AN1086: Using the Gecko Bootloader with Silicon Labs *Bluetooth*® Applications

This application note includes detailed information on using the Silicon Labs Gecko Bootloader with Silicon Labs Bluetooth applications for Gecko SDK (GSDK) 4.1.0 and higher. If you are not familiar with the basic principles of performing a firmware upgrade or want more information about upgrading image files, refer to *UG103.6: Bootloader Fundamentals*.

KEY POINTS

- Gecko Bootloader overview
- Using Gecko Bootloader for BGAPI UART DFU
- Using Gecko Bootloader for Bluetooth OTA upgrade
- Using Gecko Bootloader to update firmware from the user application
- Delta DFU

1 Overview

The Silicon Labs Gecko Bootloader is a common bootloader for all the newer MCUs and wireless MCUs from Silicon Labs. The Gecko Bootloader can be configured to perform a variety of bootloader functions, from device initialization to firmware upgrades. The Gecko Bootloader uses a proprietary format for its upgrade images, called GBL (Gecko Bootloader). These images are produced with the file extension ".gbl". Additional information on the GBL file format is provided in *UG103.6: Bootloader Fundamentals*.

The Gecko Bootloader has a two-stage design, where a minimal first stage bootloader is used to upgrade the main bootloader. The first stage bootloader only contains functionality to read from and write to fixed addresses in internal flash. To perform a main bootloader upgrade, the running main bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The running main bootloader then writes the upgrade image to a fixed location in flash and issues a reboot into the first stage bootloader. The first stage bootloader verifies the integrity of the main bootloader firmware upgrade image, by computing a CRC32 checksum before copying the upgrade image to the main bootloader location.

The Gecko Bootloader can be configured to perform firmware upgrades in standalone mode (also called a standalone bootloader) or in application mode (also called an application bootloader), depending on the software component configuration. Software components can be enabled and configured through the Simplicity Studio IDE.

This document describes how to configure and use the Gecko Bootloader for device firmware upgrades over UART, and for OTA (over-the-air) upgrades using Bluetooth.

The Gecko Bootloader does not come bundled into the application download image. Therefore, you must compile and load the bootloader separately from the application image.

2 BGAPI UART Device Firmware Upgradde (DFU)

This is the firmware upgrade used in Network Co-Processor (NCP) Bluetooth applications. For more information on NCP applications, take a look at *AN1259: Using the v3.x Silicon Labs Bluetooth® Stack in Network Co-Processor.*

In the BGAPI UART DFU implementation a GBL image containing the new firmware is written to target device using UART as the physical interface.

2.1 UART DFU Options

The target device must be programmed with the Gecko Bootloader sample project **Bootloader - NCP BGAPI UART DFU**. Gecko Bootloader is configured automatically for the selected radio board. The BGAPI UART DFU bootloader is a standalone bootloader, so no storage area needs to be configured. During UART DFU upgrade the bootloader writes the new firmware image directly on top of the old firmware image and therefore no temporary download area is needed.

GPIO Settings

The default settings are suitable for testing with a WSTK (Wireless Starter Kit). These settings can be easily changed by through the Software Components tab. Select the Bootloader UART driver component. Here, Hardware Flow Control can be enabled or disabled, and the baud rate and pinout can be configured.

Flow control settings of the radio board and the WSTK must match. The WSTKs flow control can be configured through the admin console. More information on the admin console can be found in the user guide for the respective development kit. To configure the flow control through the admin console:

- In Simplicity Studios Debug Adapters view, right click on the connected device.
- Select Launch Console.
- In the Admin tab, type 'serial vcom config handshake disable/enable', depending on if you want to disable or enable flow control.

