AN1259: Using the Silicon Labs *Bluetooth*® Stack v3.x and Higher in Network Co-Processor Mode

This document is an essential reference for anyone developing a system for the Silicon Labs Wireless Gecko products using the Silicon Labs v3.x Bluetooth Stack in Network Co-Processor (NCP) mode. The document covers the C language application development flow, walks through the examples included in the stack, and shows how to customize them.

KEY POINTS

- Introduces the available tools for NCP system development.
- Walks through the NCP host and target examples.

1. Introduction

The Silicon Labs Bluetooth SDK allows you to develop System-On-Chip (SoC) firmware in C on a single microcontroller. The SDK also supports the Network Co-Processor (NCP) system model.

This document provides a guide on how to get started with software development of an NCP system. It describes the development tools and example projects, then highlights the most important steps you need to follow when writing your own application.

1.1 SoC vs NCP System Models

On an SoC system, the Application code, the Bluetooth Host, and Controller code run on the same Wireless MCU.

On an NCP system, the Application runs on a Host MCU, and the Host and Controller code run on a Target MCU. The Host and Target MCUs communicate on a serial interface. The communication between the Host and Target is defined in the Silicon Labs Proprietary Protocol called BGAPI. The physical interface is UART. BGLib v3.x is an ANSI C reference implementation of the BGAPI protocol, which can be used in the NCP Host Application.

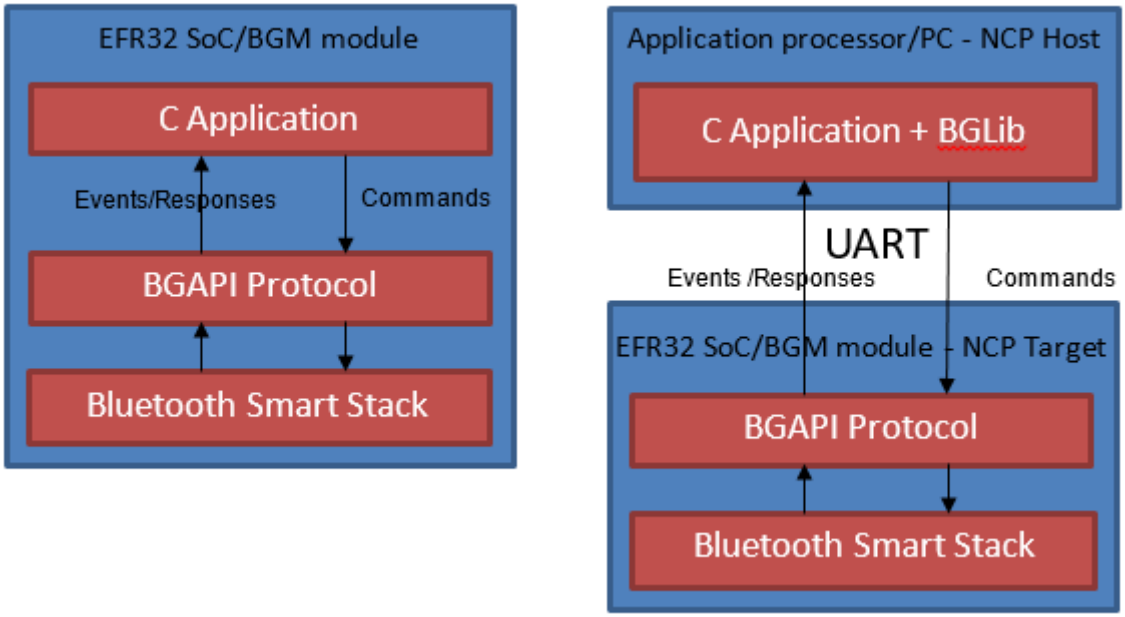


Figure 1.1. SoC vs NCP System Models

2. NCP Target Development

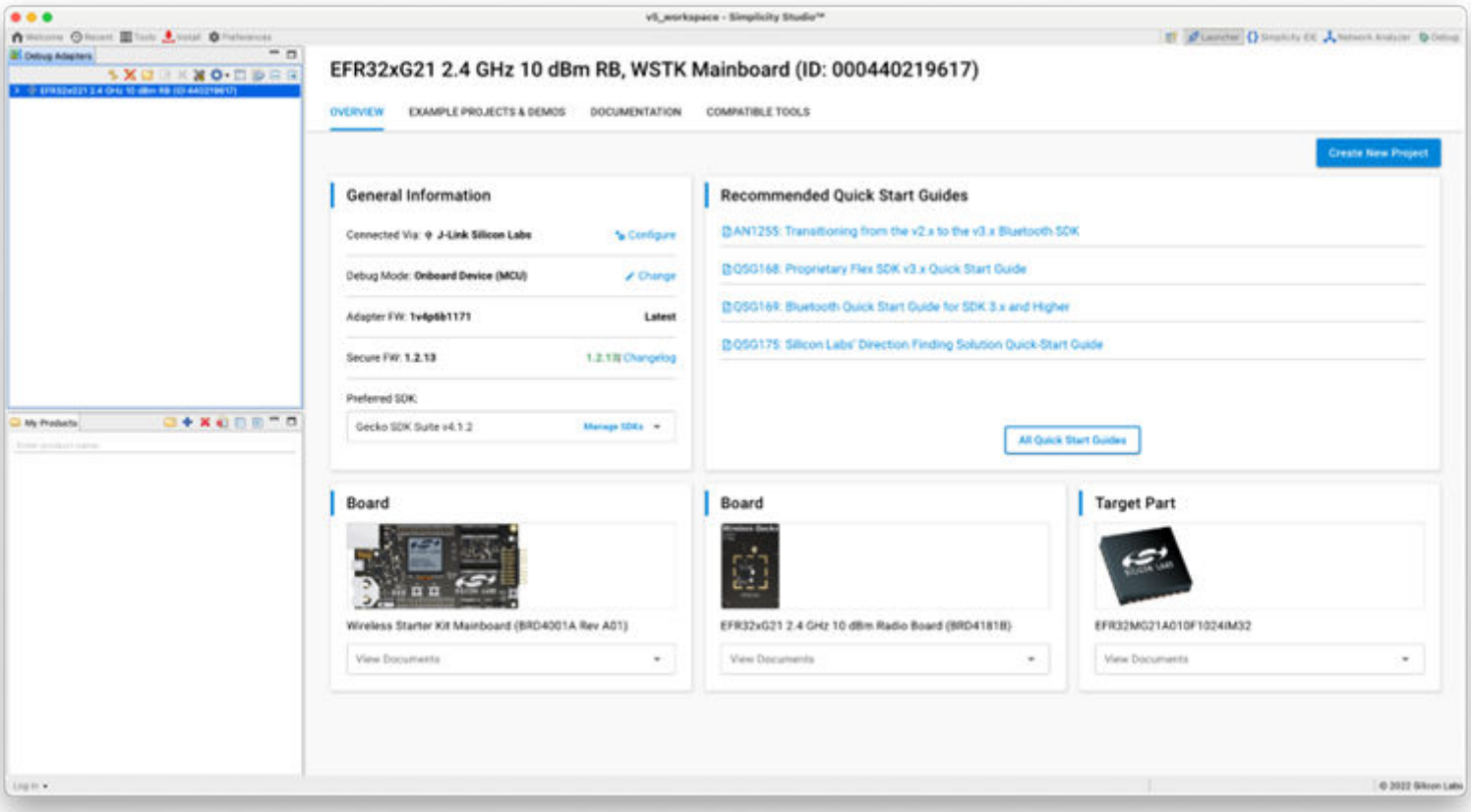
This chapter describes the available tools for compiling and flashing the NCP target firmware.

Before proceeding with compiling and flashing C-based firmware, install Simplicity Studio 5 (SSv5). You can download it from the Silicon Labs website: http://www.silabs.com/simplicity. Once Simplicity Studio is installed, follow the prompts to install the Gecko SDK Suite (GSDK) containing the Bluetooth SDK. See the online (Simplicity Studio 5 User's Guide) for details on installation and using Simplicity Studio. See *QSG169: Bluetooth® SDK v3.x Quick Start Guide* for additional detail about the Bluetooth SDK.

Note: *AN1042: Using the v2.x Silicon Labs Bluetooth® Stack in Network Co-Processor Mode* describes in detail how the NCP is implemented in the Gecko SDK v2. This application note explains extensively the code and tools on both the target and host side.

To develop in C, you not only need Simplicity Studio 5 but also a supported compiler. The Bluetooth SDK release notes and *UG434: Silicon Labs Bluetooth® C Application Developers Guide for SDK v3.x* list the supported compilers.

The NCP target firmware comes with the Bluetooth SDK. It is available in a precompiled binary format and as a project file you can build. The following procedures describe how to install the precompiled binary image and how to build and install the example project. Note that Simplicity Studio only shows the relevant examples for the preferred SDK, so you have to select **Gecko SDK Suite vn.n.n** first, as shown in the following figure. (Note: Your SDK version may be different from the one shown in the figure.)

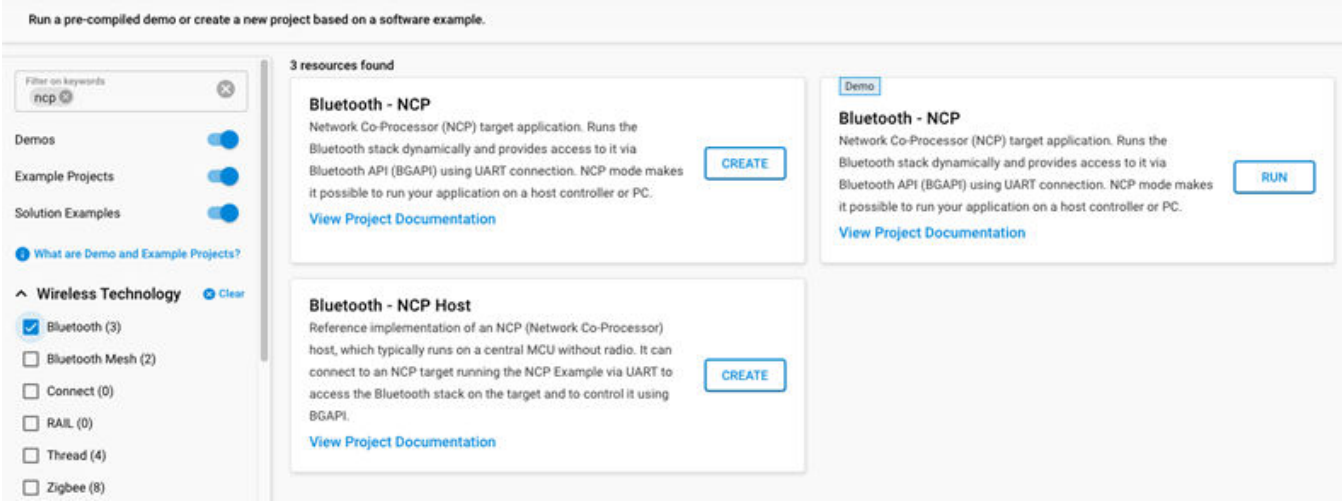


The following procedure describes how to build and load the example code. This procedure assumes you have already loaded a Gecko Bootloader in one of the following ways:

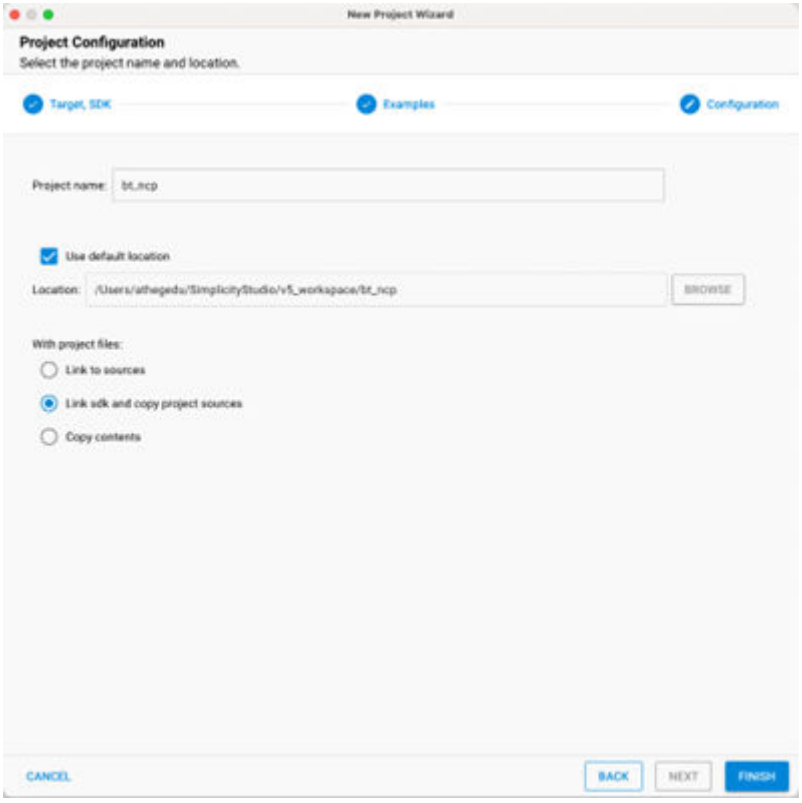
- Loaded the Gecko Bootloader precompiled binary from the list of Demos. For an NCP application, load the NCP BGAPI UART DFU bootloader.
- Built and loaded your own Gecko Bootloader as described in *UG266: Silicon Labs Gecko Bootloader User's Guide in GSDK 3.2 and Lower*/*UG489: Silicon Labs Gecko Bootloader User's Guide in GSDK 4.0 and Higher*.

1. Click **Example Projects & Demos**, select **Bluetooth - NCP** and click **Create**.

   Note: The **Bluetooth - NCP** example does not contain a GATT database. The dynamic GATT API can be used for building it. Host software examples in the Bluetooth SDK build their GATT database dynamically, by default.



2. Name the project and click **Finish**.



3. Now the project is ready to build and flash. Click **Debug** (bug icon) in the top left menu to do it in one step. You may also use the precompiled NCP demos in Simplicity Studio, which are shipped with bootloader included.

   Note: If you get an error when you click **Debug**, click the project *.isc* file in the Project Explorer view. It may not be fully selected.

3. NCP Host Development

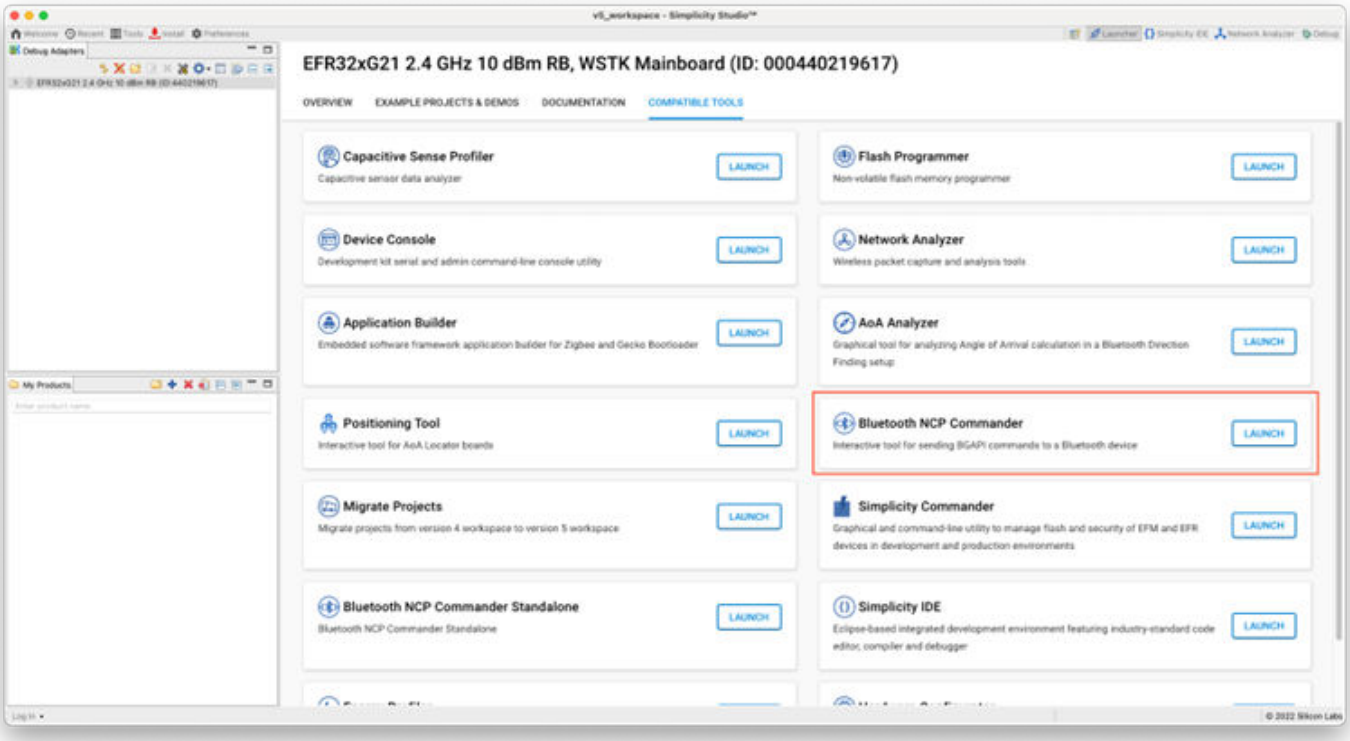
This chapter introduces the Bluetooth NCP Commander tool, which can be used to send BGAPI commands from a graphical user interface. It then walks through the process of building the PC Host examples provided in the Bluetooth SDK. And finally, it describes using Python for host side development.

3.1 Bluetooth NCP Commander

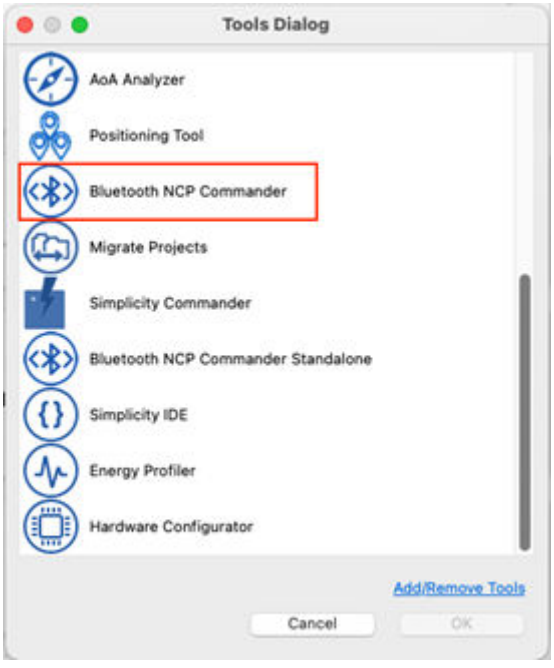
Bluetooth NCP Commander is an easy-to-use tool that can be used for testing different stack features, by sending BGAPI commands to the target device. The tool has two versions: a version built into Simplicity Studio, which makes it easy to connect to your development kit and start testing, and a standalone version to test a board in an environment where Simplicity Studio cannot be installed, or if you want to test a custom board that can be accessed on UART interface, but not through a Simplicity Studio supported debug adapter using VCOM.

Built-in Version

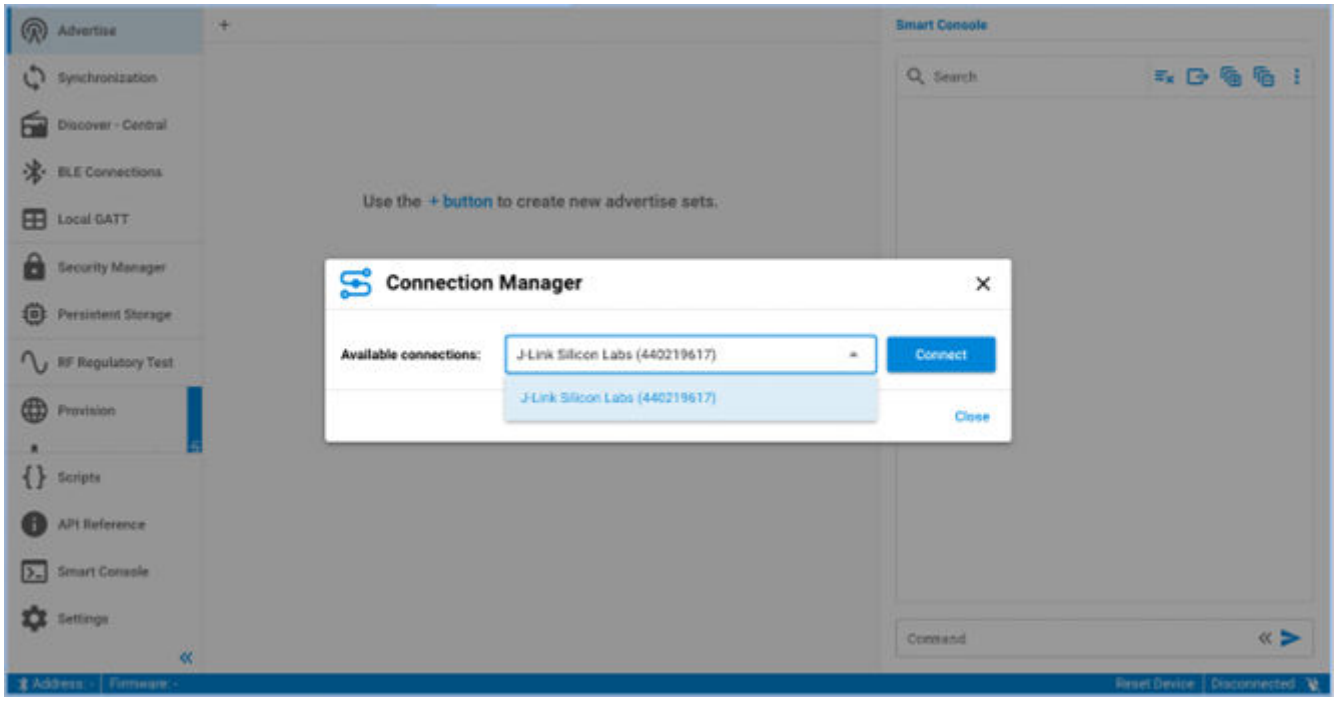
1. To open the built-in Bluetooth NCP Commander, select the target board in the **Debug Adapters** view, and check that the preferred SDK is set to **Gecko SDK Suite vn.n.n**. Select the **Compatible Tools** tab, and click **Launch** next to Bluetooth NCP Commander.



Alternatively, you can open the built-in Bluetooth NCP Commander from the **Tools** menu.



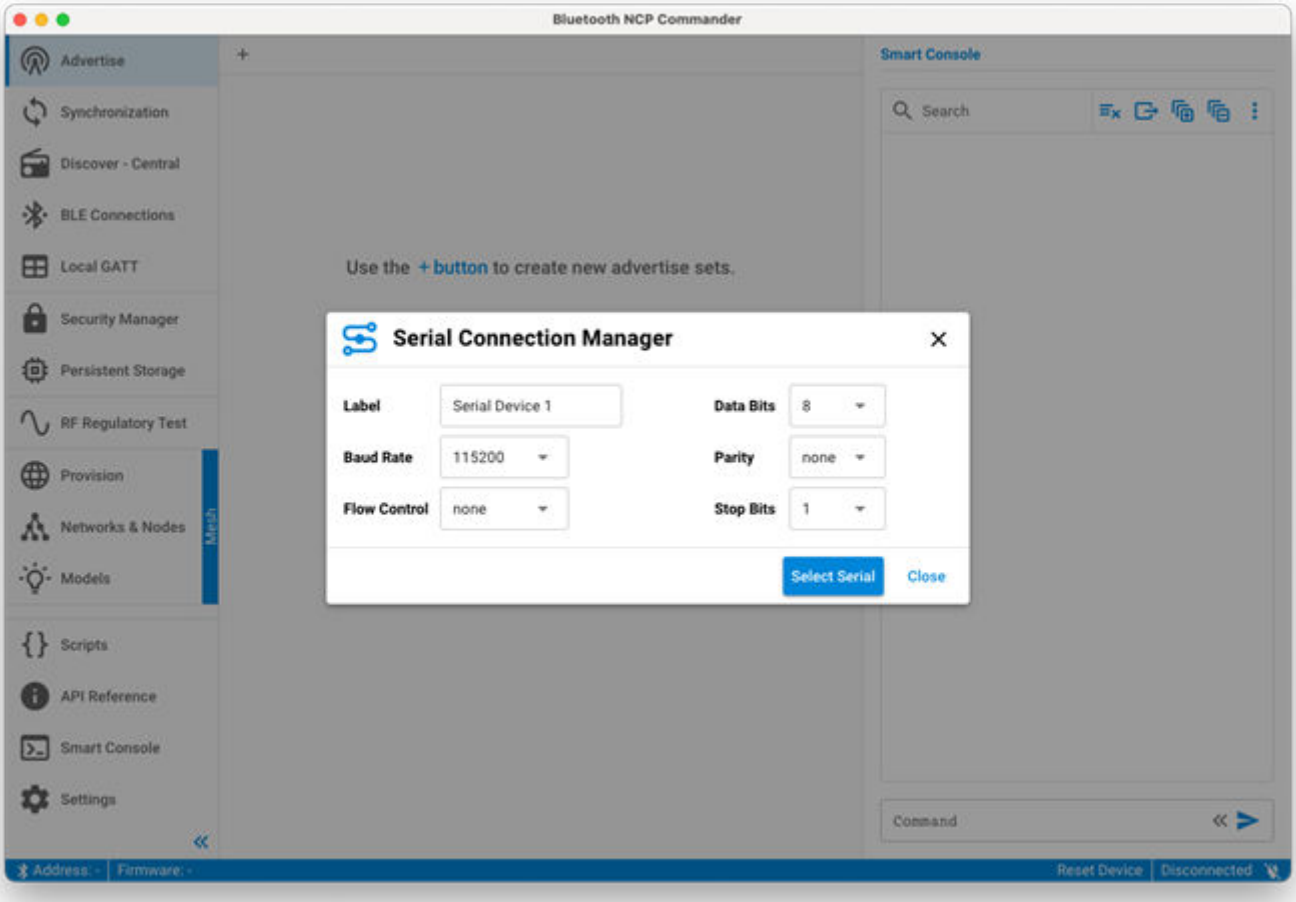
2. Select the target device, and click **Connect**.



