NXP Semiconductors
User's Guide

Document Number: KFSCIHAPIUG
Rev. 2, 12/2016

Kinetis FSCI Host Application Programming Interface

User's Guide

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 running on Kinetis KW2xD nodes that 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 'black box' firmware running on a Kinetis microcontroller.

Contents

2 Deploying Host Controlled Firmware

2.1 Thread Application Configurations

IAR Embedded Workbench for ARM® (EWARM) version 7.70 or later and Kinetis Design Studio version 3.2.0 or later are the development toolchains used to deploy the Thread stack software applications.

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

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

Table 1 Sample Application Configurations

Application	Default Capabilities
border_router	A template for all product categories 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** – section: *Thread Integration with Linux Host on Serial (UART) Tunnel Interface;* other sections that refer to THCI/FSCI.
2. **border_router** – section*: Thread Integration with Linux host on RNDIS Interface*.

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. The Thread stack makes use of an XML file that contains 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.

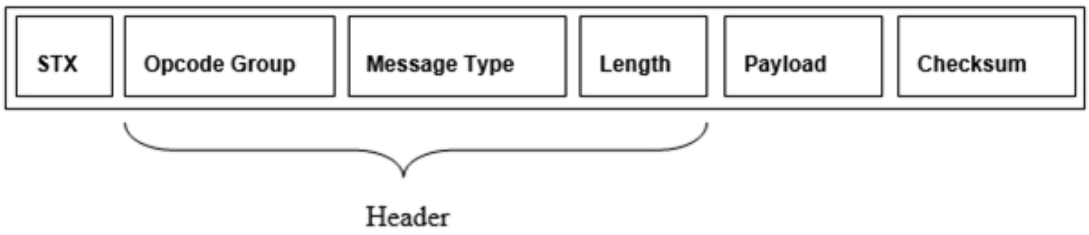


Figure 1. FSCI message format

Table 2 FSCI message format description

STX	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).
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 BLE 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 that 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 stack instance from a Linux OS system, with focus on Python language bindings.

