK libraries

Unzip the previously downloaded FRDM-KW41Z SDK, most of the files used in this document are on the path: *<SDK path>\tools\wireless\host_sdk\hsdk>*

To download FRDM-KW41Z SDK, please refer to Section 2.

5.1.2. Test Tool Application

To download Test Tool application, go to https://www.nxp.com/ and search 'Test Tool for connectivity products'.

NOTE

Test Tool application only runs on Windows OS.

5.1.3. Linux packages

Following packages are required before starting any demo:

- build-essential
- libudev-dev
- libpcap-dev

Use 'apt-get install' on Debian-based distributions.

```
$apt-get install build-essential
$apt-get install libudev-dev
$apt-get install libpcap-dev
```

Install HSDK libraries

Open a terminal in the directory: '*~\<SDK path>\tools\wireless\host_sdk\hsdk*'. Execute the following commands to build and install the libraries required by the HSDK:

```
$make
$sudo make install
```

6. Libraries description

Directory tree

You can find hsdk folder at: *<SDK path>\tools\wireless\host_sdk\hsdk*'

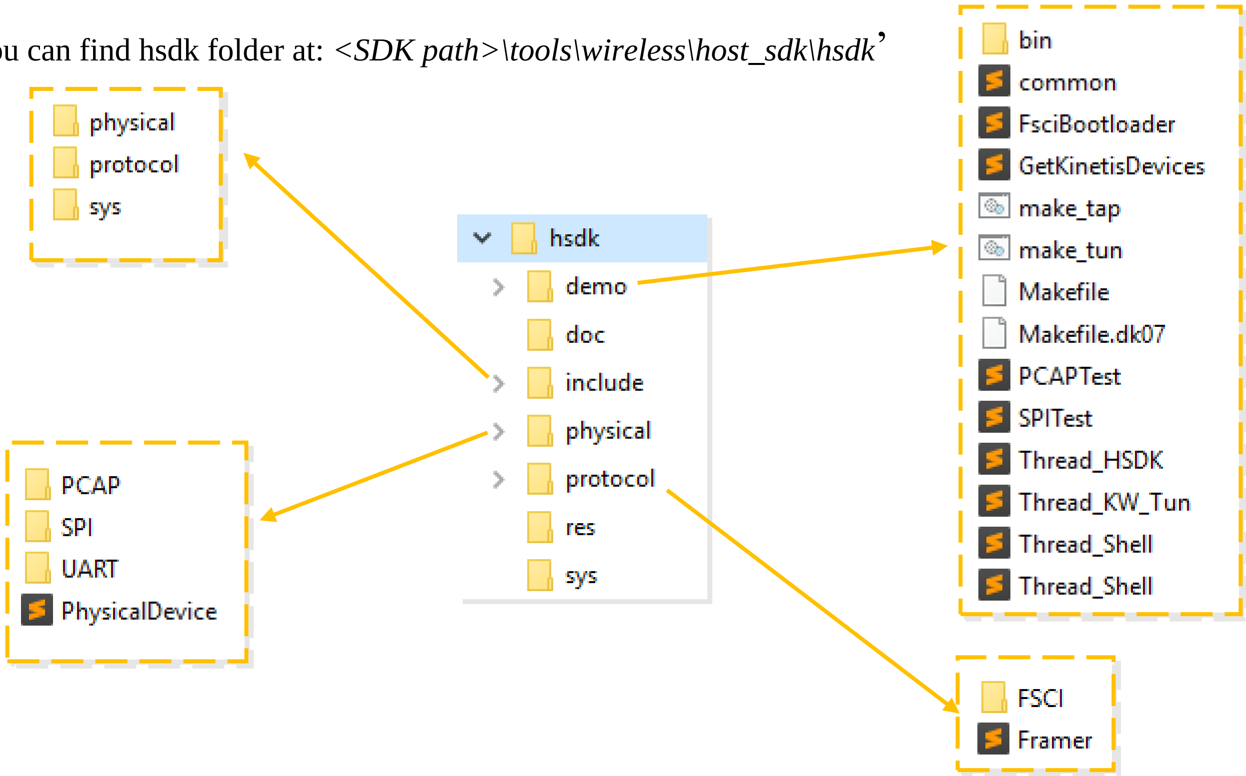


Figure 3. directory tree.

- demo
  Contains all demo source codes provided in the HSDK, and the makefile to compile them.
  - bin: contains executable files of every demo.
- include
  Every library used in demos are placed in this folder. These libraries have only prototype functions.
  - physical: has all physical protocols used to set a communication with the Kinetis device.
  - protocol: in this folder are placed the libraries to create the framer and the FSCI frame.
  - sys: has general libraries needed to work with an OS.
- physical
  Contains every .C files of the physical library find in the 'include' folder.
  - PCAP: has all functions to set a PCAP communication.
  - SPI: has all functions to set an SPI communication.
  - UART: has all functions to set a UART communication.
- protocol: contains every .C files of the protocol library find in the 'include' folder.

- FSCI: in this folder are placed the libraries to create the FSCI frame.

Below is a small description of the header files required to develop an HSDK application.

- *'Framer.h'* and '*FSCIFrame.h*' are headers needed to create, destroy and send the FSCI frames from the computer to the board. Contains information about the structure of the FSCI frame and all prototypes functions that can be used. It can be found in the following directory:

  '<SDK path>\tools\wireless\host_sdk\hsdk\include\protocol'
- *'PhysicalDevice.h'* is needed to attach the board with the computer by UART protocol (more device types are supported by the library). The library can be found in the next directory:

  '<SDK path>\tools\wireless\host_sdk\hsdk\include\physical'
- *'UARTConfiguration.h'* has all structures and enumerations to configure UART protocol communication. It can be found in the following directory:

  '<SDK path>\tools\wireless\host_sdk\hsdk\include\physical\UART'

Each HSDK demo has its own header with all operation groups and codes needed by the FSCI frame protocol. These headers are located in the following directory: *'<SDKpath>\tools\wireless\host_sdk\hsdk\demo'*.

7. Framer

A framer is required to specify every field that contains in the message packet. The API that is provided for HSDK supports a variety of protocols, allowing the developer to create his own frames manipulating every field of the framer such as the endianness, CRC field size, length field size, etc. Also, it is used to specify the port where the physical device is connected to the computer.

Create framer

The construction of this frame is made with the following functions:

```
InitPhysicalDevice(DeviceType type, void *pConfigData, char *deviceName, FsciAckPolicy
policy)
```

'PhysicalDevice.h'

This function defines the following parameters:

- Communication protocol (UART, SPI, USB…).
- Communication protocol data configuration.
- The port where the Kinetis device is connected.
- Policy for FSCI acknowledge synchronization (none, TX, RX, both, global).

```
InitializeFramer(void *connDev, FramerProtocol protocol, uint8_t lengthFieldSize, uint8_t
crcFieldSize, endianness endian);
```

'Framer.h'

This function constructs the framer fields. Its parameters are:

- Physical device. The one created with the first function `InitPhysicalDevice()`.
- Frame protocol type (FSCI, HCI, ASCII).
- Length field size.
- CRC field size.
- Endianness of the framer.

```
OpenPhysicalDevice(PhysicalDevice *);
```

‘PhysicalDevice.h’

Open the specific device created in the first function `InitPhysicalDevice()`. Parameters:

- Physical device. The one created with the first function `InitPhysicalDevice()`.

8. FSCI frame

Frame protocol allows monitoring an extensive testing of the protocol layer interfaces. It also allows the separation of the protocol stack between two protocol layers in two processing entities set up, the host processor (typically running the upper layers of a protocol stack) and the Black Box application (typically containing the lower layers of the stack, serving as modem).

Refer to Connectivity Framework Reference Manual document at section 3.11 Framework Serial Communication Interface to get more information about it. The document can be found in the directory: *<SDK path>\docs\wireless\Common.*

Obtain data frame from Test Tool Application

Test Tool for connectivity products provides several loaded commands sets for every connectivity protocol (BLE, Zigbee, Thread, SMAC). Also, it provides a serial window view where the user can watch each byte contained in every FSCI command sent and received by the host computer.

These characteristics make the Test Tool a very useful application to analyze FSCI commands.

8.1.1. Load a host demo to FRDM-KW41Z

For example, you can load a Thread ‘*host_controlled_device*’ demo to one FRDM-KW41Z board. The demo project can be found in the directory:*’ <SDK path>\boards\frdmkw41z \wireless_examples\thread\’.*

8.1.2. Use Test Tool application

Connect the board to the computer with **Windows OS,** open Test Tool application and double-click on the COMx on Active devices.

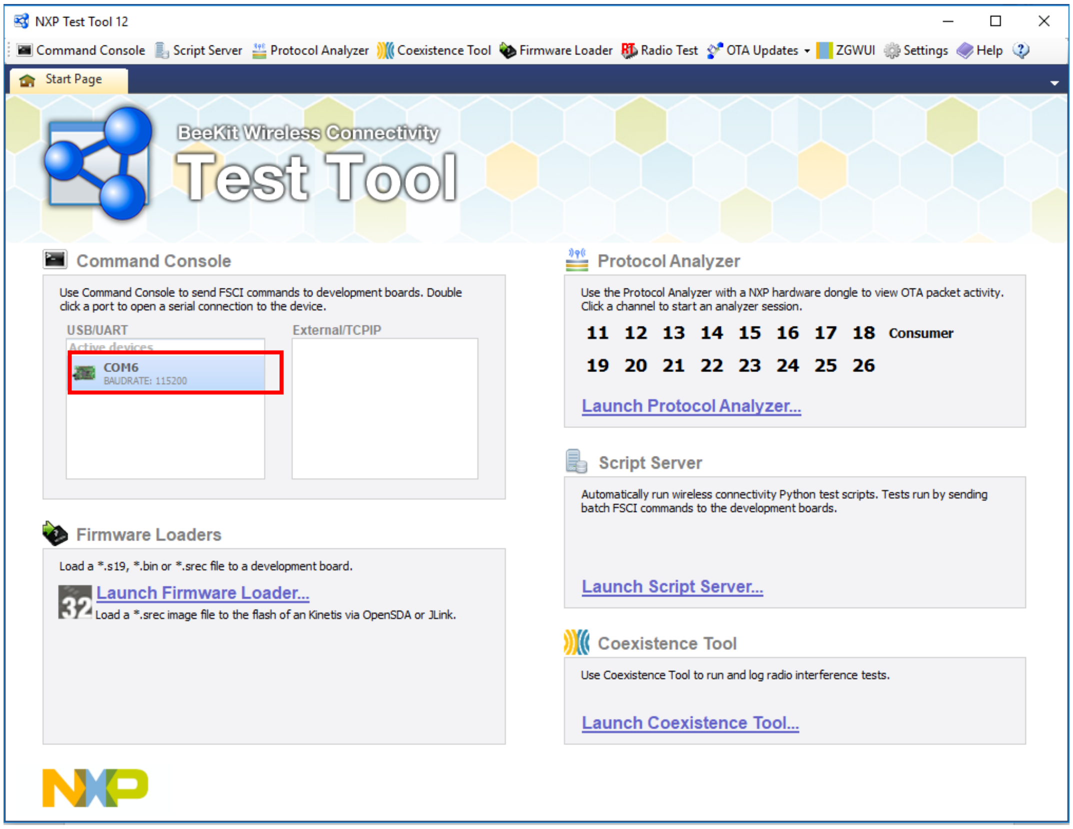


Figure 4. Test Tool home page.

Verify that the 'Loaded Command Set' corresponds to the protocol stack that you just loaded (Zigbee, Thread, BLE) and check Raw Data checkbox at the bottom.

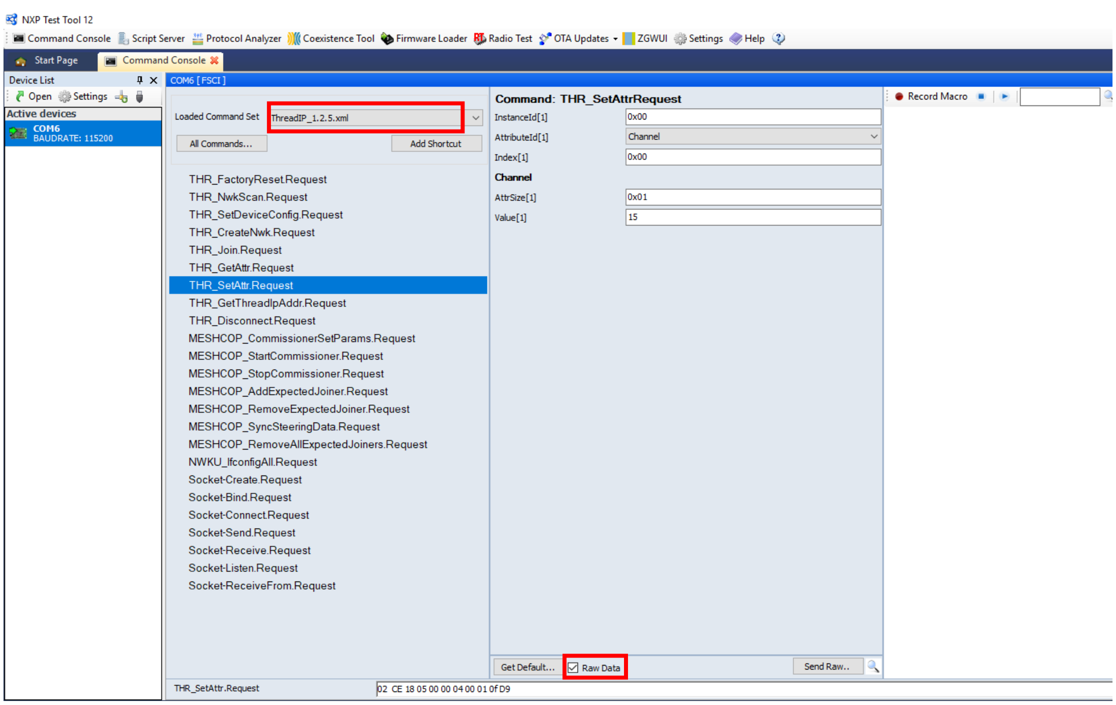


Figure 5. Test Tool command window.

8.1.3. Send and receive packets

The FSCI frame example described below sets the **channel 15** as an attribute in a **Thread** network. Frame data is obtained from the Test Tool application specifying each one of the fields and indicating if the message is an RX or TX packet.

Below is a description of each one of the steps that the developer should follow:

1. Select the command to be sent.
2. Select 'Channel' on the "AttributeId" field, set the channel number on the value field and send the command.

