NXP Semiconductors
User's Guide

Document Number: KFSCIHAPIUG
Rev. 7, 08 April 2019

Kinetis FSCI Host Application Programming Interface

Contents

Chapter 1
About This Document

This document provides a detailed description for the Kinetis Wireless Host Application Programming Interface (Host API) implementing the Framework Serial Connectivity Interface (FSCI) on a peripheral port such as UART, USB, and SPI. The Host API can be deployed from a PC tool or 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 is the Kinetis Wireless Host Software Development Kit (SDK).

This version of the document describes the Thread IPv6 mesh stack, Bluetooth® Low Energy stack and ZigBee 3.0 stack running on Kinetis-W Series Wireless Connectivity Microcontrollers (MCUs), which are interfaced from a high-level OS (Linux® OS, Windows® OS) by the Host API and the Host SDK.

1.1 Audience

This document is for software developers who create tools and multichip partitioned systems using a serial interface to a Thread, Bluetooth LE or ZigBee 3.0 ‘black box’ firmware running on a Kinetis microcontroller.

Chapter 2
Deploying Host Controlled Firmware

2.1 Thread application configurations

IAR Embedded Workbench for Arm® (EWARM) and MCUXpresso IDE are the development toolchains used to deploy the Thread stack software applications.

Detailed information about how to build, deploy and debug an IAR or MCUXpresso IDE-based project are presented in the *Kinetis Thread Stack Demo Applications User's Guide*.

The Kinetis Thread software provides EWARM and MCUXpresso workspace projects for the application configurations, such as:

Table 1. Sample application configurations

Application	Default capabilities
`ble_thread_host_controlled_device`	A template for a thread_host_controlled_device which may run both Thread and Bluetooth LE stacks in parallel. Serial commands are distinguished by means of two virtual interfaces (0 – Thread, 1 – Bluetooth LE).
`ble_thread_router_wireless_uart`	A template for products running the dual mode Thread + Bluetooth LE stack, which is controlled by a Bluetooth LE controller, such as the Kinetis Bluetooth LE Toolbox Wireless UART, to connect to the Thread stack and perform Thread network management operations
`border_router`	A template for all product categories above which use a standalone Kinetis to forward IP packets from/to the Thread subnet and an alternate IP capable interface working via Ethernet, Wi-Fi, or USB to establish a local network or Internet end-to-end IP connectivity.
`end_device`	A template for mains powered or high-capacity battery products driven entirely by a Kinetis MCU which are NOT intended to be always-on: light fixtures, appliances, some door locks, some thermostats, and some resource constrained devices.
`host_controlled_device`	A template for products where a Kinetis MCU running the Thread stack is hosted by an application processor over UART or SPI; use of Host API tools is recommended for HLOS UNIX host systems; serves as sub-component for advanced asymmetric multiple chip border routers.
`low_power_end_device`	A template for low-capacity battery Kinetis products: sensors, remote controls, fobs, door locks.
`router_eligible_device`	A template for mains powered, always-on products driven entirely by a Kinetis MCU: security control panels, standalone sensor hubs, range extenders, smart plugs, some thermostats, wall light switches, some light fixtures, some appliances.

The current document demonstrates the host integration capabilities of the following projects:

1. `host_controlled_device` - Thread Integration with Linux OS Host on Serial (UART) Tunnel Interface on page 15; other sections that refer to THCI/FSCI.
2. `border_router` - Thread Integration with Linux OS Host on RNDIS Interface on page 23.
3. `ble_thread_host_controlled_device` - How to Control Both the Bluetooth LE and Thread Stacks on page 43.

2.2 Bluetooth LE application configuration

To exercise the Host API, the Bluetooth LE 'black box' firmware is required to be flashed on a compatible Kinetis-W platform. The Bluetooth LE 'black box' is represented by the 'ble_fsci_black_box' or the hybrid 'ble_thread_host_controlled_device' application firmware that can be interfaced and configured with FSCI commands over the serial interface.

The user can compile the black box image of the 'ble_fsci_black_box' software application using IAR Embedded Workbench for Arm (EWARM) or MCUXpresso IDE. For information on how to build the application see additional documentation provided in the package.

NOTE

The default configuration of the Bluetooth LE 'black box' application enables FSCI acknowledgement frames on both the transmission and reception paths. To use the 'black box' in conjunction with Host SDK, one needs to disable these in app_preinclude.h, i.e. set gFsciTxAck_c and gFsciRxAck_c to 0.

2.3 ZigBee 3.0 application configuration

The ZigBee 3.0 'black box' firmware is required to be flashed on a compatible Kinetis-W platform to exercise the Host API, represented by 'zb_fsci_black_box' application firmware.

Chapter 3
Host Software Overview

The FSCI (Framework Serial Communication Interface - Connectivity Framework Reference Manual) module allows interfacing the Kinetis protocol stack with a host system or PC tool using a serial communication interface.

FSCI can be demonstrated using various host software, one being the set of Linux OS libraries exposing the Host API described in this document. The NXP Test Tool for Connectivity Products PC application is another interfacing tool, running on the Windows OS. Both the Thread and Bluetooth LE stacks make use of XML files which contain detailed meta-descriptors for commands and events transported over the FSCI.

The FSCI module sends and receives messages as shown in the figure below. 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.

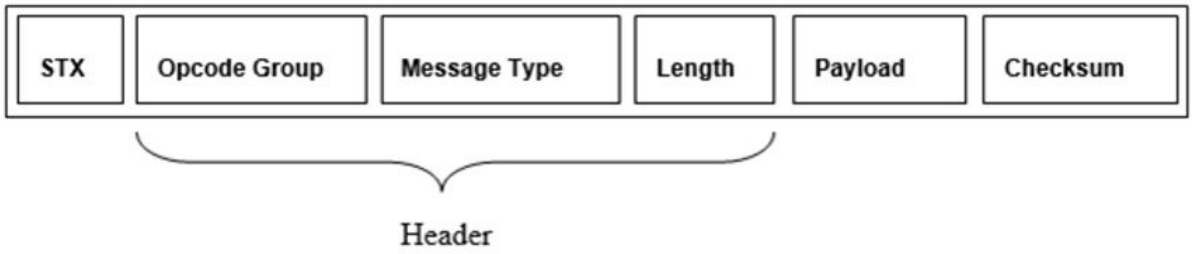

Figure 1. Sending and receiving messages

Table 2. FSCI send receive message formats

FSCI Frame FormatSTX	1	Used for synchronization over the serial interface. The value is always 0x02.
Opcode Group	1	Distinguishes between different layers (for example, Thread Management, Thread Utils, MeshCoP, DTLS – Thread; GAP, GATT, GATTDB – Bluetooth LE).
Message Type	1	Specifies the exact message opcode that is contained in the packet.
Length	2	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	Checksum field used to check the data integrity of the packet. When virtual interfaces are used to distinguish between the Bluetooth LE and Thread stacks when both run concurrently on the same device, this field expands to two bytes to embed the virtual interface number.

The Kinetis Wireless Host SDK consists 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. On top of these libraries, Python bindings provide easy development of user applications.

The Kinetis Wireless Host SDK is meant to run on Windows OS, Linux OS, Apple OS X® and OpenWrt. This version of the document describes a subset of functionality related to interfacing with a Thread/Bluetooth LE stack instance from a Linux OS system, with focus on Python language bindings.

3.1 Kinetis wireless host software system block diagram

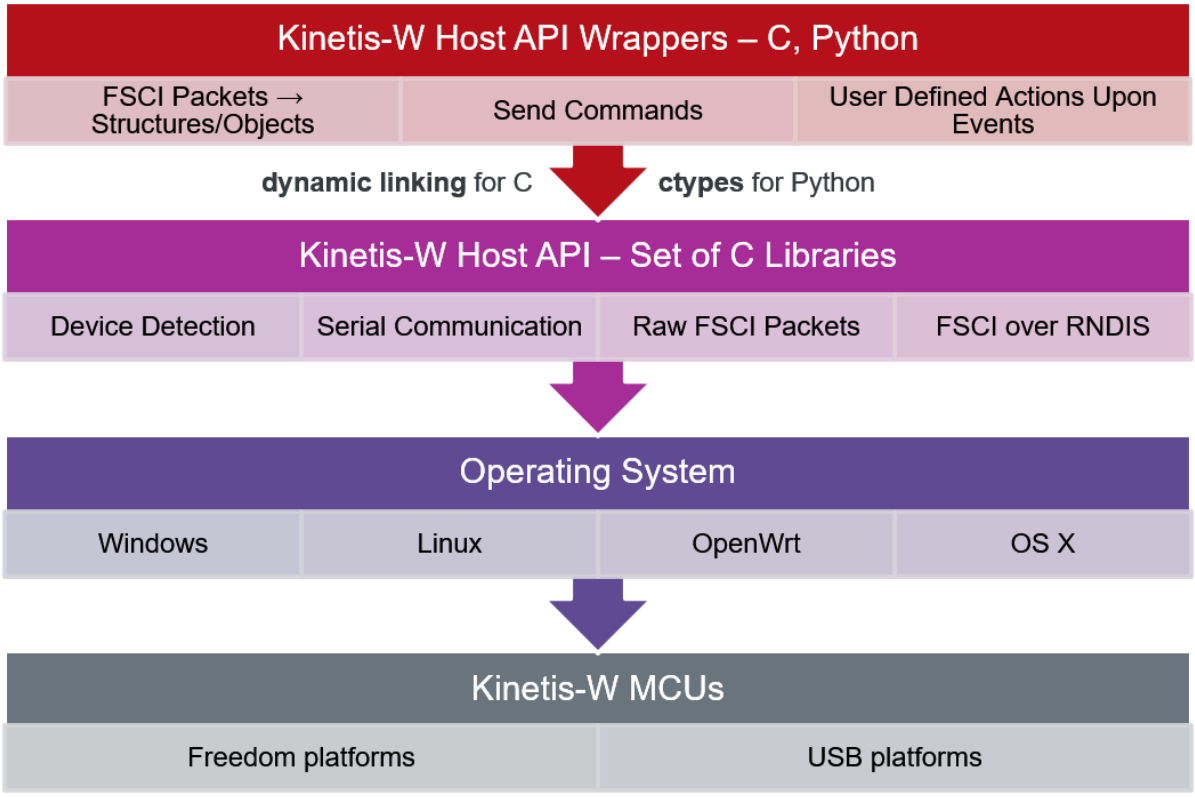

Figure 2. Kinetis host software system block diagram

3.2 Directory tree

```
├── demo                      A set of programs to demonstrate functionality.
|   ├── GetKinetisDevices.c  Outputs all Kinetis devices available on serial to the
console.
|   ├── Makefile
|   ├── make_tun.sh         Script for automating the creation of a TUN/TAP interface on
Linux.
|   ├── PCAPTest.c          Sends THCI frames over Ethernet by leveraging the RNDIS driver.
|   ├── SPITest.c           Sends THCI frames over SPI bus.
|   └── Thread_KW_Tun.c     IP Tunnel functionality interfacing the Linux and Thread IP
stacks.
```

```
├── include              All the headers used are present in this folder.
│   ├── physical             Headers specific to the physical serial bus or PCAP interface
used by the NXP device.
│   │   ├── PhysicalDevice.h
│   │   ├── PCAP
│   │   │   └── PCAPDevice.h
│   │   ├── SPI
│   │   │   ├── SPIConfiguration.h     Handles SPI slave bus configuration(max speed Hz,
bits per word).
│   │   │   └── SPIDevice.h            Encapsulates an OS SPI device node into a well-
defined structure.
│   │   └── UART
│   │       ├── UARTConfiguration.h    Handles serial port configuration (baudrate,
parity).
│   │       ├── UARTDevice.h         Encapsulates an OS UART device node into a well-
defined structure.
│   │       └── UARTDiscovery.h        Handles the discovery of UART connected devices.
│   ├── protocol                        Headers specific to the transmission of frames.
│   │   ├── Framer.h                   A state machine implementation for sending/
receiving frames.
│   │   └── FSCI                       Headers specific to the FSCI protocol.
│   │       ├── FSCIFrame.h
│   │       └── FSCIFramer.h
│   └── sys                        General purpose headers for interaction with the OS,
message queues and more.
│       ├── EventManager.h         Handles event registering, notifying and callback
submission.
│       ├── hsdk                   Macros for error reporting.
│       ├── h                      Logger implementation for debugging.
│       ├── hsdkOSCommon.h     Interaction with OS specifics.
│       ├── MessageQueue.h     A standard message queue implementation (linked list).
│       ├── RawFrame.h         Describes the format of a frame, independent of the
protocol.
│       └── utils.h          Various functions to manipulate structures and byte arrays.
├── ConnectivityLibrary.sln       Microsoft Visual Studio 2013 solution file.
├── Makefile
├── physical           Implementation of the physical UART/SPI serial bus or PCAP
interface module.
│   ├── PCAP
│   │   └── PCAPDevice.c
│   ├── PhysicalDevice.c
│   ├── SPI
│   │   ├── SPIConfiguration.c
│   │   └── SPIDevice.c
│   └── UART
│       ├── UARTConfiguration.c
│       ├── UARTDevice.c
│       └── UARTDiscovery.c
├── protocol               Implementation of the protocol module relating to FSCI.
│   ├── Framer.c
│   └── FSCI
│       ├── FSCIFrame.c
│       └── FSCIFramer.c
├── README.md
├── res
│   ├── 77-mm-usb-device-blacklist.rules      Udev rules for disabling ModemManager.
│   └── hsdk.conf                             Configuration file to control FSCI-ACKs.
└── sys                              Implementation of the system/OS portable base module.
    ├── EventManager.c
    ├── hsdkEvent.c
```

```
    ├── hsdkFile.c
    ├── hsdkLock.c
    ├── hsdkLogger.c
    ├── hsdkSemaphore.c
    ├── hsdkThread.c
    ├── MessageQueue.c
    ├── RawFrame.c
    └── utils.c
```

3.3 Device detection

The Kinetis Wireless Host SDK can detect every USB attached peripheral device to a PC. On Linux OS, this is done via `udev`. Udev is the device manager for the Linux OS kernel and was introduced in Linux OS 2.5. Using the manager, the Kinetis Wireless Host API can provide the Linux OS path for a device (for example, /dev/ttyACM0) and whether the device is a supported USB device, based on the vendor ID/product ID advertised. Upon device insertion, the USB `cdc_acm` kernel module is triggered by the kernel for interaction with TWR, USB and FRDM devices.