3.1 Kinetis Wireless Host Software System Block Diagram

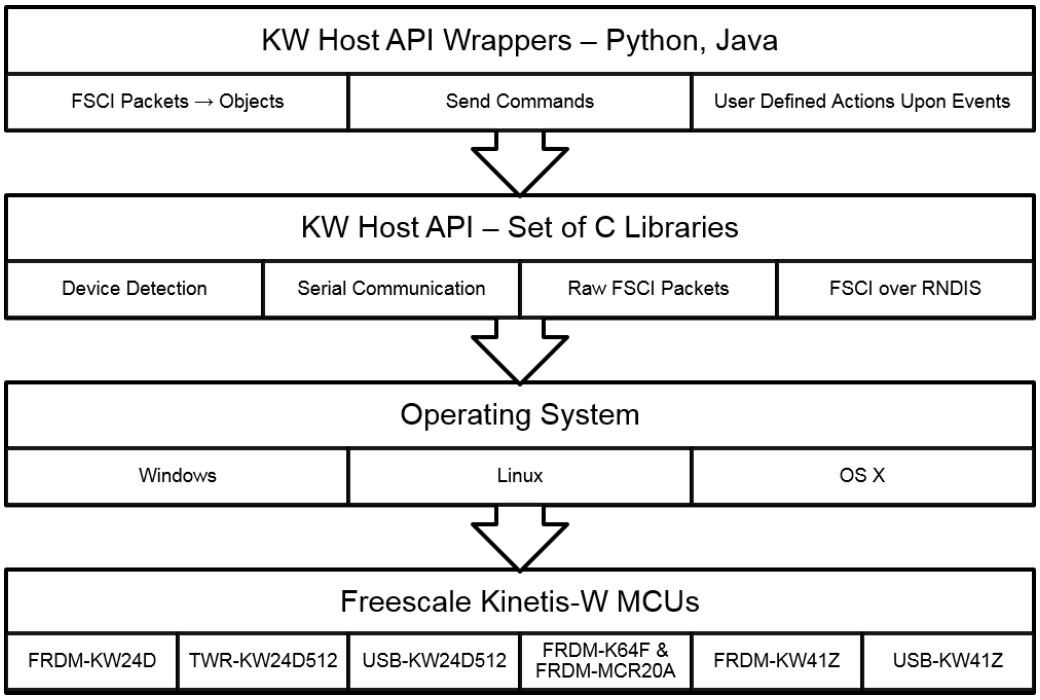


Figure 2. Kinetis Host Software System block diagram

3.2 Directory Tree

```
├── demo                                    A set of programs to demonstrate functionality.
│   ├── GetKinetisDevices.c                 Outputs all the 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.
│       ├── hsdkError.h        Macros for error reporting.
│       ├── hsdkLogger.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 that must be used as input strings for the Python scripts that 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)	1MHz
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 that 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
```

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 2.6.

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[1] 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
```

[1] http://www.freedesktop.org/wiki/Software/ModemManager/

```
$ 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 section *Thread Integration with Linux host on Serial (UART) Tunnel Interface*.
- **PCAPTest**: this program demonstrates the functionality of USB-KW24D512 running 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 Section 8.
- **SPITest**: this program demonstrates how to open and configure the speed of a device which exposes a SPI interface to the host processor.

5 Windows OS Host Software Installation Guide

5.1 Library

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 section 9 *Host API Python Bindings*.

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 THR_SERIAL_TUN_ROUTER is enabled in the project file:

middleware\wireless\nwk_ip_<version>\examples\host_controlled_device\config\config.h

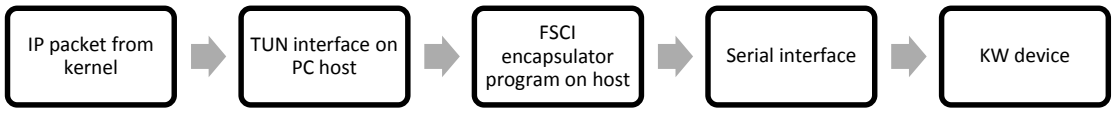
```
#define THR_SERIAL_TUN_ROUTER                    1
```

Note

Note that a Serial-TUN enabled device advertises itself as a default router for prefix FD01::/64 (first 64 bits of `THR_SERIAL_TUN_DEFAULT_ADDRESS`). When multiple Serial-TUN capable nodes coexist in the same Thread network, routing loops may occur if packets have either the source or destination address from this prefix range. To fix this, either change `THR_SERIAL_TUN_DEFAULT_ADDRESS` and ensure this is unique throughout your setup or have only one Serial-TUN capable device in the network.

To provide connectivity to the host, there are two 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.

Diagram, direction from host to serial Thread device:



Diagram, direction from Thread device to host:

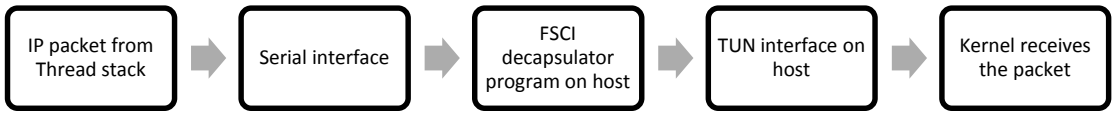


Figure 3. Host-to-serial and Thread device to host

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 that can operate with IP packets. TAP describes virtual layer 2 devices, which usually handle Ethernet frames. Although these types of virtual interfaces are common on most operating systems, various parameters differ. The following paragraphs describe the layer 3 component, TUN interfaces, from an operating system point of view.