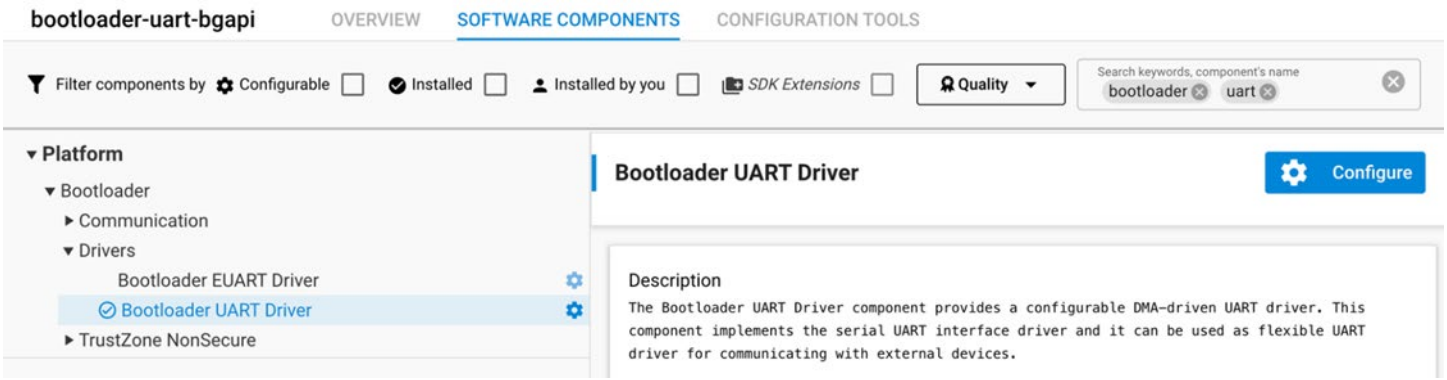


Figure 1 The bootloaders UART software component

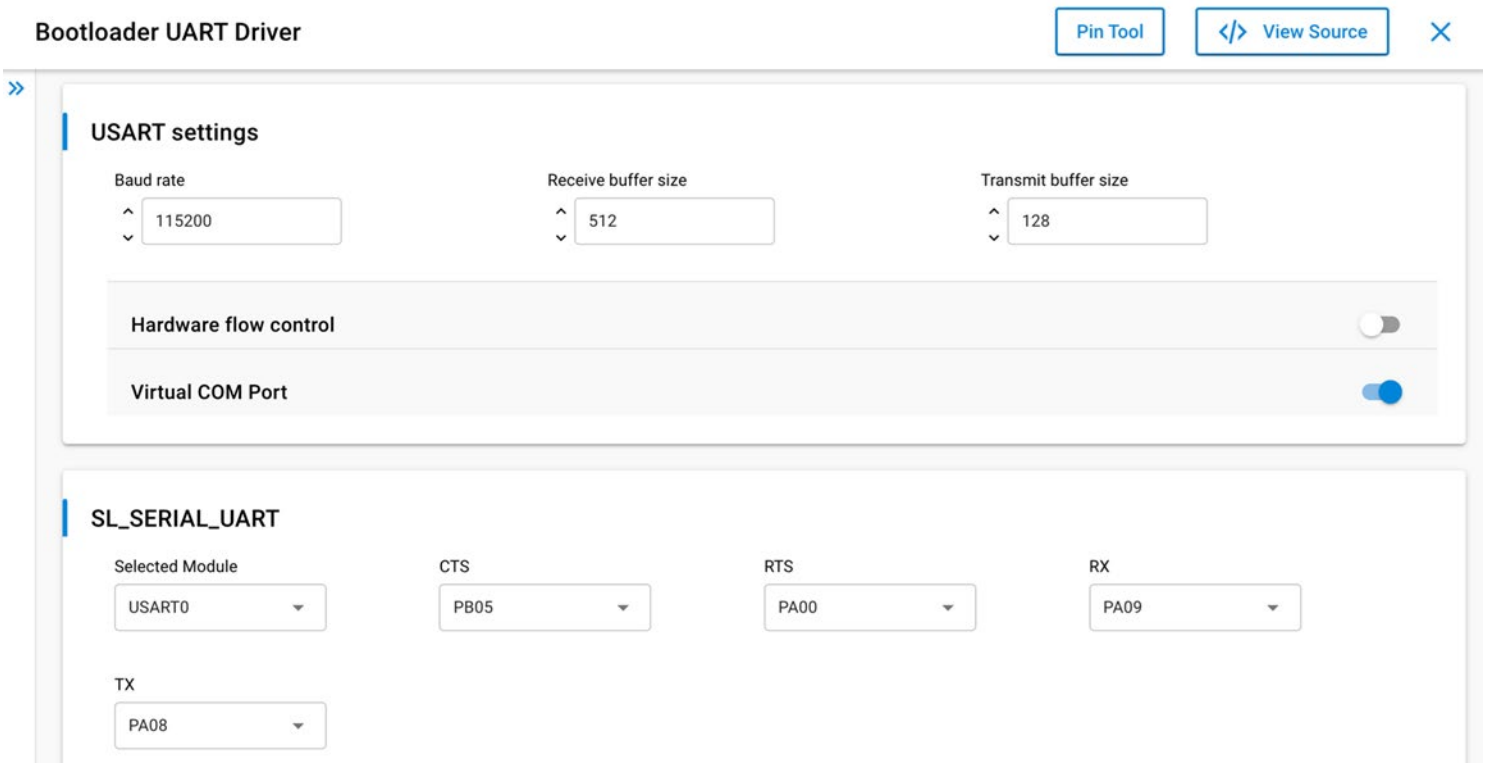


Figure 2 The bootloaders UART configuration

2.2 UART DFU Process

The basic steps involved in the UART DFU, using NCP, are as follows:

1. Boot the target device into DFU mode by calling the bootloader API `bootloader_rebootAndInstall()`. In NCP-mode, this can be achieved by calling `sl_bt_user_reset_to_dfu()`. When using Series 1 devices, reset into DFU is achieved with `sl_bt_system_reset(1)`. Alternatively, if you have GPIO activation enabled in the bootloader, press the bootloader activation pin while the device is reset.
2. Wait for the `sl_bt_evt_dfu_boot` event.
3. Send the command `sl_bt_dfu_flash_set_address(address)` to start the firmware upgrade.
4. Send the entire contents of the GBL upgrade image with `sl_bt_dfu_flash_upload(data)`.
5. After sending all data, the host sends the command `sl_bt_dfu_flash_upload_finish()`
6. To finalize the upgrade, the host resets the target device into normal mode with `sl_bt_system_reboot()`. If using Gecko SDK 4.4.x and older, the device is reset with `sl_bt_system_reset(0)`.

A detailed description of the DFU-related BGAPI commands is found in the Bluetooth Software API Reference Manual.

At the beginning of the upgrade, the NCP host uses the command `sl_bt_dfu_flash_set_address` to define the start address. The start address shall always be set as zero. During the data upload (step 4 above) the target device calculates the flash offset automatically. The host does not need to explicitly set any write offset.