On Windows OS, attached peripherals are retrieved from the registry path HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\SERIALCOMM, resulting in names such as COMxx which must be used as input strings for the Python scripts which require a device name.

3.4 Serial port configuration

The Host SDK configures a serial UART port with the following default values:

Table 3. Host SDK – UART default values

Configuration	Value
Baudrate	115200
ByteSize	EIGHTBITS
StopBits	ONE_STOPBIT
PARITY	NO_PARITY
HandleDSRControl	0
HandleDTRControl	ENABLEDTR
HandleRTSControl	ENABLERTS
InX	0
OutCtsFlow	1
OutDSRFlow	1
OutX	0

The library only allows the possibility to change the baudrate, as this is the most common scenario.

NOTE

For Kinetis devices using a USB connection interfaced directly (where the USB stack runs on the Kinetis device and the system is NOT using an OpenSDA UART to USB converter), the baudrate is not necessary and setting it has no effect.

The Host SDK configures a serial SPI port with the following default values:

Table 4. Host SDK – SPI default values

Configuration	Value
Transfer Mode	SPI_MODE_0
Maximum SPI transfer speed (Hz)	1 MHz
Bits per word	8

The library only allows the possibility to change the maximum SPI transfer speed.

3.5 Logger

The Host SDK implements a logger functionality which is useful for debugging. Adding the compiler flag USE_LOGGER enables this functionality.

When running programs that make use of the Host SDK, a file named hsdk.log appears in the working directory. This is an excerpt from the log:

```
HSDK_INFO - [6684] [PhysicalDevice]InitPhysicalDevice:Allocated memory for PhysicalDevice
HSDK_INFO - [6684] [PhysicalDevice]InitPhysicalDevice:Initialized device's message queue
HSDK_INFO - [6684] [PhysicalDevice]InitPhysicalDevice:Created event manager for device
HSDK_INFO - [6684] [PhysicalDevice]InitPhysicalDevice:Created threadStart event
HSDK_INFO - [6684] [PhysicalDevice]InitPhysicalDevice:Created stopThread event
HSDK_INFO - [6684] [PhysicalDevice]AttachToConcreteImplementation:Attached to a concrete
implementation
HSDK_INFO - [6684] [PhysicalDevice]InitPhysicalDevice:Created and start device thread
HSDK_INFO - [6684] [Framer]InitializeFramer:Allocated memory for Framer
HSDK_INFO - [6684] [Framer]InitializeFramer:Created stopThread event
HSDK_INFO - [6684] [Framer]InitializeFramer:Initialized framer's message queue
HSDK_INFO - [6684] [Framer]InitializeFramer:Created event manager for framer
```

Chapter 4
Linux OS Host Software Installation Guide

4.1 Libraries

4.1.1 Prerequisites

Packages: build-essential, libudev, libudev-dev, libpcap, libpcap-dev. Use apt-get install on Debian-based distributions.

The Linux OS kernel version must be greater than 3.2.

4.1.2 Installation

```
$ pwd
/home/user/hsdk
$ make
$ find build/ -name "*.so"
build/libframer.so
build/libsys.so
build/libphysical.so
build/libuart.so
build/librndis.so
build/libfsci.so
build/libspi.so
$ sudo make install
```

By default, make generates `shared` libraries (having .so extension). The step make install (superuser privileges required) copies these libraries to /usr/local/lib, which is part of the default Linux OS library path. The installation prefix may be changed by passing the variable PREFIX, e.g. make install PREFIX=/usr/lib. The user is responsible for making sure that PREFIX is part of the system's LD_LIBRARY_PATH. On low-resource systems where libudev or libpcap are not present, the user may opt to not link against them by passing the variables UDEV and RNDIS respectively, i.e. make UDEV=n RNDIS=n. Lastly, support for the SPI physical layer may be disabled in the same manner by passing the variable SPI=n.

Static libraries can be generated instead, by modifying the LIB_OPTION variable in the Makefile from `dynamic` to `static` (.a extension).

make install also disables the ModemManager control for the connected Kinetis devices. Otherwise, the daemon starts sending AT commands that affect the responsiveness of the afore-mentioned devices in the first 20 seconds after plug in.

4.2 Demos

4.2.1 Installation

```
$ pwd
/home/user/hsdk/demo
$ make; make spi
```

```
$ ls bin/
GetKinetisDevices  PCAPTest  SPITest  Thread_KW_Tun
```

These demos are provided in this package:

- `GetKinetisDevices`: this program detects the Kinetis boards connected to the serial line and outputs the path to the console:

  ```
  $ ./GetKinetisDevices
  NXP Kinetis-W device on /dev/ttyACM0.
  NXP Kinetis-W device on /dev/ttyACM1.
  ```

- `Thread_KW_Tun`: the functionality of the Thread serial (UART tunnel driver described in Thread Integration with Linux OS Host on Serial (UART) Tunnel Interface on page 15.
- `PCAPTest`: this program demonstrates the functionality of the border_router example application as an USB RNDIS host, where THCI frames are sent over a back channel of RNDIS. Find additional details in Thread Integration with Linux OS Host on Serial (UART) Tunnel Interface on page 15.
- `SPITest`: this program demonstrates how to open and configure the speed of a device which exposes a SPI interface to the host processor.

Chapter 5
Windows OS Host Software Installation Guide

5.1 Libraries

5.1.1 Prerequisites

Microsoft Visual Studio® 2013 is required to build the host software. Open the solution file ConnectivityLibrary.sln and build it for either Win32 or x64, depending on your setup requirements. The output directory contains a file named HSDK.dll, which can be thought of as a bundle of all the shared libraries from Linux OS, except for SPI and RNDIS (in other words, libspi.so, librndis.so). Currently, SPI and RNDIS interfaces to the board are not supported by the Windows host software.

Prebuilt HSDK.dll files are available under directory hsdk-python\lib.

5.1.2 Installation

The host software for the Windows OS is designed to work in a Python environment by contrast to the Linux OS where standalone C demos also exist.

Download and install the latest Python 2.7.x package from www.python.org/downloads/. When customizing the installation options, check **Add python.exe to Path**.

5.1.2.1 Using prebuilt library

1. Depending on your Python environment architure (not Windows architecture) copy the appropriate HSDK.dll from hsdk-python\lib\[x86|x64] to <Python Directory>\DLLs, which defaults to C:\Python27\DLLs when using the default Python installation settings.
2. Download and install Visual C++ Redistributable Packages for Microsoft Visual Studio 2013, depending on the Windows architecture of your system (vcredist_x86.exe or vcredist_x64.exe) from www.microsoft.com/en-us/download/details.aspx?id=40784.
3. Download and install the Microsoft Visual C++ Compiler for Python 2.7 from the download center .

5.1.2.2 Using local built library

1. Depending on your Python environment architure (not Windows architecture), build the appropriate Microsoft Visual Studio 2013 solution configuration and then copy HSDK.dll to <Python Directory>\DLLs (which defaults to C:\Python27\DLLs when using the default Python installation settings).
2. Download and install the Microsoft Visual C++ Compiler for Python 2.7 from the download center .

Optionally, copy the hsdk\res\hsdk.conf to <Python Directory>\DLLs to control the behavior of the FSCI-ACK synchronization mechanism.

5.2 Demos

See Host API Python Bindings on page 29.

Chapter 6
Thread Integration with Linux OS Host on Serial (UART) Tunnel Interface

The Kinetis Thread stack implements a serial Tunnel media interface which can be used to exchange FSCI encapsulated IPv6 packets with a host system. First, ensure that the macro SERIAL_TUN_IF is enabled in the project file:middleware\wireless \nwk_ip_<ver>\examples\[multicore]\host_controlled_device\config\config.h. For multi-core platforms the symbol must be enabled on all cores.

```
#define SERIAL_TUN_IF                    1
```

and then select the appropriate Neighbor Discovery (ND) setting for your setup in app_serial_tun_h: ND host (default) or ND router or disabled:

```
#define BR_ROUTER_MODE      0
#define BR_HOST_MODE        1
```

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. In addition, the media interface must be configured statically with IPv6 addresses and routes. Previous versions of the Thread stack used prefix fd01::/64 for communicating to external networks, with address fd01::1 configured for the interface itself. Host artifacts for firmware in TUN mode applying this static configuration are described in Chapter 6.2 Userspace module in TUN mode.

To provide connectivity to the host, there are 2 components needed: the TUN/TAP kernel module, which allows the operating system to create virtual interfaces and a program that knows to encapsulate/decapsulate IP packets to/from FSCI/THCI.

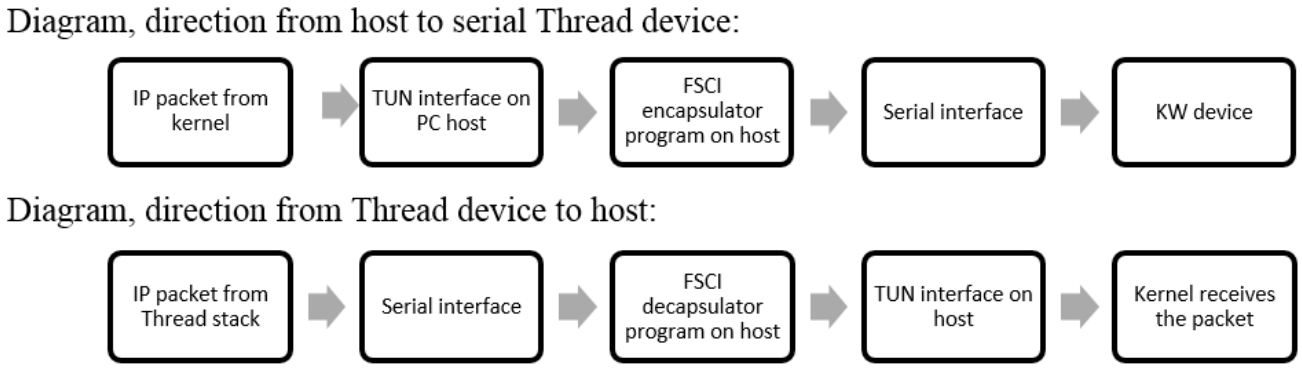

Figure 3. Host-to-serial and Thread-device-to-host

6.1 TUN/TAP

In computer networking, TUN/TAP devices are virtual network adapters that are not backed up by a physical interface. TUN represents a virtual layer-3 device which can operate with IP packets. TAP describes virtual layer 2 devices which usually handle Ethernet frames.

On Linux OS, the kernel modules which handle TUN/TAP are usually built in. If not, these can be easily compiled from the Linux OS source, by enabling Universal TUN/TAP device driver when issuing make menuconfig, or by enabling CONFIG_TUN=y or m

in the build system .config file. If the kernel module is built as an external module, insmod tun.ko enables the functionality in the system.