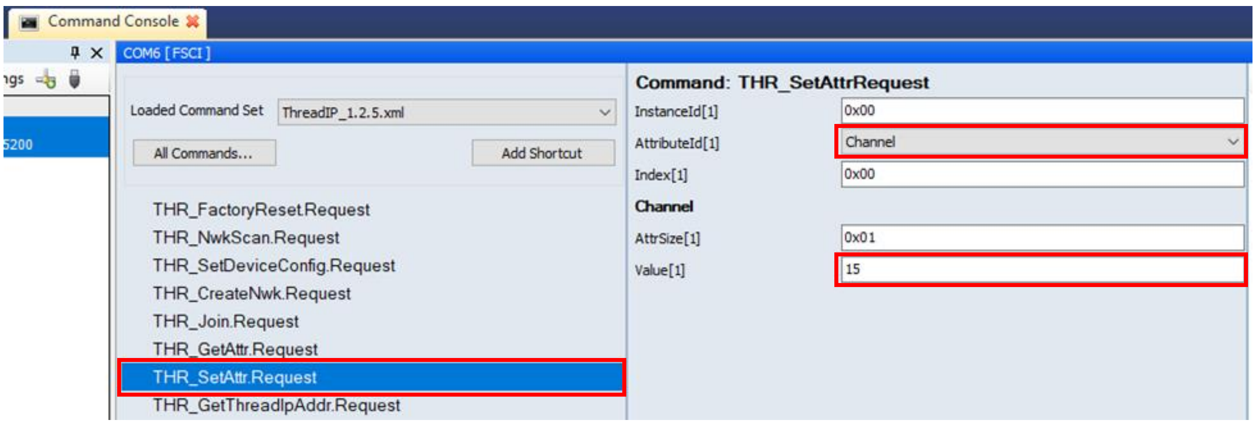


Figure 6.

RX and TX messages appear on the right side of the screen. With this information, the developer can watch all fields in the sent frame.

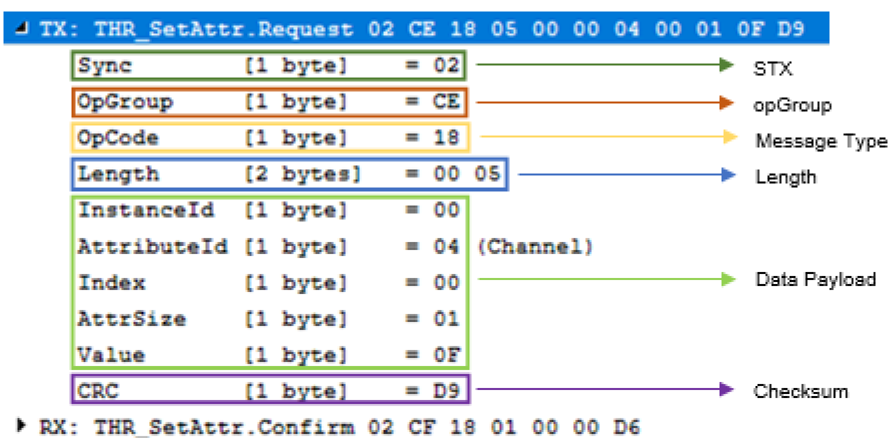


Figure 7. Frame fields.

The payload data from the TX message is shown at the bottom of the screen. Taking this raw data, the developer can create the buffers for every command that can be sent to the board.

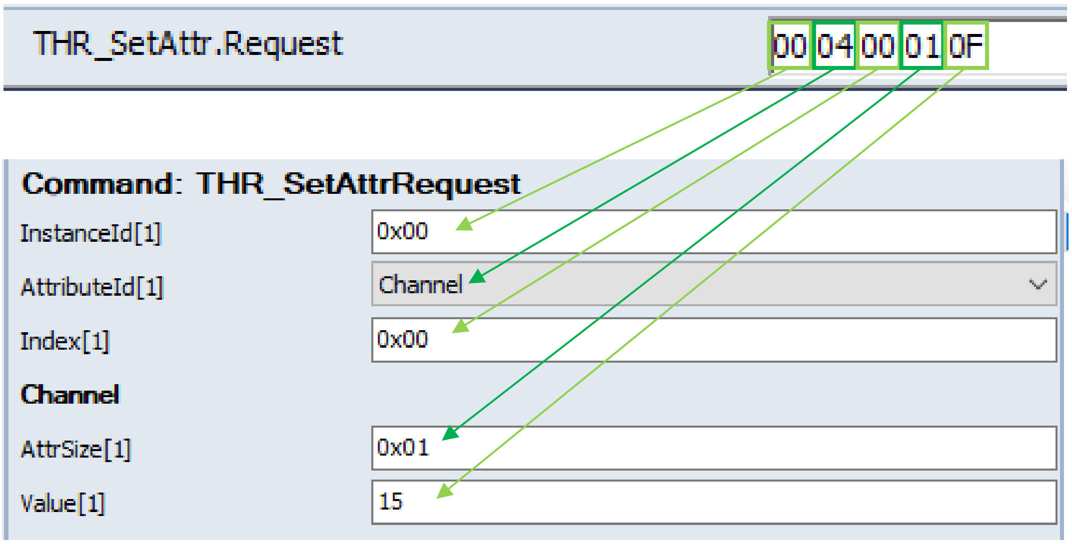


Figure 8. Frame payload.

Analyzing the frame fields of the RX message:

- opGroup: Is different than the TX message opGroup byte.
- Message Type: Is the same as the TX message opCode byte.
- Status (payload): A zero response indicates a successful execution command.

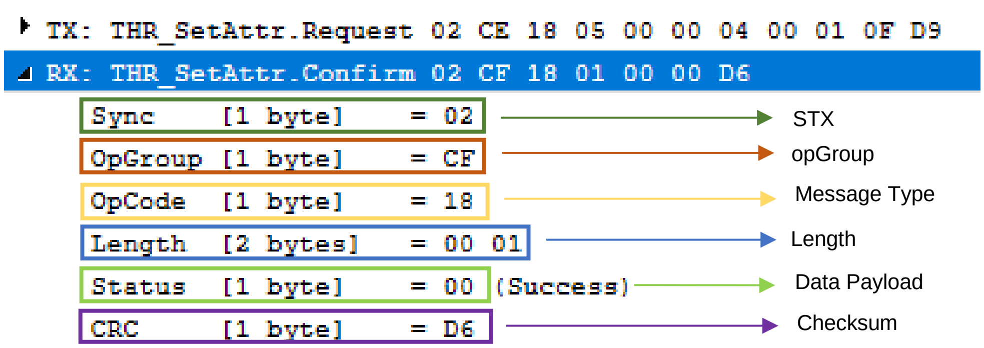


Figure 9. Frame fields.

Coding the frames

8.2.1. Function description

'*FSCIFrame.h*' offers several functions to create FSCI frames. This is a brief description of the '`CreateFSCIFrame()`' function.

The function receives six parameters:

- framer: The function uses this parameter to set the length and the endianness frame size. Refer to 'Chapter 6. Framer' from this document to obtain more information about it.
- opGroup: Every FSCI frame has a code to identify if the frame is an RX or TX packet. The opGroup codes that are supported in this example can be found in 'common.h' with an 'OG' ending name.
- opCode: Every FSCI frame has its own opCode that specifies the command that wants to be executed by the board (factory reset, create network, join to network, send coap message, etc.). Every opCode that is supported can be found in 'common.h' with an 'OC' ending name.
- data: Pointer to the data payload that is going to be sent to the board.
- length: Payload length.
- virtualId: Specifies if the frame uses a virtual interface or not.

8.2.2. Select the opGroup & opCode

TX message opGroup and opCode provided in 'Section 7.1.3 - Send and Receive Packets' are the ones that will be used to construct the frame.

opGroup	0xCE	This byte corresponds to a TX message. Every command sent to the board will have this opGroup.
opCode	0x18	This byte corresponds to set an attribute to the network.

Every opGroup and opCode supported can be found in 'common.h' file.

8.2.3. Create data payload buffer

A data buffer is required to send different FSCI commands to the board. This buffer can be created using the information collected in 'Send and receive packets 8.1.3'. In this case, the payload buffer has the next bytes:

```
static uint8_t set_ch_buf[] = {THR_INST_ID, 0x04, 0x00, 0x01, 0x0F};
```

8.2.4. Create FSCI frame

By now, the developer has all information required to create his own FSCI frame. The `'CreateFSCIFrame()'` function and the parameters used to create the frame is shown below.

```
FSCIFrame *set_channel = CreateFSCIFrame(framer, TX_OG, THR_SET_ATTR_OC, set_ch_buf,
sizeof(set_ch_buf), VIF);
```

8.2.5. Send frame

'Framer.h' header provides the `'SendFrame()'` function. Following are the parameters that are used by this function:

- framer: this parameter is used to set the length and the endianness frame size. To obtain more information about it see chapter 7 Framer.

- frame: can be any FSCI frame created by the developer.The FSCI frame created in section 8.2.4 Create FSCI Frame is used in this document.

```
SendFrame(framer, set_channel);
```

8.2.6. Destroy FSCI Frame

'Framer.h' library provides the `'DestroyFSCIFrame()'` function. This function is used to deallocate the memory space required by the frame. The function receives the FSCI frame to be destroyed.

```
DestroyFSCIFrame(FSCIFrame *);
```

9. Frame callback

In this example, every time a TX message is sent to the client device, the board responds with an RX message.

First, the developer needs to attach a callback to the framer created in Section 6.1 Create Framer. This callback is executed on every RX packet. '*Framer.h'* header provides *'*`AttachToFramer()`*'* function:

```
AttachToFramer(framer, NULL, callback);
```

Analyzing the frame received in section 7.1.3 Send and Receive Packets, the developer can execute specific tasks depending on the received message. The opCode field can be used to filter the messages.

```
static void callback(void *callee, void *response)
{
    FSCIFrame *frame = (FSCIFrame *) response;
   if (frame->opGroup != THR_RX_OG && frame->opGroup != MWS_RX_OG) {
       DestroyFSCIFrame(frame);
       return;
    }
   switch (frame->opCode) {
       case THR_SET_ATTR_OC:
              printf("RX: THR_SetAttr.Confirm");
              if (frame->data[0] == 0x00) {
                  printf(" -> Success\n");
              }
             break;
    DestroyFSCIFrame(frame);
```

```
}
```

10. Add source files

This application note consists of eight source files that are not included in the SDK folder. To run the following demos correctly, you need to copy these files in the 'hsdk\demo' path:

'<SDK path> \tools\wireless\host_sdk\hsdk\demo'

Make sure to replace the old Makefile with the new one to compile all the source files with a single command.

These source files can be found as a ZIP named 'Host SDK on Linux OS' at Application Note Software section.

11. Configure makefile

Go to: *'<SDK path> \tools\wireless\host_sdk\hsdk\demo'* and open the file named 'Makefile' with a text editor of your preference. This file is used to resolve any dependency on the used libraries and compile every demo.

If the developer wants to add his own file, make sure the name of your application file is written in the line that is shown below, if it is not, write it.

```
build: clean pre-build FsciBootloader GetKinetisDevices Thread_KW_Tun Thread_Shell PCAPTest
```

Then, go to the end of the makefile and add the build profile of your project as it-s shown in the figure below.

```
PCAPTest: PCAPTest.o
	$(CC) $(BUILDDIR)/$^ -o $(BINDIR)/$@ $(HSDK_LIBS) $(LDFLAGS)
PCAPTest.o: PCAPTest.c
	$(CC) $(CFLAGS) $(BUILDFLAGS) $^ -o $(BUILDDIR)/$@

SPITest: SPITest.o
	$(CC) $(BUILDDIR)/$^ -o $(BINDIR)/$@ $(HSDK_LIBS)
SPITest.o: SPITest.c
	$(CC) $(CFLAGS) $(BUILDFLAGS) $^ -o $(BUILDDIR)/$@

File_Name: File_Name.o
	$(CC) $(BUILDDIR)/$^ -o $(BINDIR)/$@ $(HSDK_LIBS)
File_Name.o: File_Name.c
	$(CC) $(CFLAGS) $(BUILDFLAGS) $^ -o $(BUILDDIR)/$@
```

Add here

Figure 10. Makefile view.

Substitute 'File_Name' with your own file name.

12. Compile an application

To compile any demo application, follow the next step on the computer with **Linux OS**:

- Open a terminal in the next directory: *'<SDK path>\tools\wireless\host_sdk\hsdk\ demo'* and execute the next command:

```
$make
```

If user wants to compile his own application, make sure to previously configure the makefile file (refer to Chapter 10. Configure makefile).

13. Thread Shell demo

This demo allows the developer to experiment and become familiar with the Thread Host Control Interface (THCI) messages.

To get more information about this demo, please refer to the document: Kinetis Thread Stack Shell Interface User's Guide. The document can be found in the following directory:*' <SDK path>\docs\wireless\Thread'*.

Prerequisites

To make this example work correctly, the boards will have an identification letter as is shown in the next figure:

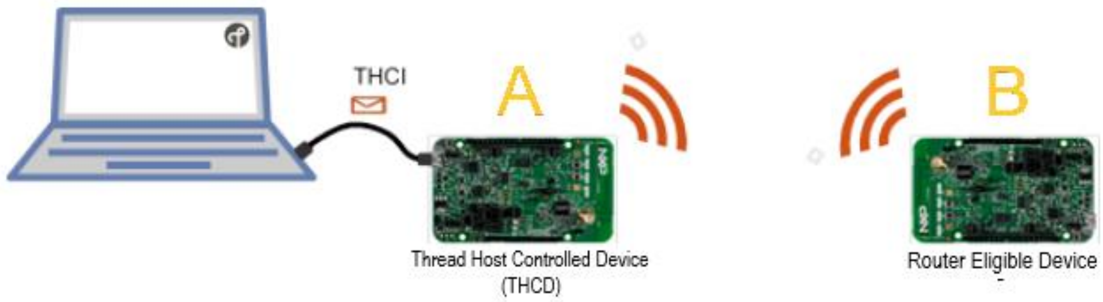


Figure 11. Board setup.

Load THCD firmware to both boards as follows:

- **Board A**:
  Load '*host_controlled_device*' example. The project can be found:
  ' <SDK path>\boards\frdmkw41z \wireless_examples\thread\'.
- **Board B**:
  Load '*router_eligible_device*' demo. The project can be found:
  ' <SDK path>*\boards\frdmkw41z \wireless_examples\thread\'.*

Run demo application

To execute this demo, follow the next steps:

- Compile demo. See Chapter 10. Compile an Application.
- Open a terminal in the next directory: *'<SDK path>\tools\wireless\host_sdk\hsdk\ demo\bin'*.
- Execute 'GetKinetisDevices' program to obtain the port where the Kinetis device is connected.

- Execute ‘Thread_Shell’ application.

```
$sudo ./GetKinetisDevices
 NXP Kinetis-W device on /dev/ttyACM0

$sudo ./Thread_Shell /dev/ttyACM0 15 115200
```

Parameters: Port Channel Baud rate

Note:

If you are running these demos on a Virtual machine you need to enable the USB controllers in the virtual machine settings, then the board will be recognized as a connected device.