Standalone Version

1. To open the standalone tool, navigate to *C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\ncp_commander*, and start NcpCommander.exe.

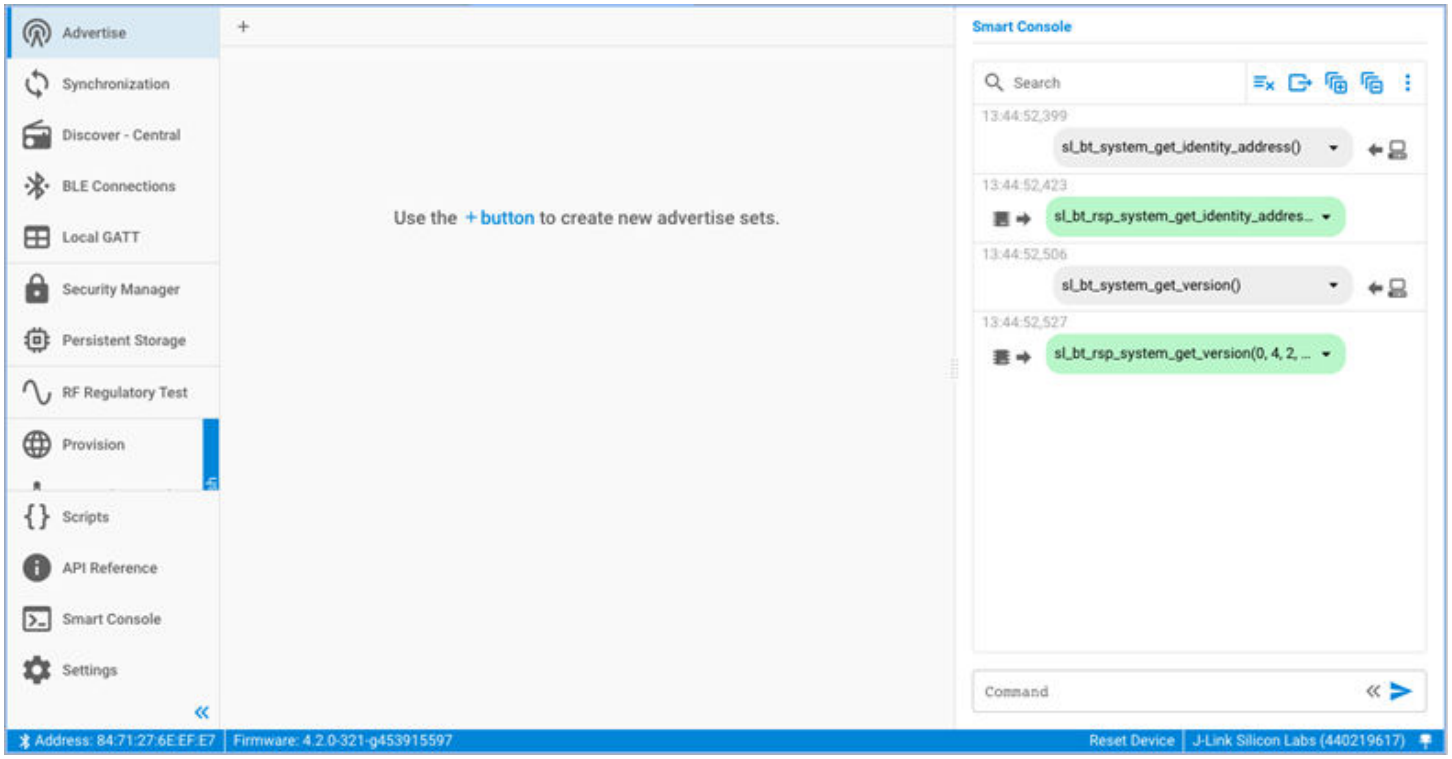
2. In the standalone tool, provide the UART interface settings, and then select the COM port on which the device can be accessed.



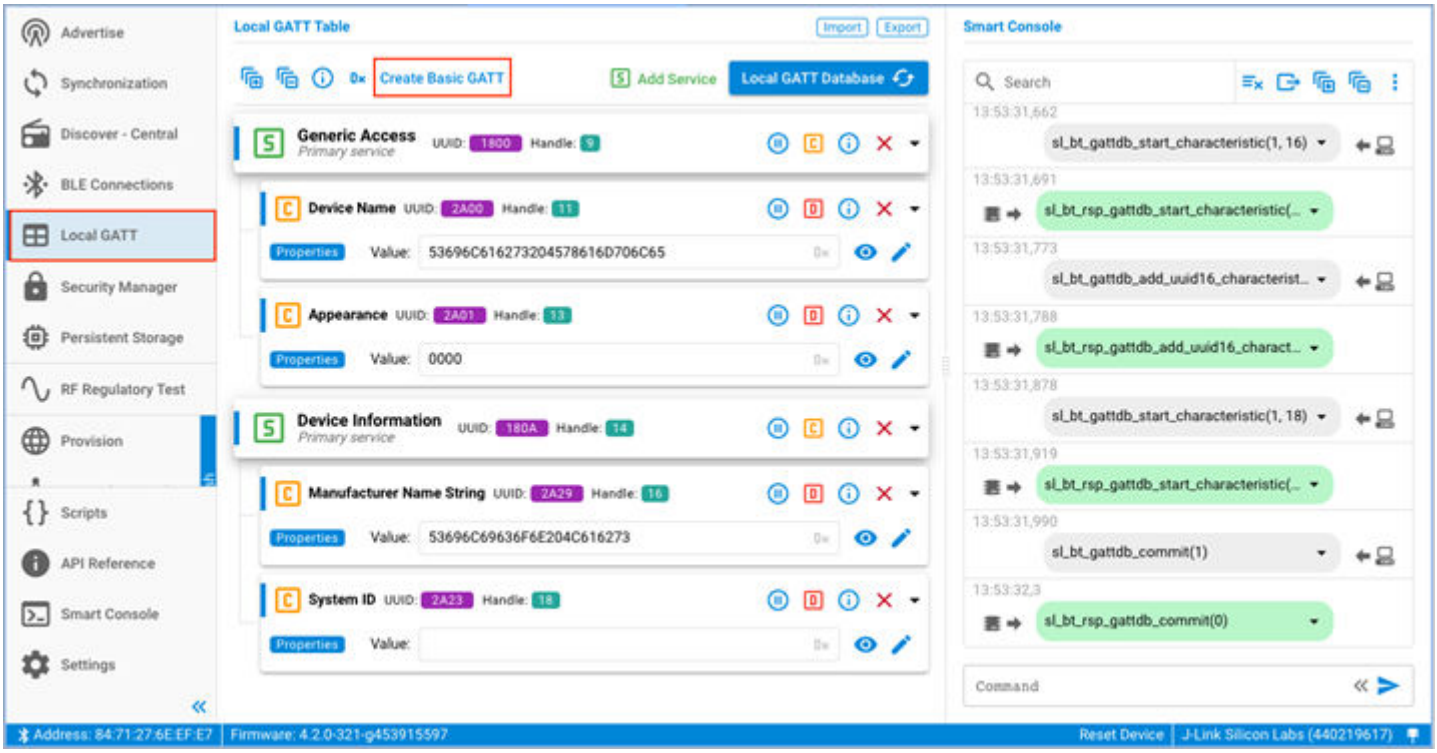
3.1.1 Bluetooth NCP Commander Functions

The following procedure covers most NCP Commander functions.

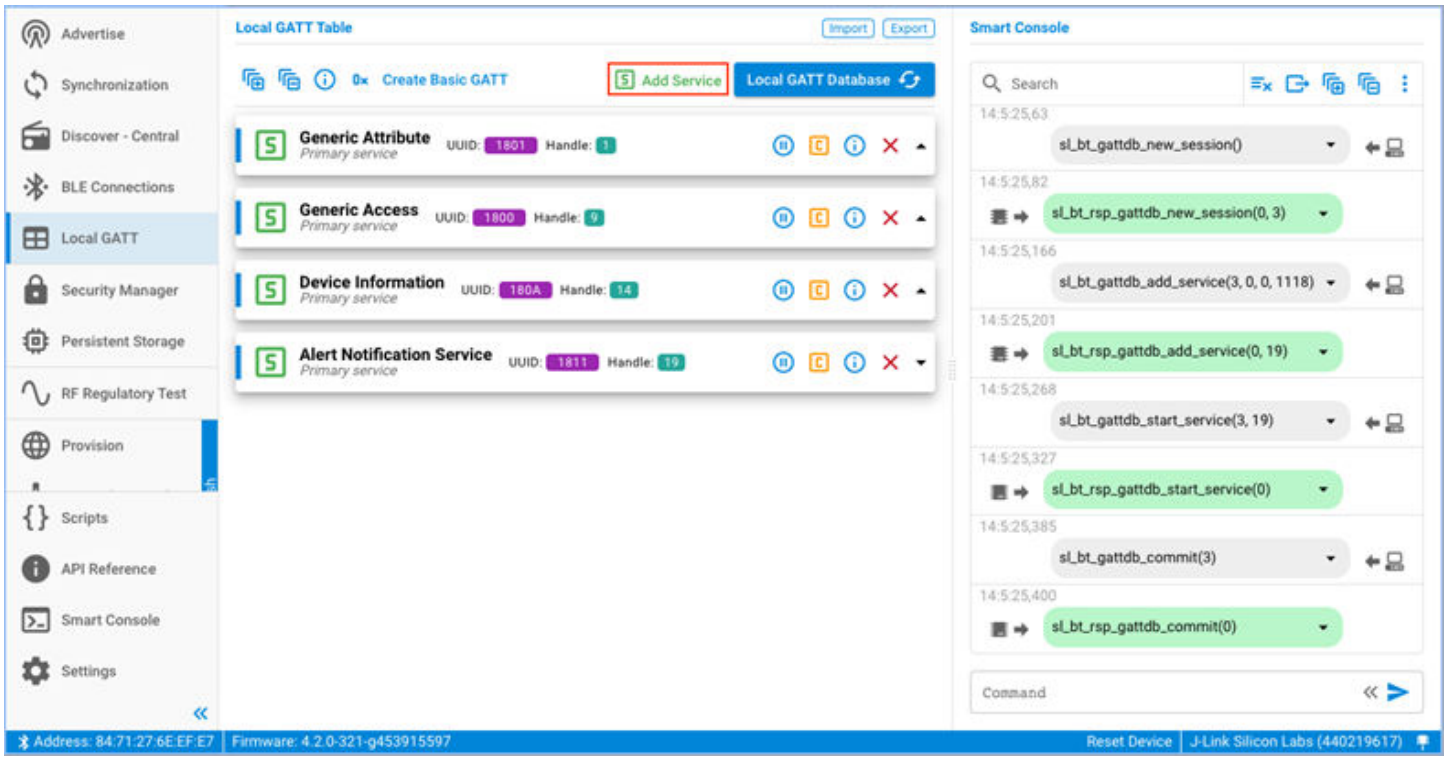
1. After the device is connected, you should see the result of the "sl_bt_system_get_identity_address" command displayed in green



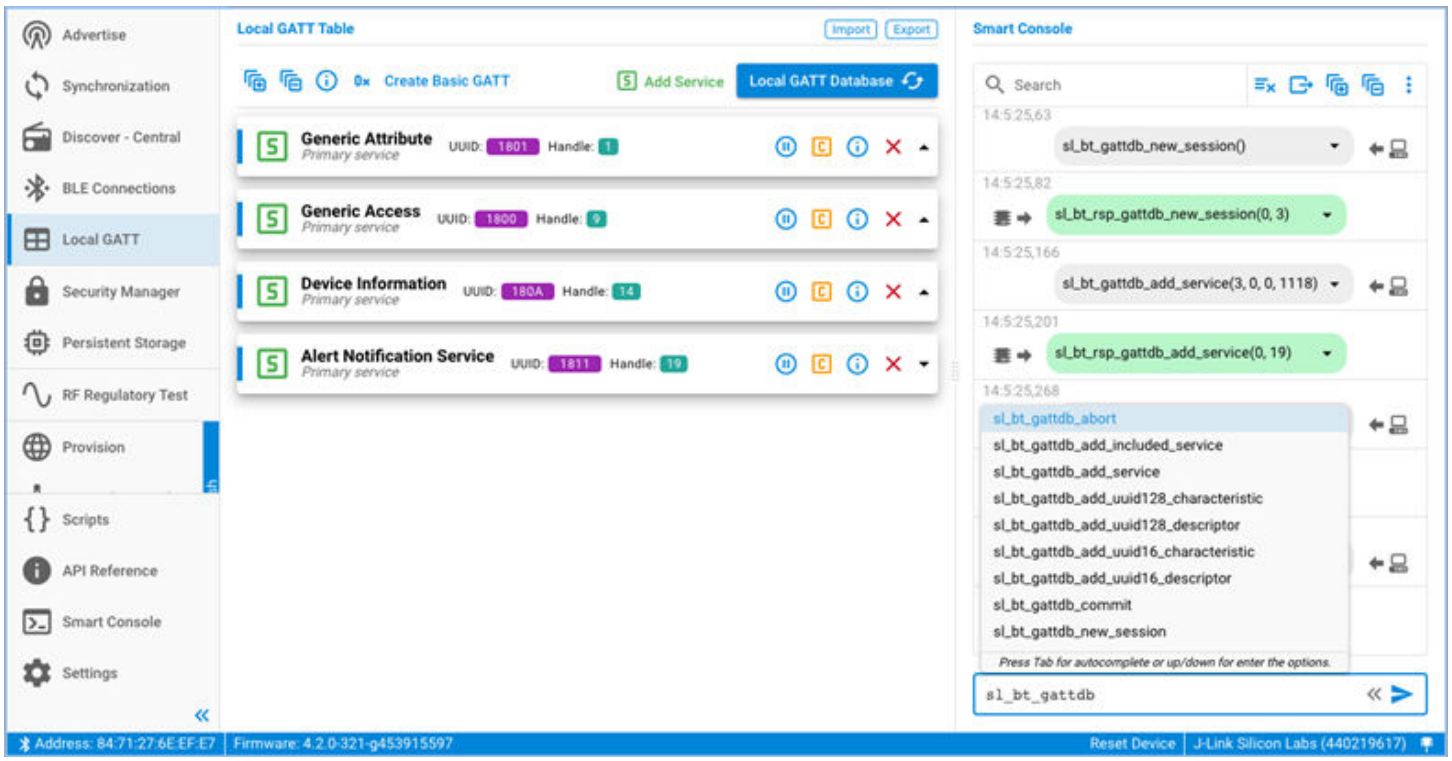
2. Unlike SoC examples, the NCP demo does not have a built-in GATT database. It expects the host to build the GATT database using the dynamic GATT database BGAPI commands. To create a basic GATT database, select the Local GATT menu, and click **Create Basic GATT**. This triggers a series of BGAPI commands that will build a basic database. You can modify this GATT database as you want. You can also change the device name here by changing the value of the Device Name characteristic.



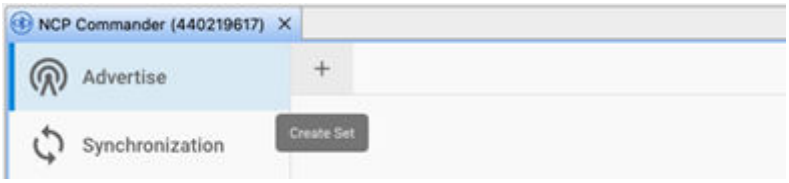
3. To extend the database with new services, characteristics, and descriptors, click **Add Service**. You can then add new characteristics for the service.



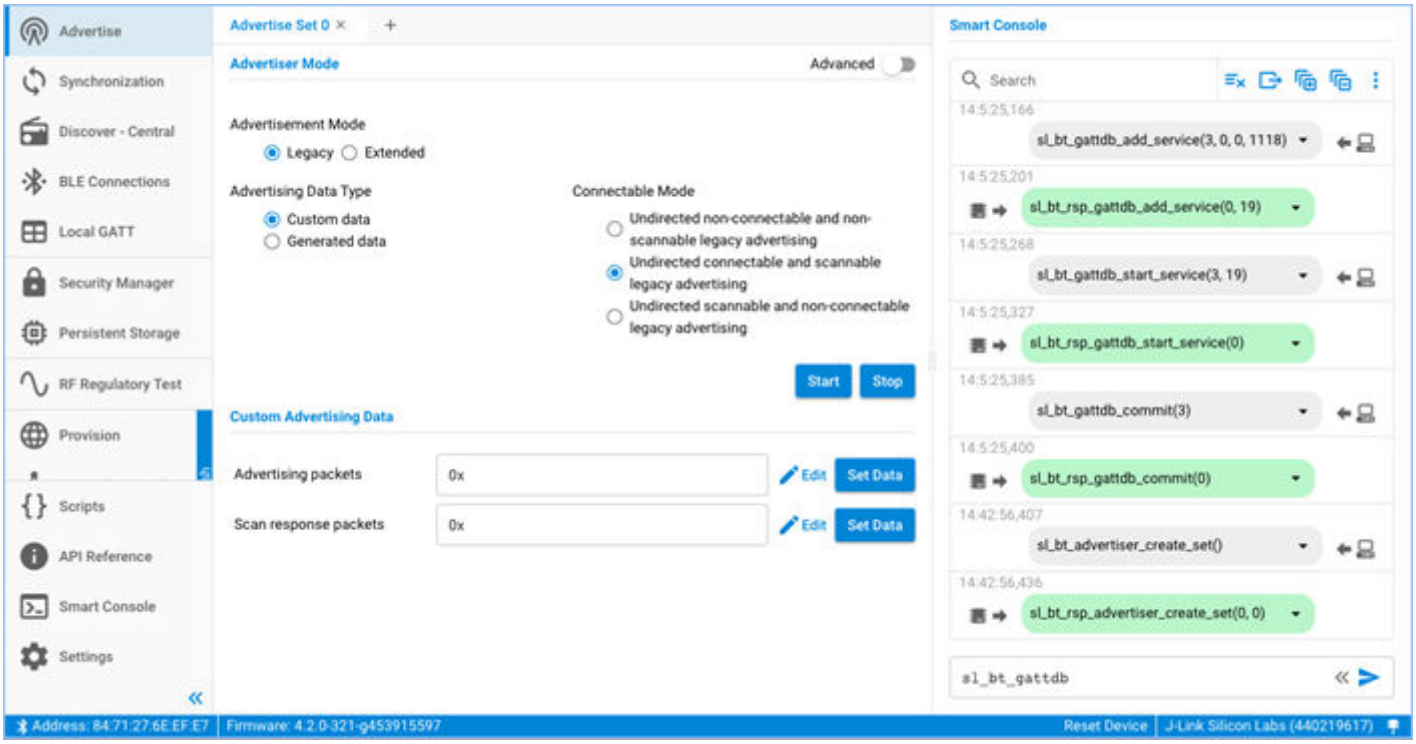
4. To read out the GATT database from the device, click **Local GATT Database**. The smart console also supports API calls for creating entries.



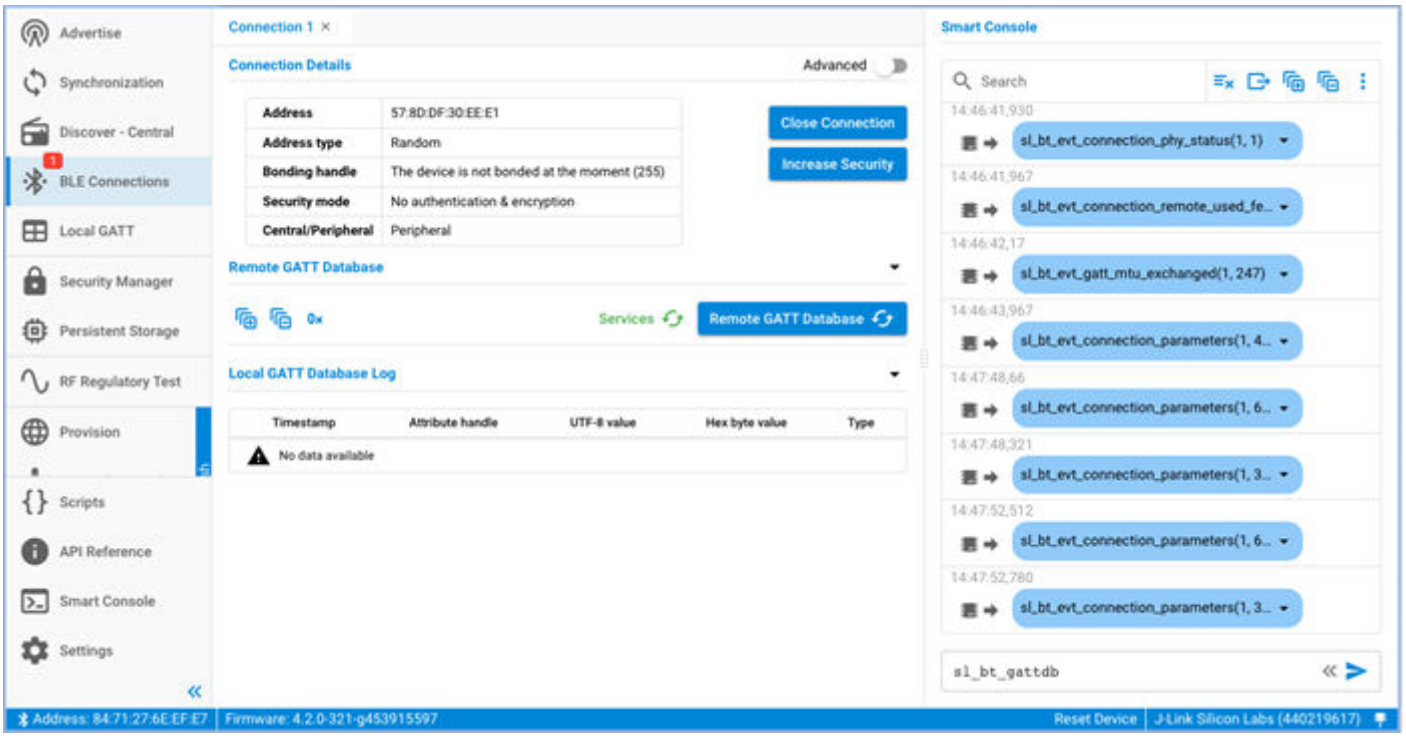
5. To start advertising your device so that other devices can discover it and connect to it, in the Advertise menu click '**+**' (Create Set) to create an advertiser set.



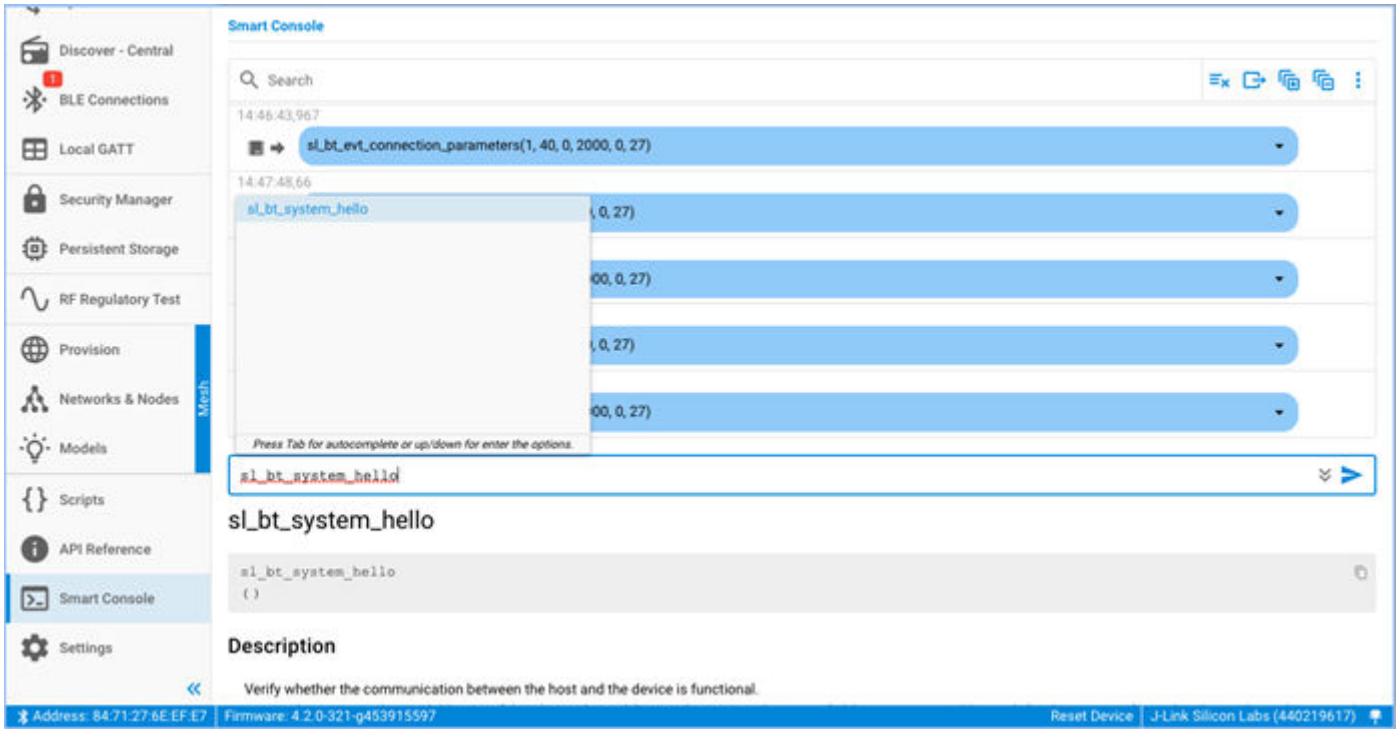
6. To populate the advertisement payload with the device name, set the Advertising Data Type to **Generated data** and click **Start** to start advertising.