After ensuring that the Linux OS enables this component, a new file with path /dev/net/tun appears in the filesystem. Any operations that involve a virtual TUN/TAP interface open this file and any subsequent read or write is to be done with respect to this file. Creating a new virtual TUN/TAP interface and configuring it can be done with basic iproute2 commands:

```
$ sudo ip tuntap add mode tun threadtun0
$ ip link show threadtun0
5: threadtun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN mode DEFAULT qlen 500
    link/none
$ sudo ip -6 addr add FD01::2 dev threadtun0
$ ip address show threadtun0
5: threadtun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN qlen 500
    link/none
    inet6 fd01::2/128 scope global
       valid_lft forever preferred_lft forever
```

Note the DOWN state of the interface. A virtual TUN or TAP interface becomes UP as soon as a userspace program attaches to it, as in opening the special /dev/net/tun file. For the Thread stack integration scenario, the userspace is the FSCI encapsulator/decapsulator from the above diagrams.

After all the modules are connected, the userspace program can read packets injected by the kernel into the TUN/TAP interface, encapsulate it as a FSCI packet and send it to the development board. On the return path, the program can read data coming from the board, extract the Ethernet/IPv6 packet and write it to the TUN/TAP file descriptor, reaching back into the kernel IP stack.

6.2 Userspace module in TUN mode

In this chapter, various references to a userspace demo program capable of sending IPv6 packets to a Kinetis device over FSCI are made. This program is part of the deliverable under the path: hsdk/demo/Thread_KW_Tun.c. This program operates by default on TUN interfaces; set the define SERIAL_TAP to 1 in order to make it operate on TAP interfaces.

The program can be compiled by calling make in the hsdk/demo directory, and the executable file can be found under hsdk/demo/bin directory.

The program takes two mandatory arguments and three optional. These are, respectively: the path of the Kinetis board port in the /dev file system, the name of the serial TUN/TAP interface, whether a factory reset occurs at the start of the program, the 802.15.4 channel to be used and the baudrate for configuring the serial port. The path of the Kinetis board port, depending on the operating system, should resemble /dev/ttyACMX on the Linux OS. The second argument must be the name of the previously created virtual interface on Linux OS (such as threadtun0 from the previous example). The first two optional arguments must be numbers. A factory reset resembles a Boolean value where value of 1 resets the device and 0 does not change state. The 802.15.4 channel must be a number within 11 and 25 (25 is the default value if none entered). The fifth and last parameter is the baudrate which should be entered as a plain number, for example, 115200. Note that 115200 is the default baudrate used if none entered.

The program starts with configuring the device to create a Thread network on the specified channel with commissioning capabilities. This behavior may be eliminated by setting the PROVISION macro to 0. Afterwards it attaches to the virtual TUN/TAP interface and, in an infinite loop, starts reading Ethernet/IPv6 packets coming from the kernel. It then encapsulates them in the FSCI format, sends the packet and waits for packets on the return path on a separate thread. The packets received are decapsulated and the Ethernet/IPv6 remainder is injected back into the TUN/TAP interface to be handled by the kernel. Both the packets read from and injected back in the TUN/TAP interface are dumped to the console, having either the transmit tags (data read is transmitted further to the device) or receive tags respectively. Define DEBUG to 0 to disable the console output.

Before using the interface, it must be configured for proper interaction with the serial TUN media interface on the Kinetis device side. By default, the Kinetis side is configured with the IPv6 address `FD01::1`, when ND is turned off.

The routers and end devices that further join the network have assigned a Thread Unique Local Address in range `FD01:0000:0000:3EAD::/64`. To adhere to these requirements, in the hsdk/demo folder there is a script that automates the creation of the TUN interface, while configuring it with the proper IPv6 addresses and routes.

```
$ cat make_tun.sh
#!/bin/bash
# Create a new TUN interface for Thread interaction.
ip -6 tuntap add mode tun threadtun0
# Assign it a global IPv6 address.
ip -6 addr add FD01::2 dev threadtun0
# Add route to default address of Serial TUN embedded interface.
ip -6 route add FD01::1 dev threadtun0
# Add route to Unique Local /64 Prefix via threadtun0.
ip -6 route add FD01:0000:0000:3EAD::/64 dev threadtun0
# The interface is ready.
ip link set threadtun0 up

# Enable IPv6 routing on host.
sysctl -w net.ipv6.conf.all.forwarding=1
$ chmod +x make_tun.sh
$ sudo ./make_tun.sh
```

After calling this script, the environment is prepared for sending native IPv6 packets to a Thread remote destination via an attached Thread Border Router. The script adds a route to the IPv6 space of `FD01:0000:0000:3EAD::/64` and the host may now reach remote Thread nodes on their Unique Local address. Other routes may be added in a similar manner, depending on the application requirements.

```
$ ip address show dev threadtun0
3: threadtun0: <NO-CARRIER,POINTOPOINT,MULTICAST,NOARP,UP> mtu 1500 qdisc pfifo_fast state DOWN qlen 500
    link/none
    inet6 fd01::2 scope global
       valid_lft forever preferred_lft forever

$ ip -6 route show
[...]
fd01:0:0:3ead::/64 dev threadtun0  metric 1024
[...]
```

Example: Considering the following simple topology: a Thread Leader/Border Router (project thread_host_controlled_device) connected to a Linux system (assigned /dev/ttyACM0) and another Thread Router joined to the Leader. The assumption is that the Thread Router is a thread_router_eligible_device, which exposes the management server over the USB/UART shell. Its IPv6 addresses can be retrieved by issuing 'ifconfig' from screen/putty, which prints the following to the console:

```
$ ifconfig
Interface 0: 6LoWPAN
        Link local address (LL64): fe80::a49d:71ab:5f12:b200
        Mesh local address (ML16): fd2f:62d8:42f3::ff:fe00:400
        Unique local address: fd01::3ead:7c16:37b5:e6ef:701a
        Link local all Thread Nodes (MCast):  ff32:40:fd2f:62d8:42f3::01
        Realm local all Thread Nodes (MCast): ff33:40:fd2f:62d8:42f3::01
```

Afterwards, the Thread Router can be pinged directly from the Linux OS on its Unique Local Address as shown below:

```
Terminal 1: $ sudo ./bin/Thread_KW_Tun /dev/ttyACM0 threadtun0
Terminal 2: $ ping6 -c 1 fd01::3ead:7c16:37b5:e6ef:701a
Output from terminal 1: hexdump of the TX/RX packets
TX:
  0000  68 00 60 00 00 00 00 40 3a 40 fd 01 00 00 00 00  h.`....@:@......
  0010  00 00 00 00 00 00 00 00 00 02 fd 01 00 00 00 00  ................
```

```
  0020  3e ad 7c 16 37 b5 e6 ef 70 1a 80 00 f2 93 15 d8  >.|.7...p.......
  0030  00 01 1d 2a 01 56 00 00 00 00 51 3c 05 00 00 00  ...*.V....Q<....
  0040  00 00 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d  ................
  0050  1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d  .. !"#$%&'()*+,-
  0060  2e 2f 30 31 32 33 34 35 36 37                    ./01234567
RX:
  0000  60 00 00 00 00 40 3a 3e fd 01 00 00 00 00 3e ad  `....@:>......>.
  0010  7c 16 37 b5 e6 ef 70 1a fd 01 00 00 00 00 00 00  |.7...p.........
  0020  00 00 00 00 00 00 00 02 81 00 f1 93 15 d8 00 01  ................
  0030  1d 2a 01 56 00 00 00 00 51 3c 05 00 00 00 00 00  .*.V....Q<......
  0040  10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f  ................
  0050  20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f   !"#$%&'()*+,-./
  0060  30 31 32 33 34 35 36 37                         01234567
```

Output from terminal 2:

```
PING fd01::3ead:7c16:37b5:e6ef:701a(fd01::3ead:7c16:37b5:e6ef:701a) 56 data bytes
64 bytes from fd01::3ead:7c16:37b5:e6ef:701a: icmp_seq=1 ttl=62 time=33 ms
```

NOTE

When the program starts with a factory reset of the device and the Linux OS re-enumerates the port to the lowest available ttyACM number. Situations where /dev/ttyACM1 becomes /dev/ttyACM0 are possible. However, the program does not handle them. Ensure that a ttyACM device is used which does not get re-enumerated to a different number upon reset (for example, /dev/ttyACM0 always re-enumerates to itself).

6.3 Userspace module in TAP mode and global addressing scheme

One can use either hsdk/demo/Thread_KW_Tun.c with SERIAL_TAP defined on 1 or hsdk/demo/Thread_Shell.c. The latter is preferred because it simulates a user-friendly shell over FSCI interface that resembles the one present in the router_eligible_device projects, plus features designed to re-establish communication after a power failure/reset.

hsdk/demo/make_tap.sh can be used to create a virtual TAP interface named threadtap0, like in the previous chapter. Thread_Shell has only two arguments: the device node and 802.15.4 channel to be used; it automatically opens the threadtap0 interface. Example of usage:

./Thread_Shell /dev/ttyACM0 25

In case make_tap.sh is not used, the user is requested to create a TAP interface manually and update the TAP_IFNAME symbol in Thread_Shell.c to the new tunnel interface name.

This section shows how to test external IPv6 connectivity through the Thread border router device which is provisioned an external IPv6 prefix by an external OpenWrt based device, when ND host mode is enabled. The application acts as a Border Router with respect to propagating the network provisioning information to the downstream Thread network (such as IPv6 address assignment based on prefix delegation) and interfacing end-to-end IP layer connectivity to the Thread-only nodes.

To run the application use case, the board must be connected to a Linux box that is itself connected to a router or switch which provides IPv6 prefix provisioning by means of DHCPv6-PD (Prefix Delegation) and IPv6 Neighbor Discovery Protocol (ND). The TAP interface must be bridged with the uplink interface towards the DHCPv6-PD capable OpenWrt router. Assuming eth0 is the

name of the uplink interface and threadtap0 is the name of the TAP interface, the following commands are needed on the Linux box to bridge the Thread TAP and uplink interfaces:

```
# brctl addbr br0
# brctl addif br0 eth0
# brctl addif br0 threadtap0
```

Starting Thread_Shell enables the IPv6 traffic flow between the OpenWrt router and the Thread border router.

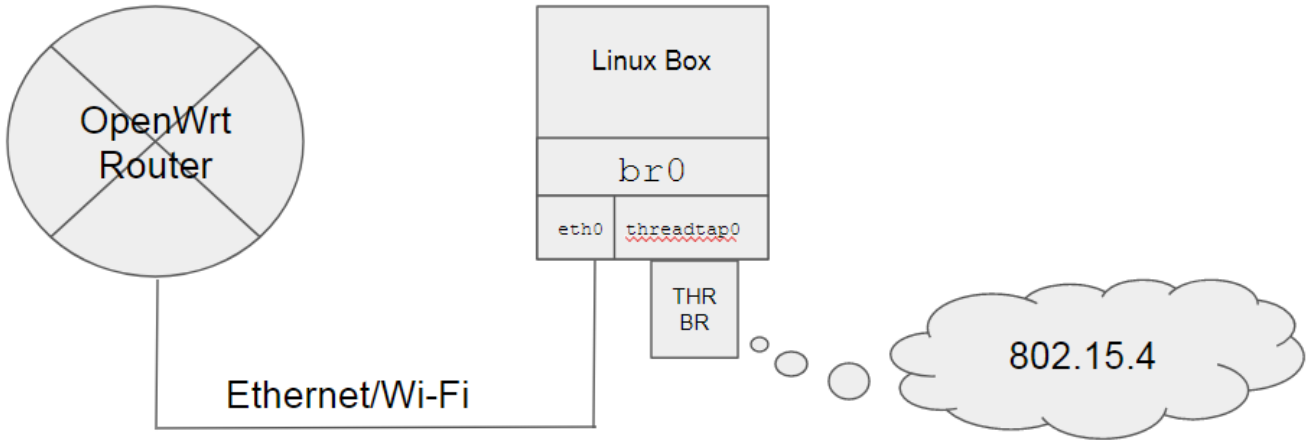

Figure 4. Topology

When the Thread stack is started, either when a new network is created or an existing one joined to, the border router tries to acquire global addresses via DHCPv6-PD. For this operation to succeed, the OpenWrt router must implement RFC3633 and must be configured with a /48 ULA prefix. If the request is successful, a /64 prefix sub-range from the /48 ULA prefix range will be assigned to the border router, for it to distribute into the Thread network. A global address will be configured to the 6LoWPAN interface of the border router and to other existing devices in the Thread network or newly joined ones. Communication that uses this global ULA-based addressing scheme will work with no other added static routes, i.e. the Thread end nodes are accessible from the LAN hosts and vice-versa.