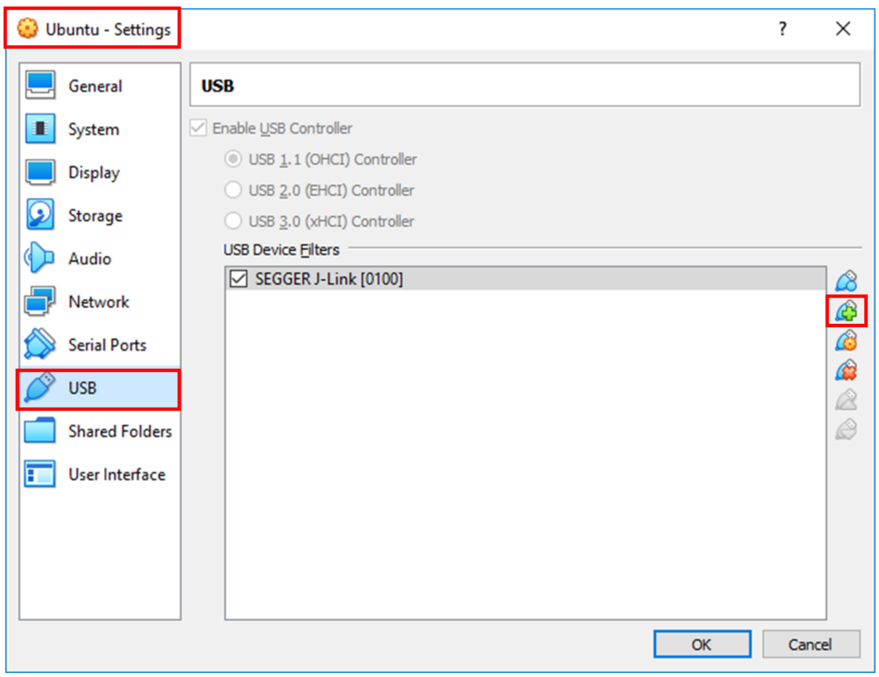


Figure 12. Virtual machine configuration.

14. Thread HSDK demo

Refer to ‘Kinetis Thread Host Control Interface Reference Manual’ document to have a more detailed description of each one of the available Thread messages. The document can be at: *’<SDK path>\docs\wireless\Thread’.*

The *Thread HSDK demo* makes use of the HSDK APIs to send several packets with 1 second interval between each one. The host device sends the following commands to the serial interface and the client device will execute them:

- Create network.
- Start Commissioner.
- Allow nodes to connect.

When a node joins the network, perform the following using Mesh Local Address:

- Print node data.
- Send one ICMP Echo request (Ping).
- Send one CoAP message “/led toggle” with no ACK. (User verify if the LED was toggled).
- Send one CoAP message “/temp” with ACK.
- Open a Socket and send data to the remote node.
- Print any Socket request coming from the network.

Prerequisites

To make this example work correctly, the boards have an identification letter as is shown in below figure:

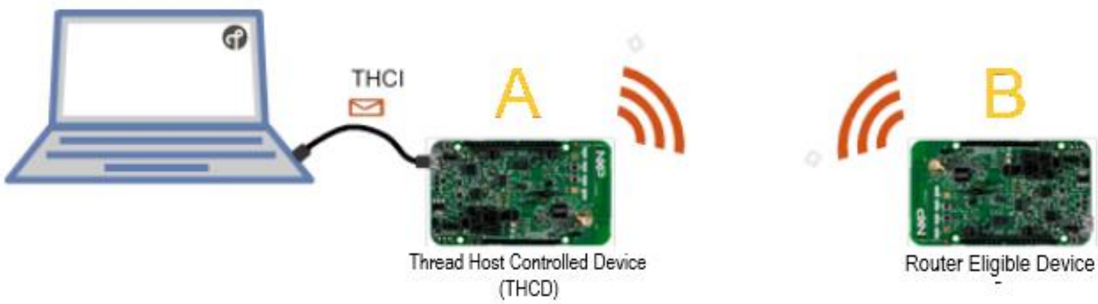


Figure 13. Board identification.

Load THCD firmware to both boards as follows:

- **Board A**:

  Load ‘*host_controlled_device*’ example. The project can be found:

  ’ <SDK path>\boards\frdmkw41z \wireless_examples\thread\’.
- **Board B**:

  Load ‘*router_eligible_device*’ demo. The project can be found:

  ’<SDK path>*\boards\frdmkw41z \wireless_examples\thread\’.*

  To enable the router_eligible_end_device to send and receive socket messages it is necessary to set SOCK_DEMO to 1. This macro can be found in the ‘config.h’ file.

Run demo application

To run any demo application (or your own application) follow these steps:

- Compile demo. See chapter 10 Compile an Application.
- Connect the **Board A** to the computer with **Linux OS**.
- Open a terminal in the next directory: *‘<SDK path>\tools\wireless\host_sdk\hsdk\ demo\bin’*.
- Execute ‘GetKinetisDevices’ program to obtain the port where the Kinetis device is connected.
- Execute ‘Thread_HSDK’ program.

```
$sudo ./GetKinetisDevices
 NXP Kinetis-W device on /dev/ttyACM0

$sudo ./ Thread_HSDK /dev/ttyACM0 15 115200
```

Parameters: Port Channel Baud rate

Demo description

14.3.1. Create Thread Network

The *Thread HSDK* application will starts and you see the information as in below image on your terminal:

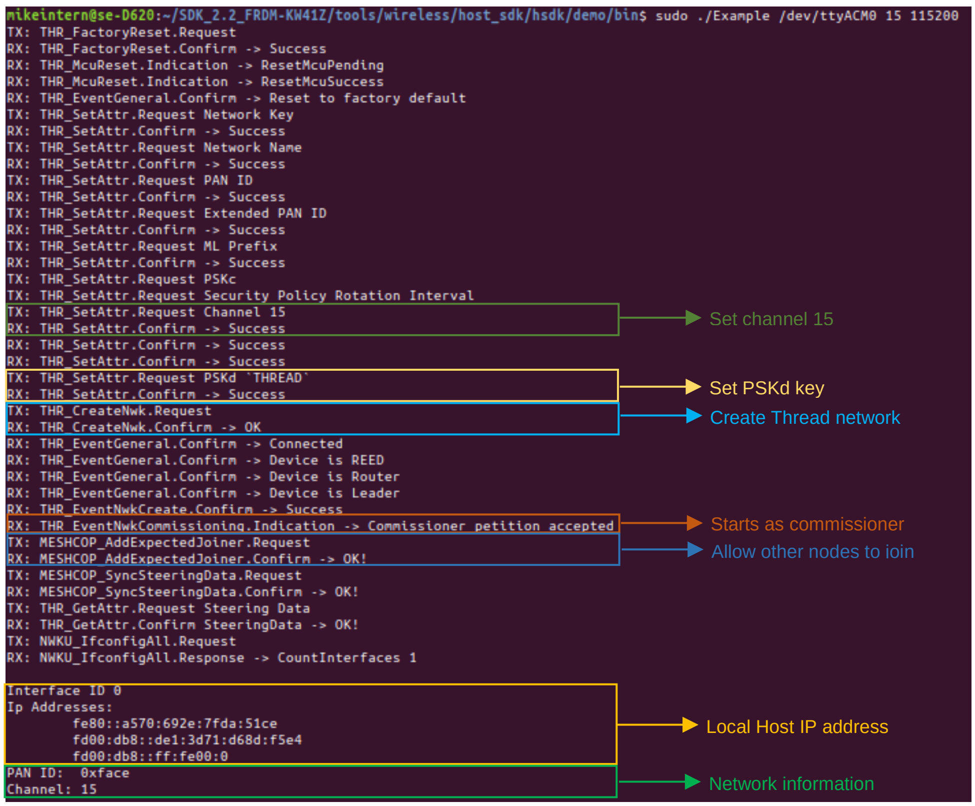


Figure 14. Board A, Thread Network creation.

The host sets the channel, PAN ID, Extended PAN ID, and other several attributes also starts the network as commissioner.

When network is created, the demo displays the host IP addresses and the network information by sending ‘ifconfig’ FSCI frame. You can enable or disable the use of this command by setting USE_IFCONFIG at the top of the ‘Thread_HSDK.c’ file.

14.3.2. Join New Node

Connect the **Board B** and open a serial terminal with Tera Term or PuTTY. Configure the serial terminal with the following parameters: Baud rate: 115200, 8-bit data, 1 stop bit, No flow control, No parity.

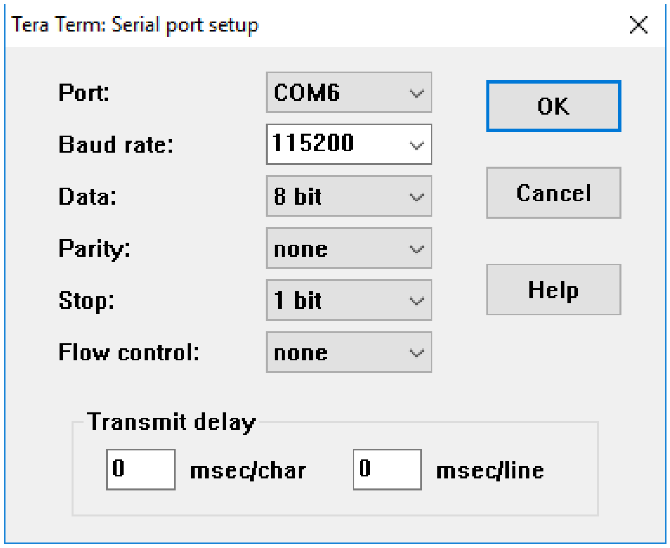


Figure 15. Serial terminal configuration.

Press the reset button on the **Board B** and enter the next thr commands on the serial terminal:

```
$ thr scan allchannel
$ thr set channel 15
$ thr join
```

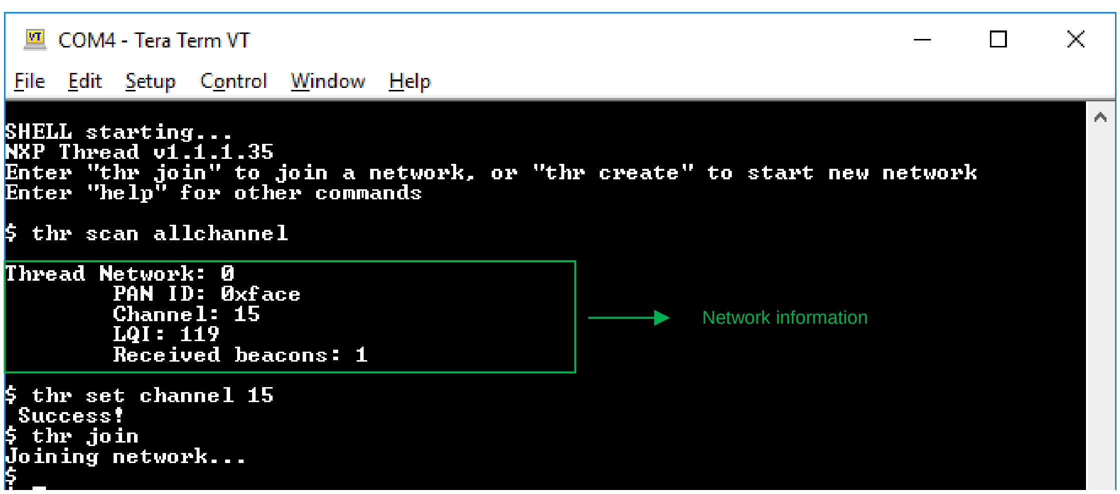


Figure 16. Board B, 'thr' commands.

The end device joins to the Thread network. You can see the outputs as below on the Linux terminal showing the commissioner status:

```
Interface ID 0
Ip Addresses:
        fe80::ad0c:c780:e866:128f
        fd00:db8::e12c:1693:5c7:a877
        fd00:db8::ff:fe00:0
PAN ID:  0xface
Channel: 15

RX: THR_EventNwkCommissioning.Indication -> Commissioner Joiner DTLS session started
RX: THR_CommissioningDiagnostic.Indication -> IN JOIN_FIN_REQ
RX: THR_CommissioningDiagnostic.Indication -> OUT JOIN_FIN_RSP
RX: THR_EventNwkCommissioning.Indication -> Commissioner Joiner accepted
RX: THR_EventNwkCommissioning.Indication -> Joiner DTLS alert
RX: THR_CommissioningDiagnostic.Indication -> OUT JOIN_ENT_REQ
RX: THR_EventNwkCommissioning.Indication -> Providing the security material to the Joiner
RX: THR_CommissioningDiagnostic.Indication -> IN JOIN_ENT_RSP
```

A node has joined

Figure 17. Board A, commissioner status.

```
$ thr join
Joining network...
$
Commissioning successful

Attached to network with PAN ID: 0xface
```

Join to the Thread network with the same PAN ID

Figure 18. Board B, joining network.

14.3.3. Print joiner's IPv6 addresses

This information is obtained by sending 'getnodesip' FSCI frame. You can enable or disable this command by setting USE_GET_NODES_IP.

```
TX: THR_MgmtDiagnosticGet.Request
RX: THR_MgmtDiagnosticGet.Confirm -> OK!
RX: THR_MgmtDiagnosticGetRsp.Indication -> OK!
        Node Short Address: 0001
                fe80::b042:ba2f:4050:7fa6
                fd00:db8::b569:eaa2:a61:d65
                fd00:db8::ff:fe00:1
```

IP addresses of new joiner

Figure 19. Joiner IPv6 address.

14.3.4. Ping request/response

You can enable or disable this command by setting USE_PING.

```
Ping to fd00:db8::ff:fe00:1 with 20 bytes of data
Reply from fd00:db8::ff:fe00:1: bytes=20 time=22ms
```

Ping req/rsp

Figure 20. Ping request/response.

14.3.5. CoAP messages

First, the host sends a create instance request and receives the confirmation. Then, sends both CoAP messages ("/led toggle" and "/temp") and receives only the temperature response.

You can enable or disable this command by setting USE_COAP.

```
RX: NWKU_CoapCreateInstance.Confirm -> OK!
RX: NWKU_CoapMsgReceived.Indication
        Status -> Success
        Payload -> Temp:23.00
```

CoAP messages req/rsp

Figure 21. CoAP message.

LED control.

You can put in different states your led end device.

Figure 22. LED status.

14.3.6. Socket messages

The host opens a socket port and send data to the end device.

Figure 23. Socket message.

See the received socket data in the router_eligible_end_device serial terminal.

Figure 24. Socket received data.

You can enable or disable this command by setting USE_SOCKET.

Send CoAP and Socket messages from the HSDK

14.4.1. CoAP

14.4.1.1. Modifications on the board side

User can send his own CoAP messages to the host. To do this, a new uri-path must be added on the host-controlled device. Follow the next community post to add a new uri-path: https://community.nxp.com/docs/DOC-333784.

Once the new uri-path callback has been added, the user can use `THCI_EventData()` function to send data to the host (Linux computer) through thci message.

```
THCI_EventData(uint8_t opCode, uint8_t length, uint8_t *pData);
```

This function can be found in 'thci.h' file. It can be used to send data from **board A** to the host computer, specifying the opCode and a pointer to the sending data.

NOTE

All these modifications must be implemented on the board side (MCUXpresso or IAR) only on **board A**.

14.4.1.2. Modifications on the host side

The only thing that must be added on the host side is on the reception callback. These modifications depend on the opCode used by the developer and the functionality that wants to be added.

14.4.2. Socket

To send socket messages from the end device to the leader (host_controlled_device), enter the below commands on the serial terminal (**board B**):

```
$ help socket
$ socket open udp fd00:db8::ff:fe00:0 1233
$ socket send 0 ABCD
```

Developer can see the available socket commands and its parameters.

Figure 25. Socket commands.

Open a socket port using UDP protocol and the leader IP address. Send 'ABCD' using the socket id obtained in the socket open response.

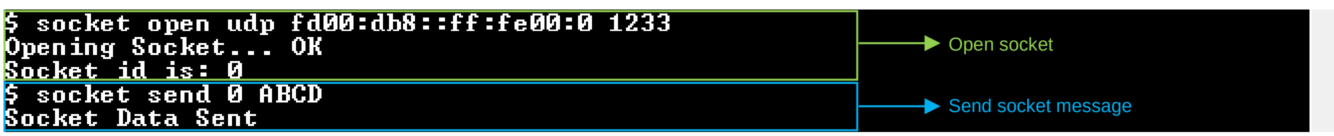


Figure 26. Socket commands 2.