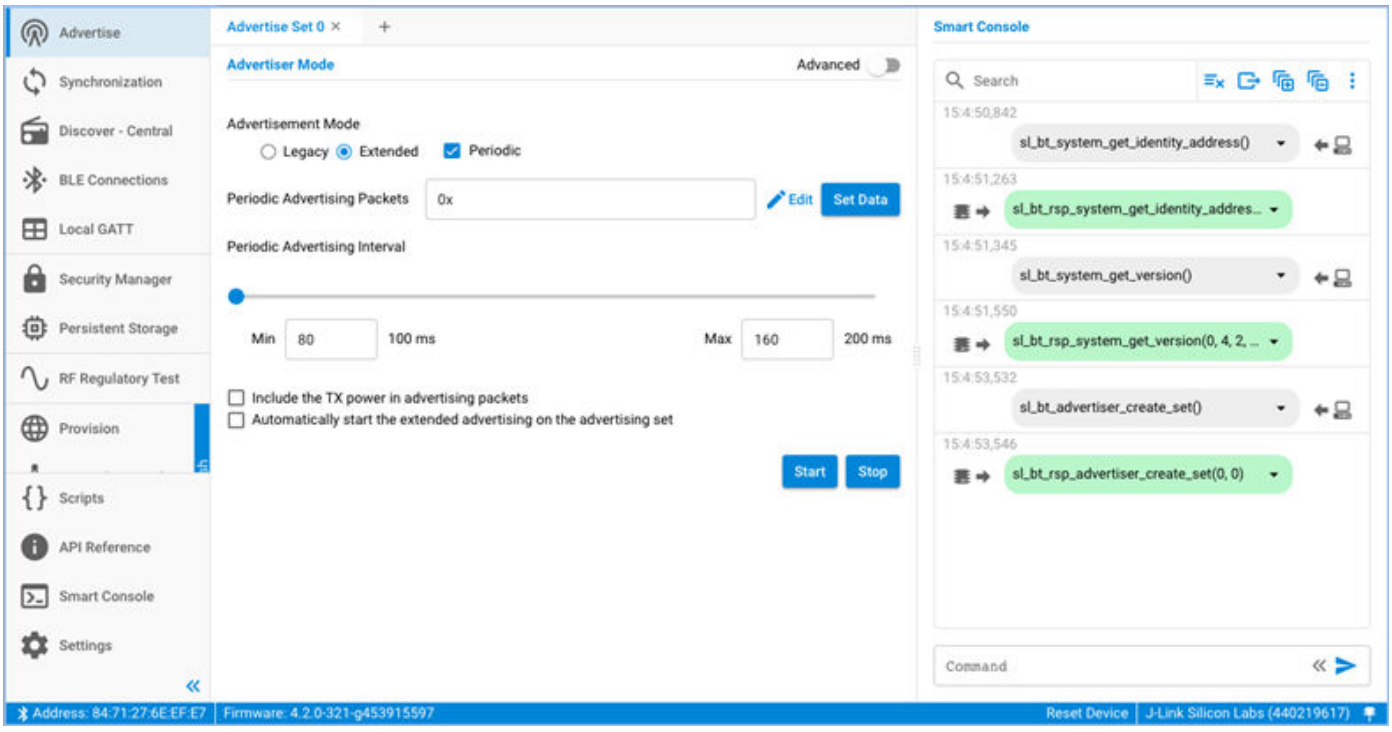
7. When advertising, the NCP target example accepts Bluetooth connections. If you connect to the mainboard or with another central device (for example with your phone), you can see the events and commands on the log.



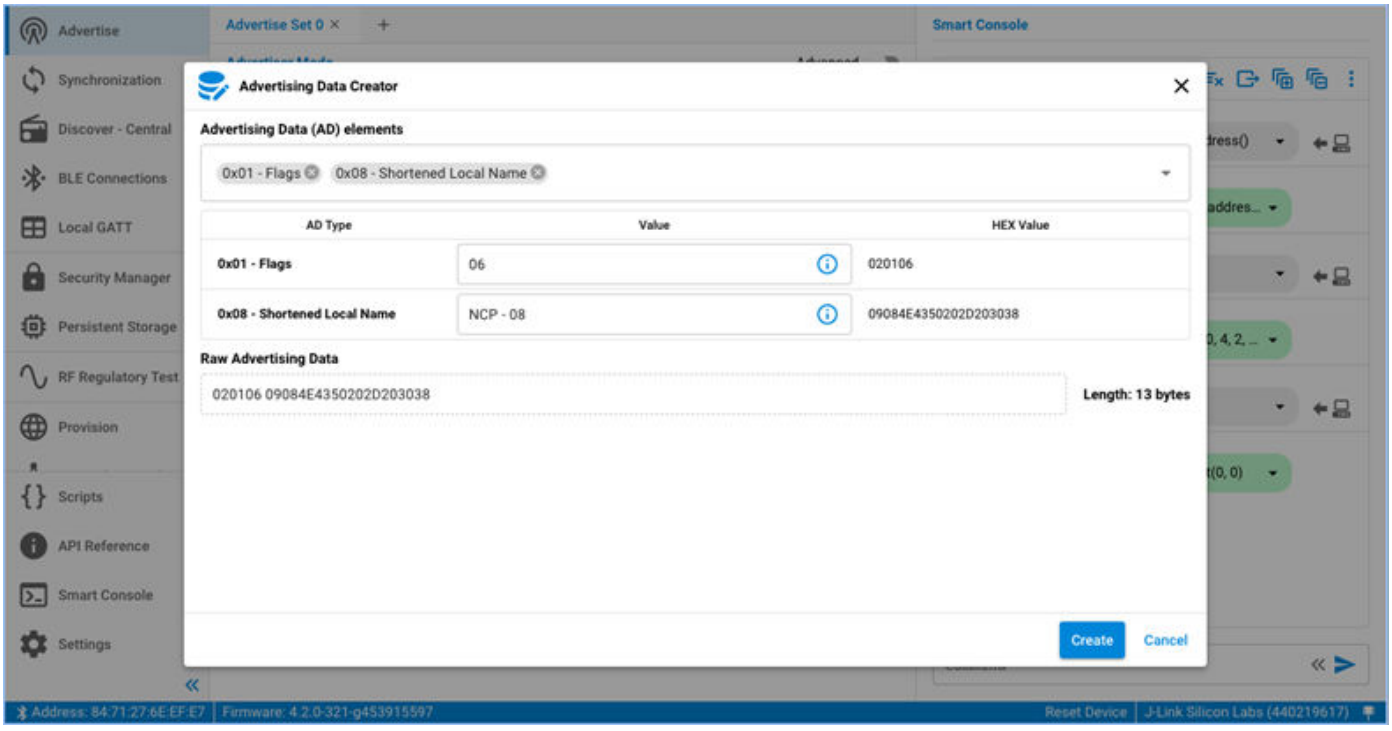
8. You can also issue commands manually. For example, you can issue the 'system hello' command at any time to verify that communication between the host and the device is working. The Smart Console provides auto-completion and documentation for the possible commands. To open/close the documentation, click the arrows at the right side of the input field.



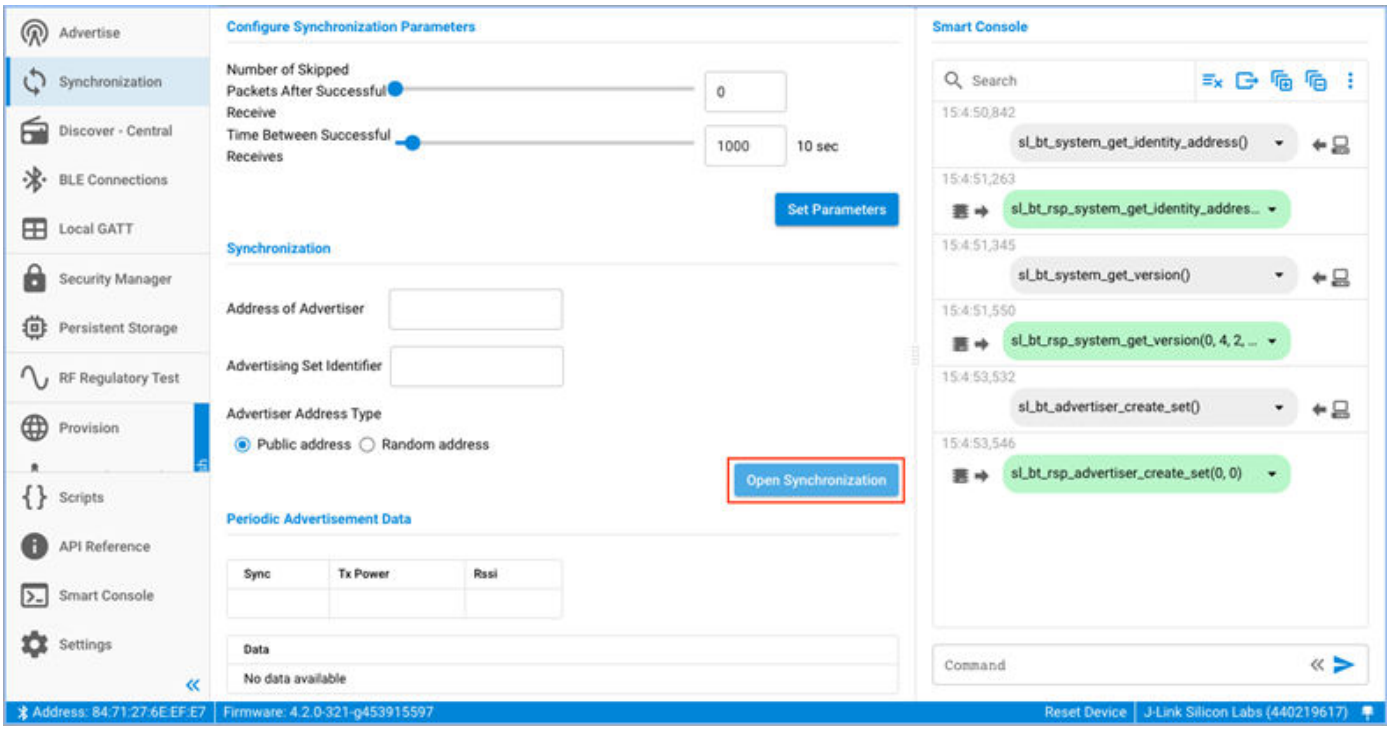
9. To create periodic advertisement sets, select **Advertisement mode: Periodic**. To set the content of the packet, use the **Edit** option next to "Periodic Advertising Packets".



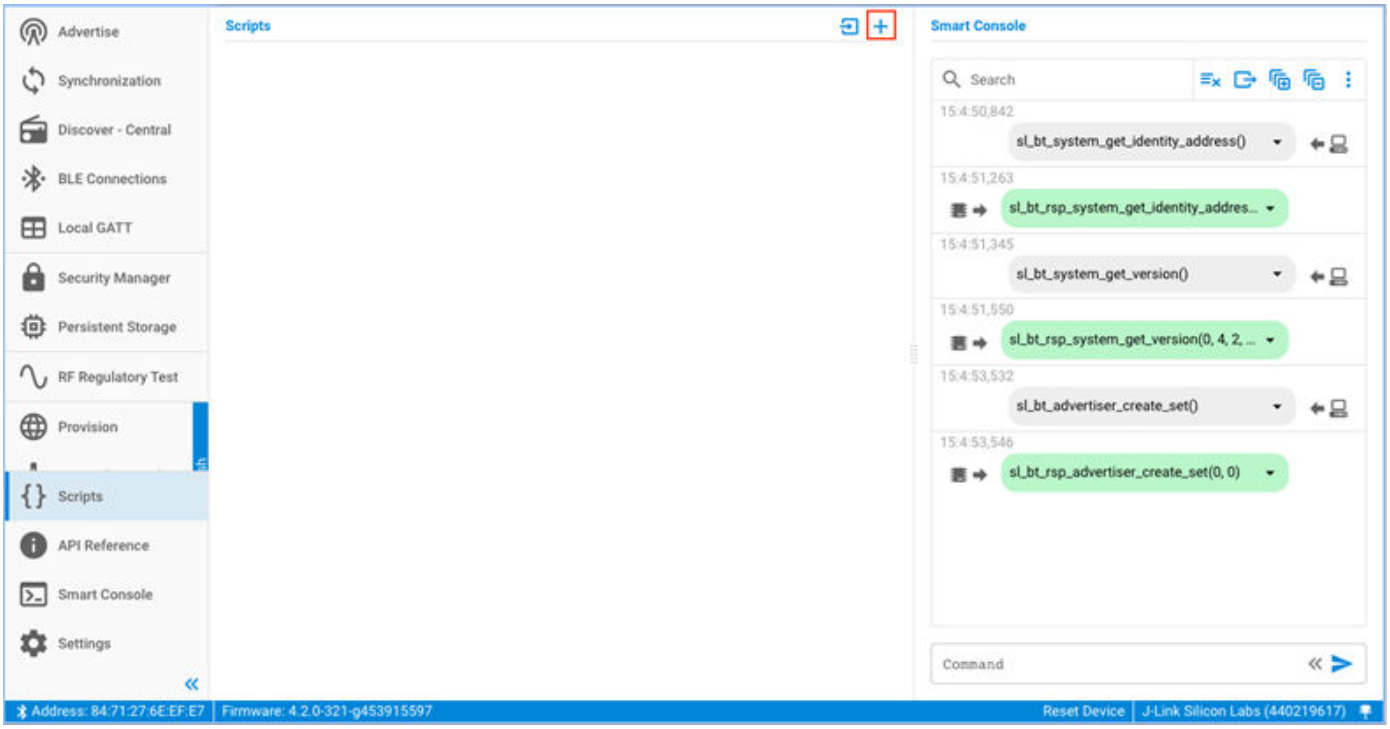
This opens a new dialog where you can edit the contents of the package. Click **Set Data** after the data is edited.



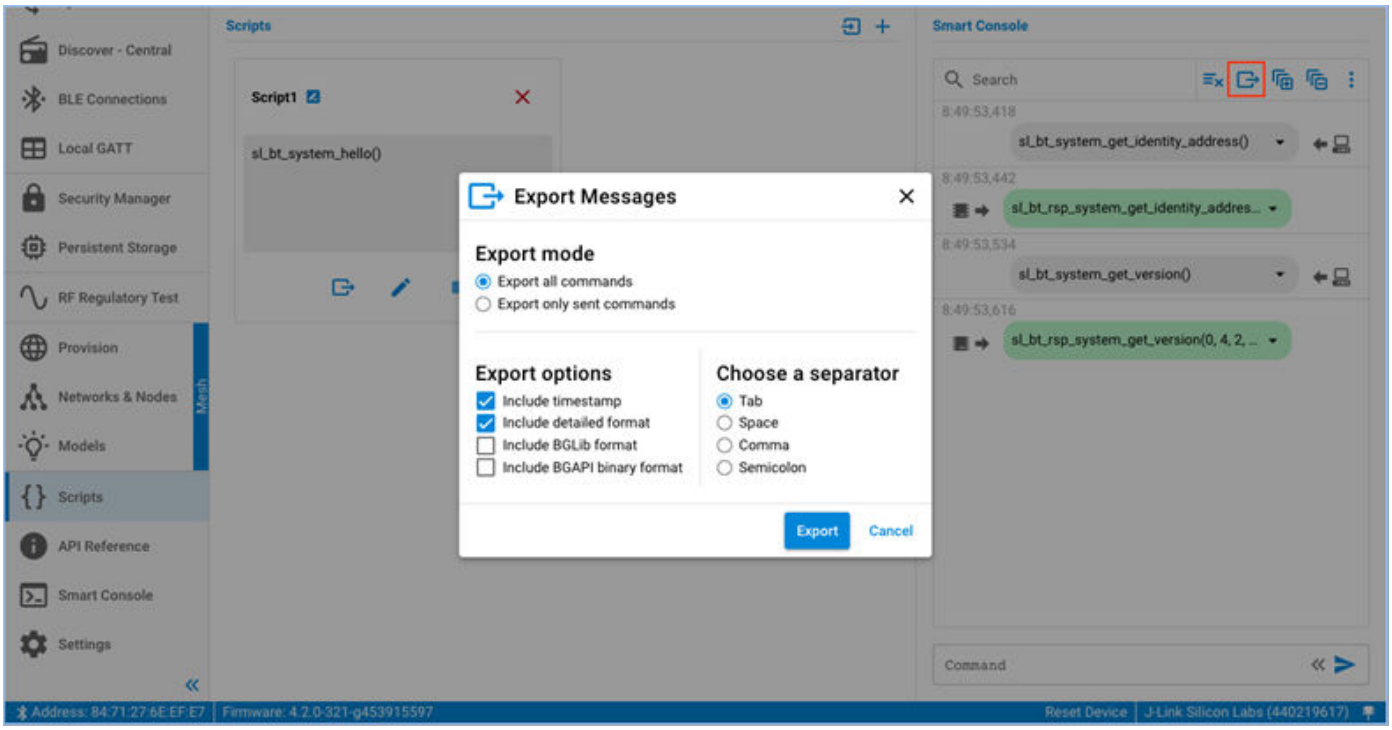
10. It is also possible to synchronize to periodic advertisement trains. To do this, click **Synchronization** on the left menu, input the Advertiser Address and advertising Set identifier, and click **Open Synchronization**.



11. NCP Commander also provides a simple scripting feature. You can create or import an existing script with the controls on the top right corner. You can use any BGAPI commands in the script, but there are no additional features, such as branching or error handling.



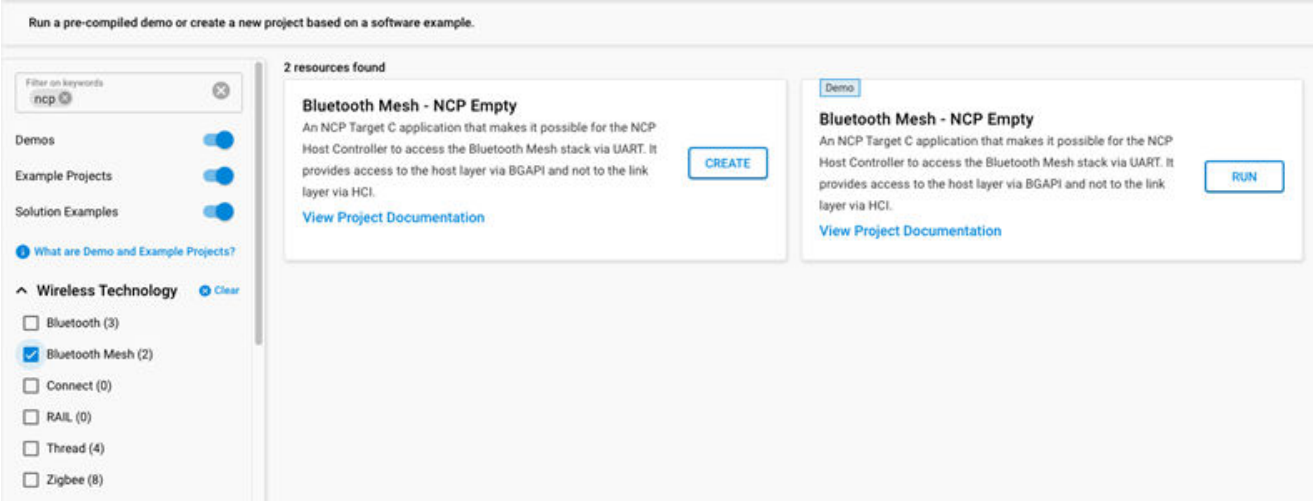
You can export the commands sent in the Smart console with the **Export** control. This saves the sent commands to a file that can be imported back as a script. You can also export the raw script using the **Export** button under the editor.



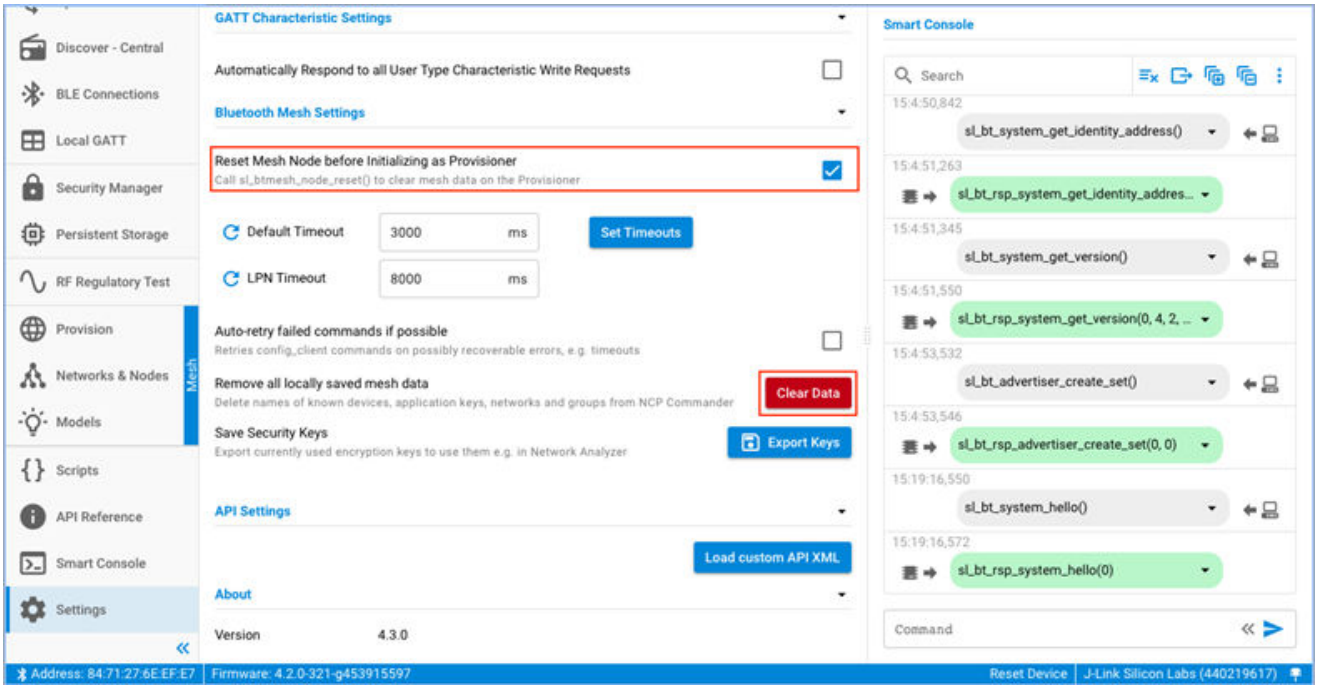
3.1.2 Host Provisioner with Bluetooth NCP Commander

Bluetooth NCP Commander also supports Bluetooth mesh features. You can issue Bluetooth mesh commands manually in the command box of Smart Console or use the host provisioner feature from the left menu. You can use the feature to provision and configure mesh nodes and to manage mesh networks rather than using a Bluetooth Mesh mobile application.

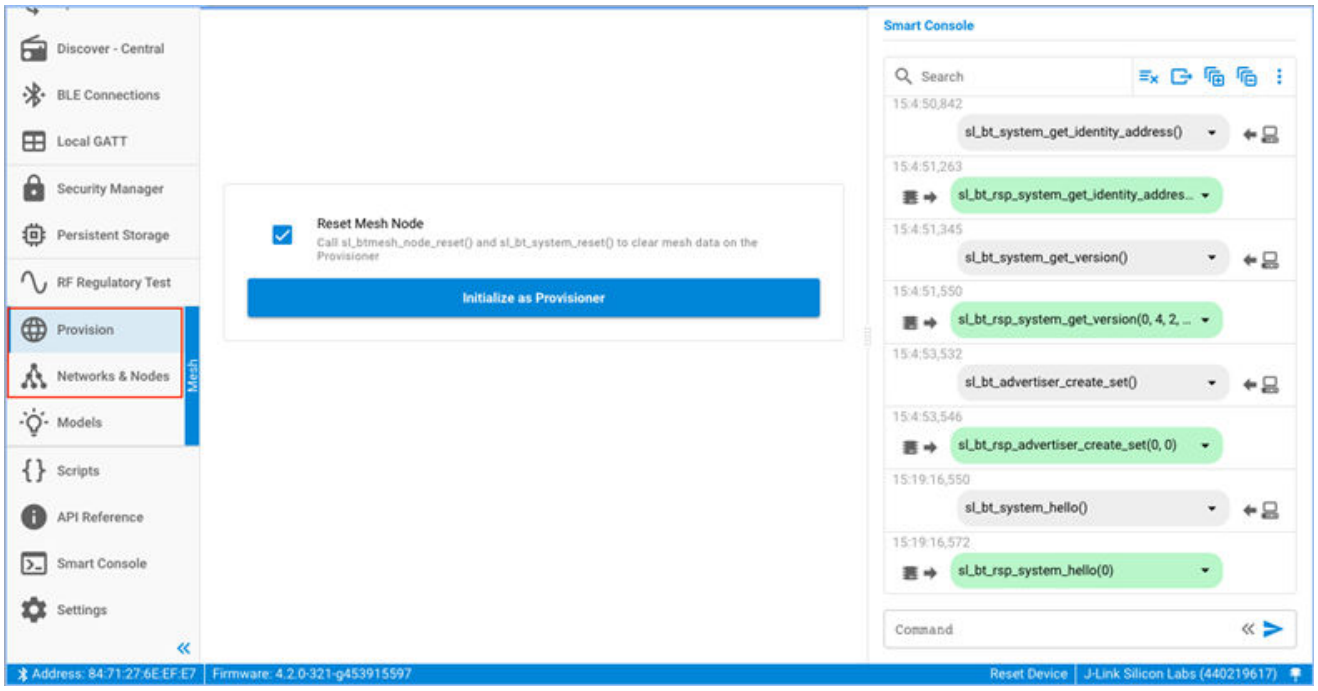
To use the Bluetooth Mesh features, create, build, and flash the device with an NCP example supporting Mesh features. Otherwise, the provisioner initialization attempt returns SL_STATUS_NOT_SUPPORTED (0x000f).