Thread_Shell is also capable of detecting link changes on the bridged interface (eth0), useful for situations where the OpenWrt router resets. Make sure to define UPLINK_IFNAME to the appropriate uplink interface name in Linux. In the unlikely event of a router reset, Thread_Shell notifies the Thread border router to re-establish the communication with the router, by means of a new DHCPv6-PD request.

Chapter 7
Applications Using the TUN Interface

7.1 External commissioning of untrusted Thread radio devices

Terms such as Border Router, Commissioner, Joiner, Joiner Router, and MeshCoP appear in this chapter. See Thread 1.1, Specification for an in-depth overview of the commissioning protocol used in Thread networks.

A Border Router, as defined by the Thread specification, is a device capable of forwarding packets between a Thread Network and a non-Thread Network where the Commissioner is reachable. NXP offers two solutions for a Border Router:

1. NXP FRDM-K64F – interface to Commissioner over physical Ethernet interface.
2. Kinetis-W MCUs – any form-factor – connected to a physical or virtual Linux OS machine which provides various means of interfacing with the Commissioner, depending on the capabilities: Ethernet, Wi-Fi, Virtual Machine Software Adapter, and so on.

This chapter focuses on the second Border Router solution because it is tightly coupled with the Host API solution over a TAP interface described throughout this document.

The Thread Group provides an Android application that acts as an external Commissioner over Wi-Fi for the Thread network. Install Thread 1.1 Commissioning App from Google Play Store prior to starting the setup.

The following steps are required to establish a MeshCoP channel between the Android smartphone and the Thread Border Router:

- Bridge the TAP and uplink interfaces as described in chapter 6.3 "Userspace module and global addressing scheme". Then run Thread_Shell to enable the IPv6 traffic flow between the LAN hosts and the Thread border router and create/join a Thread network:

```
#./bin/Thread_Shell /dev/ttyACM0 25
            > thr create
            [... output elided ...]
```

- Connect the Android smartphone to the LAN
- Open Thread 1.1 Commissioning App. A "BorderRouter THREAD" entry should be available under "AVAILABLE BORDER ROUTERS". Click on it and enter "THREAD" as the Thread Admin Password. "THREAD" is the Border Router device password (PSKd) and it is configured by Thread_Shell on network creation. After a few seconds the MeshCoP connection is established and Thread_Shell will print out the following messages as an indication:

```
RX: THR_EventNwkCommissioning.Indication -> Commissioner->BR Accepted
RX: THR_EventNwkCommissioning.Indication -> Commissioner petition alert
```

  If the application does not detect any available border routers, please check the network settings and disable firewall rules that may be filtering out mDNS traffic.

- Prepare a Joiner device to be added to the network. For simplicity let us assume this device is running the router_eligible_device project. Make sure both the Border Router and Joiner operate on the same 802.15.4 channel. Open Joiner's shell console to retrieve its IEEE EUI-64 address and get or set the PSKd:

  $ thr get eui eui: 0x006037831472D1BA

  $ thr get pskd pskd: THREAD

  If you plan to change the password, please be advised that the PSKd must respect the following format, according to the Thread 1.1 Specification: "A Joining Device Credential is encoded as uppercase alphanumeric characters (base32-thread:

0-9,A-Y excluding I,O,Q, and Z for readability) with a minimum length of 6 such characters and a maximum length of 32 such characters."

Create a QR code of type Text from text "v=1&cc=THREAD&eui=006037831472D1BA". Please note that the EUI must not contain the "0x" prefix. Continue and scan this QR code from the Android app and then issue `thr join` in Joiner's shell. The Android user interface will confirm if the Joiner was successfully added, while Thread_Shell will print out the following indications:

```
RX:THR_EventNwkCommissioning.Indication -> Commissioner<-Joiner Outbound Data Relayed
RX:THR_EventNwkCommissioning.Indication -> Commissioner->Joiner Inbound Data Relayed
RX:THR_EventNwkCommissioning.Indication -> Commissioner<-Joiner Outbound Data Relayed
RX: THR_EventNwkCommissioning.Indication -> Commissioner->Joiner Inbound Data Relayed
RX:THR_EventNwkCommissioning.Indication -> Commissioner<-Joiner Outbound Data Relayed
RX:THR_EventNwkCommissioning.Indication -> Commissioner->Joiner Inbound Data Relayed
RX:THR_EventNwkCommissioning.Indication -> Commissioner<-Joiner Outbound Data Relayed
RX:THR_EventNwkCommissioning.Indication -> Commissioner->Joiner Inbound Data Relayed
RX:THR_EventNwkCommissioning.Indication -> Commissioner<-Joiner Outbound Data Relayed
RX:THR_CommissioningDiagnostic.Indication -> OUT JOIN_ENT_REQ
RX:THR_EventNwkCommissioning.Indication -> Providing the security material to the Joiner
RX:THR_EventNwkCommissioning.Indication -> Commissioner->Joiner Inbound Data Relayed
RX:THR_CommissioningDiagnostic.Indication -> IN JOIN_ENT_RSP
```

The Joiner is now connected to the Thread network and other devices can be added in the same manner.

7.2 CoAP applications

At this point, a variety of IPv6-enabled applications can be built to control remote Thread nodes seamlessly from the Linux OS. The subsequent sections describe the Thread stack support for CoAP datagrams and it uses the border router in TUN mode, i.e. with ND turned off. The first example uses a CoAP POST message to change the color of the RGB LED on a compatible Kinetis-W Freedom board, while the second uses a CoAP GET message to get the temperature from supporting devices

Note that for the below examples to work, two boards are needed, such that:

- The Linux OS-attached Thread Border Router runs a thread_host_controlled_device image with the macro THR_SERIAL_TUN_ROUTER enabled.
- The Thread Router runs a thread_router_eligible_device image. Use a supporting Freedom board for the first example to see the color of the RGB led change each second.

Host software prerequisites: `pip install txThings` --upgrade

`rgb_led.py` and `temp.py` scripts are in the package com.nxp.wireless_connectivity.test in the hsdk-python/src folder.

7.2.1 LED

The following application shows how to use the CoAP to communicate with the Thread stack. This specific demo changes the LED color on a Thread Router, which is joined to a Thread Border Router attached to a Linux OS host. The host uses CoAP POST messages.

[Terminal 1]$ sudo ./bin/Thread_KW_Tun /dev/ttyACM0 threadtun0

Then, the led application, destination being the Unique Local Address of the joined Kinetis-W Freedom node:

[Terminal 2]$ python rgb_led.py fd01::3ead:7c16:37b5:e6ef:701a

At this point, transmit and receive packets appear in Thread_KW_Tun's output and the the node's LED changes colors randomly each second.

7.2.2 Temperature

The above example handles the transmission of CoAP POST datagrams from the Linux OS system to a Thread remote router. CoAP GET messages are also supported. This example finds the host acquiring temperature information from the Thread Router.

First, start the TUN interface:

```
[Terminal 1]$ sudo ./bin/Thread_KW_Tun /dev/ttyACM0 threadtun0
```

Then, the temperature application, with the same parameter as above:

```
[Terminal 2]$ python temp.py fd01::3ead:7c16:37b5:e6ef:701a
[…]

[timestamp] [-] Result: Temp:28.56

[…]
```

This reports the obtained temperature value each second.

Chapter 8
Thread Integration with Linux OS Host on RNDIS Interface

NOTE

THCI over RNDIS may be used only in the configuration border_router. To enable this feature, set the THREAD_USE_THCI, gFsciIncluded_c and THCI_USBENET_ENABLE macro definitions to 1 and THREAD_USE_SHELL macro definition to 0 in middleware\wireless\nwk_ip_<ver>\examples\border_router \config\config.h. Modify/place the definitions in the correct conditional group for the tested board.

The Kinetis Thread Stack introduces the `border_router` example which acts as a ND_ROUTER at the IPv6 level on the emulated Ethernet over USB link, supporting THCI management commands over RNDIS. The Remote Network Driver Interface Specification is a protocol used mostly on top of USB which provides a virtual Ethernet link to the operating system. In Linux OS, the handling of these links is assigned to the `rndis_host` driver. Following the connection of the RNDIS interface, the Linux OS kernel logs describe the creation of a new network interface:

```
[692568.101376] usb 1-1.4: new full-speed USB device number 44 using ehci-pci
[692568.196601] usb 1-1.4: New USB device found, idVendor=1fc9, idProduct=0301
[692568.196606] usb 1-1.4: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[692568.196609] usb 1-1.4: Product: KINETIS REMOTE NDIS
[692568.196612] usb 1-1.4: Manufacturer: NXP SEMICONDUCTORS
[692568.196614] usb 1-1.4: SerialNumber: 00000001
[692568.199363] rndis_host 1-1.4:1.0 eth1: register 'rndis_host' at usb-0000:00:1a.0-1.4, RNDIS
device, 00:60:37:44:4d:04
[692568.213493] systemd-udevd[21269]: renamed network interface eth1 to eth6
```

Inspecting the newly created interface:

```
$ ip address show dev eth6
eth3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN group default qlen
1000
    link/ether 00:60:37:44:4d:04 brd ff:ff:ff:ff:ff:ff
    inet6 fd01::420a:d72:3ba6:5273:a05c/64 scope global temporary dynamic
       valid_lft 604478sec preferred_lft 85478sec
    inet6 fd01::420a:260:37ff:fe44:4d04/64 scope global dynamic
       valid_lft forever preferred_lft forever
    inet6 fe80::260:37ff:fe44:4d04/64 scope link
       valid_lft forever preferred_lft forever
```

The interface is auto-configured with its IPv6 addresses because of enabling ND_ROUTER nature of the Thread device. From this point on, the IPv6-based operations are like others performed on IP media interfaces such as Ethernet. However, a different type of physical layer needs to be implemented to maintain compatibility with the previously described FSCI layer.

The Host SDK implements two physical layers for transporting THCI: UART for direct UART, USB CDC, and PCAP for RNDIS. The UART layer handles sending and receiving THCI frames over a serial interface. This layer was the underlying basis for all previously discussed examples, but it is not suitable for networking operations mainly because it does not provide support for Ethernet headers and network packet injection.

The implementation using the PCAP layer attempts to maintain the compatibility with the upper THCI framing module, while being capable of network interface binding, packet injection and Ethernet header manipulation. To achieve these, the well-known third party library libpcap is used. Through *libpcap*, the user can send THCI frames that get delivered to the board on a special EtherType value: 0x88B5. This Ethertype value is available for public use for prototype and vendor-specific protocol development, as defined in Amendment 802a to IEEE Std 802.

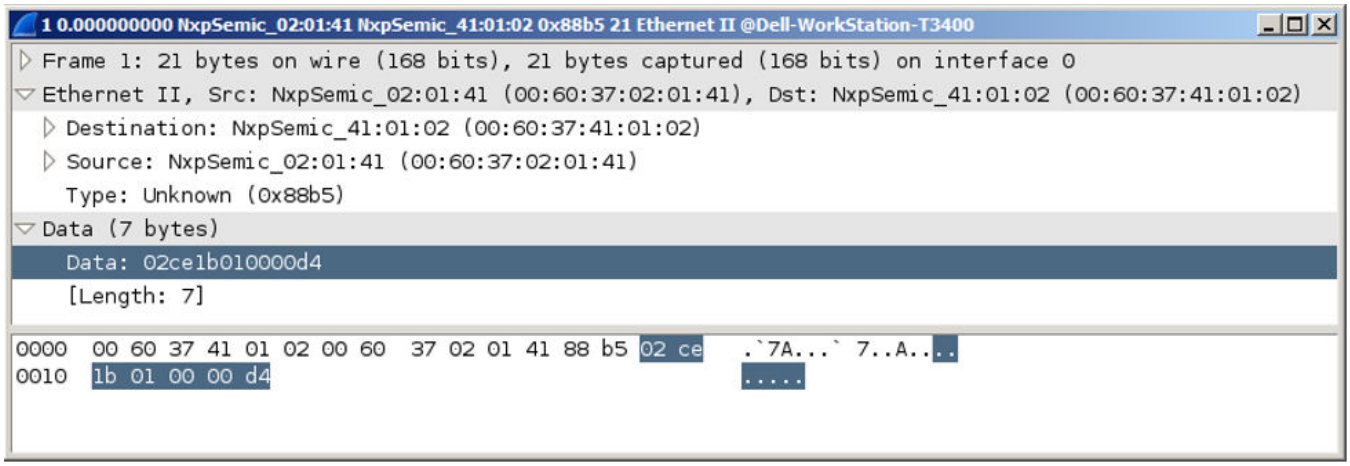

Figure 5. Host injects a packet over RNDIS. Data represents a Thread Create Network Request frame.