See the information sent by the end device printed in the host terminal.

Figure 27. Socket information from host terminal.

15. Thread TUN/TAP HSDK

TUN and TAP interfaces allows the programmer to send and receive network traffic (MAC or IP level) on his applications. Both are software interfaces, they have no physical hardware components. The Kernel manages the created interfaces and decides when to send data through those TUN or TAP interfaces or through any physical hardware component. The developer can only use one interface on his application, you can't use both at the same time. Depending on the network requirements, you can choose a TUN or TAP interface.

The Kinetis Thread Stack implements a serial Tunnel media Interface which can be used to exchange IPv6 packets encapsulated in THCI commands with a host system.

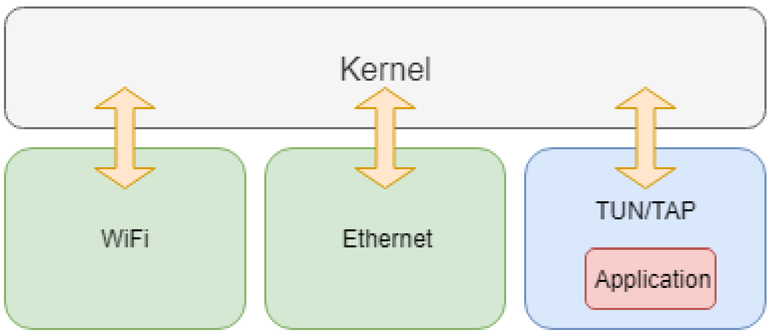


Figure 28. TUN/TAP software interface

TUN Interface

TUN interface works at layer three (Network), which can operate with IP packets. This interface provides routing between different nodes. Since TUN runs at layer three, it can only accept IP packets. This type of interface is used when programmers want to enable routing.

TAP Interface

TAP interface works at layer two (Data Link Layer), which handle MAC frames. This interface provides node-to-node data transfer or point-to-point connection. Since TAP runs at layer two, the device can send data to any layer three protocol added on the device. This type of interface is used when programmers want to create a network bridge and avoid routing between the host_controlled_device and the HSDK host.

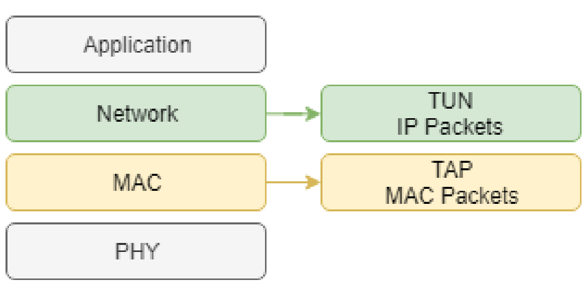


Figure 29. TUN/TAP layers

NOTE

Previous versions of the Thread stack permitted to disable ND completely on the serial tunnel media interface, with communication being network layer only (layer-3 TUN mode). The current version of the Thread stack disables this mode to promote link layer communication (layer-2 TAP mode) between the Thread border router and external networks. If needed, one may modify the firmware to use TUN mode instead of TAP mode by stripping out the Ethernet header from network frames handled in functions IP_SerialTunRecv and IP_SerialTunSend6.

A bridge is used to join two independent networks together and form a larger network. Having this in mind, you could create an interconnection between those networks and exchange data.

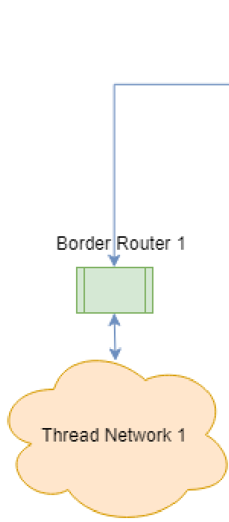


Figure 30. Bridge between 2 networks.

Topology

Two components are required to provide connectivity to the host: the TUN/TAP kernel module, which allows the OS to create virtual interfaces and a program that knows how to encapsulate/decapsulate IP packets from/to FSCI/THCI.

The image below describes the communication stages for each message sent by the host (PC) and the Thread device (Border Router).

Figure 31. Direction from host (PC) to serial Thread device.

Figure 32. Direction from Thread device to host (PC)

15.3.1. Linux Host

In this scenario, the Linux host is a bridge between the OpenWrt Router and the Kinetis Border Router. The PC receive/send IPv6 packets encapsulated by THCI serial commands. To do this, the PC have to enable a TUN/TAP interface. Refer to Figure 33.

15.3.2. Border Router

The Border Router will act as an interface to forward IPv6 packets between the Thread network and the Linux host. To do this, the Kinetis device will have to support a TUN/TAP interface. Refer to Chapter *15.4 General setup* to enable it. Refer to Figure 33.

15.3.3. OpenWrt Router

A router with DHCPv6-PD support is required. This router enables you to connect your Thread network to any other system that uses IPv6 packets. The Prefix Delegation feature delegates the IP address assignation. Refer to Figure 33.

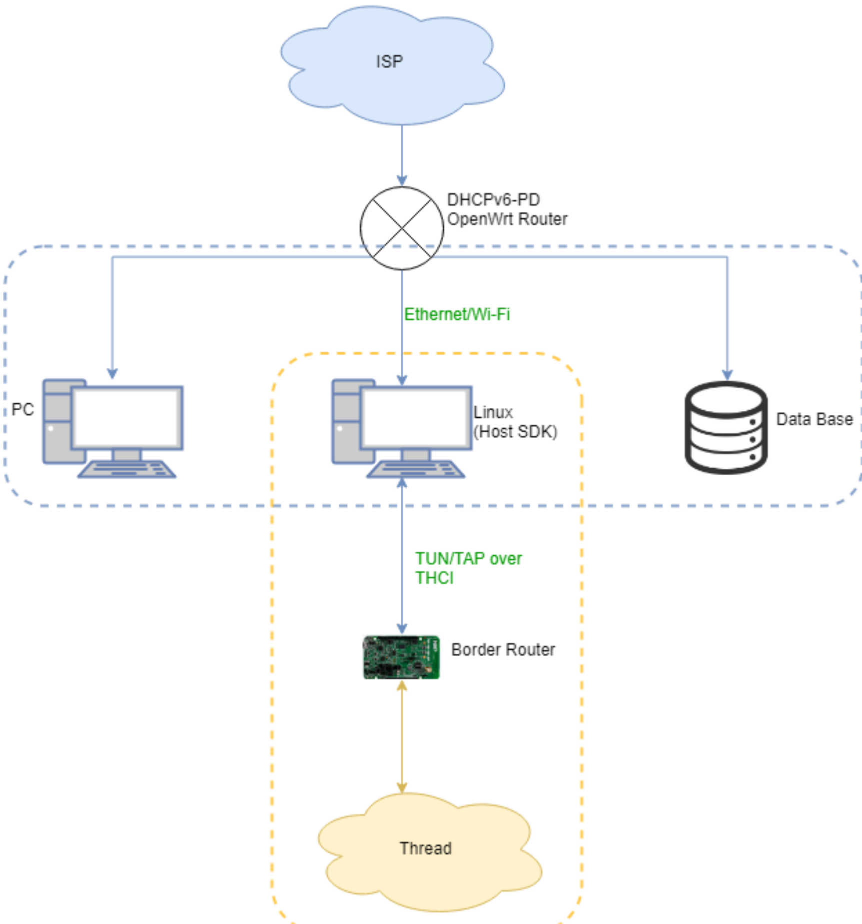


Figure 33. Overall network topology using TUN/TAP and external host

General setup

- Linux Host.
- OpenWrt AP/Router with DHCPv6-PD support.

15.4.1. Embedded setup

First, verify that you have the latest SDK version (this document uses 2.2 version). Refer to Chapter *4.1.1. SDK Builder* of this document to see the download link.

- 1 FRDM-KW41Z as Border Router (Host Controlled Device)
- 1 FRDM-KW41Z as joiner device.

15.4.1.1. Configuration

Border Router (Host Controlled Device)

Use the Host Controlled Device demo provided in the FRDM-KW41 SDK. The project can be found in the following directory: *' <SDK path>\boards\frdmkw41z\wireless_examples\thread\ '.*

The following changes are needed to enable the **TAP** interface:

1. Enable THR_SERIAL_TUN_IF /source/config.h

Joiner device.

Use the Router Eligible End Device demo provided in the FRDM-KW41 SDK. The project can be found at: *'<SDK path>\boards\frdmkw41z\wireless_examples\thread\'*. No changes are required on this demo.

To create a network with TUN/TAP devices, follow the instructions in the next tutorial: https://community.nxp.com/docs/DOC-334294. You will find step by step instructions on how to enable this functionality.

To have a more detailed description of the Thread Tunnel Interface please refer to the document: Kinetis FSCI Host Application Programming Interface on 'Chapter 6. Thread Integration with Linux OS Host on Serial (UART) Tunnel Interface' and 'Chapter 7 Applications Using the TUN Interface'. The document can be found at: *'<SDK path>\docs\wireless\Thread'*.

16. ZigBee HSDK Demo

There are two demo examples based on ZigBee protocol:

1. Zigbee Black Box: Can be used as coordinator or as router. There are two different files, one for each implementation. The files are named 'Zigbee_BBC_HSDK.c' and 'Zigbee_BBR_HSDK.c' respectively. Both files use 'Zigbee_BlackBox_HSDK.h' as header.
2. ZigBee Control Bridge: Provides a means of controlling ZigBee devices within a ZigBee network. The device would typically act as ZigBee IoT Gateway.

Both ZigBee HSDK demos send several packets with 1 second interval between each one. The host device is programmed to send the following commands through the serial interface and the client device will execute them:

- Start ZigBee network.
- Install code to allow Unique Link Key.
- Permit join.

When a ZigBee node (**OnOffLight Router**) joins the network:

- Send a Discover Command.
- Send a Simple Descriptor request.
- Send a Read attributes for OnOff status.
- Send an OnOff Set State command.
- Find & Bind – Initiator.
- Receive attribute report periodically from the OnOffLight Router.

Black Box

16.1.1. Add your own OpCode – embedded side

All Zigbee FSCI command codes supported by the device are found in 'SerialLink.h' file as an enumeration named 'teSL_MsgType'. Follow the steps below to add a new opcode implementation.

1. Select a name and a number that are not already listed in 'teSL_MsgType' enumeration and add it. It is recommended to follow the naming standard.
2. Go to 'app_Znc_cmds.c' file and look for the following function: 'APP_vProcessIncomingSerialCommands(uint8 u8RxByte)'.
3. Add a new 'case' statement with your new command code selected in the first step.
4. Add your code implementation inside the case statement.

16.1.2. FSCI Black Box as Coordinator

16.1.2.1. Prerequisites

To make this example work correctly, the boards have an identification letter as is shown in the below figure:

Figure 34. Board setup.

Load ZHCD firmware to both boards as follows:

- **Board A**:

  Load '*fsci_black_box*' example. The project can be found:

  '*<SDK path>\boards\frdmkw41z \wireless_examples\zigbee_3_0*'.

  Follow the steps bellow to enable the Install Code functionality. These changes are needed to run this demo correctly. (If you are creating a new project and don't want to use Install Code, there is no need to do it):

  1. In bdb_options.h change #define BDB_JOIN_USES_INSTALL_CODE_KEY from FALSE to TRUE.
  2. Add a new opcode to support the install code FSCI command on the embedded side. To

do this, refer to chapter 13.2. Add your own Opcode – embedded side, and add the following code:

As command code in 'teSL_MsgType' enumeration (SerialLink.h file):

```
E_SL_MSG_INSTALL_CODE = 0x0015
```

As command code implementation (app_Znc_cmds.c):

```
case (E_SL_MSG_INSTALL_CODE):
{
uint64_t u64Addr;
uint8_t i;
uint8_t offset = 0;
uint8_t Key[16];
ZPS_tsAplAib * psAplAib;

u64Addr = ZNC_RTN_U64_OFFSET(au8LinkRxBuffer,offset,offset);
   for(i = 0; i < 16 ; i++)
{
Key[i] = au8LinkRxBuffer[offset + i];
}
/* Install the new code */
ZPS_eAplZdoAddReplaceInstallCodes( u64Addr, Key, 16,
   ZPS_APS_UNIQUE_LINK_KEY);

psAplAib = ZPS_psAplAibGetAib();
for(i=0; i<16;i++)
{
Key[i] = psAplAib->psAplDeviceKeyPairTable-
   >psAplApsKeyDescriptorEntry[0].au8LinkKey[i];
}
vSL_WriteMessage(E_SL_MSG_INSTALL_CODE, 16, Key);
}
break;
```

- **Board B**:

  Load '*router*' demo. The project can be found at:

  '<SDK path>*\boards\frdmkw41z \wireless_examples\ zigbee_3_0\'.*

  Follow the steps bellow to enable the Install Code functionality. These changes are required to run this demo correctly. (If you are creating a new project and don't want to use Install Code, there is no need to do it):

  1. In app_router_node.c in the APP_vInitialiseRouter() function add the following before

the call to ZPS_eAplAfInit():

```
/* Enable use of install codes */
ZPS_tsAplAib *psAib = ZPS_psAplAibGetAib();
psAib->bUseInstallCode = BDB_JOIN_USES_INSTALL_CODE_KEY;
```

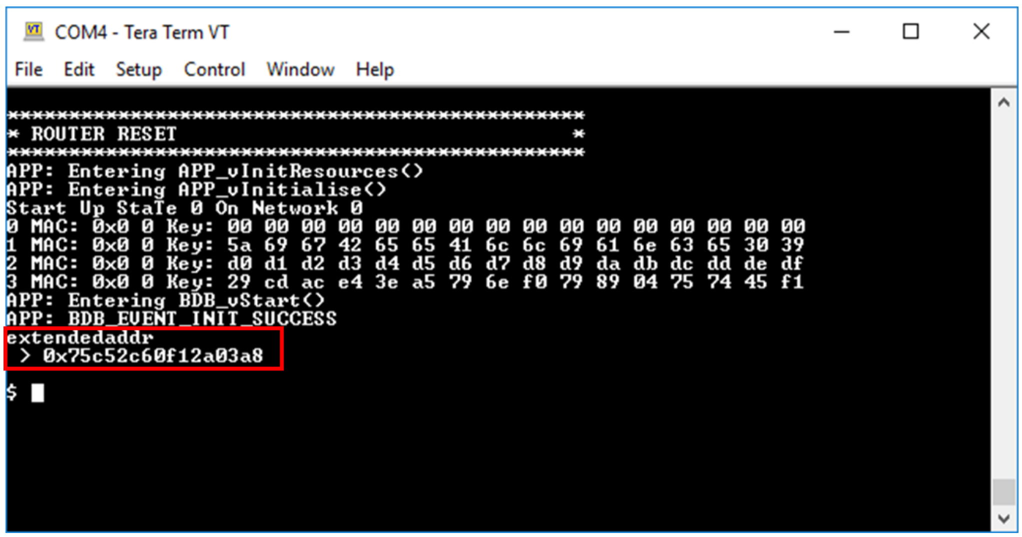
2. In bdb_options.h change #define BDB_JOIN_USES_INSTALL_CODE_KEY from FALSE to TRUE.

