Document identifier: KFSCIHAPIUG
Rev. 8, 06 August 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 Bluetooth® Low Energy 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 Bluetooth LE ‘black box’ firmware running on a Kinetis microcontroller.

Chapter 2
Deploying Host Controlled Firmware

2.1 Bluetooth LE application configuration

To exercise the Host API, the Bluetooth LE 'black box' firmware is required to be flashed on a compatible platform. The Bluetooth LE 'black box' is represented by the 'ble_fsci_black_box' application firmware that can be interfaced and configured with FSCI commands over the serial interface. One could use the binary image provided in the package, `tools\wireless\binaries\ble_fsci_blackbox.bin`, which uses UART as serial interface with 115200 baud rate.

As an alternative, 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.

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.

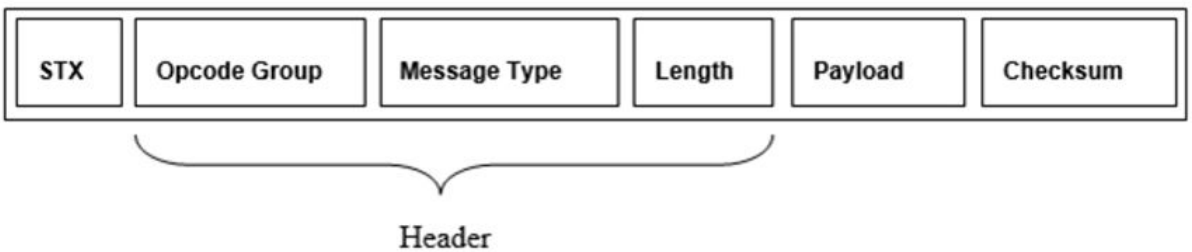

Figure 1. Sending and receiving messages

Table 1. 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, 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 Bluetooth LE stack instance from a Linux OS system, with focus on Python language bindings.

3.1 Kinetis wireless host software system block diagram

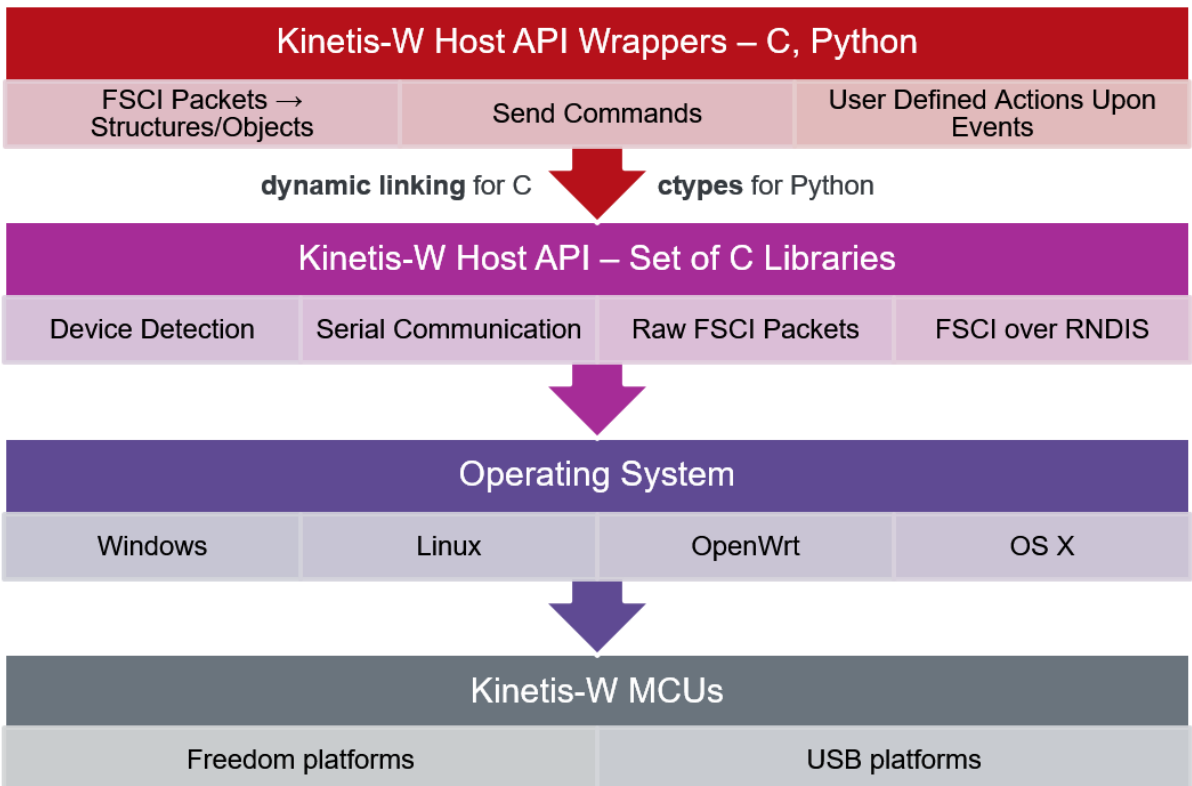

Figure 2. Kinetis host software system block diagram

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

├── 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 2. 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 3. Host SDK – SPI default values

Configuration	Value
Transfer Mode	SPI_MODE_0

Table continues on the next page...

Table 3. Host SDK – SPI default values (continued)

Configuration	Value
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
$ ls bin/
GetKinetisDevices
```

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.
```

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.

Chapter 6 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.

6.1 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.
```

6.2 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
```

6.3 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 7
Host API Python Bindings

7.1 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.

7.2 Platform setup

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

7.2.1 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')
```

7.2.2 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\src.

7.3 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

│   │   │   ├── 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.
│   │   │       │   └── __init__.py

│   │   │       ├── hrs.py                Script implementing a Bluetooth LE Heart Sensor profile .
```

```
|   |   |       ├── __init__.py

|   |   └── __init__.py
|   └── __init__.py
└── __init__.py
```

7.4 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.
```

7.5 Development

7.5.1 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)
```

7.5.2 Events

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

```
def callback(devName, 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 (devName and expectedEvent in the example above) which are used to gain access to the event object and identify the serial port.

7.5.3 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.

7.5.4 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.

7.6 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)
```

7.6.1 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
```

7.6.2 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)
```

7.6.3 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.

Chapter 8 How to Reprogram a Device Using the FSCI Bootloader

To deploy a Bluetooth LE application with FSCI bootloader support, build the FSCI Bootloader application using an IDE from the projects located at `boards\[board]\wireless_examples\framework\bootloader_fsci` and flash it to the board using J-Link.

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.

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 ble_fsci_black_box -e
```

The script does the following:

- 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 9
Revision History

This table summarizes revisions to this document.

Table 4. 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	07/2019	Updated for K32W and QN90xx
7	04/2019	Updates for Thread KW41Z Maintenance Release 3
8	08/2019	Updates for the Bluetooth LE KW37A PRC1 Release

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 06 August 2019

Document identifier: KFSCIHAPIUG