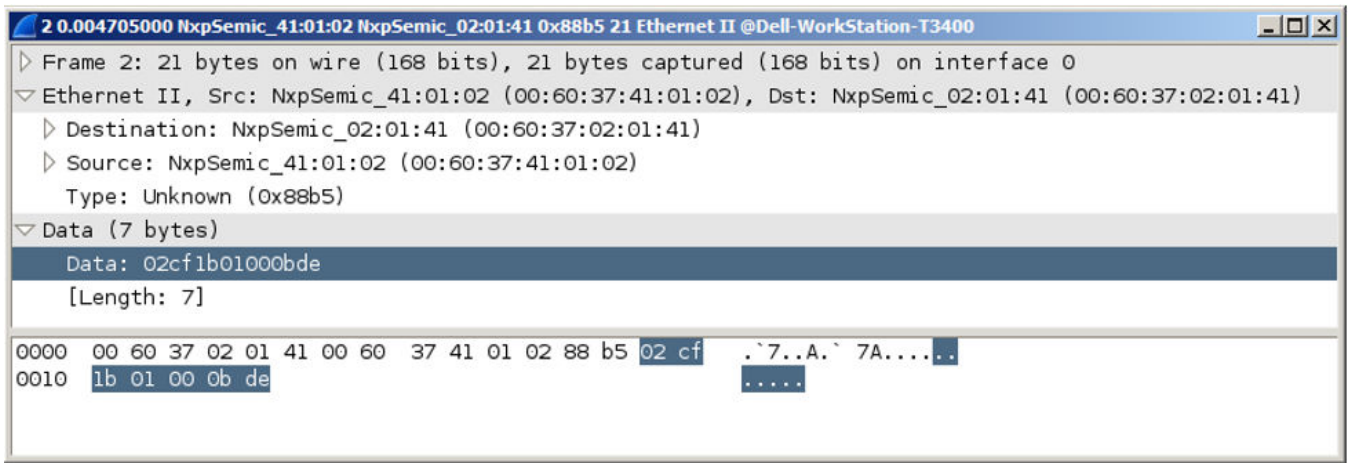

Figure 6. Board replies. Data represents a Thread Create Network Confirmation frame.

Sample code is provided in `hsdk/demo/PCAPTest.c`.

Chapter 9
Host API C Bindings

Starting with version 1.8.0, the Host SDK includes a set of C bindings to interface with a Kinetis-W black-box. Bindings are generated from the matching FSCI XML file that is available in the stack software package under tools\wireless\xml_fsci. The bindings are designed to be platform agnostic, with a minimal set of OS abstraction symbols required for building. Thus, the files can be easily integrated on a wide range of host platforms.

9.1 Directory Tree

```
 hsdk-c/
├── demo
│   ├── HeartRateSensor.c     Source file that implements the Bluetooth LE Heart Sensor Profile
│   └── Makefile
├── inc
│   ├── ble_sig_defines.h     Standard Bluetooth SIG UUID values
│   ├── cmd_<name>.h       Generated from the matching FSCI <name>.xml file.
│   └── os_abstraction.h     Provides OS dependent symbols for building the interface.
├── README.md
└── src
    ├── cmd_<name>.c             Generated from the matching FSCI <name>.xml file.
    ├── evt_<name>.c             Generated from the matching FSCI name.xml file.
    └── evt_printer_<name>.c     Generated from the matching FSCI <name>.xml file.
```

9.2 Tests and examples

Tests and examples that make use of the C bindings are placed in the hsdk-c/demo directory. Example of usage:

```
$ cd hsdk-c/demo/
$ make
$ ./HeartRateSensor /dev/ttyACM0
[...]
-->  Setup finished, please open IoT Toolbox -> Heart Rate -> HSDK_HRS
```

9.3 Development

Header file cmd_<name>.h is generated from the corresponding <name>.xml FSCI XML file.

- Enumerations

```
/* Indicates whether the connection request is issued for a specific device or for all the devices in
the White List - default specific device */
typedef enum GAPConnectRequest_FilterPolicy_tag {
    GAPConnectRequest_FilterPolicy_gUseDeviceAddress_c = 0x00,
```

```
    GAPConnectRequest_FilterPolicy_gUseWhiteList_c = 0x01
} GAPConnectRequest_FilterPolicy_t;
```

- Structures

```
typedef PACKED_STRUCT GAPConnectRequest_tag {
    uint16_t ScanInterval;  // Scanning interval - default 10ms
    uint16_t ScanWindow;  // Scanning window - default 10ms
    GAPConnectRequest_FilterPolicy_t FilterPolicy;  // Indicates whether the connection request is
issued for a specific device or for all the devices in the White List - default specific device
    GAPConnectRequest_OwnAddressType_t OwnAddressType;  // Indicates whether the address used in
connection requests will be the public address or the random address - default public address
    GAPConnectRequest_PeerAddressType_t PeerAddressType;  // When connecting to a specific device,
this indicates that device's address type - default public address
    uint8_t PeerAddress[6];  // When connecting to a specific device, this indicates that device's
address
    uint16_t ConnIntervalMin;  // The minimum desired connection interval - default 100ms
    uint16_t ConnIntervalMax;  // The maximum desired connection interval - default 200ms
    uint16_t ConnLatency;  // The desired connection latency (the maximum number of consecutive
connection events the Slave is allowed to ignore) - default 0
    uint16_t SupervisionTimeout;  // The maximum time interval between consecutive over-the-air
packets; if this timer expires, the connection is dropped - default 10s
    uint16_t ConnEventLengthMin;  // The minimum desired connection event length - default 0ms
    uint16_t ConnEventLengthMax;  // The maximum desired connection event length - default maximum
possible, ~41 s
    bool_t usePeerIdentityAddress;  // TRUE if the address defined in the peerAddressType and
peerAddress is an identity address
} GAPConnectRequest_t;
```

- Container for all possible event types

```
typedef struct bleEvtContainer_tag
{
    uint16_t id;
    union {
        [...]
        GAPConnectionEventConnectedIndication_t GAPConnectionEventConnectedIndication;
        [...]
    } Data
} bleEvtContainer_t;
```

- Prototypes for sending commands

```
    memStatus_t GAPConnectRequest(GAPConnectRequest_t *req, void *arg, uint8_t fsciInterface);
```

Header file os_abstraction.h provides the required symbols for building the generated interface. When integrating in a project different that Host SDK, the user needs to provide the implementation for

```
            void FSCI_transmitPayload(void *arg,            /* Optional argument passed to the
function */
                          uint8_t og,                       /* FSCI operation group */
                          uint8_t oc,                       /* FSCI operation code */
                          void *msg,                        /* Pointer to payload */
                          uint16_t msgLen,                  /* Payload length */
                          uint8_t fsciInterface             /* FSCI interface ID */
                         );
```

that creates and sends a FSCI packet (0x02 | og | oc | msgLen | msg | crc +- fsciInterface) on the serial interface. Source files cmd_<name>.c, evt_<name>.c and evt_printer_<name>.c are generated from the correspondent <NAME>.xml FSCI XML file.

- Functions that handle command serialization in cmd_<name>.c

```
memStatus_t GAPConnectRequest(GAPConnectRequest_t *req, void *arg, uint8_t fsciInterface)
{
    /* Sanity check */
    if (!req)
    {
        return MEM_UNKNOWN_ERROR_c;
    }

    FSCI_transmitPayload(arg, 0x48, 0x1C, req, sizeof(GAPConnectRequest_t), fsciInterface);
    return MEM_SUCCESS_c;
}
```

- Event dispatcher in evt_<name>.c

```
void KHC_BLE_RX_MsgHandler(void *pData, void *param, uint8_t fsciInterface)
{
    if (!pData || !param)
    {
        return;
    }

    fsciPacket_t *frame = (fsciPacket_t *)pData;
    bleEvtContainer_t *container = (bleEvtContainer_t *)param;
    uint8_t og = frame->opGroup;
    uint8_t oc = frame->opCode;
    uint8_t *pPayload = frame->data;
    uint16_t id = (og << 8) + oc, i;

    for (i = 0; i < sizeof(evtHandlerTbl) / sizeof(evtHandlerTbl[0]); i++)
    {
        if (evtHandlerTbl[i].id == id)
        {
            evtHandlerTbl[i].handlerFunc(container, pPayload);
            break;
        }
    }
}
```

- Handler functions to perform event de-serialization in evt_<name>.c

```
static memStatus_t Load_GAPConnectionEventConnectedIndication(bleEvtContainer_t *container, uint8_t
*pPayload)
{
    GAPConnectionEventConnectedIndication_t *evt = &(container-
>Data.GAPConnectionEventConnectedIndication);

    uint32_t idx = 0;

    /* Store (OG, OC) in ID */
    container->id = 0x489D;

    evt->DeviceId = pPayload[idx]; idx++;
    FLib_MemCpy(&(evt->ConnectionParameters.ConnInterval), pPayload + idx, sizeof(evt-
>ConnectionParameters.ConnInterval)); idx += sizeof(evt->ConnectionParameters.ConnInterval);
    FLib_MemCpy(&(evt->ConnectionParameters.ConnLatency), pPayload + idx, sizeof(evt-
```

```
>ConnectionParameters.ConnLatency)); idx += sizeof(evt->ConnectionParameters.ConnLatency);
    FLib_MemCpy(&(evt->ConnectionParameters.SupervisionTimeout), pPayload + idx, sizeof(evt-
>ConnectionParameters.SupervisionTimeout)); idx += sizeof(evt-
>ConnectionParameters.SupervisionTimeout);
    evt->ConnectionParameters.MasterClockAccuracy =
(GAPConnectionEventConnectedIndication_ConnectionParameters_MasterClockAccuracy_t)pPayload[idx]; idx+
+;
    evt->PeerAddressType = (GAPConnectionEventConnectedIndication_PeerAddressType_t)pPayload[idx]; idx
++;
    FLib_MemCpy(evt->PeerAddress, pPayload + idx, 6); idx += 6;
    evt->peerRpaResolved = (bool_t)pPayload[idx]; idx++;
    evt->localRpaUsed = (bool_t)pPayload[idx]; idx++;

    return MEM_SUCCESS_c;
}
```

- Event status console printer in evt_printer_<name>.c

```
void SHELL_BleEventNotify(void *param)
{
    bleEvtContainer_t *container = (bleEvtContainer_t *)param;

    switch (container->id) {
        [...]
        case 0x489D:
            shell_write("GAPConnectionEventConnectedIndication");
            shell_write(" -> ");
            switch (container->Data.GAPConnectionEventConnectedIndication.PeerAddressType)
            {
                case GAPConnectionEventConnectedIndication_PeerAddressType_gPublic_c:
                    shell_write(gPublic_c);
                    break;
                case GAPConnectionEventConnectedIndication_PeerAddressType_gRandom_c:
                    shell_write(gRandom_c);
                    break;
                default:
                    shell_printf("Unrecognized status 0x%02X", container-
>Data.GAPConnectionEventConnectedIndication.PeerAddressType);
                    break;
            }
            break;
        [...]
    }
```

Chapter 10
Host API Python Bindings

10.1 Prerequisites

Python 2.7.x is necessary to run the Python bindings. If Python 3.x is needed, the 2to3 code translator can be used, yet the user is requested to fix the possible remaining issues from the translation.

The bindings use the Host API C libraries. On Linux and OS X operating systems, these are called from the installation location which is /usr/local/lib, while on Windows OS the library file is loaded in <Python Install Directory>\DLLs.

10.2 Platform setup

To run scripts from the command line, the PYTHONPATH must be set accordingly, so that the interpreter can find the imported modules.

10.2.1 Linux OS

Adding the source folder to the PYTHONPATH can be done by editing ~/.bashrc and adding the following line:

```
export PYTHONPATH=$PYTHONPATH:/home/.../hsdk-python/src
```

Most of the Python scripts operate on boards connected on a serial bus and superuser privileges must be employed to access the ports. After running a command prefixed with sudo, the environment paths become those of root, so the locally set PYTHONPATH is not visible anymore. That is why /etc/sudoers is modified to keep the environment variable when changing user.

Edit /etc/sudoers with your favorite text editor. Modify:

```
Defaults env_reset -> Defaults env_keep="PYTHONPATH"
```

As an alternative to avoid modifying the *:sudoers* file, the PYTHONPATH can be adjusted programmatically, as in the example below:

```
import sys
if sys.platform.startswith('linux'):
    sys.path.append('/home/user/hsdk-python/src')
```

10.2.2 Windows OS

Add the source folder to the PYTHONPATH by following these steps:

1. Navigate to My Computer > Properties > Advanced System Settings > Environment Variables > System Variables.
2. Modify existing or create new variable named PYTHONPATH, with the absolute path of `tools\wireless\host_sdk\hsdk-python`.