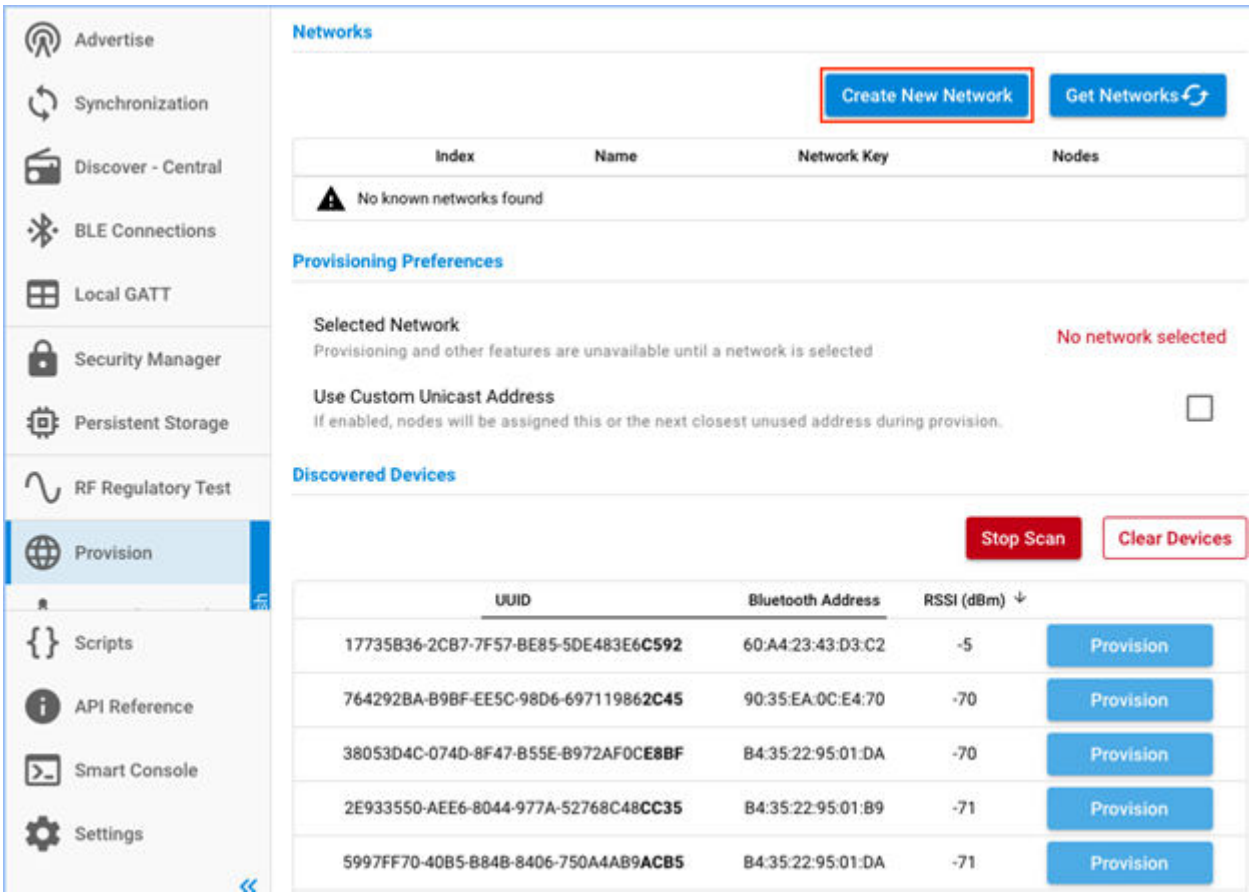
In **Settings**, if the **Reset Mesh Node before Initializing as Provisioner** option is enabled, the host provisioner does a factory reset (the **node_reset** command) on the NCP target device before initializing the node. Clicking **Clear Data** next to **Remove all locally saved mesh data** removes the network and application keys that were configured during initialization.



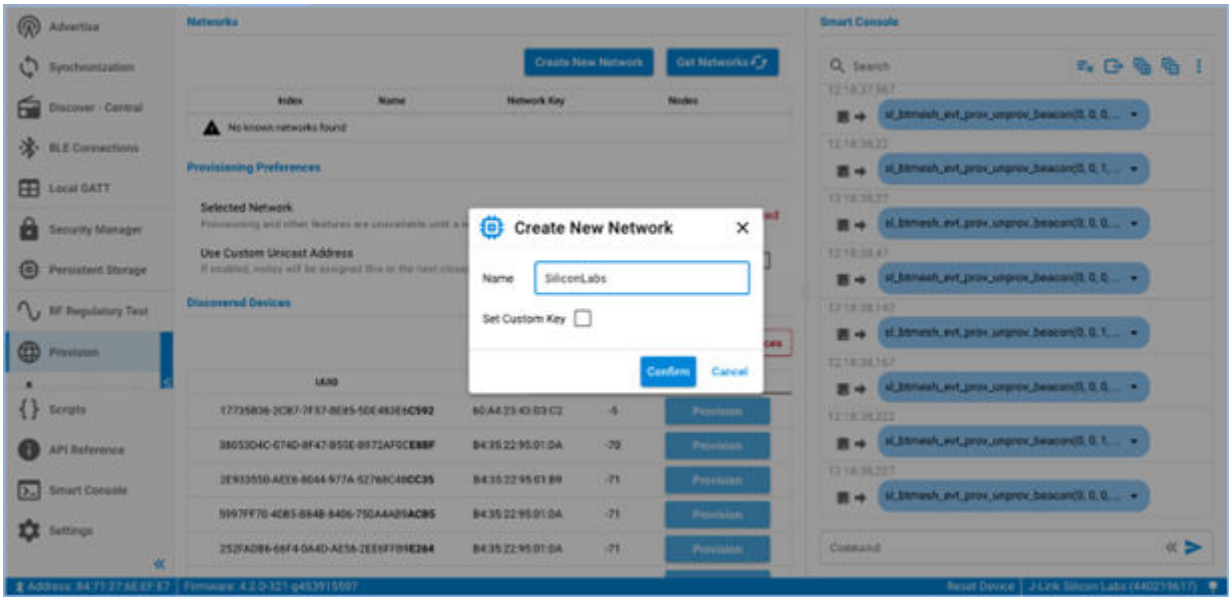
1. To start using the host provisioner, select either **Provision** or **Networks & Nodes** on the left menu, and click **Initialize as Provisioner**.



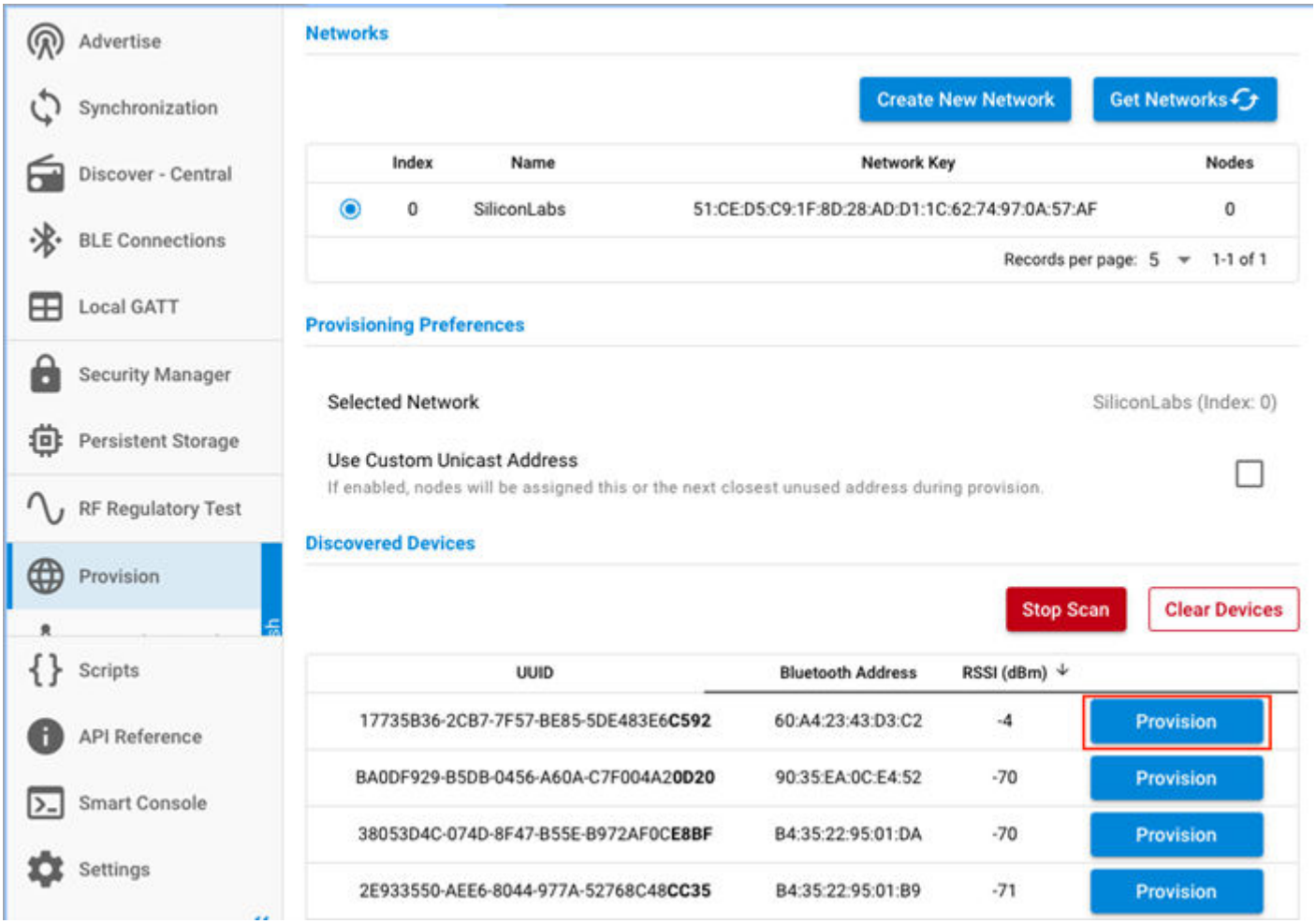
2. To provision devices, select **Provision** on the left menu and click **Start Scan** in the right panel. The devices that are transmitting unprovisioned beacons are shown in the **Discovered Devices** section. If you do not have a network from a previous configuration or have reset the provisioner node, you must create a new network with **Create New Network**.



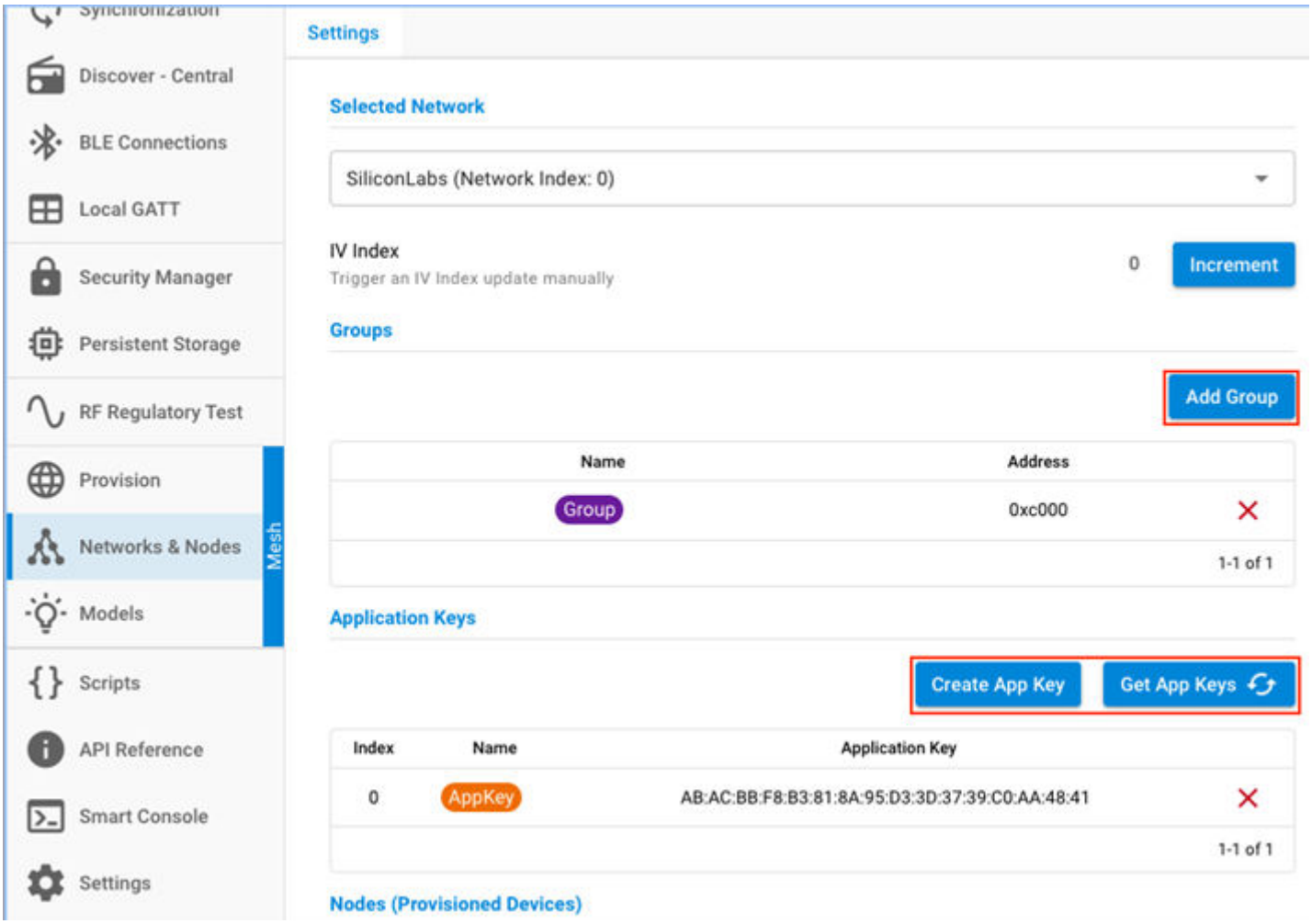
3. Enter the name of the new network and click **Confirm**.



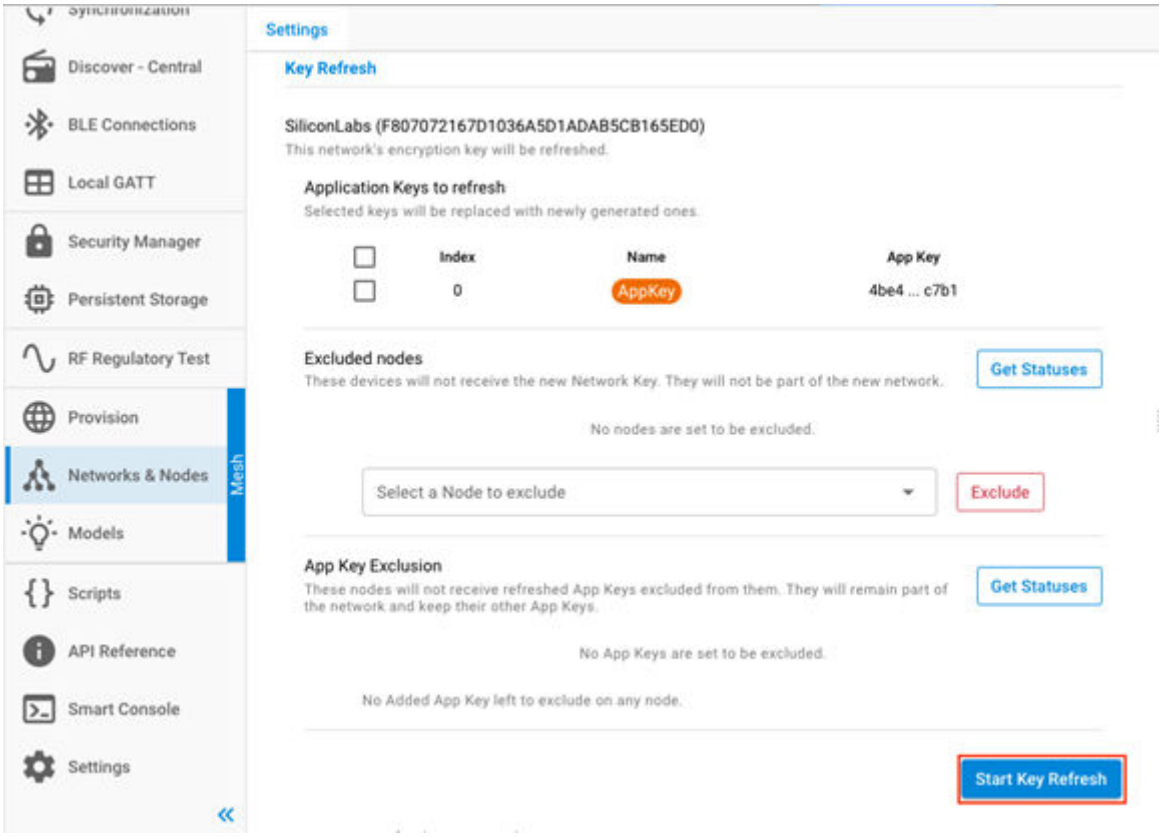
4. Click **Provision** next to the device you want to provision.



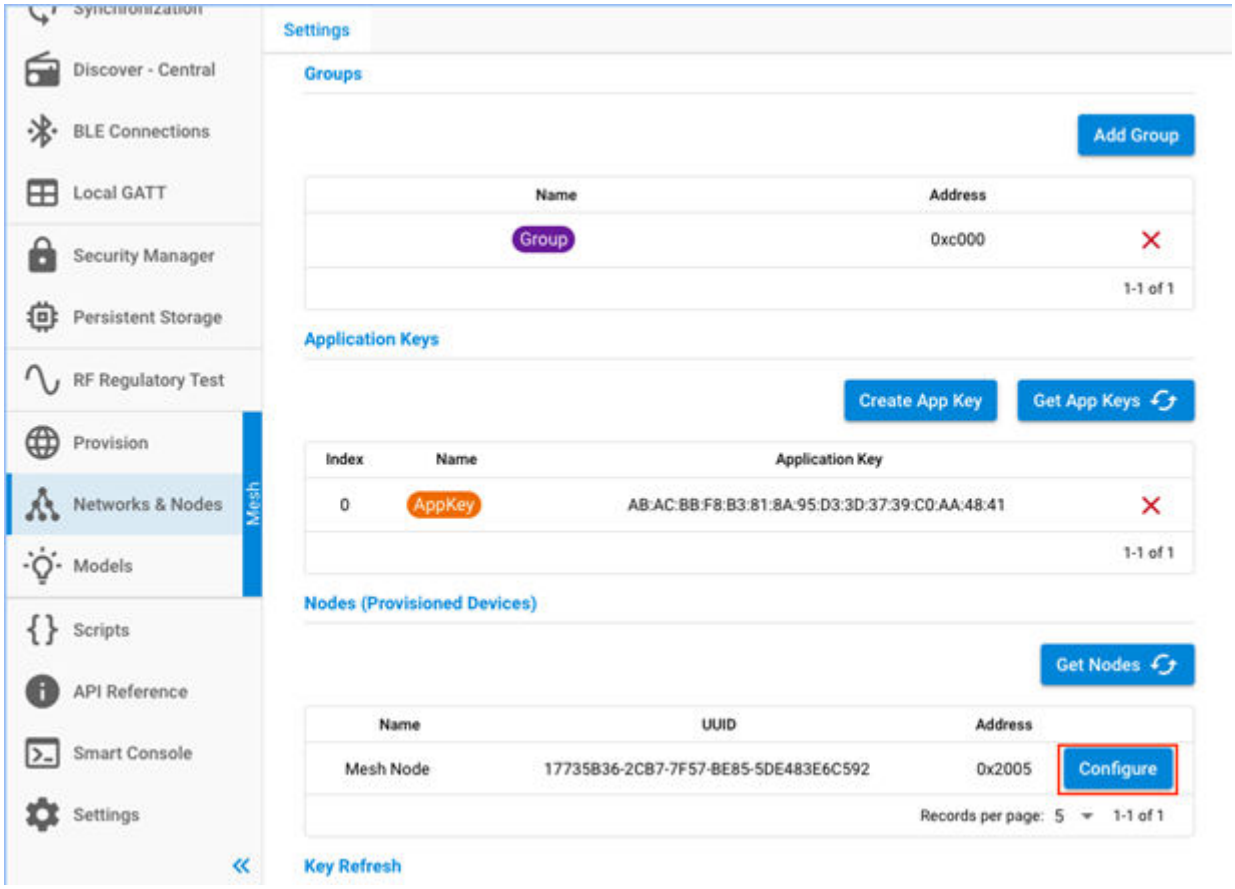
5. Before configuring devices, you may need to create application keys and groups. Application keys, groups, and other network settings can be managed in the **Settings** tab of the **Networks & Nodes** menu item. To create an application key, click **Create App Key**, name the key, and click **Confirm**. You can create as many application keys as you need. If you have created any application keys before, you can click **Get App Keys** to retrieve them. To create a group, click **Add Group**, name the group, and click **Confirm**. You can create as many groups as you need.



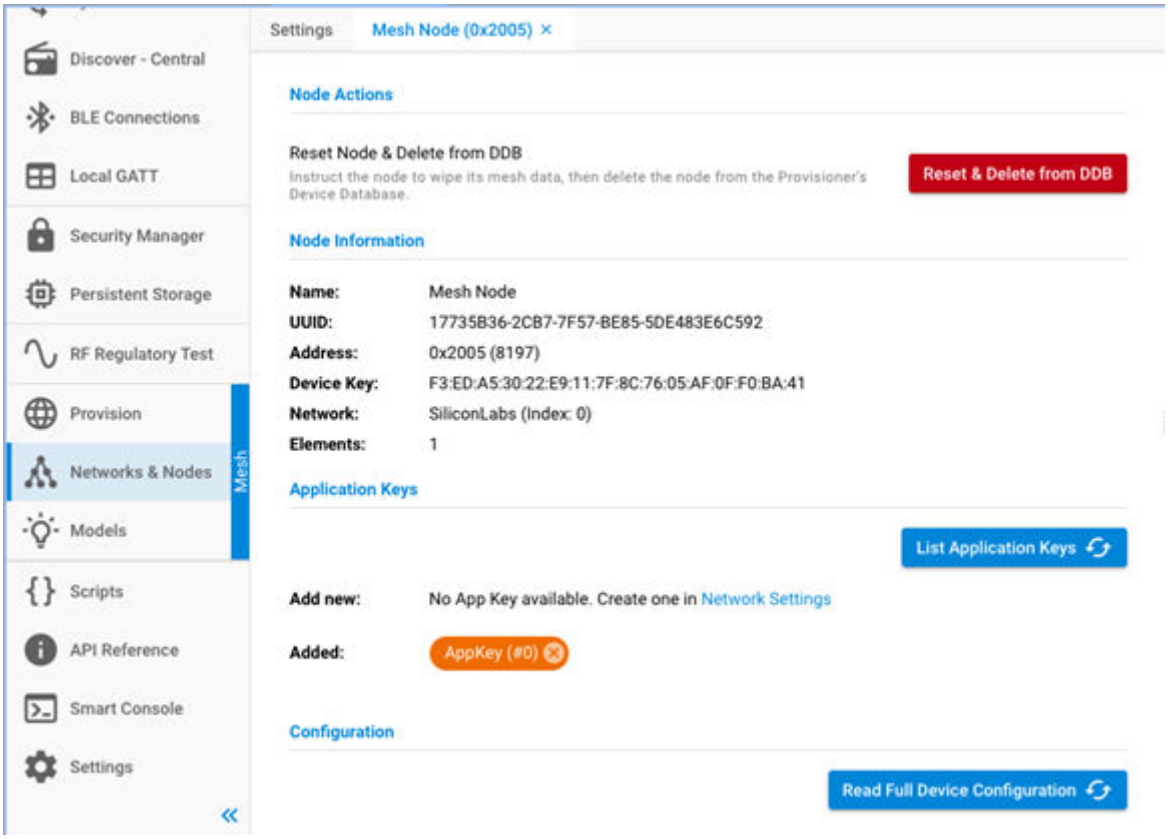
6. In the same tab, you can induce a full network-wide key refresh or exclude nodes.