On Linux OS, the kernel modules that handle TUN 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 interface open this file and any subsequent read or write is to be done with respect to this file. Creating a new virtual TUN interface and configuring it can be done with basic iproute2 commands:

```
$ sudo ip tuntap add mode tun fslthr0
$ ip link show fslthr0
5: fslthr0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN mode DEFAULT
qlen 500
    link/none
$ sudo ip -6 addr add FD01::2 dev fslthr0
$ ip address show fslthr0
5: fslthr0: <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 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 IPv6 packet and write it to the TUN file descriptor, reaching the kernel layer 3 IP stack.

6.2 Userspace Module

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.

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 interface, whether or not 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 `fslthr0` 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 interface and, in an infinite loop, starts reading 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 IPv6 remainder is injected back into the TUN interface to be handled by the kernel. Both the packets read from and injected back in the TUN interface are dumped to the console, having either the transmit tags (data read is transmitted further to the device) or receive tags respectively.

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`.

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 fslthr0
# Assign it a global IPv6 address.
ip -6 addr add FD01::2 dev fslthr0
# Add route to default address of Serial TUN embedded interface.
ip -6 route add FD01::1 dev fslthr0
# Add route to Unique Local /64 Prefix via fslthr0.
ip -6 route add FD01:0000:0000:3EAD::/64 dev fslthr0
# The interface is ready.
ip link set fslthr0 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 fslthr0
3: fslthr0: <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 fslthr0  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 fslthr0
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 is not re-enumerated to a different number upon reset (for example, /dev/ttyACM0 always re-enumerates to itself).

The console output may be switched off by setting the DEBUG macro on 0 in Thread_KW_Tun.c. Thread network creation and MeshCoP initialization may be switched off by toggling the macro-definition PROVISION. If a USB connection is used (i.e. USB-KW24 or the USB port of other form-factors) and the factory reset argument is provided, one needs to enable the macro-definition CLOSE_OPEN_PORT in order to reconnect to the device after MCU reset.

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 Chapter 8, “Mesh Commissioning Protocol”, from the Thread 1.0 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. NXP KW2xD – any form-factor – connected to a physical or virtual Linux OS machine that 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 TUN interface described throughout this document. The software artefacts for enabling commissioning on the Linux OS side are:

```
$ pwd
/home/user/hsdk-python/misc/commissioning/linux

$ ls
CC_Helper.py  make_tun.sh  relay.py SerialTUNCommissioner.py thread.service  VTun.py
```

7.1.1 Software description

Prerequisites:

- Avahi zeroconf implementation. Most modern Linux OS systems already have the Avahi suite installed. The user can check whether Avahi is installed by issuing `which avahi-daemon` and looking for output. If there is no output, Avahi can be deployed on Debian-based systems with: `apt-get install avahi-daemon avahi-discover libnss-mdns`
- Third party Python module pytun. Install with: `pip install python-pytun`
- `export PYTHONPATH=$PYTHONPATH:/home/.../hsdk-python/src`

For establishing a commissioning channel, a single script must be issued – relay.py.

```
$ ./relay.py -h
usage: relay.py [-h] [-r] [-6] [-c 802.15.4 RF channel]
                [serial_port] [nwk_interface]

Starts a Thread Border Router with Serial TUN capabilities, capable of
allowing external commissioners.

positional arguments:
  serial_port           Kinetis-W system device node.
  nwk_interface         The network interface used to communicate with the
```

```
                        Commissioner.