10.3 Directory Tree

```
lib/                                          Compiled host SDK libraries for Windows.
├── README.md
├── x64
│   └── HSDK.dll
└── x86
    └── HSDK.dll

src/
├── com
│   ├── nxp
│   │   ├── wireless_connectivity
│   │   │   ├── commands
│   │   │   │   ├── ble                      Generated files for Bluetooth LE support.
│   │   │   │   │   ├── ble_sig_defines.py
│   │   │   │   │   ├── enums.py
│   │   │   │   │   ├── events.py
│   │   │   │   │   ├── frames.py
│   │   │   │   │   ├── gatt_database_dynamic.py
│   │   │   │   │   ├── heart_rate_interface.py
│   │   │   │   │   ├── __init__.py
│   │   │   │   │   ├── operations.py
│   │   │   │   │   ├── spec.py
│   │   │   │   │   └── sync_requests.py
│   │   │   │   ├── comm.py
│   │   │   │   ├── firmware                 Generated files for OTA/FSCI bootloader support.
│   │   │   │   │   ├── enums.py
│   │   │   │   │   ├── events.py
│   │   │   │   │   ├── frames.py
│   │   │   │   │   ├── __init__.py
│   │   │   │   │   ├── operations.py
│   │   │   │   │   ├── spec.py
│   │   │   │   │   └── sync_requests.py
│   │   │   │   ├── fsci_data_packet.py
│   │   │   │   ├── fsci_frame_description.py
│   │   │   │   ├── fsci_operation.py
│   │   │   │   ├── fsci_parameter.py
│   │   │   │   ├── __init__.py
│   │   │   │   └── thread/zigbee           Generated files for THCI/ZigBee 3.0 support.
│   │   │   │       ├── enums.py
│   │   │   │       ├── events.py
│   │   │   │       ├── frames.py
│   │   │   │       ├── __init__.py
│   │   │   │       ├── operations.py
│   │   │   │       ├── spec.py
│   │   │   │       └── sync_requests.py
│   │   │   ├── hsdk
│   │   │   │   ├── CFsciLibrary.py
│   │   │   │   ├── config.py                Configuration file for the Python Host SDK subsystem.
│   │   │   │   ├── CUartLibrary.py
│   │   │   │   ├── device
│   │   │   │   │   ├── device_manager.py
│   │   │   │   │   ├── __init__.py
│   │   │   │   │   └── physical_device.py
│   │   │   │   ├── framing
```

```
|   |   |   |   |   ├── fsci_command.py
|   |   |   |   |   ├── fsci_framer.py
|   |   |   |   |   └── __init__.py
|   |   |   |   ├── __init__.py
|   |   |   |   ├── library_loader.py
|   |   |   |   ├── ota_server.py
|   |   |   |   ├── singleton.py
|   |   |   |   ├── sniffer.py
|   |   |   |   └── utils.py
|   |   |   ├── __init__.py
|   |   |   └── test                          Test and proof of concept scripts.
|   |   |       ├── bootloader
|   |   |       |   ├── fsci_bootloader.py    Details How to Reprogram a Device Using the FSCI
Bootloader on page 44.
|   |   |       |   └── __init__.py
|   |   |       ├── border_router.py
|   |   |       ├── coap_thci.py               Demonstrates CoAP control via THCI.
|   |   |       ├── commissioning_nfc.py       Demonstrates Commissioning Credential Exchange via
NFC. To be used in collaboration with the Jabil i.MX 6 UltraLite-based gateway.
|   |   |       ├── commissioning.py
|   |   |       ├── get_neigh_ula.py           Presents two ways of obtaining a neighbor's IPv6 ULA
address.
|   |   |       ├── getaddr.py
|   |   |       ├── hrs.py                Script implementing a Bluetooth LE Heart Sensor profile -
How to Control Both the Bluetooth LE and Thread Stacks on page 43.
|   |   |       ├── __init__.py
|   |   |       ├── macfiltering.py
|   |   |       ├── multimode.py            How to Control Both the Bluetooth LE and Thread Stacks on
page 43.
|   |   |       ├── ota
|   |   |       |   ├── __init__.py
|   |   |       |   ├── README.txt
|   |   |       |   ├── test_dongle_mode_coap.py
|   |   |       |   ├── test_dongle_mode.py
|   |   |       |   ├── test_standalone_mode.py
|   |   |       |   └── VTun.py
|   |   |       ├── rgb_led.py
|   |   |       └── temp.py
|   |   |       └── zb_simple_nwk.py    Simple script demonstrating formation of a ZigBee 3.0 network.
|   |   └── __init__.py
|   └── __init__.py
└── __init__.py
```

10.4 Tests and examples

Tests and examples are placed in the package com. nxp.wireless_connectivity.test.

Example of usage:

- Linux

```
$ cd hsdk-python/src/com/nxp/wireless_connectivity/test/
$ sudo python commissioning.py <LEADER_TTY> <JOINER_TTY>
$ sudo python commissioning.py /dev/ttyACM0 /dev/ttyACM1
```

Python support for THCI over RNDIS is in the early experimental phase and needs extensive testing. Theoretically, the user can perform the same operations as with a serial ttyACM device if the network interface name, created by the rndis_host driver, starts with “eth”. The subsequent examples focus on the serial connection. However, note that the only modification required when using an RNDIS-enabled board is to change the name of the device from /dev/ttyACMX to ethX.

- Windows OS

```
> cd hsdk-python\src\com\nxp\wireless_connectivity\test\
> python commissioning.py <LEADER_COM> <JOINER_COM>
> python commissioning.py COM3 COM4
```

10.5 Functional description

The interaction between Python and the C libraries is made by the ctypes module. Ctypes provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python. Because the use of shared libraries is a requirement, the LIB_OPTION variable must remain set on "dynamic" in hsdk/Makefile. Ctypes made into mainline Python starting with version 2.5.

The Python Bindings expose Thread and Bluetooth LE familiar API in the com.

nxp.wireless_connectivity.commands package. Such a package contains the following modules:

```
thread | bluetooth le | firmware
├── enums.py - Classes that resemble enums, constants are generated here.
├── events.py - Observer classes that can build an object from a byte array and deliver it to the user.
├── frames.py - Classes that map on the Thread THCI or Bluetooth LE/Firmware FSCI messages.
├── operations.py - Each Operation class encapsulates a request and one or multiple events that are to
be generated by the request.
├── spec.py - This file describes the name, size, order and relationship between the command parameters.
└── sync_requests.py - Each Synchronous Request is a method which sends a request and returns the
triggered event.
```

10.6 Development

10.6.1 Requests

Sending a request consists of three steps: opening a communication channel, customizing the request, and sending the bytes.

```
comm = Comm('/dev/ttyACM0')-Linux or comm = Comm('COM42')-Windows
request = SocketCreateRequest(
SocketDomain=SocketCreateRequestSocketDomain.AF_INET6,
SocketType=SocketCreateRequestSocketType.Datagram,
SocketProtocol=SocketCreateRequestSocketProtocol.UDP
)
comm.send(Spec().SocketCreateRequestFrame, request)
```

10.6.2 Events

To obtain the event triggered by the request, an observer and callback must be added to the program logic.

```
def callback(expectedEvent):
print 'Callback for ' + str(type(expectedFrame))
observer = SocketCreateConfirmObserver()
comm.fsciFramer.addObserver(observer, callback)
```

NOTE

1. If the callback argument is not present, by default the program outputs the received frame in the console.
2. The callback method **must** have a single parameter (expectedEvent in the example above) which is used to gain access to the event object.

10.6.3 Operations

An operation consists in sending a request and obtaining the events via observers, automatically.

```
request = SocketCreateRequest(
SocketDomain=SocketCreateRequestSocketDomain.AF_INET6,
SocketType=SocketCreateRequestSocketType.Datagram,
SocketProtocol=SocketCreateRequestSocketProtocol.UDP
)
operation = SocketCreateOperation('/dev/ttyACM0', request)
operation.begin()
```

This sends the request and prints the SocketCreateConfirm to the console. Adding a custom callback is easy:

```
operation = SocketCreateRequest('/dev/ttyACM0', request, [callback])
```

The third argument (callbacks) when defining an operation is expected to be a list. The reason is that a single request can trigger multiple events, let's assume a confirmation and an indication. When it is known that two or more events should occur (inspect self.observers of each class from operations.py for the specific events that are to occur), multiple callbacks **must** be added. If one event is not to be processed via a callback, *None* must be added, and the event gets printed to console. The order in which callbacks are entered is important, that is the first callback is executed by the first observer, and so on.

10.6.4 Synchronous requests

These methods greatly reduce the code needed for certain operations. For example, starting a Thread device resumes to:

```
confirm = THR_CreateNwk(device='/dev/ttyACM0', InstanceID=0)
```

This removes the need for adding a custom callback to obtain the triggered event, since it is already returned by the method.

10.7 Thread network start with custom parameters use case

The next example walks through a standard scenario, where a user configures a Thread device with various parameters (channel, extended address, and so on), and starts the Thread stack and creates the Thread network.

This chapter describes all Python classes involved in the process. The entire example may be found in Appendix A, along with other examples handling sockets and using 'ifconfig all' for retrieving the IPv6 addresses of a node interfaces.

First, users must connect to the physical board using serial bus port. It is assumed that the device OS path is known and hardcoded in the script or found through device discovery. Alternatively, it might be received as a command line argument.

Hardcoded in the script	Found through device discovery

`device = '/dev/ttyACM0'`	`device_manager = DeviceManager()`
	`device_manager.initDeviceList()`
	`# Operates on the first discovered device`
	`device = device_manager.getDevices()[0].name`

Next, Thread parameters can be set. The `THR_SetAttrRequest` Thread command maps the following Python object:

```
class THR_SetAttrRequest(object):

    '''
    Sets the value of a Thread attribute.
    '''

    def __init__(self, InstanceId=bytearray(1), AttributeId=bytearray(1), Index=bytearray(1),
AttrSize=bytearray(1), AttributeValue=[]):
    '''
    @param InstanceId:    1            The instance of the Thread stack.
    @param AttributeId:    1            The ID of the attribute.
    @param Index:        1            The index of the attribute, usually 0, except for tables.
    @param AttrSize:    1            The size of the attribute.
    @param AttributeValue:    AttrSize        The value of the attribute.
    '''
    self.InstanceId = InstanceId
    self.AttributeId = AttributeId
    self.Index = Index
    self.AttrSize = AttrSize
    # Array length depends on AttributeSize.
    self.AttributeValue = AttributeValue
```

A Thread frame maps on a simple Python object with just an initializer. The initializer, by default, maps each parameter to a `bytearray` of its length. This is both advisory if one class does not have documentation, the user knows the expected size and for error proofing, if the user does not fill in a field, it is zero-filled at the specified size and does not cause errrors when sending the package on the serial interface. When defining such an object, the parameters may take simple integer, boolean or even list values instead of byte arrays (the values are automatically converted to bytes when transmitted on the wire). The following example presents the setting of the IEEE 802.15.4 channel attribute. The request is defined as follows:

```
request = THR_SetAttrRequest(
    InstanceId=0,
    AttributeId=THR_SetAttrRequestAttributeId.Channel,
    Index=0,
    AttrSize=1,
    AttributeValue=26
)
```

After defining the request, one can send it and check that it has been processed by the Thread stack. For the latter, the confirmation message `THR_SetAttrConfirm` may be used.

```
class THR_SetAttrConfirm(object):

    '''
    Confirmation of the set attribute request.
```

```
    '''

    def __init__(self, Status=THR_SetAttrConfirmStatus.Success):
        '''
        @param Status: 1    Permitted values defined in THR_SetAttrConfirmStatus.
        '''
        self.Status = Status
```

where the `THR_SetAttrConfirmStatus` is a simple enum:

```
class THR_SetAttrConfirmStatus(GenericEnum):
    Success = 0x00
    Invalidinstance = 0x02
    Invalidparameter = 0x03
    Notpermitted = 0x04
    UnsupportedAttribute = 0x07
    EmptyEntry = 0x08
    InvalidValue = 0x09
```

Users should begin an operation that sends the request, adds an observer for the event and a callback that checks if the status is set to *OK*. The operation is defined as follows:

```
class THR_SetAttrOperation(FsciOperation):

    def subscribeToEvents(self):
        self.spec = Spec.THR_SetAttrRequestFrame
        self.observers = [THR_SetAttrConfirmObserver(
            THR_SetAttrConfirm), ]
        super(THR_SetAttrOperation, self).subscribeToEvents()
```

This operation inherits the FsciOperation class which handles all the backend operations. `FsciOperation` has the following initializer:

```
def __init__(self, deviceName, request=None, callbacks=[], protocol=Protocol.Thread,
baudrate=Baudrate.BR115200, sync_request=False)
```