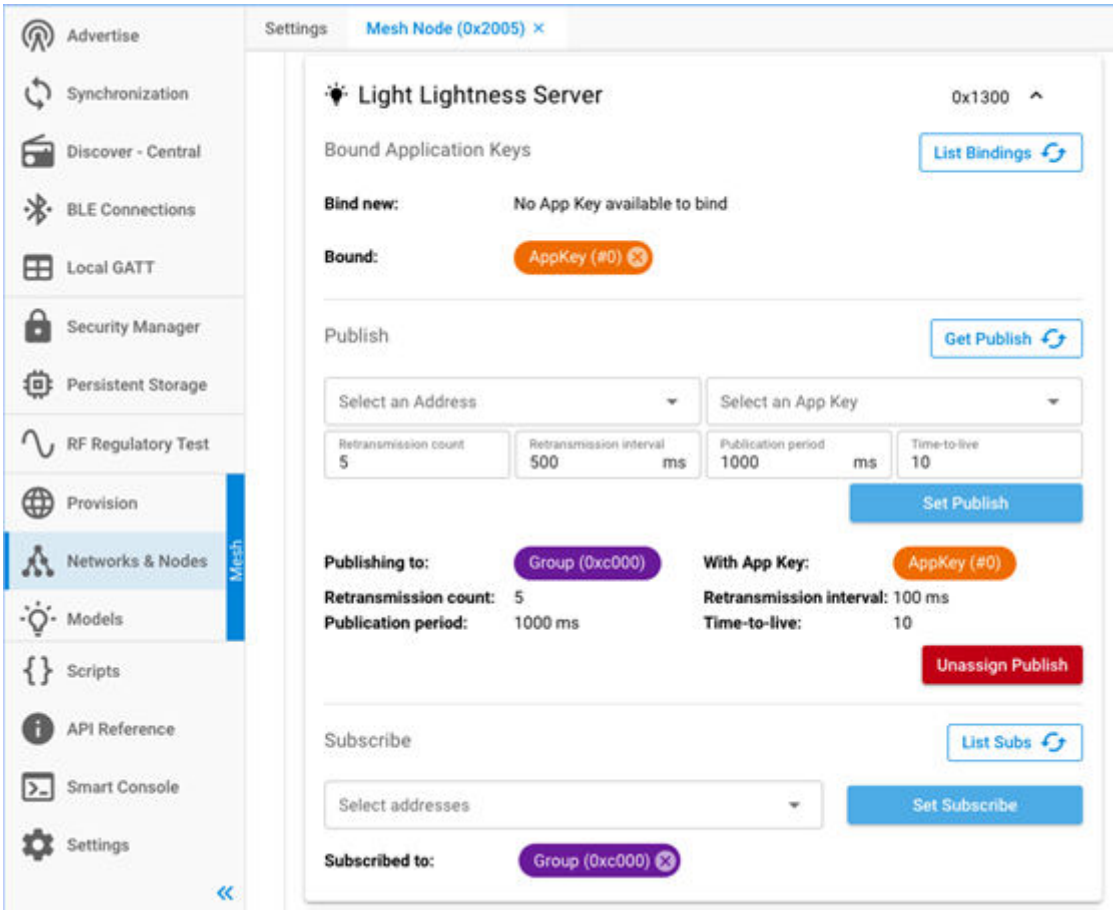
7. To configure a provisioned device, select **Networks & Nodes** on the left menu. The devices you provisioned are shown in the **Nodes (Provisioned Devices)** section of the **Settings** tab. Click **Configure** and a **Mesh Node** tab opens in which you can configure the device.



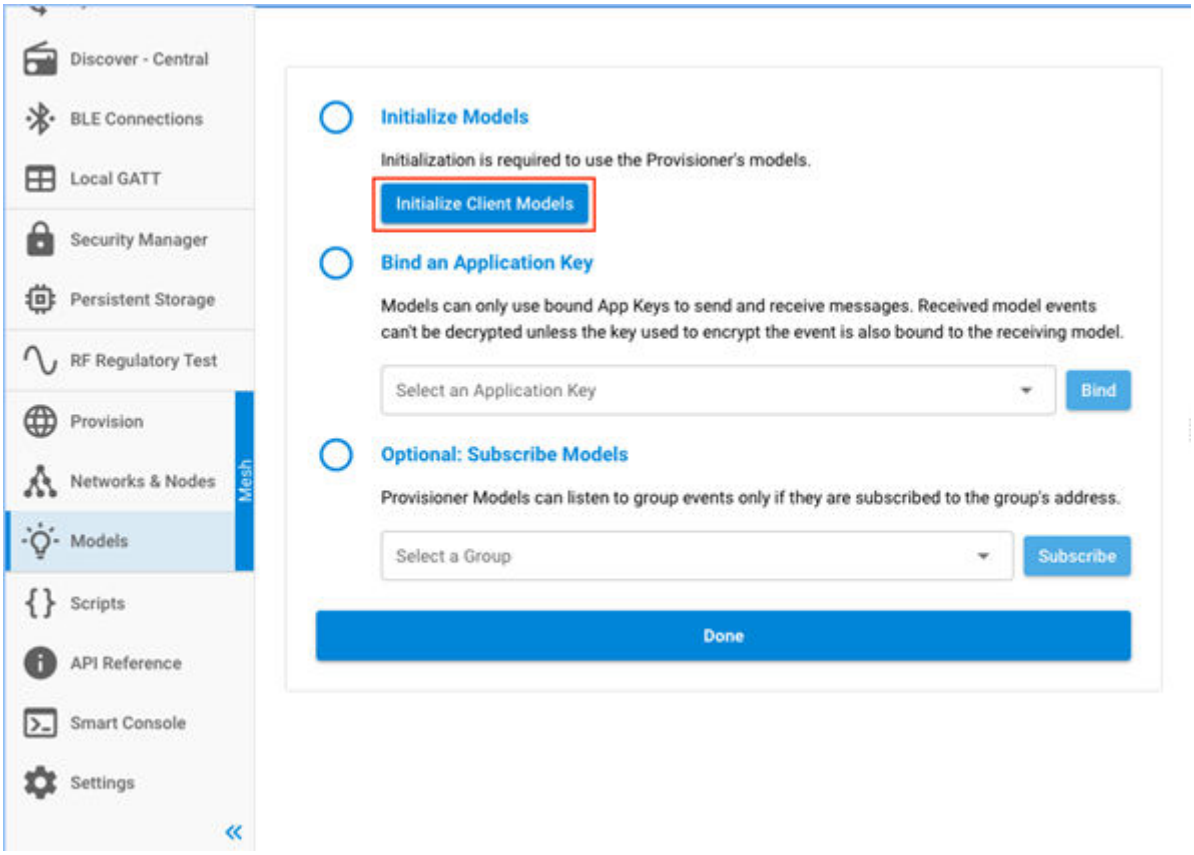
8. In the **Application Keys** section of the **Mesh Node** tab, select an application key from the drop-down list and then click **Add**.



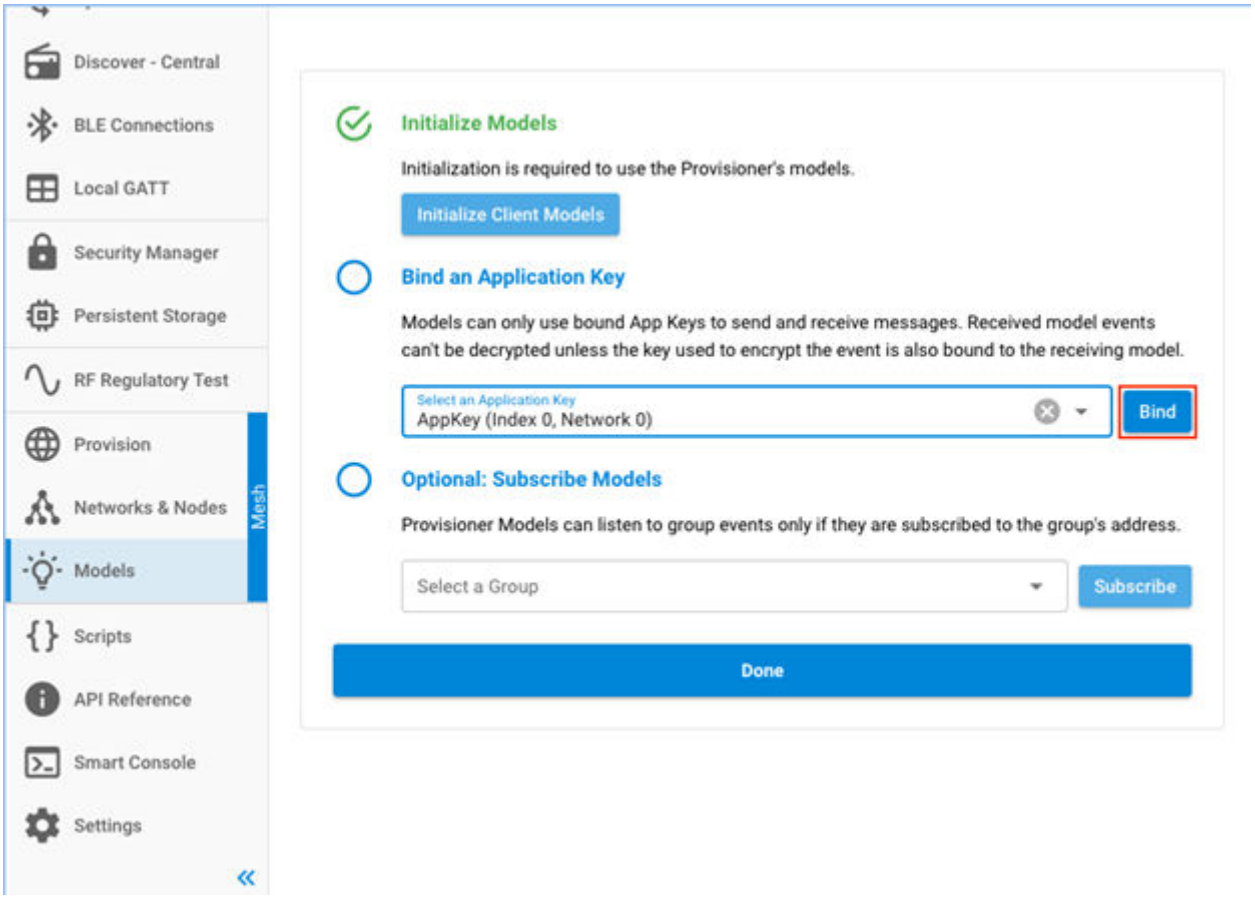
9. Click **Get DCD** to configure all the Models available on your node(s), bind to app keys, set publishing or subscription to groups, fine tune parameters, and so on.



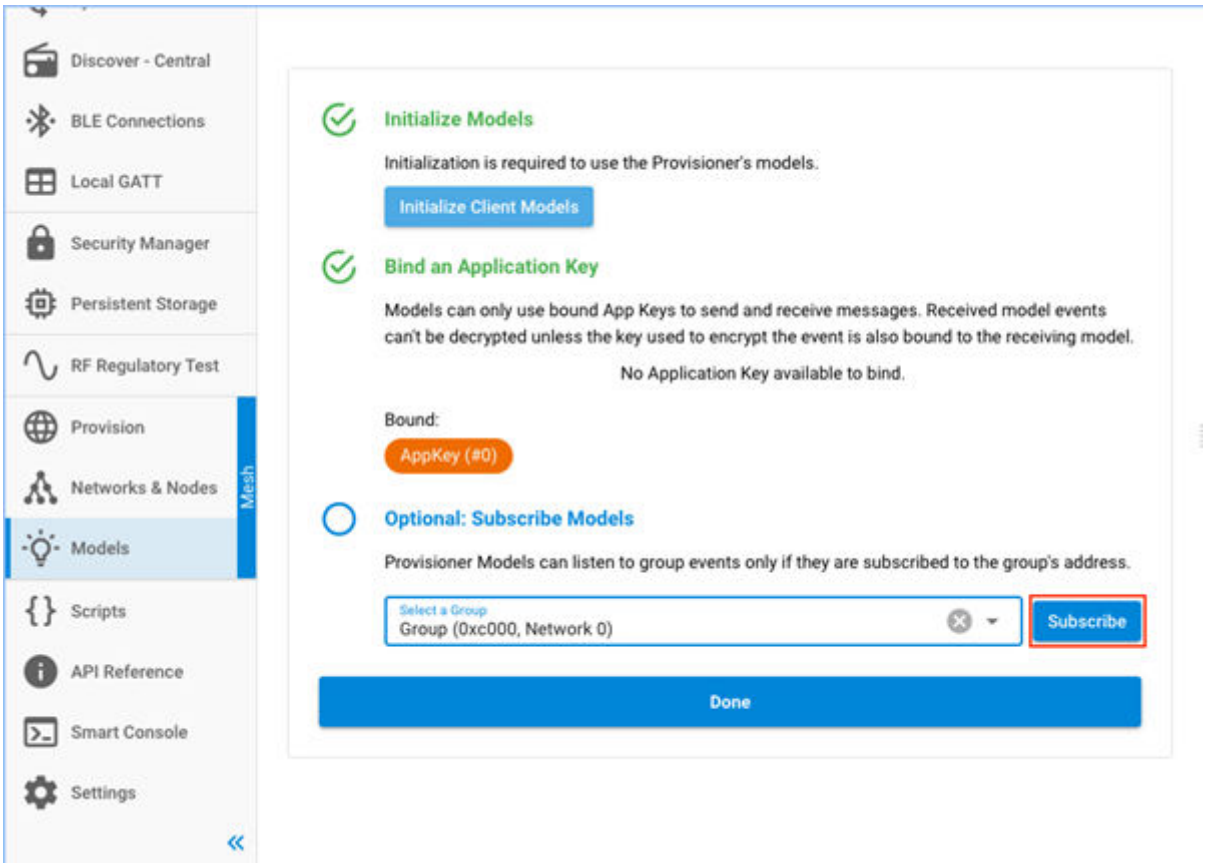
10. To configure the Provisioner, the Models must first be initialized using **Initialize Client Models**.



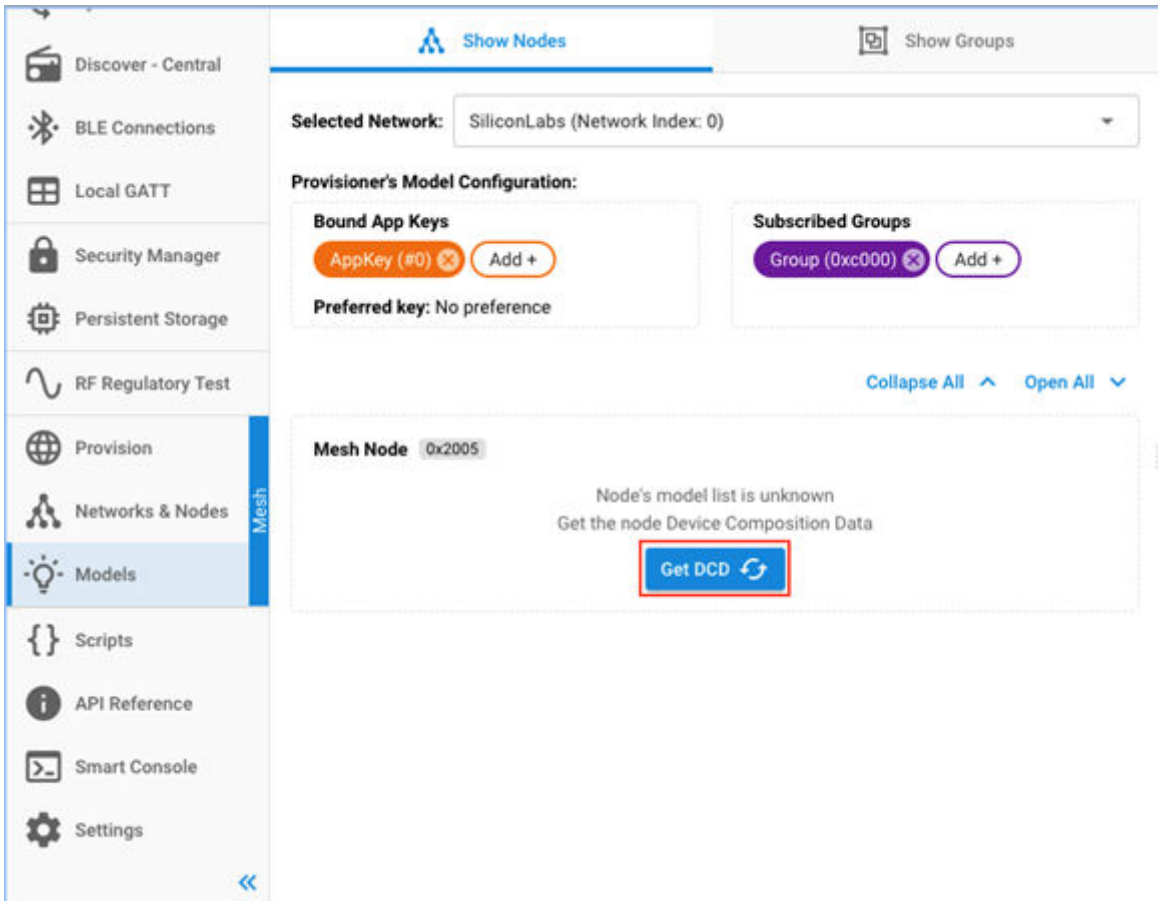
11. The Application Key must be bound to the Models in order for them to decrypt received messages. Press **Bind**.



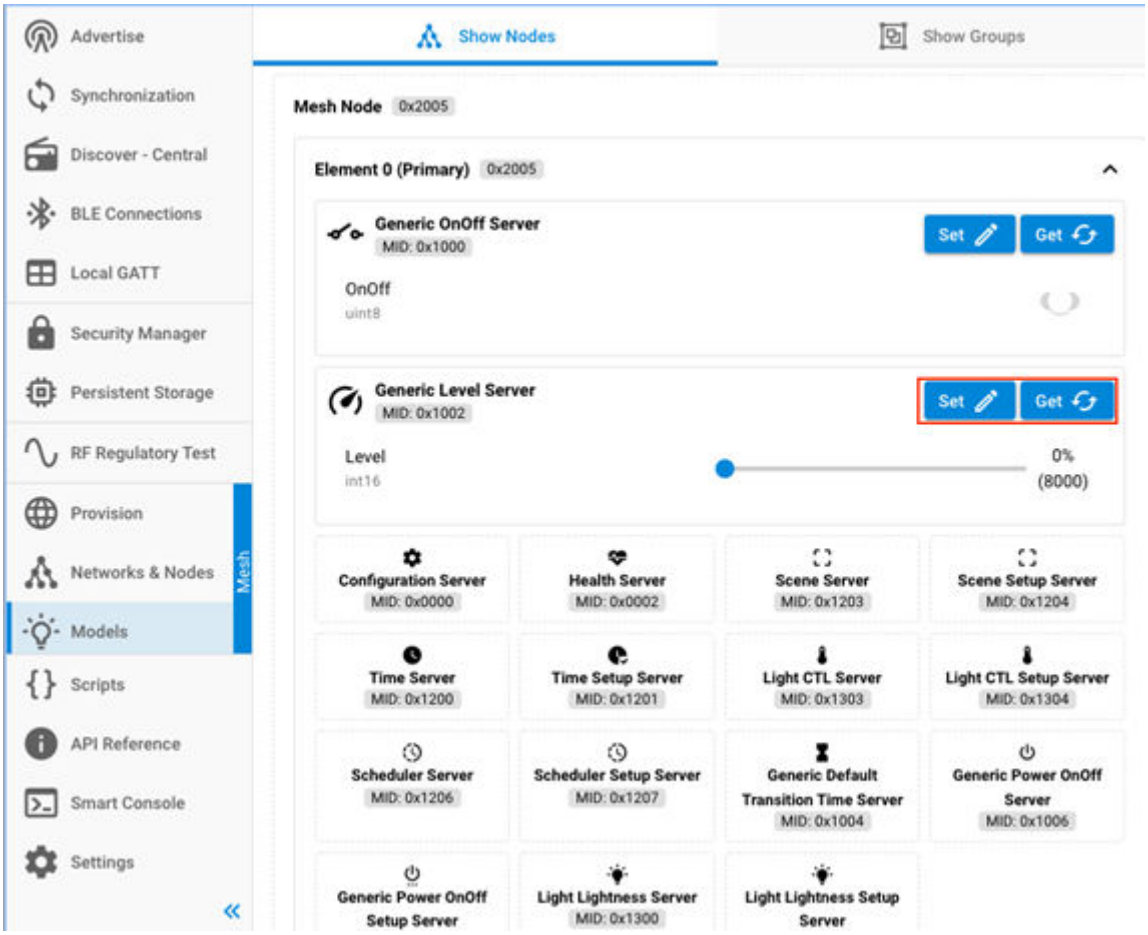
12. To subscribe the Models to the messages of the recently created Group (optional), select the chosen Group from the dropdown and click **Subscribe**.



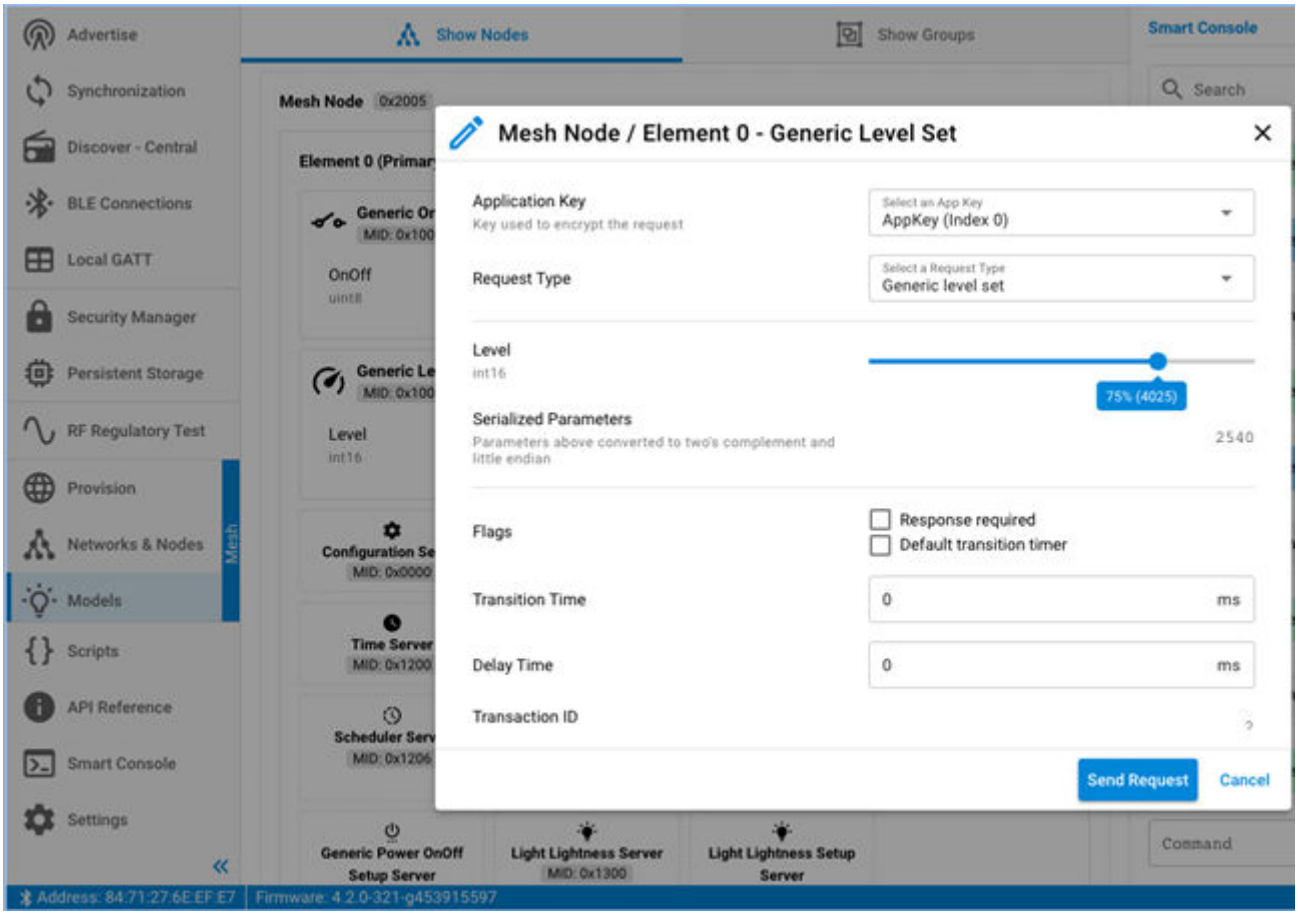
13. When configuration is complete, click **Done**. The **Show Nodes** tab is displayed, where you can **Get DCD** of the provisioned Node(s).



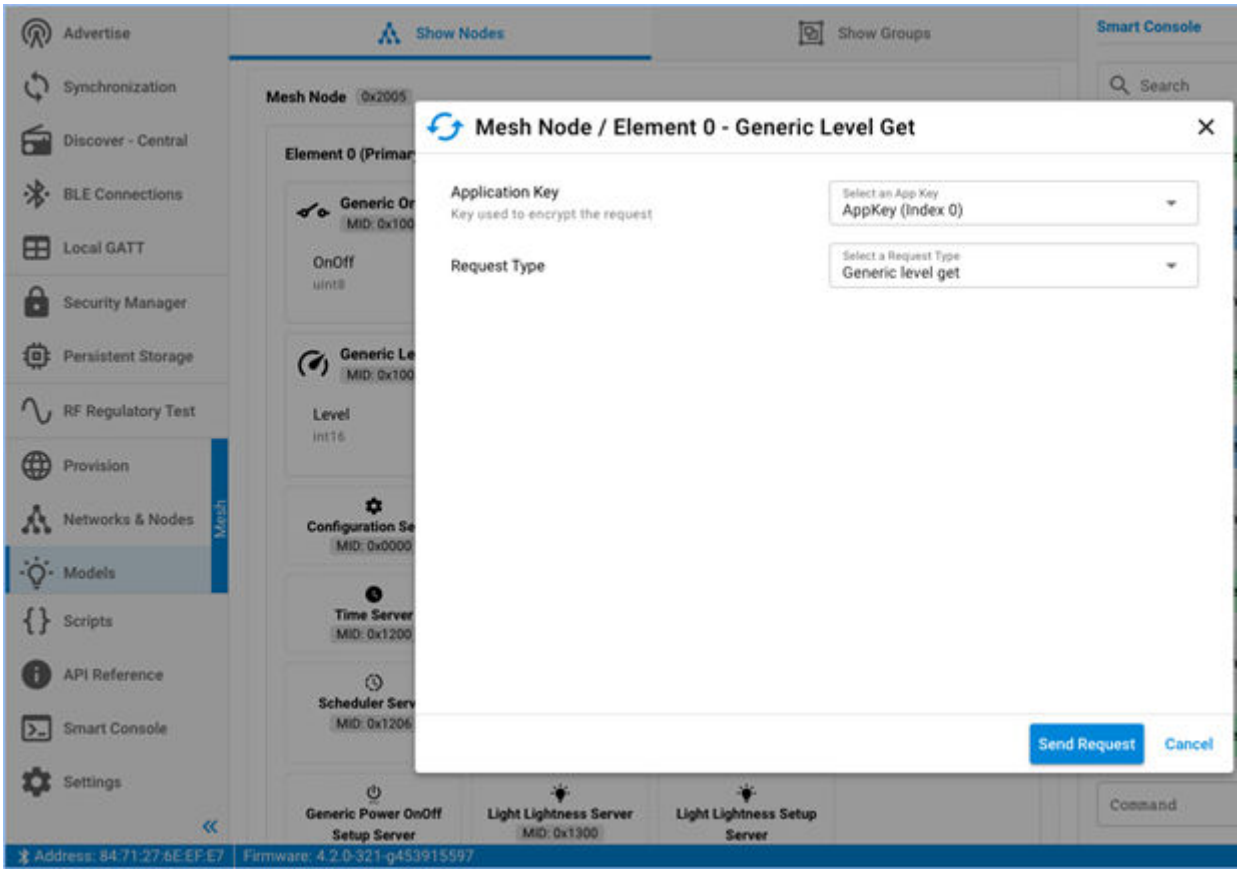
14. After listing, you can **Get** or **Set** the Server states of the Node(s).