optional arguments:
  -h, --help            show this help message and exit
  -r, --reset           The device undergoes a factory reset at the
                        beginning of the script. If not, make sure the
                        device is in a factory reset state.
  -6, --ipv6            The communication between the Commissioner and the BR
                        is IPv6.
  -c 802.15.4 RF channel, --channel 802.15.4 RF channel
                        RF channel for the Thread network
$ sudo ./relay.py /dev/ttyACM0 eth0 --reset -c 15
[... output elided ...]
```

The main purpose of this script is to establish a bidirectional channel between a Commissioner and a Thread Network behind the Border Router. It achieves this goal by receiving datagrams on a “commissioner” socket and passing it to a “Thread” socket to send datagrams back and forth to the TUN interface. The script manages the following procedures:

- Sets up a new service for the avahi-daemon to advertise. The second parameter – interface to Commissioner – is used at this step for configuring the new service with the IPv6 address of the Border Router. More on this topic in Section 7.1.2.
  Files used: *thread.service*.
- Sets up the TUN interface in the same manner as discussed throughout this document.
  Files used: *make_tun.sh*.
- Configures and starts the Thread stack on the Thread device (parameter /dev/ttyACMx). The device is expected to be in a factory state at the beginning of the script. This can be achieved with either supplying the optional argument "--reset", or by performing a manual reset (press and hold SW1 for at least six seconds). The configuration defaults to:
  - IEEE® 802.15.4 channel: 11 – may be changed from the command line
  - IEEE EUI-64: 0x146E0A000000000B

  The user is expected to modify the IEEE EUI-64 attribute depending on each application requirements in the start_router() method from relay.py:106.
  Files used: *CC_Helper.py*.
- Starts a new thread for handling data receiving and sending from/to the TUN interface previously created.
  Files used: *VTun.py*, which is a Python mirror of Thread_KW_Tun.c

- Opens two datagram sockets and two threads responsible for datagram forwarding in both directions: Commissioner –> Thread network and Thread network -> Commissioner:
  - The first is intended for receiving and sending data to the Commissioner. If the optional argument "-6" is provided, the socket listens on IPv6. Otherwise, only IPv4 datagrams can be used on the Commissioner – Border Router channel.
  - The second is intended for receiving and sending data to the Thread device, using IPv6 only.
  - Both sockets bound to 49191 – see Section 7.1.2.

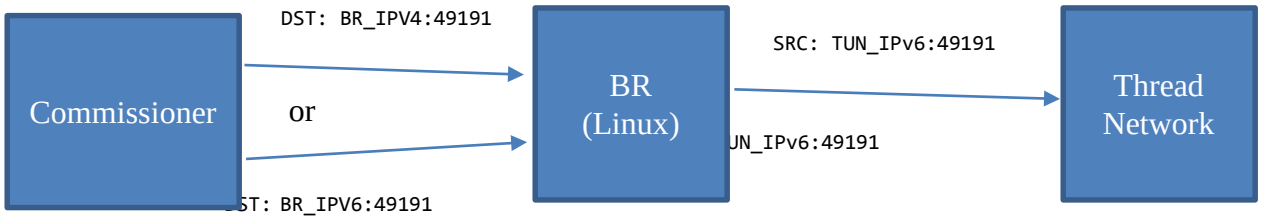


Figure 4. Flow of datagrams on path Commissioner -> Thread network

7.1.2 Commissioner discovery

The first step when initiating a commissioning procedure is Commissioner Discovery over Ethernet/Wi-Fi, as ratified by the Thread specification, Section 8.4.1.1.2:

“Devices implementing commissioning Border Agents with external Ethernet, Wi-Fi, or some other type of interface to a non-constrained IPv4 and/or IPv6 network with DNS query and DNS response capabilities SHALL advertise on this external interface using DNS-SD (DNS Service Discovery, [RFC 6763]) protocol to indicate availability of commissioning services to Commissioner Candidates. A DNS-SD Service Instance Name advertised by a Border Agent SHALL use the following <Service> section: _meshcop._udp.”

On Linux OS, the standard way of advertising services over mDNS/DNS-SD is by using a daemon (equivalent to a Windows OS service) called Avahi. Upon startup, avahi-daemon interprets its configuration file/etc/avahi/avahi-daemon.conf and reads XML fragments from /etc/avahi/services/*.service, which may define static DNS-SD services. The template XML fragment is defined in thread.service:

```
$ cat thread.service
<?xml version="1.0" standalone='no'?><!--*-nxml-*-->
<!DOCTYPE service-group SYSTEM "avahi-service.dtd">