The UART DFU procedure may fail if the update image is either corrupted or data upload is interrupted for some reason. Failure due to either of these conditions is detected by the CRC check performed by the UART DFU bootloader before jumping into the main program. In this case, a `dfu_boot_failure` event is sent by the stack. The returned reason codes align with the sl_status codes, as shown in the platform documentation.

2.3 Creating Upgrade Images for the Bluetooth NCP Application

Building a C-based NCP project in Simplicity Studio does not generate the UART DFU upgrade images (GBL files) automatically. The GBL files need to be created separately by running a script located in the application projects root folder. Before running the script, the application must be compiled.

Two scripts are provided in the SDK examples:

- create_bl_files.bat (for Windows)
- create_bl_files.sh (for Linux / Mac)

The GBL files can be generated by invoking the script from the project directory.

You need to define two environment variables, `PATH_SCMD` and `PATH_GCCARM` before running the script, as shown in the following table.

Table 1 Environment variables required for creating .gbl files

Variable Name	Example Variable Value
PATH_SCMD	C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\commander
PATH_GCCARM	C:\SiliconLabs\SimplicityStudio\v5\developer\toolchains\gnu_arm\10.2_2020q4

Running the **create_bl_files** script creates multiple GBL files in a subfolder named output_gbl. The file named **full.gbl** is the upgrade image used for UART DFU. The other files are related to OTA upgrades, and they can be ignored.