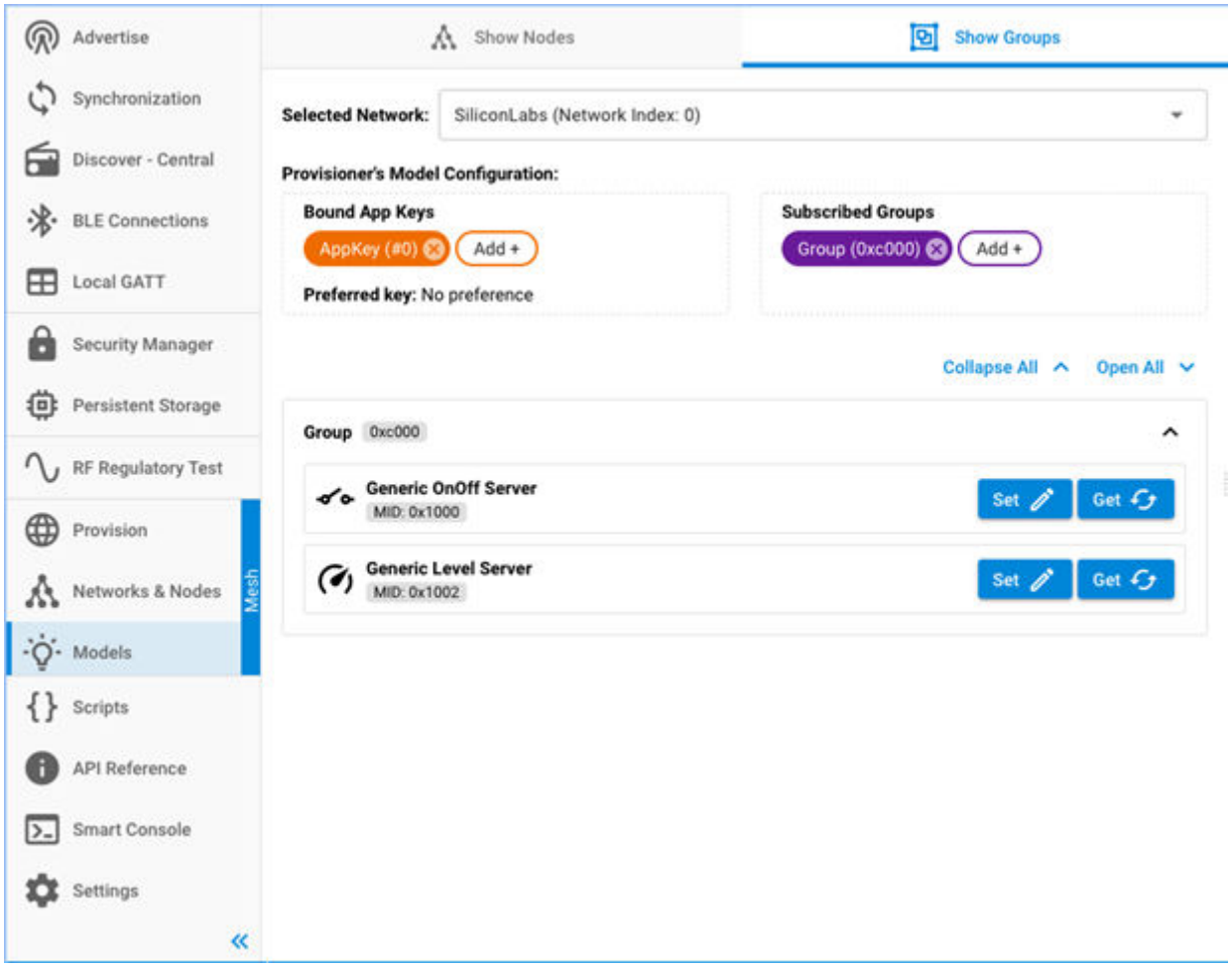
15. Click Set to set the current state of the selected Server of the selected Node.



16. Click **Get** to get the current state of the selected Server of the selected Node.



17. On the **Show Groups** tab, you can **Set** or **Get** the Server Model(s) states on a Group level.



3.2 Building the NCP Host Examples on Windows

The Silicon Labs v3.x Bluetooth SDK contains a generic NCP Host example project for the PC. This example can be compiled on Windows or any POSIX OS. This section goes through the build process on Windows.

Note: The host example projects in the SDK use the dynamic GATT database feature. They are to be used with the **Bluetooth – NCP** target application.

1. To build the examples properly, the MSYS2 development toolchain must be installed on your PC. Download MSYS2 at https://www.msys2.org/.
2. After MSYS2 is installed, update the package database as described at https://www.msys2.org/.
3. Start MSYS2 bash and install mingw-64 with the following command:

```
pacman -S make mingw-w64-x86_64-gcc
```

4. Close MSYS2 and start MSYS2 MinGW 64-bit.

5. Change to the NCP Host example folder, where <version> varies by SDK version:

```
cd c:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\v3.x\app\bluetooth\example_host\bt_host_empty\
```

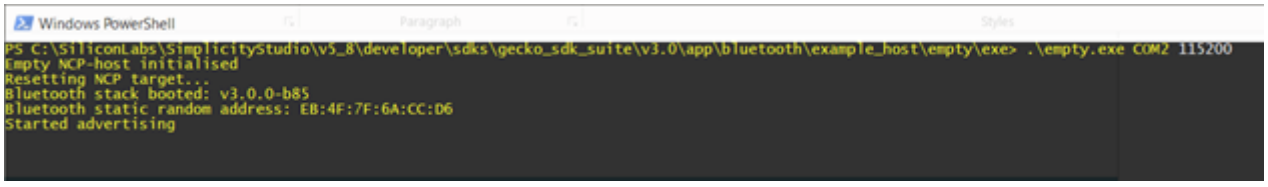
or

```
cd c:\Users\<username>\SimplicityStudio\SDKs\gecko_sdk\app
```

6. Create an export of the example with the command `make export`. After the project files are exported, the export directory will be a working directory that is completely detached from the SDK but has the same folder structure inside. The benefit of using an export is that changes in the (config) files during development will not affect the SDK content, and multiple instances can coexist, for example for testing different variants. You can also use `make export EXPORT_DIR=/my/custom/export/path` to export the example to a custom directory.
7. Within the export folder navigate to the */app/bluetooth/example_host/bt_host_empty* folder.
8. If you want to add any service/characteristic to the GATT database, edit the */config/btconf/gatt_configuration.btconf* file. Edit it either with a text editor or drag-and-drop the file onto Simplicity Studio to edit it with the GATT Configurator. Do not forget to save the file after editing.
9. Generate GATT database source files from the *.btconf* file by running `make gattdb` (in the */bt_host_empty* folder). Note: The generator script requires installing Python 3 and the Jinja2 package by calling `pip install jinja2`.
10. Build the exported project with the command: `make`. (Run it in the */bt_host_empty* folder, where you can find the makefile).
11. The build output is created in a new *exe* folder. Go to this folder with `cd exe`, and then run`bt_host_empty.exe`. The COM port and the IP address of the target are passed as command line parameters. The COM port should be the same as the one used by the JLink CDC UART Port, as shown in section 3. NCP Host Development. To see how to pass the different parameters, first run the exe with the `-h` (help) switch.

```
.\bt_host_empty.exe -h
```

12. Once the UART connection with the device is established, you should see the following:



13. Now you can connect to the device over Bluetooth.

3.3 Using Python for Host Side Development

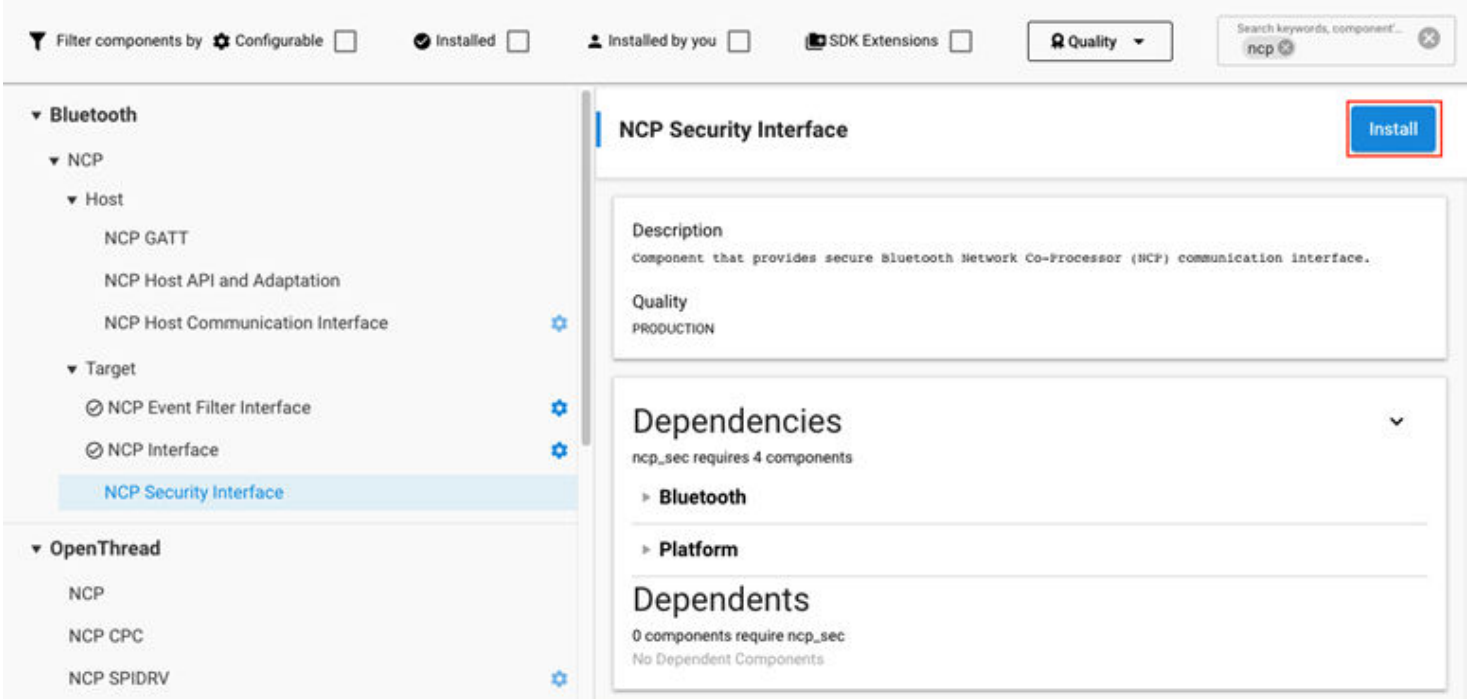
You can also implement a host application using Python. A Python package is available at https://pypi.org/project/pybgapi/. This package parses the API description file of the Bluetooth SDK and makes it possible to issue BGAPI commands and get BGAPI events in the Python environment. See the referred website for further documentation.

4. Secure NCP

Secure NCP secures communication between the NCP Host and target by encrypting the commands, events, and any data transmitted between the target and the host.

4.1 Target Side

To enable this feature on the target side, install the NCP Security Interface component.



By default, the NCP target boots without using this encryption. It will be requested by the Host part, and after the security is increased, only encrypted messages are sent and accepted by the target.

4.2 Host Side

To build the NCP Host project with secure mode, use the following command:

```
make SECURITY=1
```

This requires the openssl package to be installed. Install it to your MSYS2 environment with:

```
pacman -S mingw-w64-x86_64-openssl
```

After the project is built, the encryption can be enabled by calling the .exe file with the command line parameter `-s`:

```
.\empty.exe -s
```

```
$ ./empty.exe -u COM21 -s
[I] NCP host initialised.
[I] Resetting NCP target...
[I] Press Ctrl+C to quit

[I] Start encryption
[I] Communication encrypted
[I] Bluetooth stack booted: v3.2.1-b216
[I] Bluetooth public device address: 00:0B:57:A7:84:15
[I] Started advertising.
```

Running the exe file without this option will start a normal NCP Host application without encryption.

5. Using NCP with CPC (Co-Processor Communication)

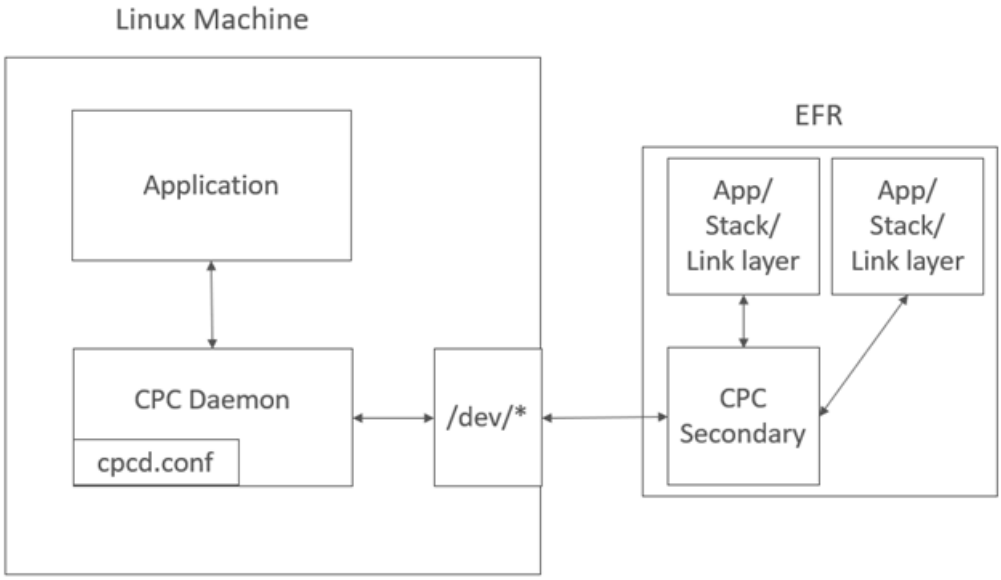
5.1 Co-Processor Communication Overview

The purpose of the Co-Processor Communication (CPC) Protocol is to act as a serial link multiplexer that allows data sent from multiple applications to be transported over a secure shared physical link. In CPC, data transfers between processors are segmented in sequential packets over endpoints. Transfers are guaranteed to be error-free and sent in order.

Find more information about the CPC at https://docs.silabs.com/gecko-platform/4.1/service/cpc/overview.

5.2 Usage

The CPC daemon acts as a bridge between the host and the target application. It was designed to make a reliable connection between two ends through UART or SPI. Reliability is achieved by an HDLC-like header and CPC. It offers multi-channel communication, and security is turned on by default. It is a connection-based protocol, so that if a message arrives incorrectly, it notifies the other end, which then can re-send that message.



The NCP host by default does not contain usage of CPC. You need to build the application with the command line option `CPC=1`.

5.3 Use Cases

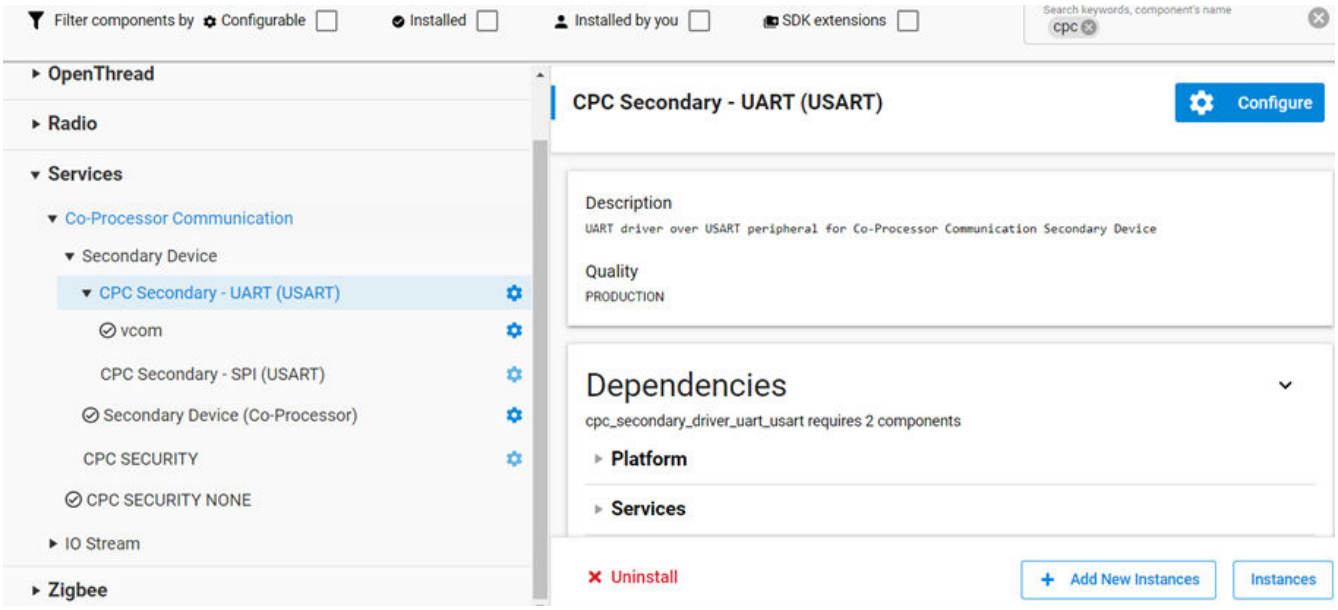
Adding the CPC functionality is recommended for the following use cases, as they cannot be used with the simple UART interface:

- Using SPI as the transport layer: SPI communication is only supported with CPC.
- DMP projects: the CPC protocol contains a multiplexer, which makes it possible to use the same interface for different applications.

5.4 Building the Target

To make the target use CPC communication, replace the USART component in the **bt_ncp** sample application with **CPC Secondary - UART (USART)** or **CPC Secondary – SPI (USART)**. This adds all the necessary components to enable CPC communication on the target. Set the pins of the selected communication interface according to the hardware design of the project.

The encryption of the communication is enabled by default. For developing and debugging, Silicon Labs recommends adding the **CPC SECURITY NONE** component so that the packet traces can be easier analyzed.



5.5 Host Side

Step 1: Build the CPC daemon

Download the code for the CPC daemon and follow the instructions to build it from https://github.com/SiliconLabs/cpc-daemon.

After the build is finished, open the *cpcd.conf* file and set the **bus_type**, and configure the pins and bitrate according to the settings on the Secondary side.

If the **CPC SECURITY NONE** component was added to the target, set **disable_encryption** to true.

Step 2: Build the host application

Find the *ncp_host_bt.mk* file in the `<SDK folder>/app/Bluetooth/component_host/` folder, and set `CPC_DIR` to the path of the CPC daemon folder on your machine.

Next, go to the **bt_host_empty** sample application in `<SDK folder>/app/Bluetooth/example_host/bt_host_empty`, and build it with this command line option to enable CPC: `make CPC=1`.

Step 4: Run the application

Start the CPC daemon `cpcd -c ./cpcd.conf`.

Start the host application by passing the **instance_name** set in the *cpcd.conf* file: `./bt_host_empty -C cpcd_0`.

6. Example Project Walkthrough

This chapter describes the structure of the example NCP Host and Target projects, and highlights the parts that can be important if you create your own project.

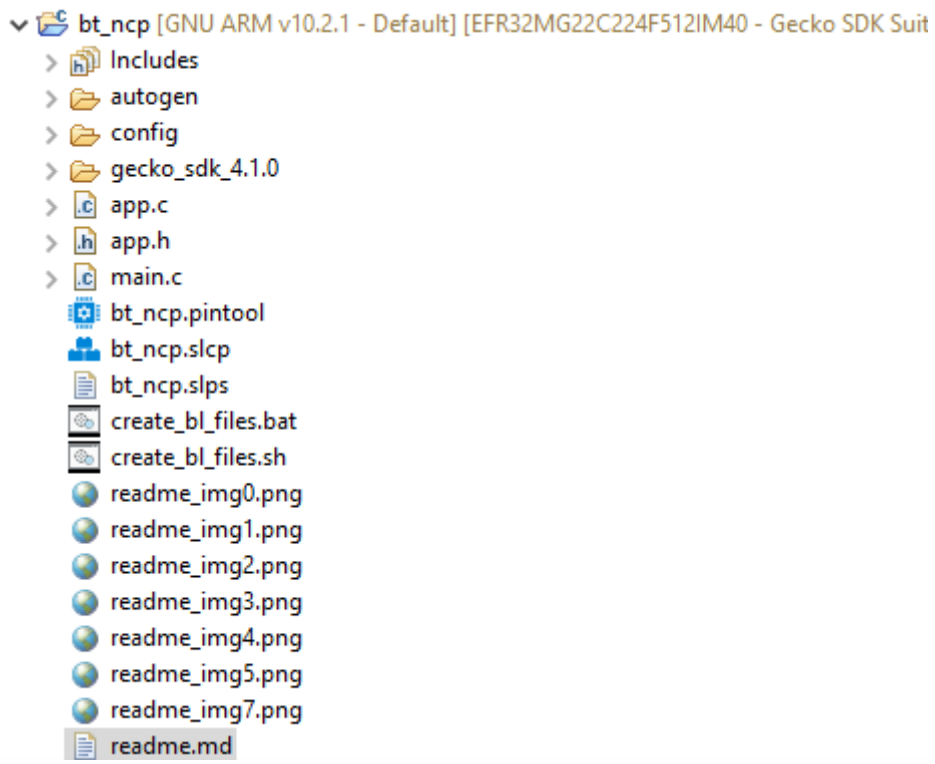
6.1 NCP Target

This section focuses on the NCP-specific part of the **Bluetooth - NCP** SSv5 project. You can find a general project description in *UG434: Silicon Labs Bluetooth® C Application Developers Guide for SDK v3.x*.

The **Bluetooth - NCP** example does not contain a GATT database. The dynamic GATT API can be used for building it. This is recommended because the target code does not need to be modified and synchronized with the Host code when the GATT database is updated.

6.1.1 Project File Structure

A common directory and file structure are used across all examples in the Bluetooth SDK v3.x. The following figure shows this layout.

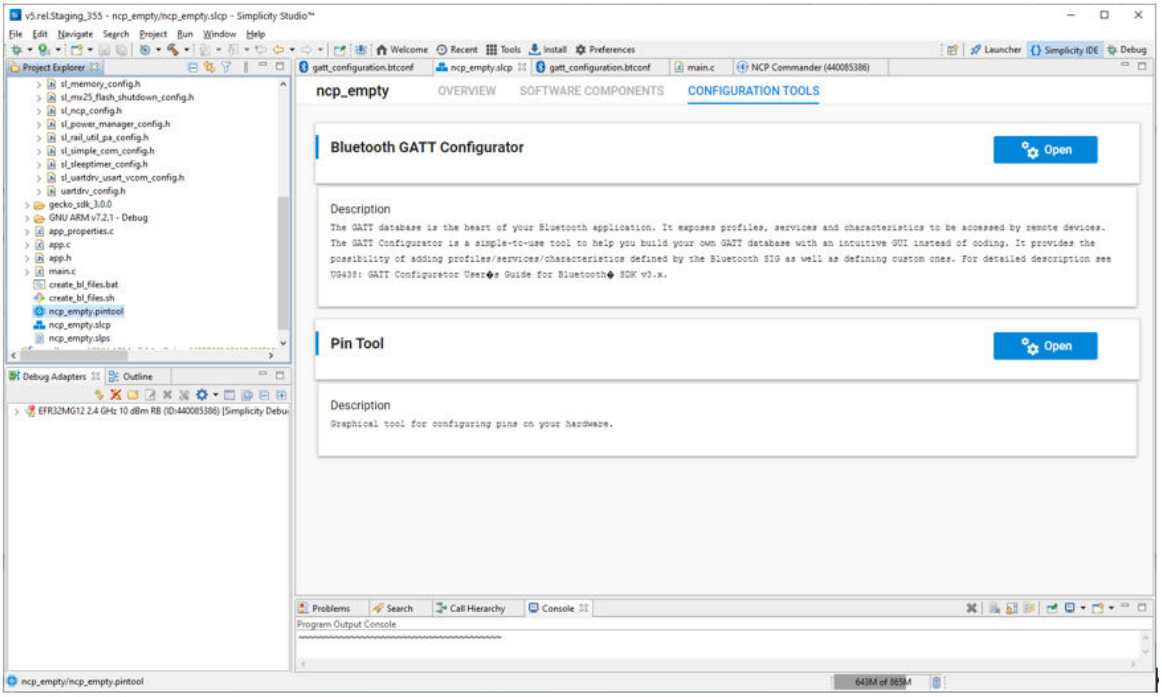


These files and directories are present in the root directory of the project:

- *main.c* and *app.c/h* – the C application code
- *bt_ncp.pintool* – the hardware configuration file and user interface
- *bt_ncp.slcp* – the component configuration and user interface
- *bt_ncp.slps* – the project properties XML file
- *GNU ARM v<X.Y.Z>* – the build directory
- *gecko.sdk_3.<X.Y>* – the Bluetooth SDK source code
- *config* – the C configuration files of the hardware and Bluetooth stack. This directory contains the output files of the Pin Tool and Component Manager.
- autogen – the C configuration code of the application. This directory typically contains the stack definition and initialization C files, as well as the generated GATT database C declaration files (*gatt_db.c/h*).