<service-group>
  <name replace-wildcards="yes">%h THREAD</name>

  <service>
    <type>_meshcop._udp</type>
    <!-- Port abbreviation :MC -->
    <port>49191</port>
    <txt-record>rv=1</txt-record>
```

```
    <txt-record>tv=1.1.0</txt-record>
    <txt-record>sb=52000000</txt-record>
    <txt-record>nn=</txt-record>
    <txt-record>xp=</txt-record>
    <txt-record>vn=</txt-record>
    <txt-record>mn=</txt-record>
    <txt-record>br_ipv6=</txt-record>
  </service>
</service-group>
```

The service type is the namespace as required by the specification. Avahi daemon automatically adds the suffix ".local.". Section 8.4.1.2 also requires the port to have the value of :MC, i.e. 49191:

"The Commissioning Session SHOULD use the assigned UDP (User Datagram Protocol) port number advertised during the discovery phase. This port is known as the Commissioner Port (":MC")."

Following, two txt-record fields are mandatory and another 2 are optional: Network Name and Extended PAN ID, respectively Vendor Name and Vendor Model. These txt-records are populated by relay.py, script that also copies the service file in the proper location, which is /etc/avahi/services. This is an example of a completed service file that gets loaded by the avahi-daemon:

```
$ cat /etc/avahi/services/thread.service
<?xml version="1.0" standalone='no'?><!--*-nxml-*-->
<!DOCTYPE service-group SYSTEM "avahi-service.dtd">

<service-group>
  <name replace-wildcards="yes">%h THREAD</name>

  <service>
    <type>_meshcop._udp</type>
    <!-- Port abbreviation :MC -->
    <port>49191</port>
    <txt-record>rv=1</txt-record>
    <txt-record>tv=1.1.0</txt-record>
    <txt-record>sb=52000000</txt-record>
    <txt-record>nn=Kinetis_Thread</txt-record>
    <txt-record>xp=0D33CCE4011D8FB9</txt-record>
    <txt-record>vn=NXP</txt-record>
    <txt-record>mn=Kinetis_Demo</txt-record>
    <txt-record>br_ipv6=fe80::5ef9:ddff:fe6f:9064</txt-record>
  </service>
</service-group>
```

Our solution introduces an additional txt-record specifying the Border Router IPv6 Link Local address, in other words, the IPv6 address of the Linux machine, visible to external Commissioners on the local network. This is because some mDNS/DNS-SD resolvers do not implement discovery over IPv6, which makes it impossible to start a commissioning procedure natively over IPv6.

From this point on, the Commissioner Discovery may be implemented in many different ways. The information obtained via discovery is demonstrated by leveraging the Python zeroconf 3rd party module and by using the code snippet that it provides[2]:

```
>>> from zeroconf import ServiceBrowser, Zeroconf
>>> class MyListener(object):
        def remove_service(self, zeroconf, type, name):
                print("Service %s removed" % (name,))
        def add_service(self, zeroconf, type, name):
                info = zeroconf.get_service_info(type, name)
                print("Service %s added, service info: %s" % (name, info))