If signing and/or encryption keys (named **app-sign-key.pem**, **app-encrypt-key.txt**) are present in the same directory, then the script also creates secure variants of the GBL files. More information on creating secure firmware for DFU can be found in our training document: Secure Firmware Upgrade using OTA.

2.4 UART DFU Host Example

The UART DFU host example is a C program that is located under the Bluetooth SDK examples in the following directory (the exact path depends on the installed SDK version):

```
C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\<sdk_suite>\<version>\app\bluetooth\example_host
\bt_host_uart_dfu
```

In Windows this program can be built using MinGW. In Linux or Mac, the program can be built using the GCC toolchain.

The project is built by running make (or mingw32-make) in the project root directory. After a successful build, an executable is created in the subfolder named **exe**. The executable filename is **bt_host_uart_dfu.exe**

Before running the example, you need to check the COM port number associated with your NCP target. For more details, see *AN1259: Using the v3.x Silicon Labs Bluetooth® Stack in Network Co-Processor.*

The bt_host_uart_dfu.exe program requires three command line arguments:

- COM port number
- Baud rate
- Name of the (full) GBL file

Furthermore, the device must be flashed with the **BGAPI UART DFU Bootloader** and an NCP-application.

Example usage and expected output in v4.0 or higher:

```
./bt_host_uart_dfu.exe -u COM42 -b 11520 full.gbl


[I] NCP host initialized.
[I] Reset NCP target in bootloader mode...
[I] DFU booted: v0x02010000
[I] Pressing Ctrl+C aborts the update process.
[I] WARNING! If the update process is aborted, the device will stay in bootloader mode.


207728/207728 (100%)
[I] DFU finished successfully. Resetting the device.
```

The number of bytes uploaded in one DFU flash upload command is configurable. The UART DFU host example included in the SDK uses a 48-byte payload. The maximum usable payload length is 128 bytes. The maximum number of bytes sent in one command is specified using a C preprocessor directive named MAX_DFU_PACKET. The value of MAX_DFU_PACKET must be divisible by four.

3 Bluetooth OTA Upgrade

This is the firmware upgrade method used in SoC-mode Bluetooth applications. A GBL-file containing new firmware is sent to a target device through a Bluetooth connection. The firmware upgrade image can be stored to an empty flash area and applied later by the user application, or immediately overwrite the original application using a component called Apploader. The main differences between application- and Apploader-based OTA are shown in the table below:

Feature	Apploader-based OTA	Application-based OTA
Implementation	Integrated into the bootloader	Implemented in the user application
Supported Devices	Series 1 & 2	Series 2 & 3
Storage slot required	No, the old firmware is directly overwritten.	Yes, the new firmware must be stored in flash before overwriting.
Supported PHY	1M	1M, 2M, Coded
Encryption	Not supported	Supported

Currently, it is recommended to use the application-based OTA as it enables improved security and customizability. Furthermore, Apploader is not supported in Series 3 devices. The two available options are described in more detail in the sections Application-based OTA and Apploader (Not recommended for new development). An example of creating the GBL-files and using the application-based OTA is shown in section OTA DFU Example.

3.1 Application-based OTA

Application-based OTA is completely implemented in the user-application. This makes it possible to use a custom GATT service instead of the Silicon Labs OTA service used in the Apploader. Furthermore, in case of UART DFU updates, the application can be designed to support other protocols and interfaces than BGAPI and UART. The user application can also be designed to support both OTA and UART DFU updates if needed.

To use this update mechanism, any application bootloader using internal or external storage may be used. At least one storage area must be defined and the area must be large enough to fit the full GBL file, while not overlapping with the user application. These parameters can be controlled by configuring the **bootloader_storage_slots** component included in the bootloader project.