16.1.2.2. Run demo application

Before executing the coordinator ZigBee host application, you need to edit 'install_code_buf' variable in 'Zigbee_BBC_HSDK.c' file. Replace all bytes with the corresponding **MAC address of your own joiner device** repeated. This is required to allow your joiner into the network.

16.1.2.3. Find MAC Address

To find the **MAC address of your own joiner device** run router demo and open a serial terminal as done in previous steps, then type "extendedaddr", the string displayed by the console is the MAC address.



For example, if your joiner device MAC address is: 0x75C52C60F12A03A8 you will do the following:

```
static uint8_t install_code_buf[] = {0x75, 0xC5, 0x2C, 0x60, 0xF1, 0x2A, 0x03, 0xA8, 0x75, 0xC5, 0x2C, 0x60, 0xF1, 0x2A, 0x03, 0xA8,0x75, 0xC5, 0x2C, 0x60, 0xF1, 0x2A, 0x03, 0xA8};
```

To run any demo application (or your own application) follow the next steps:

- Compile demo. See chapter 10 Compile an Application.
- Connect the **Board A** to the computer with **Linux OS**.
- Open a terminal at the next directory: *'<SDK path>\tools\wireless\host_sdk\hsdk\ demo\bin'*.

- Execute ‘GetKinetisDevices’ program to obtain the port where the Kinetis device is connected.
- Execute ‘Zigbee_BBC_HSDK’ program.

```
$sudo ./GetKinetisDevices
 NXP Kinetis-W device on /dev/ttyACM0

$sudo ./Zigbee_BBC_HSDK /dev/ttyACM0 15 115200
```

Parameters: Port Channel Baud rate

16.1.2.4. Demo description

Create Zigbee Network

The program starts doing a factory reset, setting the channel and extended PANID network parameters before starting the network.

You can enable or disable this set of the extended PAN ID parameter by setting USE_SET_XPANID in ‘Zigbee_BlackBox_HSDK.h’ file.

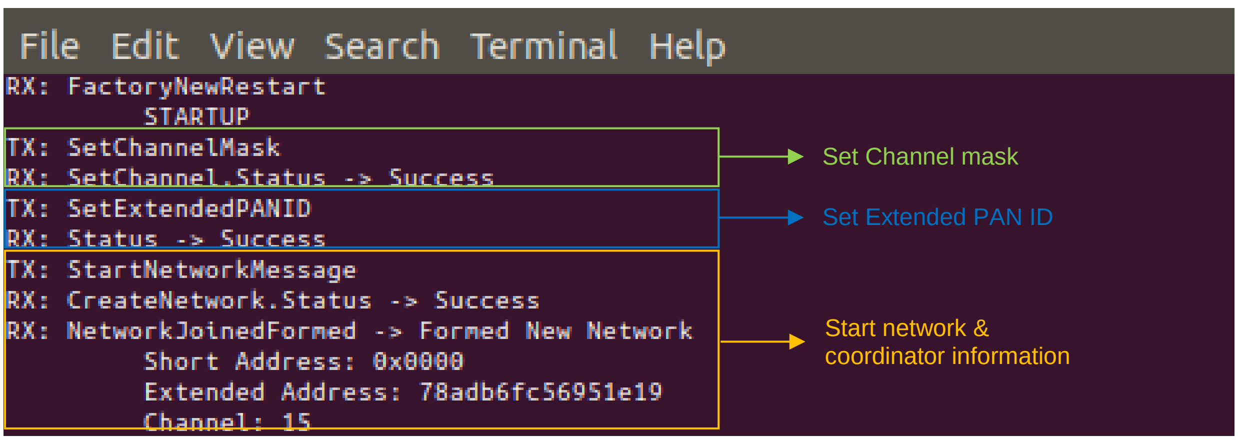


Figure 35. Creation of Zigbee network

Install Code

Once the ZigBee network has been created, the coordinator installs a Unique Link Key using the MAC address of the joiner device. This command allows to use a unique Link Key in the joining process. On the Linux terminal you must see the new Link Key for the joiner node.

You can enable or disable this command by setting USE_INSTALL_CODE in ‘Zigbee_BlackBox_HSDK.h’ file.

Figure 36. Install code

Join New Node

The coordinator sends a permit join message to allow other nodes to join and waits until a device sends a join request.

Connect the **Board B** and open a serial terminal with Tera Term or PuTTY. Configure the serial terminal with the following parameters: Baud rate: 115200, 8-bit data, 1 stop bit, No flow control, No parity.

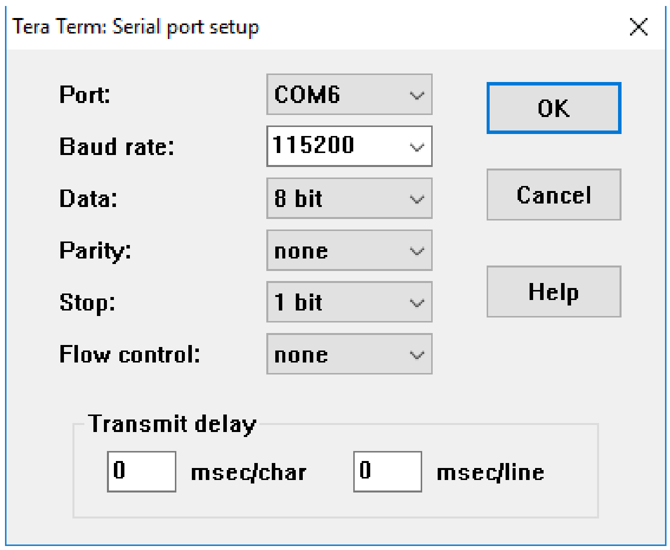


Figure 37. Serial terminal configuration

Press the reset button on the **Board B** and enter the next commands on the serial terminal:

```
$ channel 15
$ join
```

```
************************************************
* ROUTER RESET                                 *
************************************************
APP: Entering APP_vInitResources()
APP: Entering APP_vInitialise()
0 MAC: 0x0 0 Key: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1 MAC: 0x0 0 Key: 5a 69 67 42 65 65 41 6c 6c 69 61 6e 63 65 30 39
2 MAC: 0x0 0 Key: d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
3 MAC: 0x0 0 Key: cf 7c b5 12 04 a9 82 04 91 ce 6d aa 70 5d 46 7e
APP: Entering BDB_vStart()
channel 15
 > 15

$ join
BDB: Disc on Ch 15 from 0x00008000
Nwk Join 00
```

Figure 38. Board B, channel and join commands.

You can see a permit join request, a device announce and a router discovery on the host serial terminal. This messages indicates a succesfully joining request.

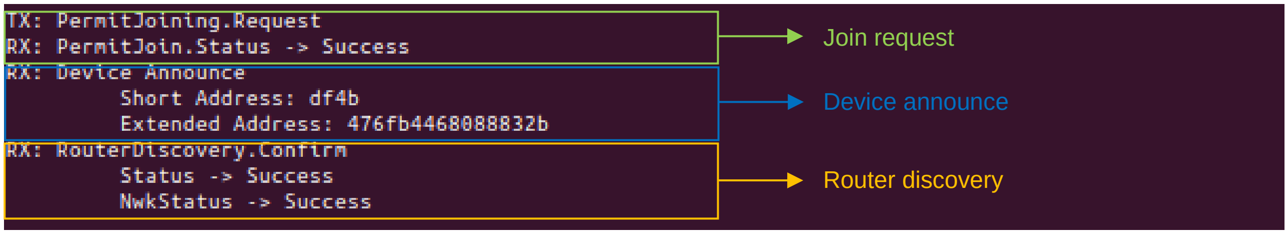


Figure 39. Board A, join request.

In the image below, you can see the asignation of the Unique Link Key generated in the Apendix A. Install Code. The Link Key generated by the coordinator is used for the joining process and then is replaced by the Trust Center Link Key sent by the coordinator.

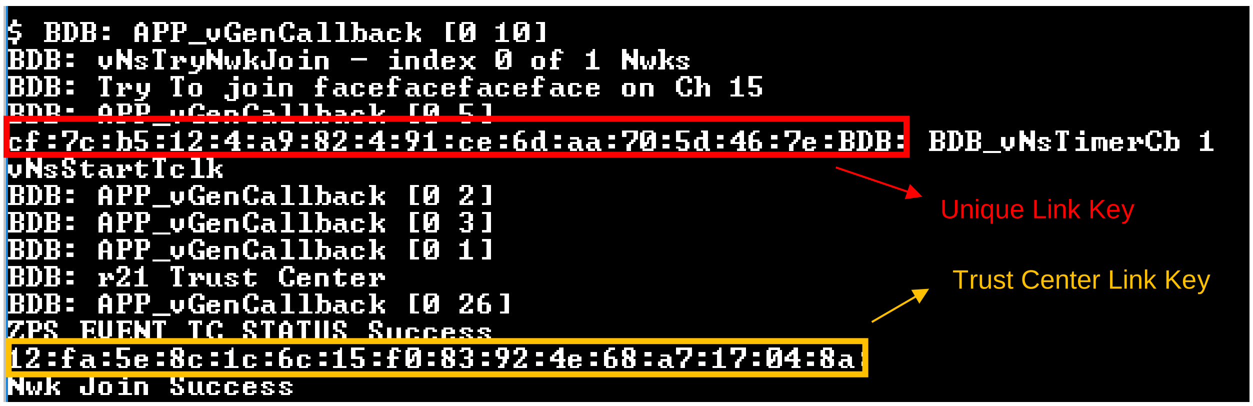


Figure 40. Board B, link keys

Match Descriptor

In this command, the developer enter a profile ID and a list of clusters to ask a specific node if they have some clusters in common. The node responds with the endpoint that supports any cluster in the list.

You can enable or disable this command by setting USE_MATCH_DESCRIPTOR in 'Zigbee_BlackBox_HSDK.h' file.

Figure 41. Board A, match descriptor.

Simple Descriptor

This command is another way to know which clusters are supported by a specific node in the ZigBee network. In this command, the developer must specify the short address of the required node and the endpoint where he wants to look up.

The node responds with the profile ID and a list of supported clusters in the selected endpoint.

You can enable or disable this command by setting USE_SIMPLE_DESCRIPTOR in 'Zigbee_BlackBox_HSDK.h' file.

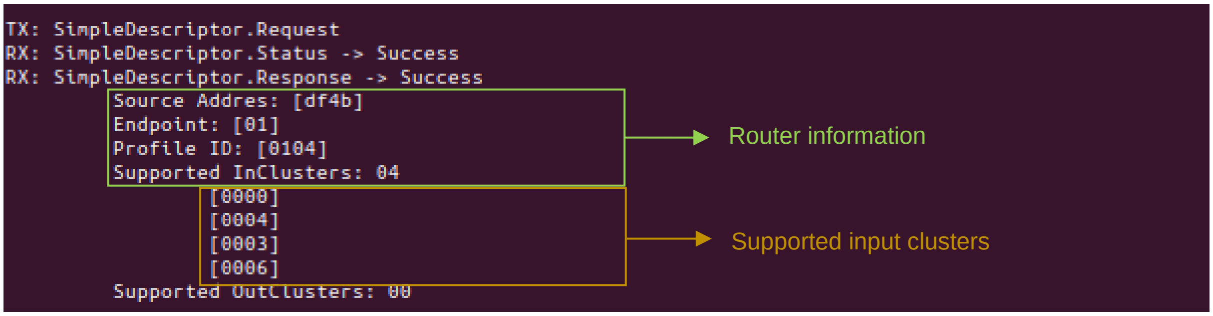


Figure 42. Board A, simple descriptor.

Read Attributes

Send this command to read one or several attributes on a specific cluster. Developer enter a list of attributes to be read.

You can enable or disable this command by setting USE_READ_ATTRIBUTES in 'Zigbee_BlackBox_HSDK.h' file.

To read multiple attributes from a cluster you can enable USE_DYNAMIC_ATTRIBUTES and modify the macro NUMBER_OF_ATTRIBUTES in 'Zigbee_BlackBox_HSDK.h'.

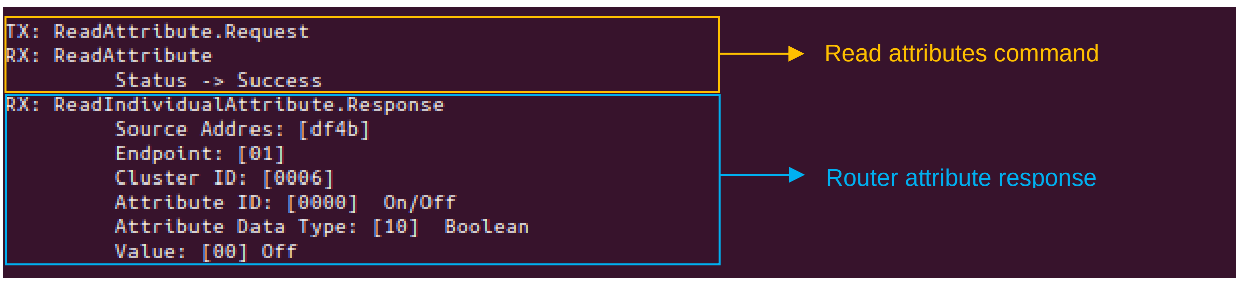


Figure 43. Board A, Read attributes.

Find & Bind

The coordinator sends a Find command as initiator on a specific endpoint and cluster. Both extended address (coordinator and requested device) are required to execute this command. Once this command is executed, the router reports its attributes every minute (time by default) automatically.

You can enable or disable this command by setting USE_FIND_AND_BIND in 'Zigbee_BlackBox_HSDK.h' file.

Figure 44. Board A, find and bind.

Cluster Command

In the example demo is used an on/off command which is supported by the OnOff Cluster (Cluster ID 0006). Every time an attribute is modified, the device reports the attribute automatically only if the binding was created using Find & Bind procedure.

You can enable or disable this command by setting USE_SET_ATTRIBUTE in ‘Zigbee_BlackBox_HSDK.h’ file.

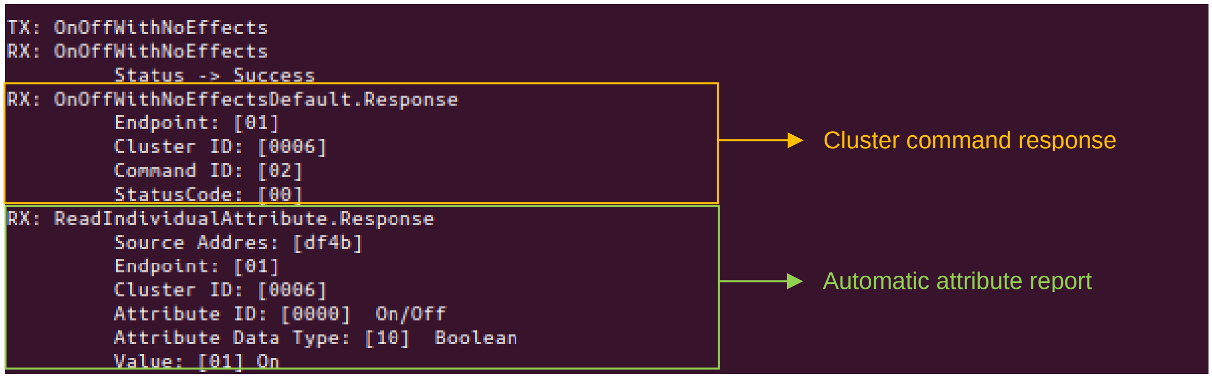


Figure 45. Board A, cluster command.