The `THR_SetAttrOperation`introduces another two concepts: a Spec object and an Observer one. The Spec object specifies the order, size and any other dependencies of the parameters of the request. In this example, `Spec.THR_SetAttrRequestFrame` is initialized by this method:

```
    def InitTHR_SetAttrRequest(self):
        cmdParams = []
        InstanceId = FsciParameter("InstanceId", 1)
        cmdParams.append(InstanceId)
        AttributeId = FsciParameter("AttributeId", 1)
        cmdParams.append(AttributeId)
        Index = FsciParameter("Index", 1)
        cmdParams.append(Index)
        AttrSize = FsciParameter("AttrSize", 1)
        cmdParams.append(AttrSize)
        AttributeValue = FsciParameter("AttributeValue", 1, AttrSize)
        cmdParams.append(AttributeValue)
        return FsciFrameDescription(0xCE, 0x18, cmdParams)
```

0xCE and 0x18 represent the values of OPGROUP and OPCODE for this specific request. This method creates a list of parameters, each parameter being defined by a name and a size. The parameter order is ensured by the list type of cmdParams ; the order is important for creating the effective raw packet that is transmitted on the physical medium

Returning to the Operation, `THR_SetAttrOperation` (which is the operation observer object) is the entity responsible for reconstructing an object from the byte array event received. It is defined as:

```
class THR_SetAttrConfirmObserver(Observer):

    opGroup = Spec.THR_SetAttrConfirmFrame.opGroup
    opCode = Spec.THR_SetAttrConfirmFrame.opCode

    @overrides(Observer)
    def observeEvent(self, event, callback, sync_request):
        # Call super, print common information
        Observer.observeEvent(self, event, callback, sync_request)
        # Get payload
        fsciFrame = cast(event, POINTER(FsciFrame))
        data = cast(fsciFrame.contents.data, POINTER(fsciFrame.contents.length * c_uint8))
        packet = Spec.THR_SetAttrConfirmFrame.getFsciPacketFromByteArray(
            data.contents, fsciFrame.contents.length)
        # Create frame object
        frame = THR_SetAttrConfirm()
        frame.Status = THR_SetAttrConfirmStatus.getEnumString(
            packet.getParamValueAsNumber("Status"))
        event_queue.put(frame) if sync_request else None

        if callback is not None:
            callback(frame)
        else:
            print_event(frame)
        fsciLibrary.DestroyFSCIFrame(event)
```

Among some ctypes operations to handle pointers, the bolded lines provide the essential functionality. A confirm frame is defined and its status field populated with the received value. Afterwards, depending on whether a callback has been added, the data is printed or handled in the callback. The callback, for demonstration purposes, is a simple function that checks whether the Status is 0x00 – OK.

```
def callback(confirm):
    # Print the string description of Status. Useful for debugging.
    print 'Status is ' + confirm.Status
    assert confirm.Status == ‘OK’, 'Something went wrong!'
```

Returning to the main script which sets the IEEE 802.15.4 channel and then starts the network:

```
[imports]

def callback(confirm):
    assert confirm.Status == ‘OK’, 'Something went wrong!'

if __name__ == '__main__':
device = ‘/dev/ttyACM0’
# Configure the MAC channel
request = THR_SetAttrRequest(
            InstanceId=0,
            AttributeId=THR_SetAttrRequestAttributeId.Channel,
            Index=0,
            AttrSize=1,
            AttributeValue=26
)
THR_SetAttrOperation(device, request, [callback]).begin()
# Start the network
```

```
request = THR_CreateNwkRequest(InstanceId=0)
THR_CreateNwk(device, request).begin()
time.sleep(0.1) # so the program won't exit before the callbacks get executed
```

For the network create command, a callback is not added and, eventually, the event data is printed to the console.

The purpose of this example was to show all internals involved in the process of sending and receiving data from a Thread device. However, using the Synchronous Requests module, the code size can be reduced:

```
if __name__ == '__main__':
device = '/dev/ttyACM0'
# Configure the MAC channel and start the Thread stack
confirm = THR_SetAttr(
    device=device,
            InstanceId=0,
            AttributeId=THR_SetAttrRequestAttributeId.Channel,
            Index=0,
            AttrSize=1,
            AttributeValue=26
)
    assert confirm.Status == 'Success'
# Start the network
confirm = THR_CreateNwk(device, InstanceID=0)
assert confirm.Status == 'OK'
```

Synchronous requests eliminate the need for custom callbacks to be added, and with a single line of code, the user can define, send the request, and obtain the event.

10.8 Bluetooth LE Heart Rate Service use case

The Heart Rate Service is presented as use case for using the API of a Bluetooth LE black box, located in the example hsdk-python/src/com/nxp/wireless_connectivity/test/hrs.py.

The example populates the GATT Database dynamically with the GATT, GAP, heart rate, battery and device information services. It then configures the Bluetooth LE stack and starts advertising. There are also two connect and disconnect observers to handle specific events.

The user needs to connect to the Bluetooth LE or hybrid black box through a serial bus port that is passed as a command line argument, for example, 'dev/ttyACM0'.

```
# python hrs.py -h
usage: hrs.py [-h] [-p] serial_port

Bluetooth LE demo app which implements a ble_fsci_heart_rate_sensor.

positional arguments:
  serial_port  Kinetis-W system device node.

optional arguments:
  -h, --help   show this help message and exit
  -p, --pair   Use pairing
```

It is important to first execute a CPU reset request to the Bluetooth LE black box before performing any other configuration to reset the Buetooth LE stack. This is done by the following command:

```
FSCICPUReset(serial_port, protocol=Protocol.BLE)
```

10.8.1 User sync request example

It is recommended for the user to access the Bluetooth LE API through sync requests.

`GATTDBDynamicAddCharacteristicDeclarationAndValue` API is used as an example:

```
def gattdb_dynamic_add_cdv(self, char_uuid, char_prop, maxval_len, initval, val_perm):
        '''
        Declare a characteristic and assign it a value.

        @param char_uuid: UUID of the characteristic
        @param char_prop: properties of the characteristic
        @param maxval_len: maximum length of the value
        @param initval: initial value
        @param val_perm: access permissions on the value
        @return: handle of the characteristic
        '''
        ind = GATTDBDynamicAddCharacteristicDeclarationAndValue(
            self.serial_port,
            UuidType=UuidType.Uuid16Bits,
            Uuid=char_uuid,
            CharacteristicProperties=char_prop,
            MaxValueLength=maxval_len,
            InitialValueLength=len(initval),
            InitialValue=initval,
            ValueAccessPermissions=val_perm,
            protocol=self.protocol
        )
```

if `ind` is None:

```
            return self.gattdb_dynamic_add_cdv(char_uuid, char_prop, maxval_len, initval, val_perm)

        print '\tCharacteristic Handle for UUID 0x%04X ->' % char_uuid, ind.CharacteristicHandle
        self.handles[char_uuid] = ind.CharacteristicHandle
        return ind.CharacteristicHandle
```

10.8.2 Sync request internal implementation

As an example, for the GATTDBDynamicAddCharacteristicDeclarationAndValue API, the command is executed through a synchronous request. The sync request code creates an object of the following class:

```
class GATTDBDynamicAddCharacteristicDeclarationAndValueRequest(object):

    def __init__(self,
UuidType=GATTDBDynamicAddCharacteristicDeclarationAndValueRequestUuidType.Uuid16Bits, Uuid=[],
CharacteristicProperties=GATTDBDynamicAddCharacteristicDeclarationAndValueRequestCharacteristicPropert
ies.gNone_c, MaxValueLength=bytearray(2), InitialValueLength=bytearray(2), InitialValue=[],
ValueAccessPermissions=GATTDBDynamicAddCharacteristicDeclarationAndValueRequestValueAccessPermissions.
gPermissionNone_c):
        '''
        @param UuidType: UUID type
        @param Uuid: UUID value
        @param CharacteristicProperties: Characteristic properties
        @param MaxValueLength: If the Characteristic Value length is variable, this is the maximum
length; for fixed lengths this must be set to 0
        @param InitialValueLength: Value length at initialization; remains fixed if maxValueLength is
set to 0, otherwise cannot be greater than maxValueLength
```

```
        @param InitialValue: Contains the initial value of the Characteristic
        @param ValueAccessPermissions: Access permissions for the value attribute
        '''
        self.UuidType = UuidType
        self.Uuid = Uuid
        self.CharacteristicProperties = CharacteristicProperties
        self.MaxValueLength = MaxValueLength
        self.InitialValueLength = InitialValueLength
        self.InitialValue = InitialValue
        self.ValueAccessPermissions = ValueAccessPermissions
```

An operation is represented by an object of the following class:

```
class GATTDBDynamicAddCharacteristicDescriptorOperation(FsciOperation):

    def subscribeToEvents(self):
        self.spec = Spec.GATTDBDynamicAddCharacteristicDescriptorRequestFrame
        self.observers = [GATTDBDynamicAddCharacteristicDescriptorIndicationObserver(
'GATTDBDynamicAddCharacteristicDescriptorIndication'), ]
        super(GATTDBDynamicAddCharacteristicDescriptorOperation, self).subscribeToEvents()
```

The Spec object is initialized and set to zero in the FSCI packet any parameter not passed through the object of a class, depending on its length. Also, when defining such an object, the parameters may take simple integer, boolean or even list values instead of byte arrays, the values are serialized as a byte stream.

The observer is an object of the following class:

```
class GATTDBDynamicAddCharacteristicDeclarationAndValueIndicationObserver(Observer):
    opGroup = Spec.GATTDBDynamicAddCharacteristicDeclarationAndValueIndicationFrame.opGroup
    opCode = Spec.GATTDBDynamicAddCharacteristicDeclarationAndValueIndicationFrame.opCode

    @overrides(Observer)
    def observeEvent(self, framer, event, callback, sync_request):
        # Call super, print common information
        Observer.observeEvent(self, framer, event, callback, sync_request)
        # Get payload
        fsciFrame = cast(event, POINTER(FsciFrame))
        data = cast(fsciFrame.contents.data, POINTER(fsciFrame.contents.length * c_uint8))
        packet = Spec.GATTDBDynamicAddCharacteristicDeclarationAndValueIndicationFrame.
getFsciPacketFromByteArray(data.contents, fsciFrame.contents.length)
        # Create frame object
        frame = GATTDBDynamicAddCharacteristicDeclarationAndValueIndication()
        frame.CharacteristicHandle = packet.getParamValueAsNumber("CharacteristicHandle")
        framer.event_queue.put(frame) if sync_request else None

        if callback is not None:
            callback(frame)
        else:
            print_event(self.deviceName, frame)
        fsciLibrary.DestroyFSCIFrame(event)
```

The status of the request is printed at the console by the following general status handler:

```
def subscribe_to_async_ble_events_from(device, ack_policy=FsciAckPolicy.GLOBAL):
    ble_events = [
        L2CAPConfirmObserver('L2CAPConfirm'),
        GAPConfirmObserver('GAPConfirm'),
        GATTConfirmObserver('GATTConfirm'),
        GATTDBConfirmObserver('GATTDBConfirm'),
        GAPGenericEventInitializationCompleteIndicationObserver(
```

```
'GAPGenericEventInitializationCompleteIndication'),
        GAPAdvertisingEventCommandFailedIndicationObserver(
'GAPAdvertisingEventCommandFailedIndication'),
        GATTServerErrorIndicationObserver('GATTServerErrorIndication'),
        GATTServerCharacteristicCccdWrittenIndicationObserver(
'GATTServerCharacteristicCccdWrittenIndication')
    ]

    for ble_event in ble_events:
FsciFramer(device, FsciAckPolicy.GLOBAL, Protocol.BLE, Baudrate.BR115200).addObserver(ble_event)
```

10.8.3 Connect and disconnect observers

The following code adds observers for the connect and disconnect events in the user class:

```
class BLEDevice(object):
    '''
    Class which defines the actions performed on a generic Bluetooth lE device.
    Services implemented: GATT, GAP, Device Info.
    '''
        self.framer.addObserver(
            GAPConnectionEventConnectedIndicationObserver(
'GAPConnectionEventConnectedIndication'),
            self.cb_gap_conn_event_connected_cb)
        self.framer.addObserver(
            GAPConnectionEventDisconnectedIndicationObserver(
'GAPConnectionEventDisconnectedIndication'),
            self.cb_gap_conn_event_disconnected_cb)
```

where the callbacks are:

```
    def cb_gap_conn_event_connected_cb(self, event):
        '''
        Callback executed when a smartphone connects to this device.
        @param event: GAPConnectionEventConnectedIndication
        '''
        print_event(self.serial_port, event)
        self.client_device_id = event.DeviceId
        self.gap_event_connected.set()

    def cb_gap_conn_event_disconnected_cb(self, event):
        '''
        Callback executed when a smartphone disconnects from this device.
        @param event: GAPConnectionEventdisConnectedIndication
        '''
        print_event(self.serial_port, event)
        self.gap_event_connected.clear()
```

From an Android™ or iOS-based smartphone, the user can use the Kinetis Bluetooth LE Toolbox application in the Heart Rate profile. Random heart rate measurements in the range 60-100 are displayed every second, while battery values change every 10 seconds.