>>> zeroconf = Zeroconf()
>>> listener = MyListener()
>>> browser = ServiceBrowser(zeroconf, "_meshcop._udp.local.", listener)
>>>      Service      ubuntu      THREAD._meshcop._udp.local.      added,      service      info:
ServiceInfo(type='_meshcop._udp.local.',      name=u'ubuntu      THREAD._meshcop._udp.local.',
address='\xc0\xa8&\x80',    port=49191,    weight=0,    priority=0,    server=u'ubuntu.local.',
properties={'xp': '1122334455667788', 'vn': 'NXP', 'mn': Kinetis_demo', 'nn': 'Kinetis_Thread',
'br_ipv6': 'fe80::20c:29ff:fe28:59d5'})
```

Service information address='\xc0\xa8&\x80' translates to 192.168.38.128, which is the IPv4 address of the Ubuntu machine we have used in this example (likewise, br_ipv6 matches the IPv6 address of the same machine interface). We conclude the Commissioning Discovery part by reminding that a successful mDNS query provides the following relevant information: IPv4 and IPv6 addresses of the Border Router, the port intended for MeshCoP activity and Thread attributes such as Extended PAN ID, Network Name, Vendor Name and Vendor Model.

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. The first example uses a CoAP POST message to change the color of the RGB LED on a FRDM-KW24D, 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 FRDM-KW24D for the first example to see the color of the RGB led change each second.

Host software prerequisites: pip install txThings --upgrade

[2] https://pypi.python.org/pypi/zeroconf

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

7.3 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 fslthr0

Then, the led application, destination being the Unique Local Address of the joined FRDM-KW device:

[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 FRDM-KWdevice LED changes colors randomly each second.

7.4 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 fslthr0

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.

8 Thread Integration with Linux OS Host on RNDIS Interface

NOTE

THCI over RNDIS is enabled by default only in the configuration border_router on usbkw24d512. To ensure that THCI over RNDIS works on different platforms, 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_<version>\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 KW2xD configuration in the **border_router** example that acts as a ND_ROUTER at the IPv6 level on the emulated Ethernet over USB link, and 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

eth6: <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 604658sec preferred_lft 85658sec

    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 similar to other 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 all of these, the well-known third party library *libpcap* is used. Through libpcap, the user can send THCI frames that are 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.

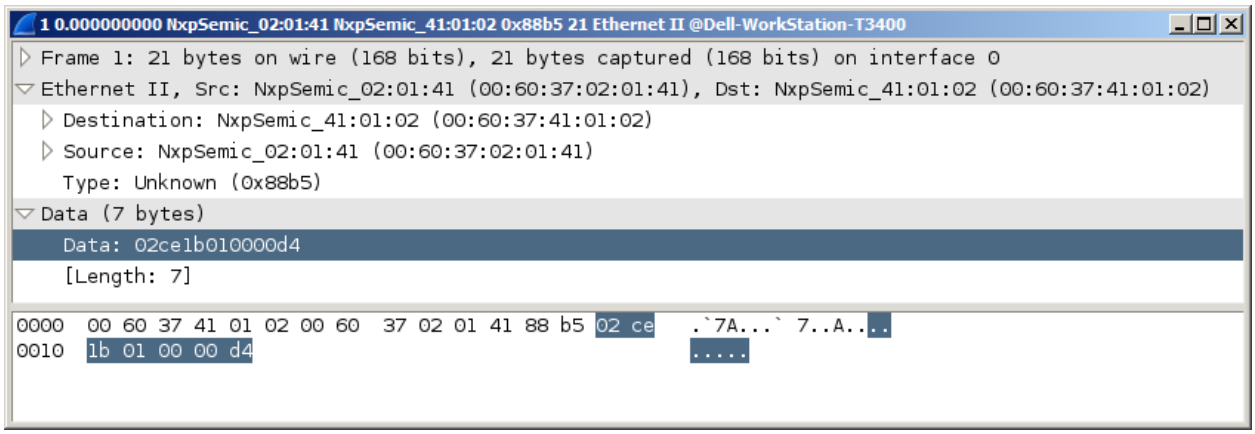


Figure 5. Host injects a packet over RNDIS. Data represents a Thread Create Network Request frame.

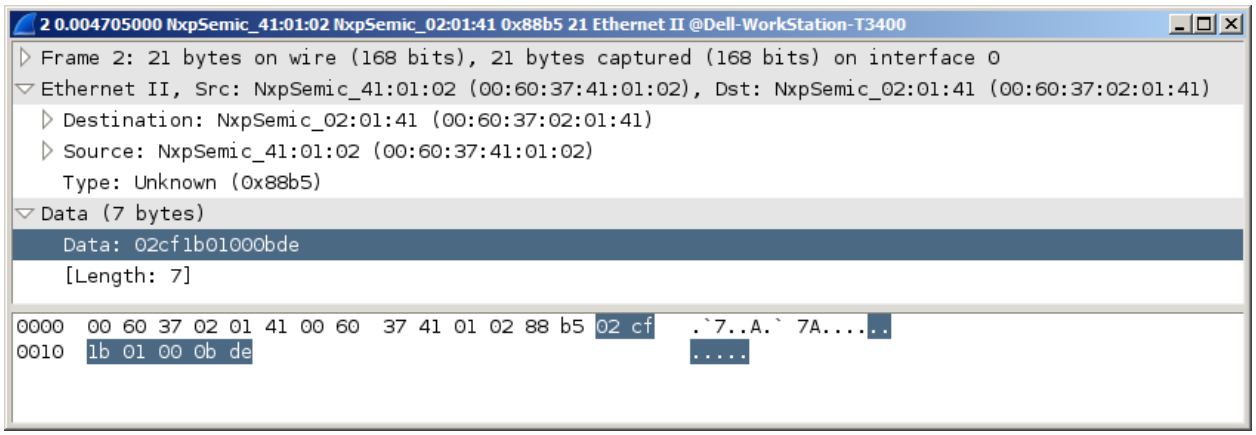


Figure 6. Board replies. Data represents a Thread Create Network Confirmation frame.

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

9 Host API Python Bindings

9.1 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 KW 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.

9.2 Platform setup

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

9.2.1 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. When 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')
```