The basic steps for the update process are described below:

1. Application initializes the Gecko bootloader by calling `bootloader_init()`.
2. The download area for the new firmware is erased by calling `bootloader_eraseStorageSlot(0)`.
3. The update image (full GBL file) is received either over-the-air or through some physical interface like UART.
4. The application writes the received bytes to the download area by calling `bootloader_writeStorage()`.
5. (optional) Application can verify the integrity of the received GBL file by calling `bootloader_verifyImage()`.
6. Before rebooting, call `bootloader_setImageToBootload(0)` to specify the slot ID where new image is stored.
7. Reboot and instruct Gecko bootloader to perform the update by calling `bootloader_rebootAndInstall()`.

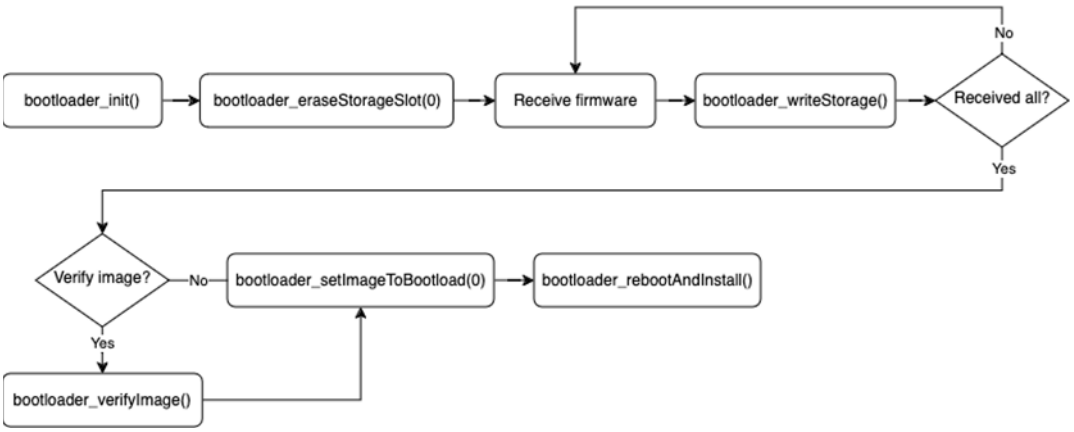


Figure 3 Application-based OTA States

It is assumed here that only one download area is configured and therefore the slot index in the above function calls is set to 0. The general firmware upgrade sequence is explained in *UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher*

Note that the erase procedure in step 2 above takes several seconds to complete. If the new image is downloaded over a Bluetooth connection, the supervision timeout must be set long enough to avoid connection drops. Alternatively, the download area can be erased in advance, before the Bluetooth connection is opened. A third alternative is to erase the download area one flash page at a time during the writing progresses. This can be done using `bootloader_eraseRawStorage()`.

3.1.1 Enabling the Gecko Bootloader API

In order to call the required Gecko Bootloader functions from the user application, the following interface sources must be included in the project:

btl_interface.c (common interface)

btl_interface_storage.c (interface to storage functionality)

These source files and their corresponding header files are copied to SDK sample projects by default and can be found in the SDK under:

```
<sdk_suite>\<version>\platform\bootloader\api\
```

When using the functions in the application, the correct header files can be included with:

```
#include "btl_interface.h"
#include "btl_interface_storage.h"
```

3.2 Apploader (Not recommended for new development)

The Apploader, introduced in Series 1, can be viewed as a small extension that manages everything related to OTA DFU. It contains a minimal version of the Bluetooth stack including only features that are necessary to perform OTA updates. Any Bluetooth features that are not necessary to support OTA updates are disabled in Apploader to minimize the flash footprint.

The Apploader features and limitations are summarized below:

- Enables OTA updating of user application.
- Only one Bluetooth connection is supported, GATT server role only.
- Bluetooth parameters used during firmware upgrade, such as advertisement and connection interval, can’t be changed.
- Encryption and other security features such as bonding are not supported.

The Apploader is placed before the user application in flash. The default linker script provided in the SDK places the user application so that it starts at the flash sector following the Apploader. The user application contains a full-featured version of the Bluetooth stack, and it can run independently of the Apploader. If in-place OTA update is not needed, the Apploader can be removed completely to free up flash for other use (code space or data storage).