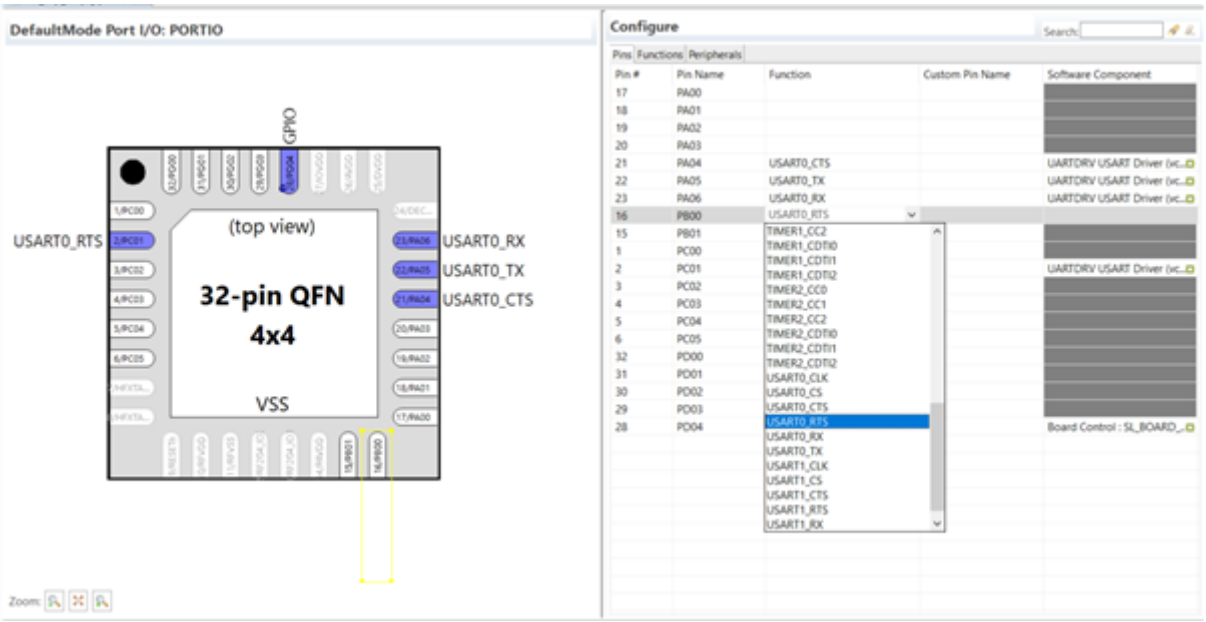
6.1.2 Pin Tool

1. Open the pin configuration tool (Pin Tool) on the project Configuration Tools tab.

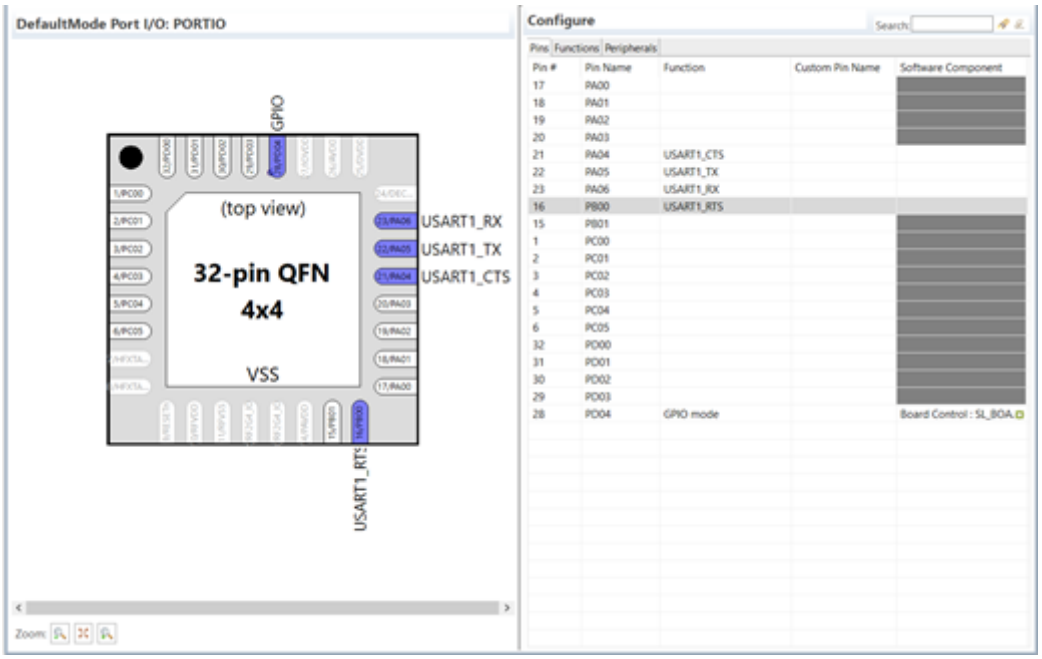


You can also double-click the *<projectname>.pintool* file in the Project Explorer view, shown highlighted in the figure above.

2. Use this tool to modify the pin configuration of the device, for example, you can reassign the pins used for USART communication to the appropriate layout for a custom board design. You do this by selecting the desired pin in the list and then selecting its functionality from the drop-down list.



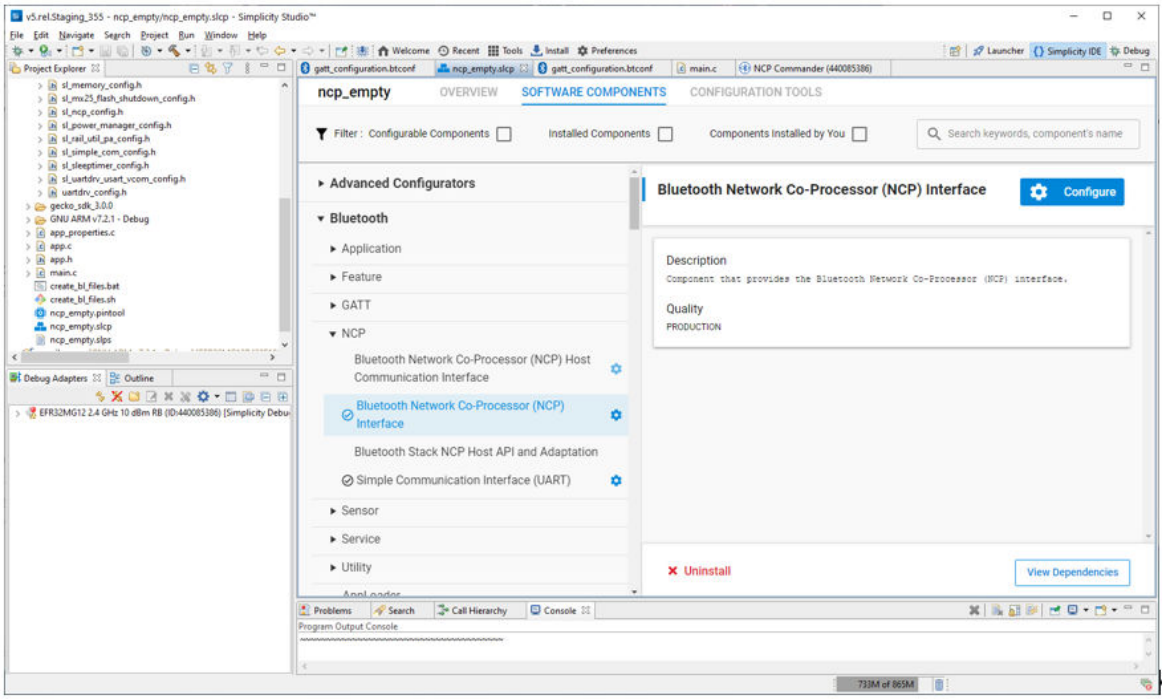
3. After clicking the selected item, the layout is updated. After saving the file, the configuration source codes are automatically generated.



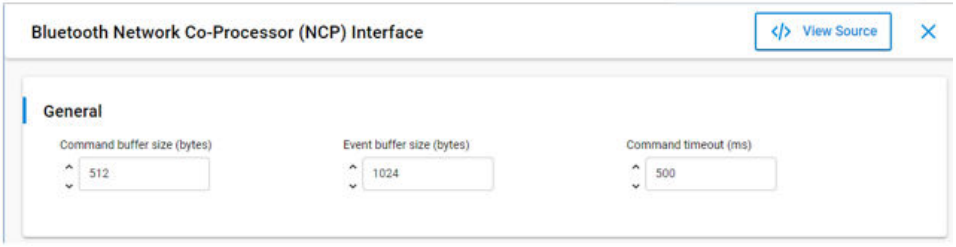
6.1.3 Project Configurator / Component Editor

You can install or uninstall components using the Project Configurator's Software Components tab. You can also configure installed components using the Component Editor. The following figures show how to change the NCP interface buffer sizes.

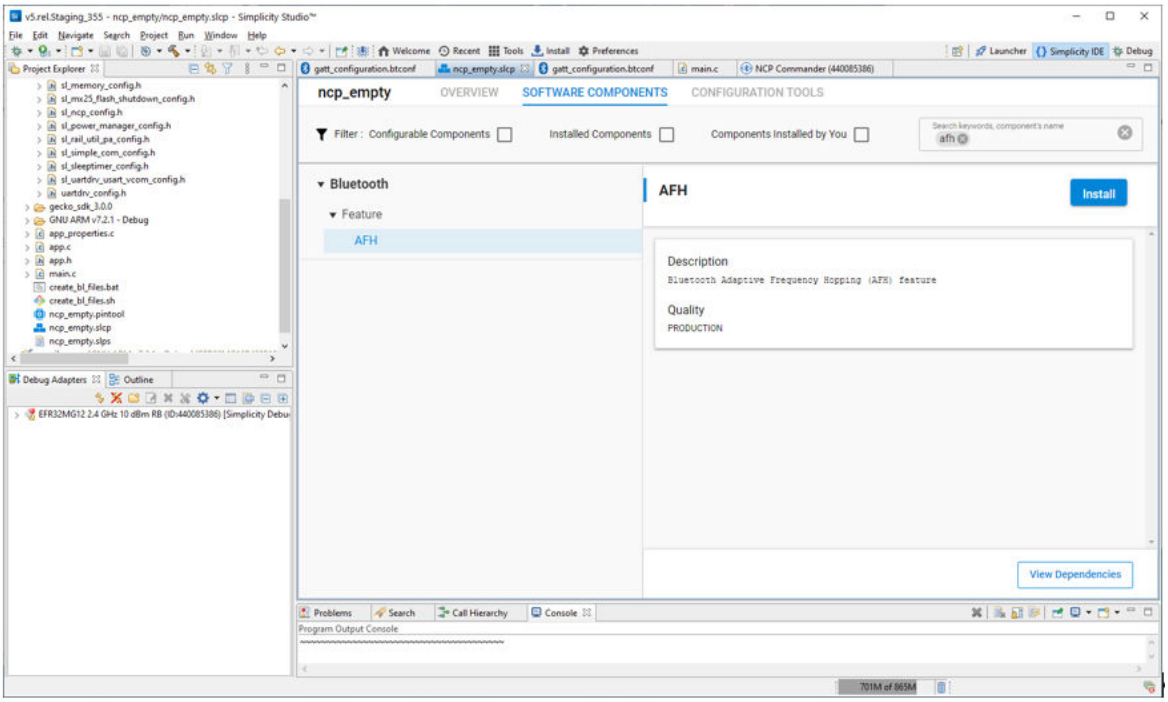
1. Select the component from the list and click **Configure**.



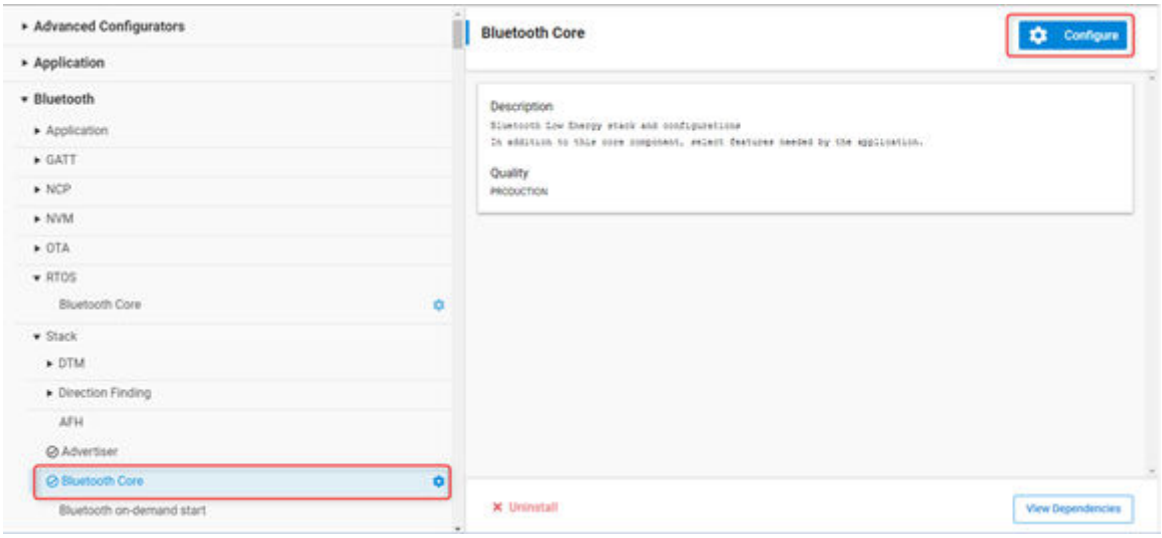
2. The Component Editor opens in a new tab with the possible configuration options. You can view the corresponding source code by clicking **Open Source**.



3. Apart from the application-specific NCP options, you can use the Project Configurator to configure the Bluetooth stack features that will be included in your project. Some advanced features are excluded from the stack by default, to save flash and memory. You can add the needed features, for example, the Adaptive Frequency Hopping (AFH) component, by clicking **Install**.



4. In many cases, you also need to change the default Bluetooth Core configuration, for example to enable more than four connections. To do so, browse for the Bluetooth Core component, and click **Configure**.



6.1.4 Enabling Hardware Flow Control

In the sample applications, Hardware Flow Control is enabled by default. On the mainboard, hardware flow control can be enabled as described in this section.

Important: If the hardware flow control settings are not the same in the SoC and mainboard, the NCP will not work.

1. Open Simplicity Studio and, in the Debug Adapters view, right-click the target device.
2. Select **Connect**.
3. Right-click the device again and select **Launch Console**.
4. Select the admin tab.
5. Set flow control with the following command:

```
WSTK> serial vcom config handshake rtscts
RTS handshake enabled
CTS handshake enabled
Serial configuration saved
```

6. Check the configuration with the following command:

```
WSTK> serial vcom
----- Virtual COM port -----
Stored port speed : 115200
Active port speed : 115226
Stored handshake : rtscts
Actual handshake : rtscts
RTS Asserted - Ready to Receive.
```

The flow can be disabled by setting the handshake parameter to `none` in step 5 above.

6.1.5 Main Walkthrough

This is a code snippet that corresponds to the `main` function. Because the Bluetooth stack and subsequent hardware are considered to be components, they are separated from the application processing that is entirely managed in *app.c/h*.

```
int main(void)
{
  // Initialize Silicon Labs device, system, service(s) and protocol stack(s).
  // Note that if the kernel is present, processing task(s) will be created by
  // this call.
  sl_system_init();

  // Initialize the application. For example, create periodic timer(s) or
  // task(s) if the kernel is present.
  app_init();

#if defined(SL_CATALOG_KERNEL_PRESENT)
  // Start the kernel. Task(s) created in app_init() will start running.
  sl_system_kernel_start();
#else // SL_CATALOG_KERNEL_PRESENT
  while (1) {
    // Do not remove this call: Silicon Labs components process action routine
    // must be called from the super loop.
    sl_system_process_action();

    // Application process.
    app_process_action();

#if defined(SL_CATALOG_POWER_MANAGER_PRESENT)
    // Let the CPU go to sleep if the system allows it.
    sl_power_manager_sleep();
#endif
  }
#endif // SL_CATALOG_KERNEL_PRESENT
   }
```

Once the USART and Bluetooth stack are initialized, the main loop continuously calls the component as well as the application state machine. The corresponding functions are `sl_system_process_action()` and `app_process_action()` respectively.

The `sl_system_process_action()` handles Silicon Labs tasks and routines. It must *not be removed* from the loop.

The default USART settings are mentioned in the Host example section. Make sure that the target and the host use the same configuration. The configuration can be adapted with the help of the Pin Tool and the Project Configurator.

6.1.6 Application Callback and Actions

Use the `app_init()` function to call application-related initializations.

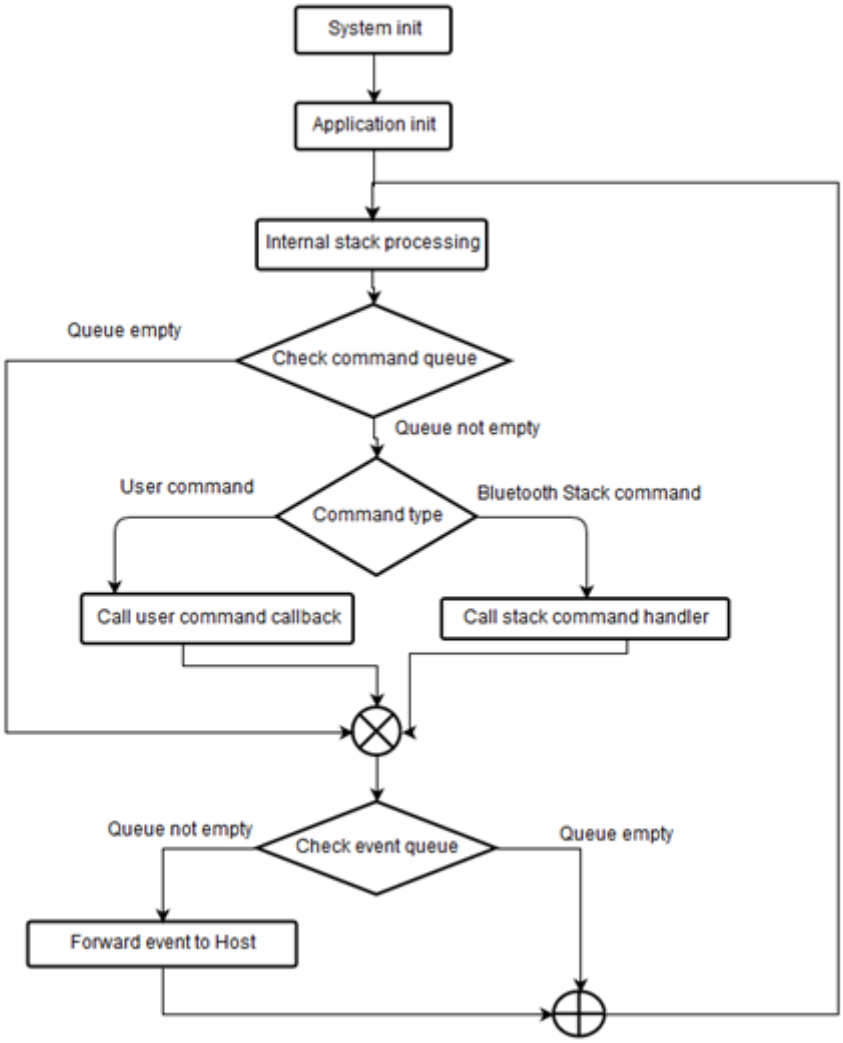
Use the `app_process_action()` function to call application-specific tasks and routines.

```
/**************************************************************************//**
 * Application Init.
 *****************************************************************************/
SL_WEAK void app_init(void)
{
  /////////////////////////////////////////////////////////////////////////////
  // Put your additional application init code here!                         //
  // This is called once during start-up.                                    //
  /////////////////////////////////////////////////////////////////////////////
}

/**************************************************************************//**
 * Application Process Action.
 *****************************************************************************/
SL_WEAK void app_process_action(void)
{
  /////////////////////////////////////////////////////////////////////////////
  // Put your additional application code here!                              //
  // This is called infinitely.                                              //
  // Do not call blocking functions from here!                               //
  /////////////////////////////////////////////////////////////////////////////
}
```

6.1.7 NCP Code Walkthrough

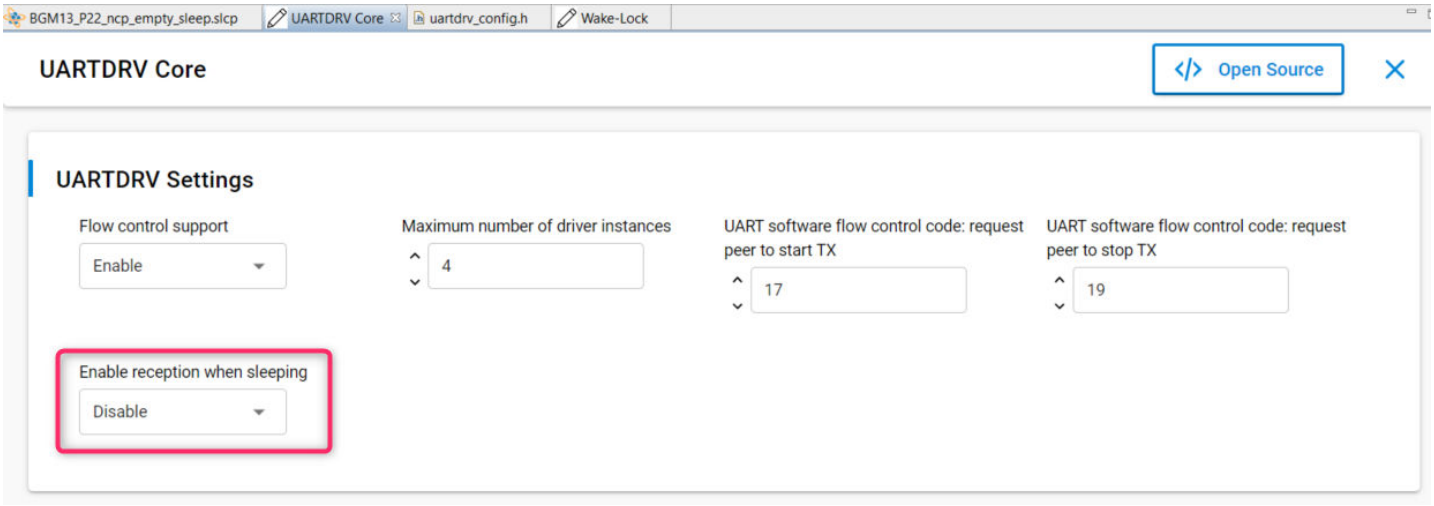
The USART communication handling is implemented in *ncp_usart.c*. Receiving any command from the Host generates an interrupt, and it will queue the received data in the command queue. Similarly, when a stack generates an event, it will be put into an event queue, which will be forwarded to the Host. These two queues will be processed in *ncp.c*, as described on the following figure.



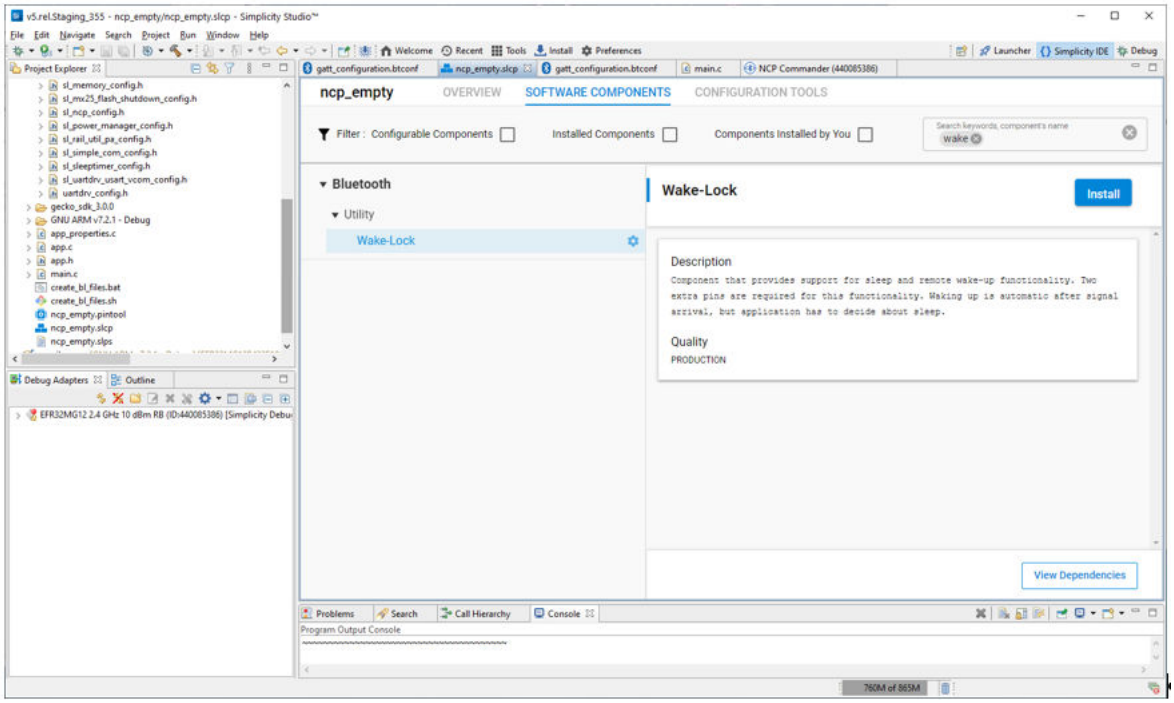
6.1.8 Sleep Modes

The NCP example project does not enable deep sleep mode (EM2) by default, because UART needs EM1 or EM0 to be able to receive commands at any time. Deep sleep mode can be enabled, but in this case, it is essential to configure a wakeup pin so that the NCP Host can wake up the target before sending any BGAPI commands to it. Any available GPIO pin can be configured as a wakeup pin and the polarity is configurable. The following example shows how to configure pin PF6 as the wakeup pin using active-high polarity.