9.2.2 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 following path: C:\NXP\MKW24D_ConnSw_1.0.1\tools\wireless\host_sdk\hsdk-python\src

9.3 Directory tree

```
lib/                                                    Compiled host SDK libraries for Windows.
├── README.md
├── x64
│   └── HSDK.dll
└── x86
    └── HSDK.dll
misc/                                   Presented in section External Commissioning of Untrusted Thread Radio Devices.
└── commissioning
    └── linux
        ├── CC_Helper.py
        ├── make_tun.sh
        ├── relay.py
        ├── SerialTUNCommissioner.py       Script to start the Commissioner on the KW Border
Router, as opposed to relay.py, which expects an external Commissioner.
        ├── thread.service
        └── VTun.py
src/
├── com
│   ├── nxp
│   │   ├── wireless_connectivity
│   │   │   ├── commands
│   │   │   │   ├── 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          Generated files for THCI 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      More about this in section How to reprogram a KW device.
│   │   │       │   └── __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
│   │   │       ├── __init__.py
│   │   │       ├── macfiltering.py
│   │   │       ├── 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
│   │   │   └── __init__.py
│   │   └── __init__.py
│   └── __init__.py
```

9.4 Tests and examples

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

Example of usage:

a) 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.

b) Windows OS

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

9.5 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 familiar API in the com. nxp.wireless_connectivity.commands package. Such a package contains the following modules:

thread | 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 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 that sends a request and returns the triggered event.

9.6 Development

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)
```

Events

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

```
def callback(deviceName,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 two parameters (deviceName and expectedEvent in the example above) which is used to gain access to the event object and the device that generated it.

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 us 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.

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.

9.7 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
)
```

Every supported AttributeId value is defined in enums. `THR_SetAttrRequestAttributeId`.

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(self.deviceName,frame)
        else:
            print_event(self.deviceName,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 or not 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 or not the Status is 0x00 – OK.

```
def callback(device_name,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(device_name,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 are 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 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*.

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.

```
$ ./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

11 Appendix A – Code Samples

11.1 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'
```

11.2 Python code sample – Thread ifconfig

```
 [imports]

def my_callback(device_name,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 particular one does not posses 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 particular device has an additional TUN interface enabled.

11.3 Python code snippet – Thread socket creation

```
[imports]

def my_callback(device_name,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
```

12 Revision History

This table 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

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document number: KFSCIHAPIUG
Rev. 2
12/2016