16.1.3. FSCI Black Box as Router

16.1.3.1. Prerequisites

To make this example work correctly, the boards have an identification letter as is shown in the below figure:

Figure 46. Board setup.

Load THCD firmware to both boards as follows:

- **Board A**:
  Load ‘*fsci_black_box*’ example. The project can be found:
  ’ <SDK path>\boards\frdmkw41z \wireless_examples\zigbee_3_0\’.
  To enable the Install Code functionality, follow the steps bellow. These changes are required to run this demo correctly. (If you are creating a new project and don’t want to use Install Code, there is no need to do it):
  1. In bdb_options.h change #define BDB_JOIN_USES_INSTALL_CODE_KEY from FALSE to TRUE.
  2. In app_zps_cfg.h change #define

ZPS_INIT_APL_DEFAULT_GLOBAL_APS_LINK_KEY from FALSE to TRUE.

3. In app_start.c in the vInitialiseApp() function add the following after the call to APP_vSetMacAddr();

```
/* Enable use of install codes */
   ZPS_tsAplAib *psAib = ZPS_psAplAibGetAib();
   psAib->bUseInstallCode = BDB_JOIN_USES_INSTALL_CODE_KEY;
```

- **Board B**:
  Load '*coordinator*' demo. The project can be found:
  ' <SDK path>\boards\frdmkw41z \wireless_examples\ zigbee_3_0\'.
  Follow the steps bellow to enable the Install Code functionality. These changes are required to run this demo correctly. (If you are creating a new project and don't want to use Install Code, there is no need to do it):
  1. In bdb_options.h change #define BDB_JOIN_USES_INSTALL_CODE_KEY from FALSE to TRUE.

16.1.3.2. Demo description

Create ZigBee Network

Now, the **Board B** starts the ZigBee network and the **Board A** joins to the network by FSCI commands.

Connect the **Board B** and open a serial terminal with Tera Term or PuTTY. Configure the serial terminal with the following parameters: Baud rate: 115200, 8-bit data, 1 stop bit, No flow control, No parity.

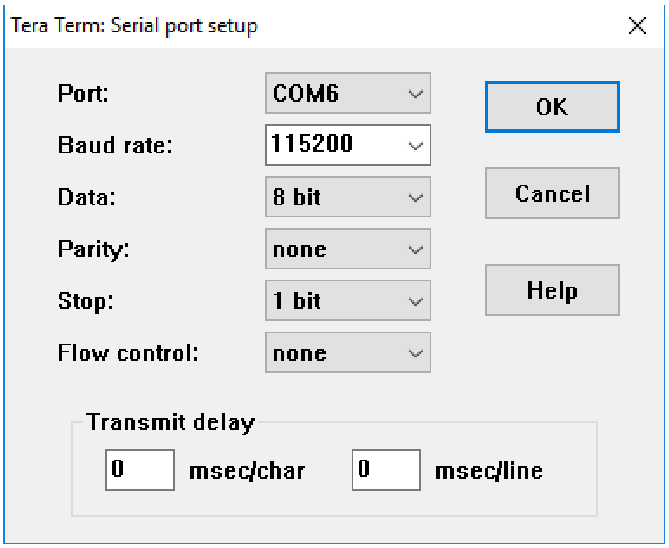


Figure 47. Serial terminal configuration.

Press the reset button on the **Board B** and enter the commands below on the serial terminal:

```
$channel 15
$form
```

See the information as in below image on your terminal:

```
*********************************************
* COORDINATOR RESET                         *
*********************************************
APP: Entering APP_vInitResources()
APP: Entering APP_vInitialise()
Recovered Application State 0 On Network 0
APP: Entering BDB_vStart()
channel 15
 > 15

$ form
BDB: Forming Centralized Nwk
Nwk Formation 00

$ BDB: APP_vGenCallback [0 4]
APP-BDB: NwkFormation Success
APP-ZDO: Network started Channel = 15
```

Figure 48. Board B, coordinator commands.

Install Code

Enter the below command on the coordinator serial terminal:

```
$ code <addr> <install_code>
```

Replace <addr> with the MAC address of the joiner node.

Replace <install_code> with the MAC address of the joiner node repeated once.

In this case, the MAC address of the joiner node is: 78ADB6FC56951E19.

To find the MAC address of the joiner device refer to step **16.1.2.3**

```
code 78ADB6FC56951E19 78ADB6FC56951E1978ADB6FC56951E19
Key Added for 78adb6fc56951e19, Status 00
0 MAC: 0x78adb6fc 56951e19 Key: f3 f5 52 ee f7 f2 f0 bb 03 a5 dd ad 61 8e eb 6b
1 MAC: 0x0 0 Key: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
2 MAC: 0x0 0 Key: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
3 MAC: 0x0 0 Key: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
4 MAC: 0x0 0 Key: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
5 MAC: 0x0 0 Key: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
6 MAC: 0x0 0 Key: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
7 MAC: 0x0 0 Key: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8 MAC: 0x0 0 Key: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
9 MAC: 0x0 0 Key: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10 MAC: 0x0 0 Key: 5a 69 67 42 65 65 41 6c 6c 69 61 6e 63 65 30 39
11 MAC: 0x0 0 Key: d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
12 MAC: 0x0 0 Key: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Install code command

MAC address (extended address) of the joiner node

Unique Link Key created for the router node

Figure 49. Board B, install code command.

See the APS table on the coordinator serial terminal showing the Unique Link Key assigned to the joiner node with the specific MAC address.

Join FSCI Router Node

Enter the next command on the coordinator serial terminal:

```
$ steer
```

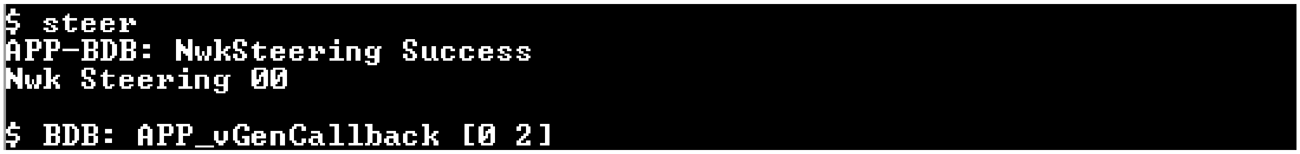


Figure 50. Board B, steer command.

This command allows other nodes to join to the ZigBee network created by the coordinator.

Then, join the FSCI router:

- Compile FSCI router demo. See chapter 10 Compile an Application.
- Connect the **Board A** to the computer with **Linux OS**.
- Open a terminal in the next directory: *'<SDK path>\tools\wireless\host_sdk\hsdk\ demo\bin'*.
- Execute 'GetKinetisDevices' program to obtain the port where the Kinetis device is connected.
- Execute 'Zigbee_BBR_HSDK' program.

```
$sudo ./GetKinetisDevices
 NXP Kinetis-W device on /dev/ttyACM0

$sudo ./Zigbee_BBR_HSDK /dev/ttyACM0 15 115200
```

Parameters: Port Channel Baud rate

Figure 51. Board A, Zigbee router view.

When a FSCI device executes a reset, sends a list of clusters and attributes to the host computer so the user can know which ones are supported by the connected device.

You can enable or disable this command by setting SEE_ATTRIBUTE_LIST in 'Zigbee_BlackBox_HSDK.h' file. By default, is set to 0.

```
BDB: APP_vGenCallback [0 9]
APP-ZDO: New Node b7f2 Has Joined
BDB: APP_vGenCallback [0 1]
APP-ZDO: Data Indication Status 00 from b7f2 Src Ep 0 Dst Ep 0 Profile 0000 Cluste
r 0013
BDB: APP_vGenCallback [0 14]
APP-ZDO: Discovery Confirm
BDB: APP_vGenCallback [0 2]
BDB: APP_vGenCallback [0 3]
BDB: APP_vGenCallback [0 26]
APP-ZDO: Trust Center Status 00
```

Figure 52. Board B, callback from router device.

Control Bridge

For the moment, the only way to manipulate Control Bridge demo is through ZGUI from the Test Tool Application. For more information about its functionality, refer to 'AN12063-MKW41Z-AN-ZigBee-3-0-ControlBridge' document provided in the following directory:

' <SDK path>\docs\wireless\ Zigbee\Application Notes\'.

17. BLE HSDK Demo

This demo allows the developer to experiment and become familiar with the Framework Serial Communication Interface (FSCI) and the BLE Host Stack to implement a Heart Rate application.

BLE Host Stack layers

The BLE Host Stack layers that offers access using FSCI are GATT, GATTDB, L2CAP, and GAP.

Each layer provides primitives that an upper layer (profile/application) uses to access services of that layer.

Here a brief description of that layers:

GATT: provides methods in which the services can be discovered and can be used, allows the access and retrieval of information between client and server.

GATTDB: provides methods in which services can be added, modified or removed.

L2CAP: allows higher level protocols and applications to transmit and receive upper layer data packets.

GAP layer provides:

- Discoverability modes and procedures.
- Connection modes and procedures.
- Security/Bonding modes and procedures.

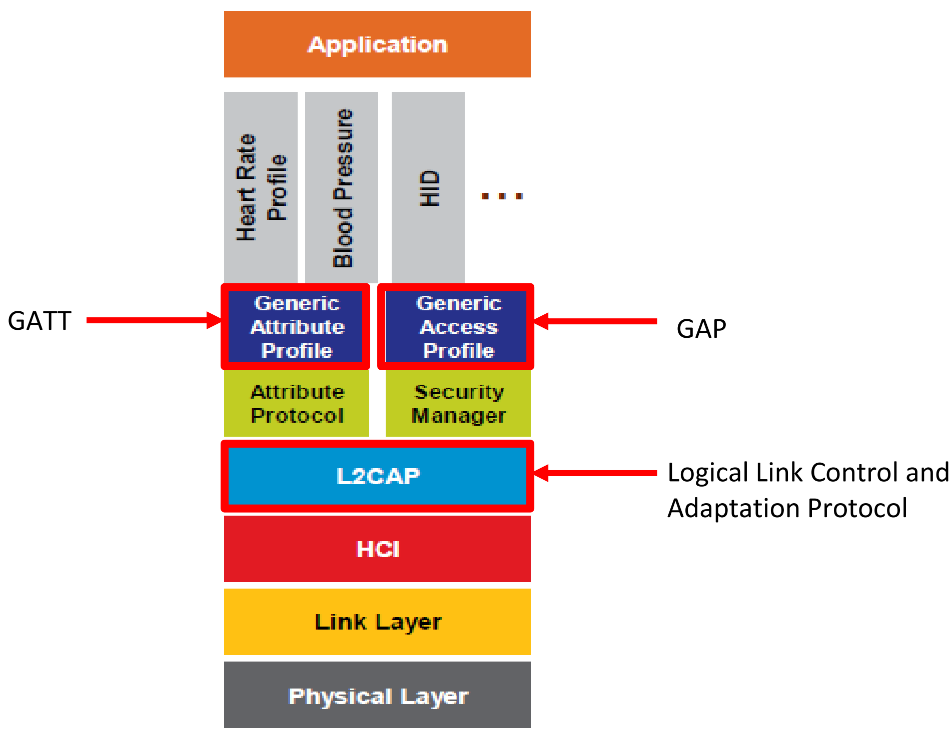


Figure 53. BLE system architecture.

For more detailed information about FSCI messages to the layers previously mentioned, refer to the document: "BLE Host Stack FSCI Reference Manual". The document can be found at: *'<SDK path>\docs\wireless\Bluetooth'.*

GATT profile hierarchy

Here is a general GATT profile hierarchy scheme that this demo follows, most of the FSCI Frames sent to the BLE Host Stack Black Box are to create this scheme and set up the Heart Rate Sensor profile.

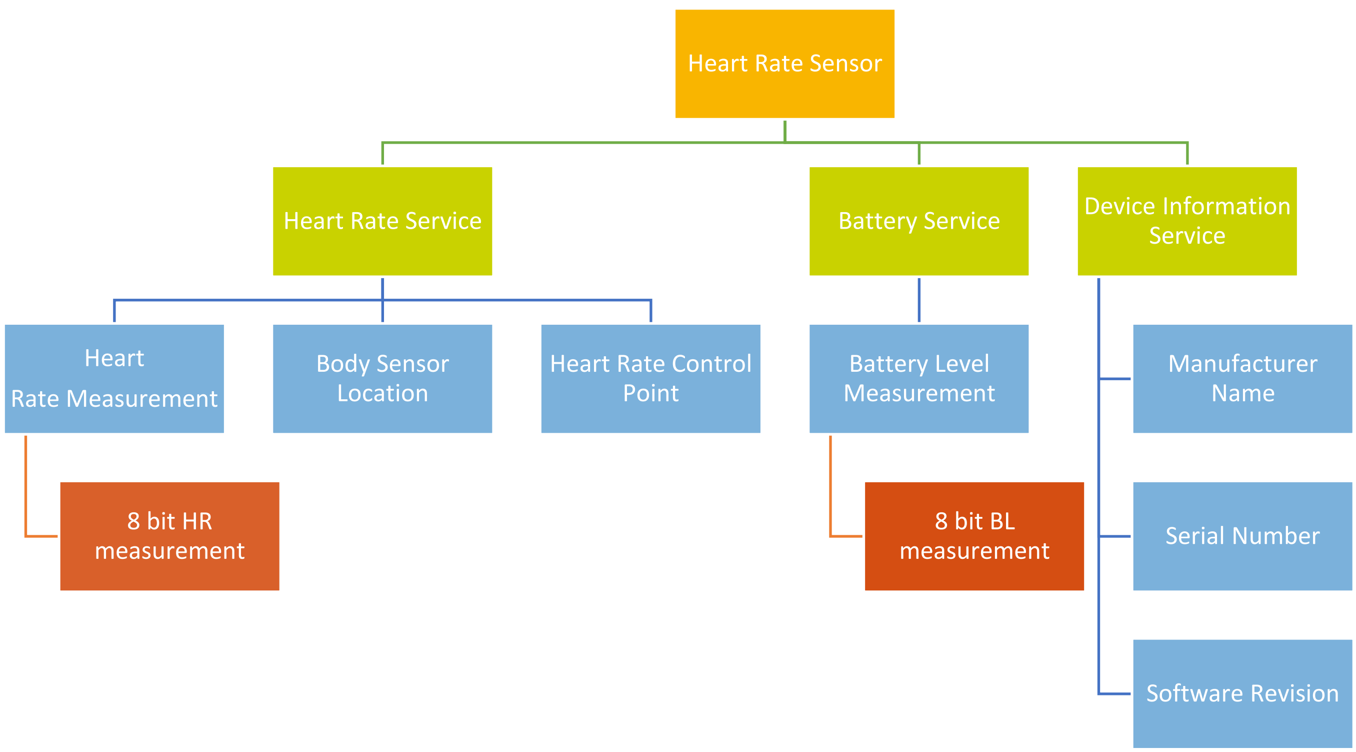


Figure 54. GATT profile hierarchy scheme.

Hierarchy level description:

1. ***Profile:*** defines the main behavior of the device, in this case a Heart Rate Sensor.
2. ***Service:*** is a collection of data and behaviors for a feature.
3. ***Characteristic:*** is the value used in the service.
4. ***Descriptor:*** describes how to read the characteristic value.

For more detailed information about BLE system architecture, refer to the Bluetooth SIG available on the link, www.bluetooth.com/specifications/bluetooth-core-specification/

Prerequisites

To make this example work correctly, use a FRDM-KW41Z board and a smartphone with the "NXP IoT Toolbox" app:

Figure 55. Board setup.

Load BLE FSCI Black Box firmware to the board as follows:

- **Board A**:

  Load '*ble_fsci_black_box*' example. The project can be found:

  ' <SDK path>\boards\frdmkw41z \wireless_examples\bluetooth\ ble_fsci_black_box'.
- **NXP IoT Toolbox**

  The IoT Toolbox is a mobile application developed by NXP Semiconductors. It is designed for the Android™ and iOS™ handheld devices. The mobile application is free in App Store® and Google Play™.

Run demo application

To execute this demo, follow the steps below:

- Compile demo. See Chapter 10. Compile an Application.
- Open a terminal in the next directory: *'<SDK path>\tools\wireless\host_sdk\hsdk\ demo\bin'*.
- Execute 'GetKinetisDevices' program to obtain the port where the Kinetis device is connected.
- Execute 'HRS_BLE_HSDK' application.