10.9 ZigBee 3.0 simplified use case

The current package includes a demonstrative ZigBee 3.0 script named ‘zb_simple_nwk.py’. In the context of the script, the behavior of a ZigBee 3.0 device is simplified to the following capabilities:

- Create a network on a given 802.15.4 channel and permit other devices to join

- Join a network on a given 802.15.4 channel

- Perform an Active Endpoint Request

To exercise these capabilities, the script requires two devices, one acting as Coordinator and another one acting as a Router.

> zb_simple_nwk.py <coord_serial_port> <router_serial_port> <802.15.4 channel>

Under normal operation, with COM7 – Coordinator and COM75 – Router, the output log is (important information emphasized):

Factory Reset

[Command 0] COM7: ErasePersistentData -> { }

[Event 1] COM7: Status -> { Status: Success }

[...]

[Event 30] COM7: FactoryNewRestart -> { Status: **STARTUP** }

[Command 31] COM75: ErasePersistentData -> { }

[Event 32] COM75: Status -> { Status: Success }

[...]

[Event 61] COM75: FactoryNewRestart -> { Status: **STARTUP** }

Set 802.15.4 Channel

[Command 62] COM7: SetChannelMask -> { ChannelMask: 0x02000000 }

[Event 63] COM7: Status -> { Status: Success }

[Command 64] COM75: SetChannelMask -> { ChannelMask: 0x02000000 }

[Event 65] COM75: Status -> { Status: Success }

Create Network and Permit Join for All Devices

[Command 66] COM7: StartNetworkMessage -> { }

[Event 67] COM7: Status -> { Status: **Success** }

[Event 68] COM7: NetworkJoinedFormed -> { Status: **FormedNewNetwork** , ShortAddr: 0x0 , ExtAddr: 0x7453a323ee3cabd6L , Channel: 0x19 }

[Command 69] COM7: PermitJoiningRequest -> { Interval: 0xff , TCSignificance: 0x0 , TargetShortAddress: ['0xFF', '0xFC'] }

[Event 70] COM7: Status -> { Status: Success }

Join Network

[Command 71] COM75: SetDeviceType -> { DeviceType: 0x1 }

[Event 72] COM75: Status -> { Status: Success }

[Command 73] COM75: StartNetworkScan -> { }

[Event 74] COM75: Status -> { Status: Success }

[Event 75] COM75: NetworkJoinedFormed -> { Status: **JoinedExistingNetwork** , **ShortAddr: 0xcee7** , ExtAddr: 0xcae790b994812b39L , Channel: 0x19 }

[Event 76] COM7: DeviceAnnounce -> { ShortAddress: 0xcee7 , IEEEAddress: 0xcae790b994812b39L , MACCapability: <com.nxp.wireless_connectivity.commands.zigbee.frames.MACCapability object at 0x023E4DB0> }

[Event 77] COM7: RouterDiscoveryConfirm -> { Status: 0x0 , NwkStatus: 0x0 }

Test Connectivity with an Active Endpoint Request

[Command 78] COM7: ActiveEndpointRequest -> { **TargetShortAddress: 0xcee7** }

[Event 79] COM7: Status -> { Status: Success }

[Event 80] COM7: **ActiveEndpointResponse** -> { Status: 0x0 , SequenceNumber: 0xbe , EndpointCount: 0x1 , Address: 0xcee7 , ActiveEndPointList: 0x1 }

Chapter 11
How to Control Both the Bluetooth LE and Thread Stacks

Project `ble_thread_host_controlled_device` may run both the Bluetooth LE and Thread stacks in parallel. Host SDK provides a script which controls both stacks in the following scenario:

1. Create a Thread network with Commissioning with two devices.
2. Start a continuous ping between them.
3. Start a Heart Rate Sensor profile on the Leader.

where steps 2 and 3 are executed in parallel, to demonstrate that both stacks can run normally at the same time.

Functionality is provided by the script: host_sdk/hsdk-python/src/com/nxp/wireless_connectivity/test/multimode.py.

```
$ python multimode.py
Usage: # python multimode.py <port thread_bluetooth_host_controlled_device_hybrid> <port
thread_host_controlled_device> <802.15.4 channel [11-25]>
```

The second argument is the port of the Thread Joiner, which may be any THCI-enabled router eligible device, while the third is the 802.15.4 channel for the Thread network to start on.

The Heart Rate Service is presented as use case for exercising the API of a Bluetooth LE black box, located in the example hsdk-python\src\com\nxp\wireless_connectivity\test\hrs.py. Functionality of hrs.py is fully integrated in multimode.py, so for one to understand the Bluetooth LE FSCI internals, hrs.py is be a good starting point. The previous section presents more information about the Heart Rate Sensor script.

In addition, the project ble_thread_router_wireless_uart allows the control of the Bluetooth LE and Thread stacks via a Bluetooth LE connection, using the Kinetis Bluetooth LE Toolbox Wireless UART application. From an Android or iOS-based smartphone, the user can connect to the multimode stack via an advertised Bluettoth LE connection and then, using the virtual hyperterminal exposed by the Thread stack, the user can send Thread management commands to manage the Thread network. More details, including application screenshots are presented in the *Kinetis Thread Stack Demo Applications User's Guide*.

Chapter 12
How to Reprogram a Device Using the FSCI Bootloader

For information on how to deploy a Thread image with FSCI bootloader support, see the document *Kinetis Thread Stack and FSCI Bootloader Quick Start Guide*. There are two options to deploy a Bluetooth LE application with FSCI bootloader support

1. Build the FSCI Bootloader application using an IDE from the projects located at `\middleware\wireless \framework_<ver>\Bootloader\bootloader_fsci\[platform]\build\` and flash it to the board using J-Link.
2. Copy the FSCI Bootloader binary located at `\middleware\wireless\framework_<ver>\Bootloader\Bin` to the J-Link mass storage device emulated via OpenSDA or flash it using a SEGGER J-Link Tool.

The Bluetooth LE application that is deployed via FSCI bootloader needs to be configured as a bootloader-compatible application. This is done by adding the gUseBootloaderLink_d=1 flag to the linker options of the application project and select the output of the build as binary. By default, the bootloader mode for a Bluetooth LE application is entered by connecting the board while holding the reset switch (this is not needed for a Thread application).

Host functionality is provided by the script: \tools\wireless\host_sdk/hsdk-python/src/com/nxp/wireless_connectivity/test/bootloader/fsci_bootloader.py providing as command line arguments the device serial port and a binary firmware file compatible with the bootloader.

```
$ python fsci_bootloader.py -h
usage: fsci_bootloader.py [-h] [-s CHUNK_SIZE] [-d] [-e]
                          serial_port binary_file
Script to flash a binary file using the FSCI bootloader.
positional arguments:
  serial_port           Kinetis-W system device node.
  binary_file           The binary file to be written.
optional arguments:
  -h, --help            show this help message and exit
  -s CHUNK_SIZE, --chunk-size CHUNK_SIZE
                        Push chunks this large (in bytes). Defaults to 2048.
  -d, --disable-crc     Disable the CRC check on commit image.
  -e, --erase-nvm       Erase the non-volatile memory.
```

For example,

```
export PYTHONPATH=$PYTHONPATH:<hsdk-path>/hsdk-python/src/
python fsci_bootloader.py /dev/ttyACM0 thread_host_controlled_device.bin -e
```

The script does the following:

- Sends the THCI command for the device to reset, then enter and remain in bootloader mode.
- Sends the command to cancel an image as a safety check and to verify the bootloader is responsive.
- Sends the command to start firmware update for a new image.
- Pushes chunks of the firmware images file sequentially until the full firmware is programmed and display intermediate progress as percent of binary file content loaded.
- Sets the flags to commit the image as valid.
- Resets the device, so it boots to the new firmware.

Chapter 13
Code Samples

13.1 Python code sample - Thread network creation and joining

```
import time

from com.nxp.wireless_connectivity.commands.thread.sync_requests import THR_CreateNwk, THR_Join

if __name__ == '__main__':
    leader_port = '/dev/ttyACM0'  # COM of the KW device designated as the initial Leader Router.
    joiner_port = '/dev/ttyACM1'  # COM of the KW device designated as a Joiner
   Router.
    # Using Synchronous Requests, Leader creates the Thread network.
    confirm_start = THR_CreateNwk(leader_port, InstanceID=0)
    assert confirm_start.Status == 'OK'

    time.sleep(2) # Allow Leader to advertise itself in the network

    # Followed by another device joining it.
    confirm_join = THR_Join(joiner_port, InstanceID=0)
    assert confirm_join.Status == 'OK'
```

13.2 Python code sample – Thread ifconfig

```
 [imports]

def my_callback(response):
    print 'Received the Ifconfig Response. We can now inspect the frame fields.'
    for interface in response.InterfaceList:
        print interface.__dict__

if __name__ == '__main__':
    com_port = '/dev/ttyACM0'  # COM of the KW device.
    # Using Synchronous Requests, Leader creates the Thread network.
    confirm_start = THR_CreateNwk(leader_port, InstanceID=0)
    assert confirm_start.Status == 'OK'

    # Create the ifconfig request. This one does not possess fields.
    request = NWKU_IfconfigAllRequest()
    # Define the operation. Begin() automatically sends the request and adds a
    # callback for execution upon the arrival of the Confirm or Response frame.
    # Multiple callbacks can be added.
    operation = NWKU_IfconfigAllOperation(com_port, request, [my_callback],
protocol=Protocol.Thread).begin()

--------------------------------------------------------------------------------
```

```
>>>execfile('my_script.py')
Received the Ifconfig All Response. We can now inspect the frame fields.
{'InterfaceID': 0, 'CountIpAddresses': 5, 'Addresses': ['fe80:0000:0000:0000:0c27:bc6f:152a:6cfd',
'fd4a:decc:39cb:0000:614a:a08d:21ae:aca9', 'fd4a:decc:39cb:0000:0000:00ff:fe00:0000',
'fd66:67b7:4126:00aa:0000:0000:0000:0000', 'fd4a:decc:39cb:0000:0000:00ff:fe00:fc02']}
{'InterfaceID': 1, 'CountIpAddresses': 2, 'Addresses': ['fe80:0000:0000:0000:0204:9fff:fe00:fa5c',
'fd66:67b7:4126:0000:0204:9fff:fe00:fa5c']}
Note: There are two interfaces displayed as the device has an additional TUN interface enabled.
```

13.3 Python code snippet – Thread socket creation

```
[imports]

def my_callback(confirm):
    print 'Received the Socket Create Confirm. We can now inspect the frame fields:',
    print 'SocketIndex', confirm.SocketIndex

if __name__ == '__main__':
    com_port = '/dev/ttyACM0'  # COM of the KW device
    # Create and tweak the request.
    request = SocketCreateRequest(
        SocketDomain=SocketCreateRequestSocketDomain.AF_INET6,
        SocketType=SocketCreateRequestSocketType.Datagram,
        SocketProtocol=SocketCreateRequestSocketProtocol.UDP
    )
    # Define the operation. This automatically sends the request and adds a callback
    # for execution upon the arrival of the Confirm or Response frame. Multiple
    # callbacks can be added.
    operation = SocketCreateOperation(com_port, request, [my_callback],
protocol=Protocol.Thread).begin()

--------------------------------------------------------------------------------
>>>execfile('my_script.py')
Received the Socket Create Confirm. Inspect the frame fields: SocketIndex 0
>>>execfile('my_script.py')
Received the Socket Create Confirm. Inspect the frame fields: SocketIndex 1
```

Chapter 14
Revision History

Table 5. Revision history on page 47 summarizes revisions to this document.

Table 5. Revision history

Revision number	Date	Substantive changes
0	08/2016	Initial release
1	09/2016	Updates for KW41 GA Release
2	12/2016	Updates for KW24 GA Release
3	03/2017	Updates for KW41 Maintenance Release
4	01/2018	Updates for KW41 Maintenance Release
5	03/2018	Updated ZigBee support
6	05/2018	Updates for Thread K32W042 Beta Release
7	04/2019	Updates for Thread KW41Z Maintenance Release 3

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer’s applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, C-5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, and UMEMS are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, µVision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© NXP B.V. 2016-2019. **All rights reserved.**

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 08 April 2019
Document identifier: KFSCIHAPIUG

arm