To enable deep sleep mode, the UARTDRV Core component's **Enable reception when sleeping** parameter must be disabled. Otherwise the UART driver will prevent the device from going into EM2 (deep sleep) and it will stay in EM1 (sleep):

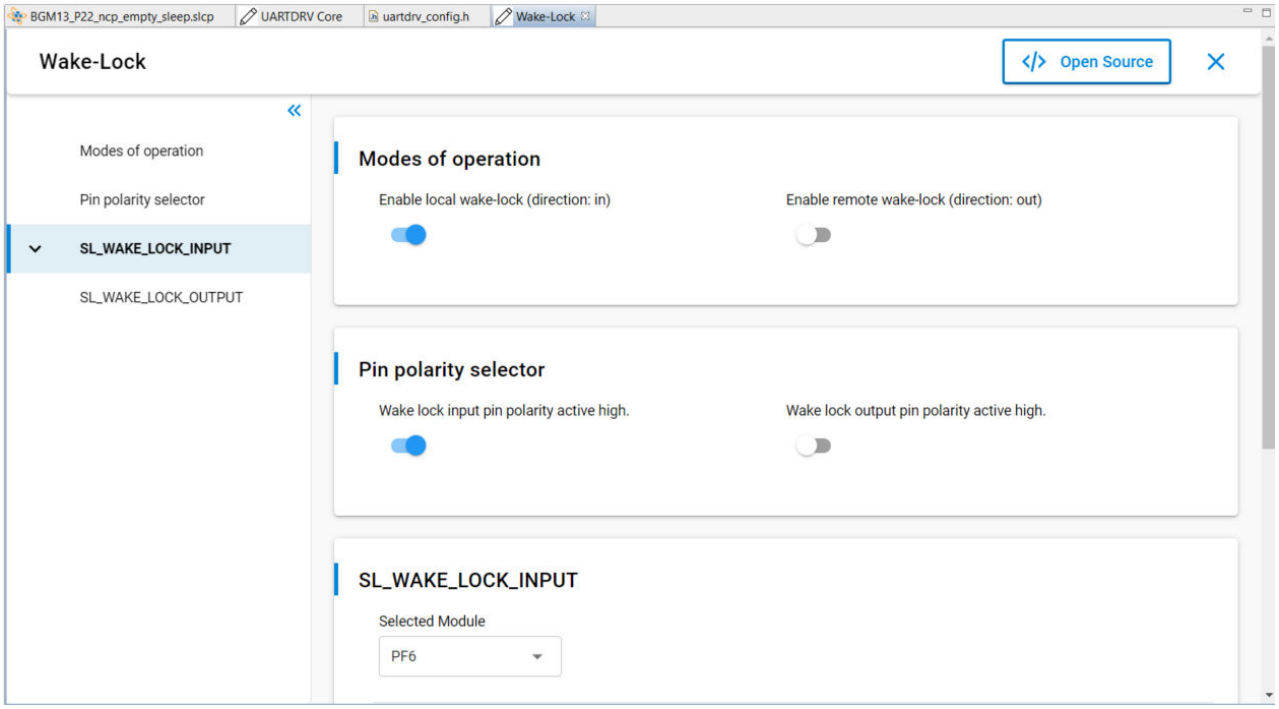


To define a wakeup pin, the Bluetooth > Utility > Wake Lock component must be added to the project:



Configure the Wake-Lock component as follows:

- Enable the wake-lock (direction in) functionality
- Set the polarity (active high in this case)
- Assign the GPIO pin (PF6 in this example)



When the Host sets the wakeup pin to the configured active value, the NCP device will wake up from deep sleep and send out the event `sl_bt_evt_system_awake` to indicate to the host that it has woken up. The host must wait for this event before sending any BGAPI commands, otherwise they might be partially or completely missed.

The remote wake-lock (direction: out) functionality can be used to wake up the host before the NCP target sends out an event. This way the host is also able to go into sleep mode, and it will be notified when it should wake up.

6.2 PC Host

The PC host application project that comes with the SDK is written in C. The host-side source files for this project are found in folders, for GSDK 3.x:

c:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\<version>\app\bluetooth\example_host\empty

or, for GSDK 4.0 and higher:

c:\Users\<username>\SimplicityStudio\gecko_sdk\app\bluetooth\example_host

The projects comprise only a few source and header files. Note, however, that many other files are referenced from the SDK in the makefile. For example, many utility functions are implemented under:

<SDK folder>\app\bluetooth\common_host

but the Bluetooth protocol folder is also heavily used as described later. To copy all the files related to the project into a single folder, take advantage of the export feature described in section 4.2 Host Side.

6.2.1 BGAPI Support Files

While the files in the previous section contain all of the application logic, the actual BGLib implementation code containing the BGAPI parser and packet generation functions is found elsewhere, in other subfolders.

Default location in GSDK 3.x, where <version> will vary by SDK version:

- c:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\<version>\protocol\bluetooth\inc\sl_bt_ncp_host.h
- c:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\<version>\protocol\bluetooth\src\sl_bt_ncp_host.c
- c:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\<version>\protocol\bluetooth\src\sl_bt_ncp_host_api.c

Default location in GSDK 4.0 and higher:

- c:\Users\<NAME>\SimplicityStudio\SDKs\gecko_sdk\protocol\bluetooth\inc\sl_bt_ncp_host.h
- c:\Users\<NAME>\SimplicityStudio\SDKs\gecko_sdk\protocol\bluetooth\src\sl_bt_ncp_host.c
- c:\Users\<NAME>\SimplicityStudio\SDKs\gecko_sdk\protocol\bluetooth\src\sl_bt_ncp_host_api.c

The SDK's specific arrangement of files is one possible way the BGAPI protocol can be used, but it is also possible to create your own library code that implements the protocol correctly with a different code architecture. The only requirement here is that the chosen implementation must be able to create BGAPI command packets correctly and send them to the module over UART. Similarly, it must be able to receive BGAPI response and event packets over UART and process them into whatever function calls are needed to trigger the desired application behavior.

The header files contain primarily #define'd compiler macros and named constants that correspond to all of the various API methods and enumerations you may need to use. The *sl_bt_ncp_host.h* file also contains function declarations for the basic packet reception, processing, and transmission functions.

The *sl_bt_ncp_host.c* file contains the implementation of the packet management functions. All functions defined here use only ANSI C code, to help ensure maximum cross-compatibility on different platforms.

Note: With structure packing, the SDK's BGLib implementation makes heavy use of direct mapping of packet payload structures onto contiguous blocks of memory, to avoid additional parsing and RAM usage. This is accomplished with the PACKSTRUCT macro used extensively in the BGLib header files. It is important to ensure than any ported version of BGLib also correctly packs structures together (no padding on multi-byte struct member variables) in order to achieve the correct operation.

With byte order, the BGAPI protocol uses little-endian byte ordering for all multi-byte integer values, which means directly-mapped structures will only work if the host platform also uses little-endian byte ordering. This covers most common platforms today, but some big-endian platforms exist and are actively used today (Motorola 6800, 68k, and so on).

6.2.2 Host Application Logic

1. Initialize BGLIB.

```
SL_BT_API_INITIALIZE_NONBLOCK(uart_tx_wrapper, uartRx, uartRxPeek);
```

2. Initialize UART.

```
if (serial_port_init(argc, argv, 100) < 0) {
  app_log("Non-blocking serial port init failure\n");
  exit(EXIT_FAILURE);
}
// Flush std output
fflush(stdout);
```

3. Reset NCP Target to ensure it gets into a defined state. Once the chip successfully boots, the `gecko_evt_system_boot_id` event should be received.

```
Sl_bt_system_reset(0);
```

4. The `sl_bt_step` function will be called from the main loop. It checks for any NCP Target events and forwards them to the handler function.

```
// Poll Bluetooth stack for an event and call event handler
static void sl_bt_step(void)
{
  sl_bt_msg_t evt;
  // Pop (non-blocking) a Bluetooth stack event from event queue.
  sl_status_t status = sl_bt_pop_event(&evt);
  if (status != SL_STATUS_OK) {
    return;
  }
  sl_bt_on_event(&evt);
} }
```

5. Process the incoming NCP target events. The example only handles the `sl_bt_evt_system_boot_id` and the `sl_bt_evt_connection_closed_id` events.

```
/* Handle events */
switch (SL_BT_MSG_ID(evt->header)) { case sl_bt_evt_system_boot_id:
// Print boot message.
      app_log("Bluetooth stack booted: v%d.%d.%d-b%d\n",
              evt->data.evt_system_boot.major,
              evt->data.evt_system_boot.minor,
              evt->data.evt_system_boot.patch,
              evt->data.evt_system_boot.build);
      sc = sl_bt_system_get_identity_address(&address, &address_type);
      app_assert(sc == SL_STATUS_OK,
                 "[E: 0x%04x] Failed to get Bluetooth address\n",
                 (int)sc);
      app_log("Bluetooth %s address: %02X:%02X:%02X:%02X:%02X:%02X\n",
              address_type ? "static random" : "public device",
              address.addr[5],
              address.addr[4],
              address.addr[3],
              address.addr[2],
              address.addr[1],
              address.addr[0]);

      // Create an advertising set.
      sc = sl_bt_advertiser_create_set(&advertising_set_handle);
      app_assert(sc == SL_STATUS_OK,
                 "[E: 0x%04x] Failed to create advertising set\n",
                 (int)sc);

      // Set advertising interval to 100ms.
      sc = sl_bt_advertiser_set_timing(
advertising_set_handle, // advertising set handle
        160, // min. adv. interval (milliseconds * 1.6)
        160, // max. adv. interval (milliseconds * 1.6)
        0,   // adv. duration
        0);  // max. num. adv. events
      app_assert(sc == SL_STATUS_OK,
                 "[E: 0x%04x] Failed to set advertising timing\n",
                 (int)sc);
      // Start general advertising and enable connections.
      sc = sl_bt_advertiser_start(
        advertising_set_handle,            // advertising set handle
        advertiser_general_discoverable,   // discoverable mode
        advertiser_connectable_scannable); // connectable mode
      app_assert(sc == SL_STATUS_OK,
                 "[E: 0x%04x] Failed to start advertising\n",
                 (int)sc);
      app_log("Started advertising\n");
      break;
}
```

```
case sl_bt_evt_connection_closed_id:
      app_log("Connection closed\n");
      // Check if need to boot to OTA DFU mode.
      if (boot_to_dfu) {
        // Enter to OTA DFU mode.
        sl_bt_system_reset(2);
      } else {
        // Restart advertising after client has disconnected.
        sc = sl_bt_advertiser_start(
          advertising_set_handle,            // advertising set handle
          advertiser_general_discoverable,   // discoverable mode
          advertiser_connectable_scannable); // connectable mode
        app_assert(sc == SL_STATUS_OK,
                   "[E: 0x%04x] Failed to start advertising\n",
                   (int)sc);
        app_log("Started advertising\n");
      }
      break;}
```

7. Custom API Support

This chapter introduces how to implement a custom binary protocol between an NCP target and host using specific features of the BGAPI. The Silicon Labs Bluetooth SDK provides the following commands and events for that purpose:

- `cmd_user_message_to_target`
- `evt_user_message_to_host`

The command and event details are documented in the API reference manual.

The `cmd_user_message_to_target` command can be used by an NCP host to send a message to the target application on a device. To send a custom message with this API command, the host must send the byte sequence specified below to the target. Byte 4..255 can be the custom message itself.

The custom message can be interpreted in any specific way, but the default implementation uses the first byte as the custom command type, and the rest handled as parameters for that command. This is the recommended way of using custom APIs.

Table 7.1. Command Byte Sequence

Byte Number	Value/Type	Description
0	0x20	Message type: Command
1	payload length	The size of the uint8array struct including its length and payload members.
2	0xFF	Message class: User messaging
3	0x00	Message ID
4..255	uint8array	The user message. The first byte is the length of the message. The next bytes are the message bytes

Once the target receives this byte sequence, it must response with the byte sequence specified below. Byte 6 to 255 can be used for the custom response.

Table 7.2. Response Byte Sequence

Byte Number	Value/Type	Description
0	0x20	Message type: Command
1	payload length	The size of the uint8array struct including its length and payload members.
2	0xFF	Message class: User messaging
3	0x00	Message ID
4-5	uint16	Result code: 0: Success / Non-0: An error occurred
6..255	uint8array	The user message. The first byte is the length of the message. The next bytes are the response message bytes.

Additionally, the `evt_user_message_to_host event` can be used by the target to send a message to NCP host. The target must send the byte sequence specified below. Byte 4..255 can be the custom message itself.

Table 7.3. Event Byte Sequence

Byte Number	Value/Type	Description
0	0xA0	Message type: Event
1	payload length	The size of the uint8array struct including its length and payload members.
2	0xFF	Message class: User messaging

Byte Number	Value/Type	Description
3	0x00	Message ID
4..255	uint8array	The user message. The first byte is the length of the message. The next bytes are the message bytes.

7.1 ncp_user_command_cb

The NCP Target calls `ncp_user_command_cb` if a command ID equals to `sl_bt_cmd_user_message_to_target_id`.

You can find the default implementation of `ncp_user_command_cb` in the *app.c* file and the declaration it in the *ncp.h* file.

In the first case, the Target echoes back the command to the Host, as a reply for the `USER_CMD_1` command. Also, it sends back the same as an event, to demonstrate how events can be sent to the Host.

It is also possible to initiate the communication from the Target. For the second user command, `USER_CMD_2` starts a timer, and when it expires, it will send a user event to the Host, using the function `sl_bt_send_evt_user_message_to_host`.

```
/**************************************************************************//**
 * User command handler.
 *****************************************************************************/
void ncp_user_command_cb(void* data)
{
  uint8array* cmd = (uint8array*)data;
  user_cmd_t* user_cmd = (user_cmd_t*)cmd->data;

  switch (user_cmd->hdr) {
    // User command 1.
    case USER_CMD_1_ID:
      ////////////////////////////////////////////
      // Add your user command handler code here! //
      ////////////////////////////////////////////

      // Example: respond to user command and send user event back too.
      // Sending back received command both as response and event.
      ncp_user_command_rsp(SL_STATUS_OK, cmd->len, cmd->data);
      ncp_user_evt(cmd->len, cmd->data);
      break;

    // User command 2.
    case USER_CMD_2_ID:
      ////////////////////////////////////////////
      // Add your user command handler code here! //
      ////////////////////////////////////////////
      sl_bt_system_set_soft_timer(2*SECOND, USER_EVENT_ID, 1)
      break;

    // Unknown user command.
    default:
      // Send error response back to host.
      ncp_user_command_rsp(SL_STATUS_FAIL, 0, NULL);
      break;
  }
}

…
/* Sending target-initiated message from the timer event handler */

  /* soft timer fired event */
   case sl_bt_evt_system_soft_timer_id:
   /* user event can be sent out to HOST */
   /* send custom message from the Target */
     sl_bt_send_evt_user_message_to_host(data_len, data)
  break;
```

7.2 Host Side

The example below shows how to send custom APIs from the Host side, and how to handle custom events. The custom command structure is defined as a struct for easier access:

```
#define USER_CMD_2_ID     0x02
#define DATA_LENGTH       0x02

PACKSTRUCT(struct user_cmd {
  uint8_t hdr;
  uint8_t data[DATA_LENGTH];
});
typedef struct user_cmd user_cmd_t;
```

The command packet is filled with Command ID and custom data, and then sent with the sl_bt_user_message_to_target. The response is logged to the console.

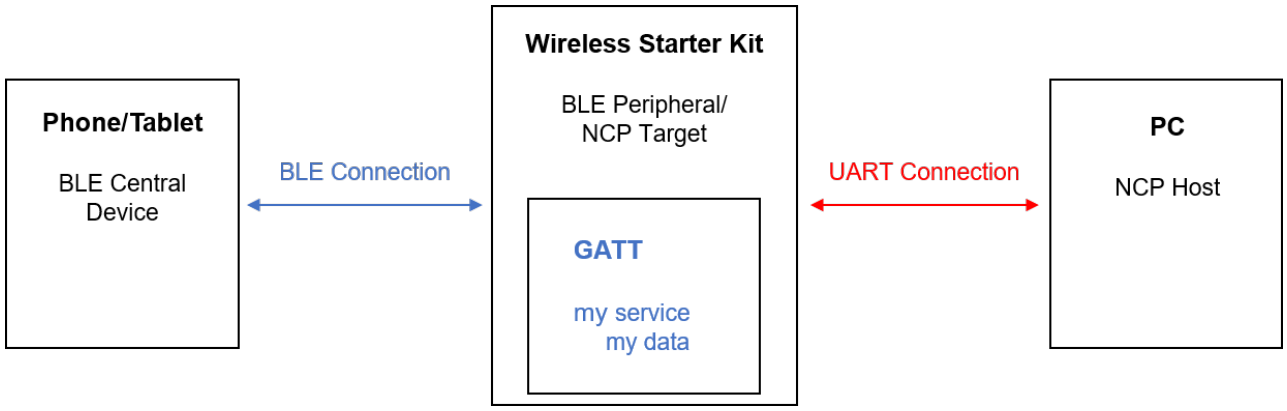
```
user_command.hdr = USER_CMD_2_ID;
user_command.data[0] = 0xAA;
user_command.data[1] = 0xBB;
sl_bt_user_message_to_target(sizeof(user_command), (uint8_t*)&user_command, sizeof(user_command),
                             &resp_length, (uint8_t*)&custom_response);
app_log_info("Custom response received. Response data: %x %x", custom_response->data[0],
 custom_response->data[1]);
```

If the target also sends an event, it can be handled from the main event handler loop as any other NCP events. The `evt->data.evt_user_message_to_host.message.data` field contains the user message:

```
case sl_bt_evt_user_message_to_host_id:
custom_event = (user_cmd_t*)evt->data.evt_user_message_to_host.message.data;
app_log_info("Custom event received. Response data: %x %x", custom_event->data[0], custom_event->data[1]);
break;
```

8. Adding New Attributes to the GATT Database

This chapter describes how to add a custom Bluetooth service to the **NCP** example using the dynamic GATT API. The service added here has one characteristic to receive data. When the central device (tablet/phone) writes this characteristic, the peripheral (starter kit – NCP Target) forwards this data to the NCP Host. The NCP Host prints out the actual data to the PC console.



To implement this application, you need to make these changes:

- Modify the GATT database in the Host project
- Handle the GATT change event (sl_bt_evt_gatt_server_attribute_value) in the Host project

8.1 Adding New Attributes Using the Configuration File

Beginning with SDK version 3.3, the NCP host sample applications contains a .btconf file with a basic GATT configuration. This configuration can be edited with the GATT Configurator. Although PC host examples are not handled by Simplicity Studio, .btconf files can still be edited individually. Open the Simplicity IDE perspective in Simplicity Studio and drag-and-drop the .btconf file onto the editor area. GATT Configurator will automatically open. Edit the file as described in *UG438: GATT Configurator User's Guide for Bluetooth® SDK v3.x* and save it. To be compatible with the code snippets, create a custom service and then add a custom characteristic with the following properties:

- ID: my_data
- Read, Write, Indicate
- Value length: 20 bytes

Once the .btconf file is saved it must be turned into source code by running `make gattdb`. Run this command in the root folder of your example, where you can find the makefile. Note: The generator script requires installing Python 3 and the Jinja2 package by calling `pip install jinja2`.

```
baleidec@NB0013794 MINGW64 /c/sbx/super/app/bluetooth/example_host/empty
$ make gattdb
Generate GATT database
```

The output (gatt_db.c / gatt_db.h) is located in the autogen folder. These values will be used as parameters for the dynamic GATT APIs. The database will be created (that is, built on the NCP target with the dynamic GATT API) automatically in the initialization phase before the boot event is sent to the application. The application is still able update the GATT database with the dynamic GATT commands, as described in the next section.

Note: If your database contains included services, the included ones need to be defined before the ones including them.

8.2 Adding New Attributes Using the Dynamic GATT API

The GATT database can be extended with the APIs provided by the Dynamic GATT Configurator component. See the Bluetooth API reference manual on docs.silabs.com, section "GATT Database", for more details.

In the code snippet below, the custom service and characteristic is added to the database. The service will be a primary service, defined with a 16-byte long UUID, and it will be advertised.

The characteristic has the following properties:

- Read, Write, Indicate
- Valuelength: 20 bytes
- Valuemax length: 20 bytes
- 16-bytelong UUID

The service and the characteristic can be added any time after the boot event was received, but adding them in the boot event handler is suggested.

```
uint8_t uuid_service[16] = {…} //define your 128bit service UUID, you can use a random number
uint8_t uuid_characteristic[16] = {…} //define your 128bit characteristic UUID

//create a session for the database update
sl_bt_gattdb_new_session(&session);
//add our service to the database, as an advertised primary service
sl_bt_gattdb_add_service(session, sl_bt_gattdb_primary_service, SL_BT_GATTDB_ADVERTISED_SERVICE, 16,
                uuid_service, &service);
//define the following properties: read, write, indicate
property = (SL_BT_GATTDB_CHARACTERISTIC_READ | SL_BT_GATTDB_CHARACTERISTIC_INDICATE |
                SL_BT_GATTDB_CHARACTERISTIC_WRITE);
//add our characteristic to the service
sl_bt_gattdb_add_uuid128_characteristic(session, service, property, 0, 0, uuid_characteristic,
                sl_bt_gattdb_fixed_length_value, 20, 20, &value, &characteristic);
//activate the new service
sl_bt_gattdb_start_service(session, service);
//activate the new characteristic
sl_bt_gattdb_start_characteristic(session, characteristic);
//store the handle of the characteristic for future reference
gattdb_my_data = characteristic;
//save changes and close the database editing session
sl_bt_gattdb_commit(session);
```

Note: The handle returned while adding a service or characteristic is not always the final one. It isonly valid until `sl_bt_gattdb_commit` is called. You can get the actual final handle with the API call `sl_bt_gatt_server_find_attribute()`.

8.3 Responding to the GATT Change Event

1. Add the callback function that reacts to the GATT change. In this case, it prints out the content of the characteristic.

```
void AttrValueChanged_my_data(uint8array *value)
{
  uint8_t i;
  for (i = 0; i < value->len; i++){
      app_log("my_data[%d] = 0x%x \r\n",i,value->data[i]);
  }
  app_log("\r\n");
}
```

2. Add the `sl_bt_evt_gatt_server_attribute_value_id` event to the switch case.

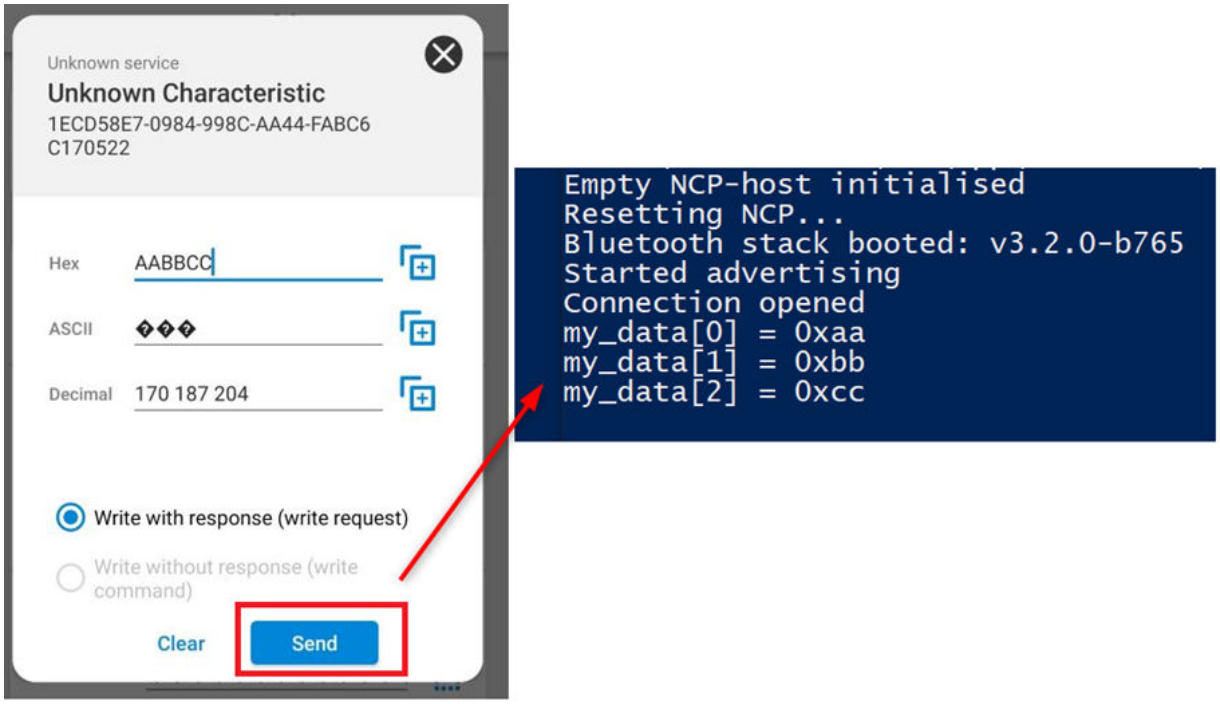
```
case sl_bt_evt_gatt_server_attribute_value_id:
  // Check if the event is because of the my_data changed by the remote GATT client
  if ( gattdb_my_data == evt->data.evt_gatt_server_attribute_value.attribute ){
   // Call my handler
   AttrValueChanged_my_data(&(evt->data.evt_gatt_server_attribute_value.value));
   }
break;
```

Note: if you edited the .btconf file, `gattdb_my_data` is defined in *gatt_db.h*. If you used the dynamic GATT API, `gattdb_my_data` is defined in the application as in the provided code snippet.

Now you can rebuild the host application. See the build process with MinGW in 3.2 Building the NCP Host Examples on Windows.

8.4 Testing

1. Start the host application from the *\exe* folder.
2. Once the PC is connected to WSTK (via UART), the WSTK starts advertising on Bluetooth.
3. If you connect via tablet/phone you can write the newly created `my_data` characteristic in the GATT. For this, you can use the EFR Connect app provided by Silicon Labs.
4. Browse to the `my_data` characteristic and write something to it. The data will be printed by the host application.



Smart. Connected. Energy-Friendly.

IoT Portfolio
www.silabs.com/products

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

SILICON LABS

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

www.silabs.com