```
$sudo ./GetKinetisDevices
 NXP Kinetis-W device on /dev/ttyACM0

$sudo ./HRS_BLE_HSDK /dev/ttyACM0 115200
```

Parameters: Port Baud rate

The HRS_BLE_SDK starts, and you should see the below information on your console.

First, a general reset request is sent, followed by the layer's confirmation response. Then, the request to add a generic GATT and GAP services, these are used to create the first level of the hierarchy scheme viewed in ‘Section 15.2 - GATT profile hierarchy’.

```
javier@javier-VirtualBox:~/Desktop/KW41_SDK/tools/wireless/host_sdk/hsdk/demo/bin$
 sudo ./HRS_BLE_HSDK /dev/ttyACM1 115200

TX: FSCI-CPUReset.Request                        General reset request
RX: L2CAP.Confirm->gBleSuccess
RX: L2CAP.Confirm->gBleSuccess
RX: L2CAP.Confirm->gBleSuccess
RX: GATT.Confirm->gBleSuccess
RX: GATTDB.Confirm->gBleSuccess
RX: L2CAP.Confirm->gBleSuccess
RX: GAP-GenericEventInitializationComplete.Indication
RX: GAP.Confirm->gBleSuccess

        ---- Add GATT Service ----

TX: GATTDBDynamic-AddPrimaryServiceDeclaration.Request      Add Generic Attribute Profile
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddPrimaryServiceDeclaration.Indication
        Service Handle: 0001

TX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Request -> Service Changed
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Indication
        Characteristic Handle: 0002

TX: GATTDBDynamic-AddCccd.Request
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCccd.Indication
        Cccd Handle: 0004

        ---- Add GAP Service ----

TX: GATTDBDynamic-AddPrimaryServiceDeclaration.Request      Add Generic Access Profile
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddPrimaryServiceDeclaration.Indication
        Service Handle: 0005

TX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Request -> Device Name
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Indication
        Characteristic Handle: 0006
```

Figure 56. Adding generic attributes and access profiles.

In this section, the Heart Rate service is added as a service, then, the characteristics of that service as heart rate measurement, body sensor location and heart rate control point are added.

```
TX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Request -> Appeareance
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Indication
        Characteristic Handle: 0008

TX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Request -> Ppcp
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Indication
        Characteristic Handle: 000a
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCccd.Indication
        Cccd Handle: 000c

        ---- Add Heart Rate Service ----

TX: GATTDBDynamic-AddPrimaryServiceDeclaration.Request
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddPrimaryServiceDeclaration.Indication
        Service Handle: 000d

TX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Request -> HR Measurement
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Indication
        Characteristic Handle: 000e

TX: GATTDBDynamic-AddCccd.Request
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCccd.Indication
        Cccd Handle: 0010

TX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Request -> Body Sensor Loca
tion
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Indication
        Characteristic Handle: 0011

TX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Request -> Control Point
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Indication
        Characteristic Handle: 0013
```

Add Heart service
UUID = 0x180D

Add Heart Rate Characteristics
- HR Measurement
- Body Sensor Location
- HR Control Point

Figure 57. Adding Heart Rate service and characteristics.

Then, the program adds Battery and Device Information services, each service followed by their necessary characteristics and in some cases descriptors of that characteristics.

```
        ---- Add Battery Service ----

TX: GATTDBDynamic-AddPrimaryServiceDeclaration.Request
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddPrimaryServiceDeclaration.Indication
        Service Handle: 0015

TX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Request -> Battery Level
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Indication
        Characteristic Handle: 0016

TX: GATTDBDynamic-AddCharacteristicDescription.Request -> Char Format
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDescriptor.Indication
        Descriptor Handle: 0018

TX: GATTDBDynamic-AddCccd.Request
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCccd.Indication
        Cccd Handle: 0019

        ---- Add Device Information Service ----

TX: GATTDBDynamic-AddPrimaryServiceDeclaration.Request
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddPrimaryServiceDeclaration.Indication
        Service Handle: 001a

TX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Request -> Manufacturer Nam
e
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Indication
        Characteristic Handle: 001b

TX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Request -> Model Number
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Indication
        Characteristic Handle: 001d

TX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Request -> Serial Number
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Indication
        Characteristic Handle: 001f
```

Add Battery service
UUID = 0x180F

Add Battery Level characteristic
UUID = 0x2A19

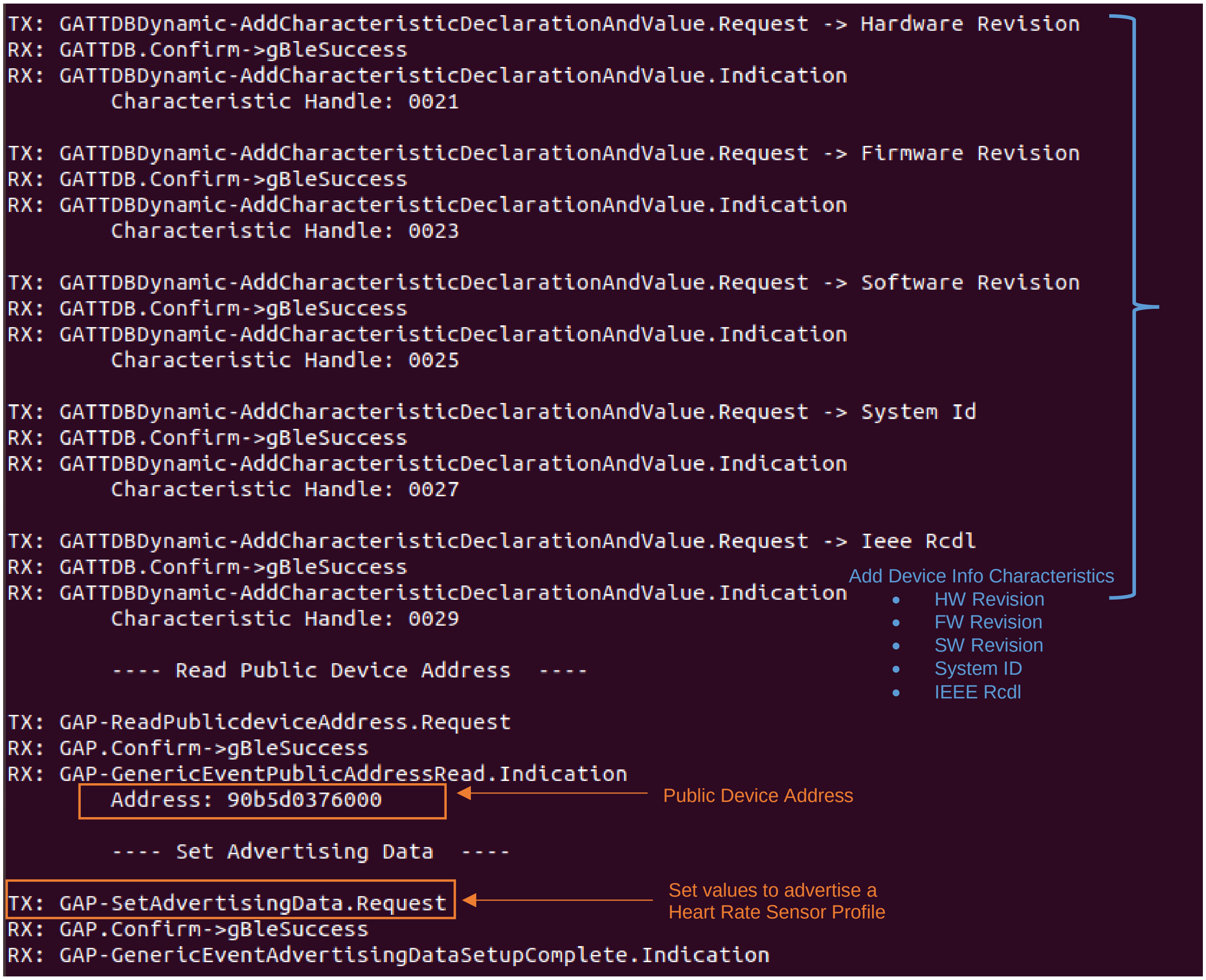
Add Descriptor for Battery Level characteristic

Add Device Information service
UUID = 0x180A

Add Device Info Characteristics
- Manufacturer name
- Model number
- Serial number

Figure 58.Adding battery and device information services, characteristics and descriptors.

Here the program is still adding device information characteristics, then the host request the device address and sets the advertising data. At this point, the device is recognized as a Heart Rate sensor profile instead of having a generic profile added at the beginning of the program.



```
TX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Request -> Hardware Revision
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Indication
        Characteristic Handle: 0021

TX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Request -> Firmware Revision
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Indication
        Characteristic Handle: 0023

TX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Request -> Software Revision
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Indication
        Characteristic Handle: 0025

TX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Request -> System Id
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Indication
        Characteristic Handle: 0027

TX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Request -> Ieee Rcdl
RX: GATTDB.Confirm->gBleSuccess
RX: GATTDBDynamic-AddCharacteristicDeclarationAndValue.Indication
        Characteristic Handle: 0029

        ---- Read Public Device Address  ----

TX: GAP-ReadPublicdeviceAddress.Request
RX: GAP.Confirm->gBleSuccess
RX: GAP-GenericEventPublicAddressRead.Indication
        Address: 90b5d0376000

        ---- Set Advertising Data  ----

TX: GAP-SetAdvertisingData.Request
RX: GAP.Confirm->gBleSuccess
RX: GAP-GenericEventAdvertisingDataSetupComplete.Indication
```


Figure 59. Device information characteristics.

To update heart rate measurements, battery level and sensor location, it's necessary to find their own handles, then, with that handles discovered, the values of the services previously added can be refreshed.

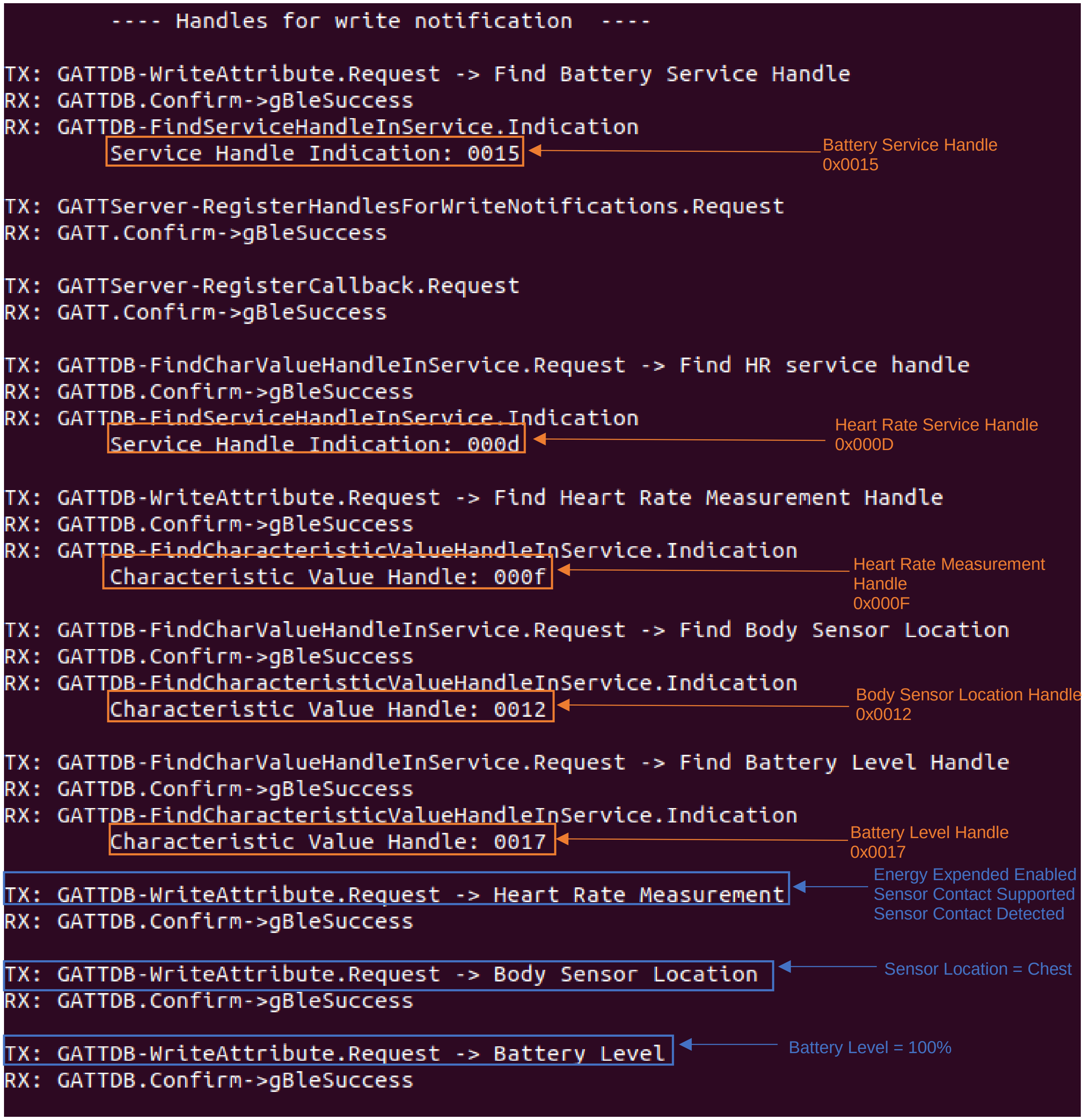


Figure 60. Handles for write notifications.

The setup is done, finally the host sends a request to start advertising and then waits for the NXP IoT Toolbox connection.

```
        ----------- Start Advertising -----------

TX: GAP-SetAdvertisingParameters.Request
RX: GAP.Confirm->gBleSuccess
RX: GAP-GenericEventAdvertisingParametersSetupComplete.Indication

TX: StartAdvertising.Request
RX: GAP.Confirm->gBleSuccess
RX: GAP-AdvertisingEventStateChanged.Indication
==================================================================================
----Please open NXP IoT Toolbox and select Heart Rate, and then NXP_HRS device.----
==================================================================================
=========== waiting for connection ===========
=========== waiting for connection ===========
=========== waiting for connection ===========
```

Set advertising parameters:
- Min & Max Connection Intervals
- Advertising type
- Address Type

Start Advertising

Configuration done, connect a smartphone with the NXP IoT Toolbox

Figure 61. Start advertising and wait for NXP IoT Toolbox connection.

Once the demo prints the "waiting for connection" message, open NXP IoT Toolbox, go to Heart Rate option and then tap on the "NXP_HRS" device.

For more detailed information about setting the NXP IoT Toolbox App, please refer to the mobile application user guide available at: nxp.com/docs/en/user-guide/KBLETMAUG.pdf?fromsite=ja

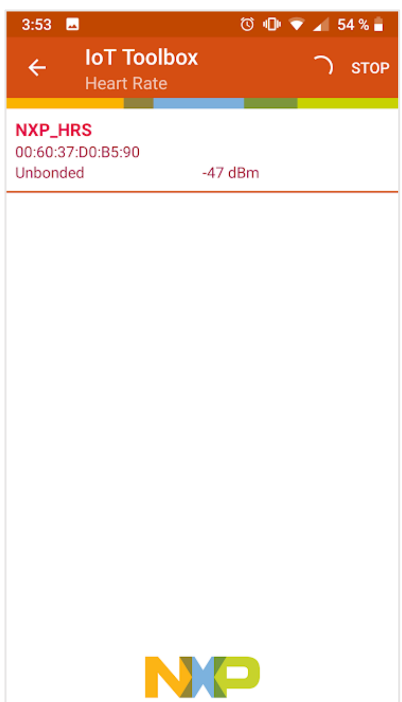


Figure 62. NXP IoT Toolbox, Heart Rate view.

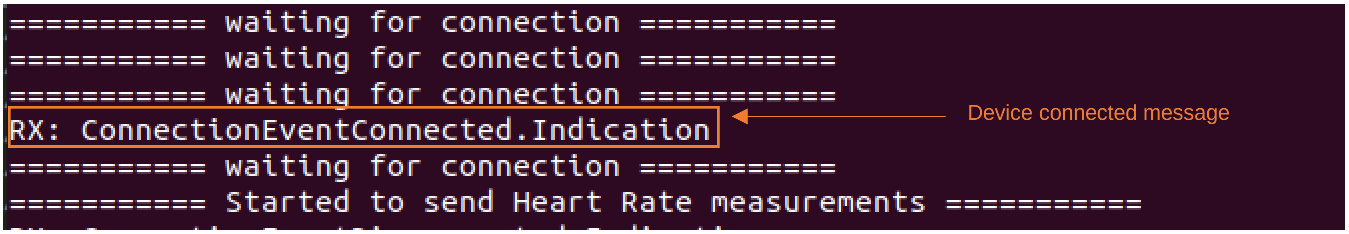


Figure 63. Device connected terminal view.

Program sending updates of heart rate measurement and battery level to the NXP IoT Toolbox app.

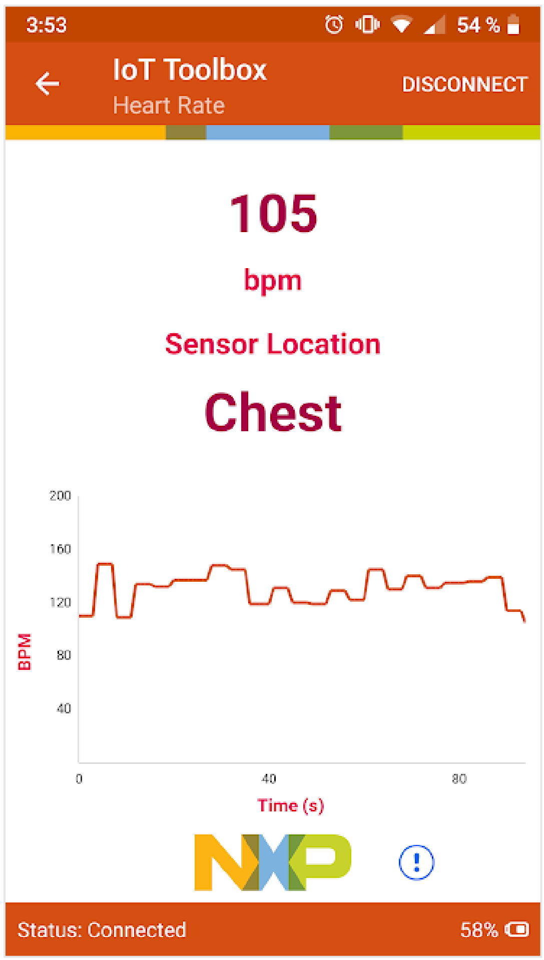


Figure 64. Application view.

18. Appendix A

Install Code.

The Install Codes allow the developer to create a temporary Unique Link Key on every joining node using its MAC address and change it once the joining process is completed.

- Global Link Key: This key is used in the joining process. Allows to connect and join a node to a ZigBee network. By default, ’ZigBeeAlliance09’ key is set.
- Unique Link Key: Has the same functionality of Link Key but it is unique to every joiner node. MAC address is used to generate these unique link key.
- Network Key: This key is used to decrypt the packets sent in the ZigBee network.
- Trust Center Link Key: Used for communications between the Trust Center and one other node. It is randomly generated by the Trust Center.

To have more information about ZigBee Security, refer to chapter 5.8 Implementing Zigbee Security on ‘Zigbee 3.0 Stack User Guide’ document that can be found in the next directory:

‘<SDK path>\docs\wireless\Zigbee\’.

18.1.1. Key Exchange Process

The internal process that is executed in this key exchange is the following:

- The coordinator creates a Unique Link Key for the joining node using install code command. To do this, the developer needs the MAC address (extended address) of the joiner node.
- The joiner node will make a join request to the network using its own Unique Link Key. If the coordinator generated key doesn’t match with the joiner key, the joiner will attempt to join again after security timeout and after 3 retries the joiner node fails with APS security fail.
- If the joiner was accepted, the coordinator will provide the Standard Network Key to the joiner node. This Transport Key packet will be encrypted with the Unique Link Key.
- The joiner router will request the Trust Center Key to the coordinator. This request will be encrypted with the Unique Link Key and the Standard Network Key provided before. This key will replace the Unique Link Key as the new Link Key.

In the figure below is described the key exchange process using the command ‘install code’ on a Zigbee network.

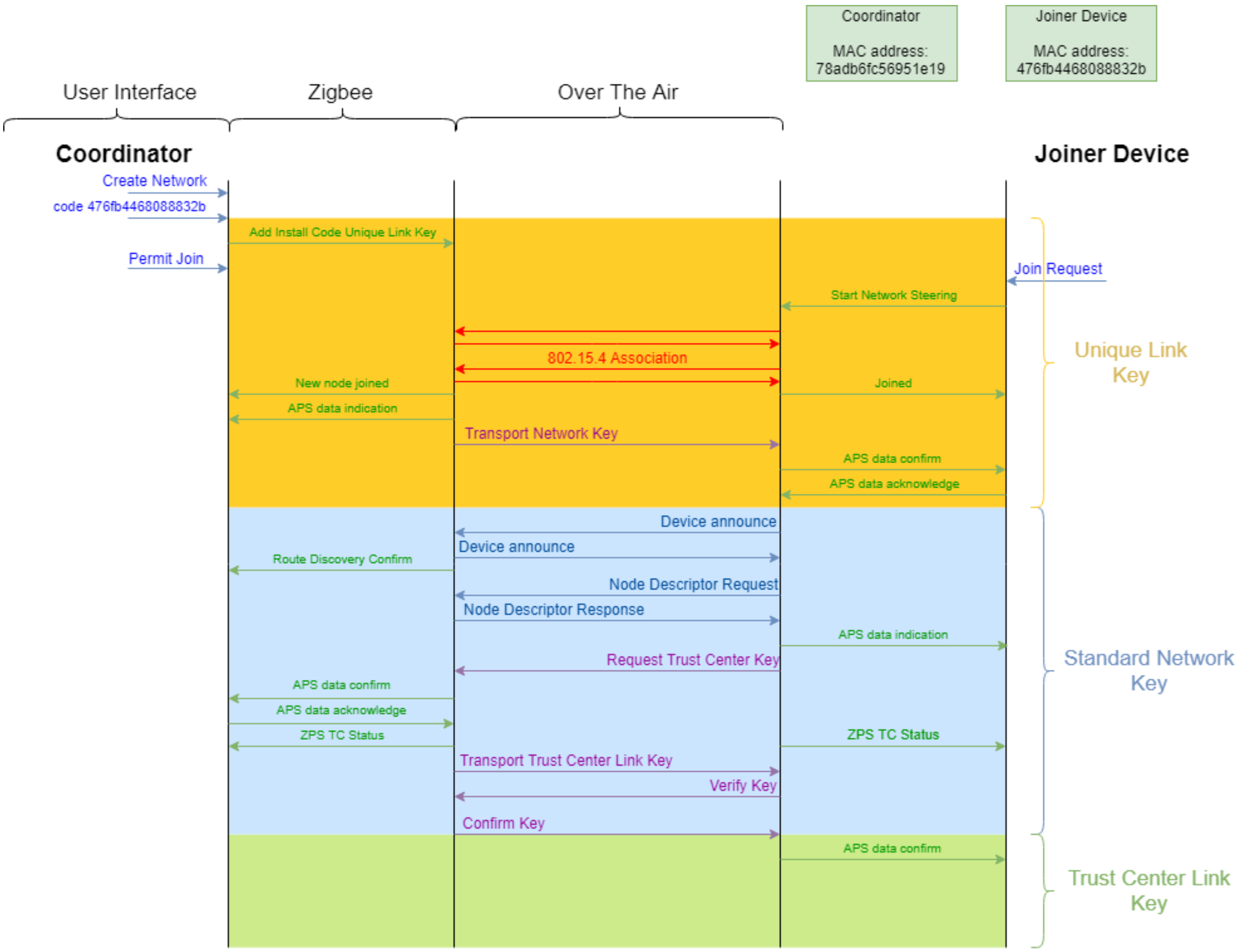


Figure 65. key exchange process.

On figure 66, find all Over the Air (OTA) packets using a sniffer and Ubiqua protocol analyzer, demonstrating the functionality described on figure 65.

6	10	11:35:58.3970	0.467472	15	ZigBee	MAC	Beacon Request		0xFFFF	124
7	28	11:35:58.3996	0.002688	15	ZigBee	NWK	Beacon: NwkOpen: RC 1: Depth…	0x0000		80
8	21	11:35:58.5424	0.142768	15	ZigBee	MAC	Association Request	47:6F:B4…	0x0000	125
9	5	11:35:58.5435	0.001056	15	ZigBee	MAC	Acknowledgement			125
10	18	11:35:59.0361	0.492624	15	ZigBee	MAC	Data Request	47:6F:B4…	0x0000	126
11	5	11:35:59.0371	0.000976	15	ZigBee	MAC	Acknowledgement			126
12	27	11:35:59.0444	0.007296	15	ZigBee	MAC	Association Response: Success	78:AD:B6…	47:6F:B…	167
13	5	11:35:59.0456	0.001264	15	ZigBee	MAC	Acknowledgement			167
14	73	11:35:59.0523	0.006640	15	ZigBee	APS	Transport Key	0x0000	0xC23D	168
15	5	11:35:59.0550	0.002736	15	ZigBee	MAC	Acknowledgement			168
16	57	11:35:59.0714	0.016432	15	ZigBee	ZDP	Device Announce	0xC23D	0xFFFF	127
17	51	11:35:59.0850	0.013520	15	ZigBee	NWK	Route Request: ManyToOne with…	0x0000	0xFFFF	169
18	57	11:35:59.1062	0.021280	15	ZigBee	ZDP	Device Announce	0x0000	0xFFFF	170
19	51	11:35:59.1847	0.078496	15	ZigBee	NWK	Route Request: ManyToOne with…	0xC23D	0xFFFF	128
20	55	11:35:59.3834	0.198688	15	ZigBee	NWK	Route Record	0xC23D	0x0000	129
21	5	11:35:59.3856	0.002160	15	ZigBee	MAC	Acknowledgement			129
22	48	11:35:59.3898	0.004208	15	ZigBee	ZDP	Node Descriptor Request	0xC23D	0x0000	130
23	5	11:35:59.3917	0.001936	15	ZigBee	MAC	Acknowledgement			130
24	45	11:35:59.4042	0.012496	15	ZigBee	APS	Acknowledgement	0x0000	0xC23D	171
25	5	11:35:59.4061	0.001840	15	ZigBee	MAC	Acknowledgement			171
26	62	11:35:59.4148	0.008720	15	ZigBee	ZDP	Node Descriptor Response	0x0000	0xC23D	172
27	5	11:35:59.4172	0.002384	15	ZigBee	MAC	Acknowledgement			172
28	45	11:35:59.4256	0.008400	15	ZigBee	APS	Acknowledgement	0xC23D	0x0000	131
29	5	11:35:59.4274	0.001840	15	ZigBee	MAC	Acknowledgement			131
30	58	11:35:59.4307	0.003280	15	ZigBee	APS	Request Key	0xC23D	0x0000	132
31	5	11:35:59.4329	0.002240	15	ZigBee	MAC	Acknowledgement			132
32	90	11:35:59.4390	0.006064	15	ZigBee	APS	Transport Key	0x0000	0xC23D	173
33	5	11:35:59.4422	0.003264	15	ZigBee	MAC	Acknowledgement			173
34	65	11:35:59.4522	0.009952	15	ZigBee	APS	Verify Key	0xC23D	0x0000	133
35	5	11:35:59.4547	0.002480	15	ZigBee	MAC	Acknowledgement			133
36	67	11:35:59.4599	0.005184	15	ZigBee	APS	Confirm Key	0x0000	0xC23D	174
37	5	11:35:59.4624	0.002528	15	ZigBee	MAC	Acknowledgement			174

Figure 66. Over the Air packets.

18.1.2. Generate Unique Link Key – Install code command

To successfully use the install code command, the developer must consider the following aspects:

To get familiar with the install code command, is recommended to see how it works. You can do this by using the coordinator demo example found in the next directory: *' <SDK path>\boards\frdmkw41z \wireless_examples\zigbee_3_0\'.*

To see the parameters needed by this command, enter 'help code' on the serial terminal:

```
$ help code
code – Provisions an install code into the APS Key Table
  code <addr> <install_code>
```

- <addr>: MAC address of the node that will be doing a join request to the network.
- <install_code>: 128-bit pre-configured key value.

Since <install_code> parameter is a pre-configured value on the joiner device, the user cannot enter any random value on the coordinator terminal. By default, this value is the joiner device MAC address repeated once.

The developer can change this value in the function 'bGetInstallCode(uint8_t* pInstallCode)' in the next lines (‘app_zb_utils.c’ file):

```
if(pIeeeAddr)
{
    /* Generate an install code that is the MAC address repeated once */
    FLib_MemCpyReverseOrder(pInstallCode, pIeeeAddr, sizeof(uint64_t));
    FLib_MemCpyReverseOrder(pInstallCode + sizeof(uint64_t), pIeeeAddr,
sizeof(uint64_t));
    result = TRUE;
}
```

How to Reach Us:

Home Page:
www.nxp.com

Web Support:
www.nxp.com/support

Document Number: AN12566
Rev. 0
09/2019