3.2.1 Series 1 vs Series 2

From GSDK v4.1.0 and higher, there is a clear distinction between the integration of Apploader for Series 1 and Series 2 devices. For Series 1 devices, the Apploader is a separate application from both the main and the bootloader.

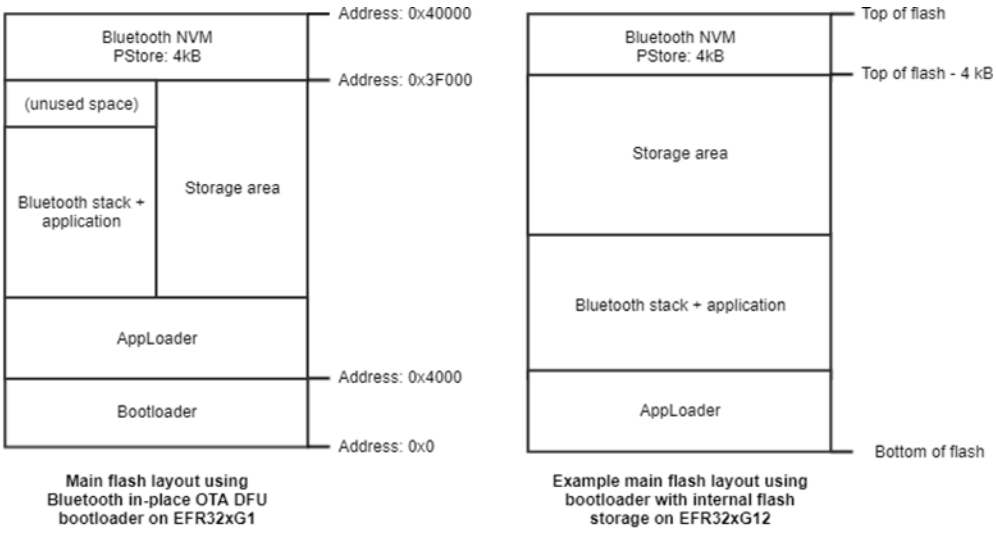


Figure 4 Example flash layout for Series 1 devices using Apploader

For Series 2 devices, the Apploader is added as a part of the bootloader. This results in the following for Series 2 devices:

- The Apploader cannot be upgraded or flashed by itself.
- The Apploader is upgraded by upgrading the bootloader.
- The OTA advertising data cannot be changed.
- The size of the combined bootloader and Apploader is much larger than that of a regular bootloader, and it does not fit into the regular bootloader area.

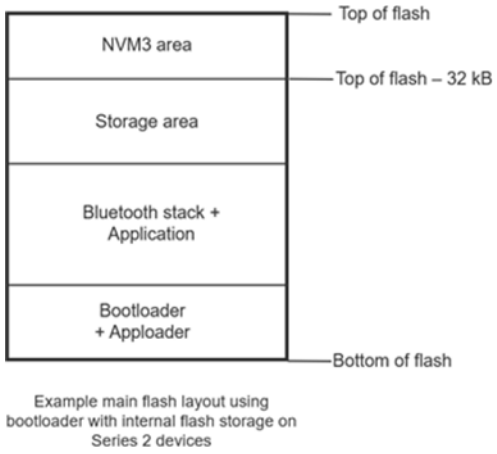


Figure 5 Example flash layout of a Series 2 device using the combined bootloader and apploader

3.2.2 Apploader OTA Process

Most of the OTA functionality is handled autonomously by the Apploader. The minimum requirement for the user application is to trigger a reboot into DFU mode. After the upload is complete, Apploader will reboot the device back into normal mode.

Reboot into DFU mode can be triggered in a variety of ways. It is up to the application developer to decide which is most applicable. Most of the example applications provided in the Bluetooth SDK already have OTA support built into the code. In these examples, the DFU mode is triggered through the Silicon Labs OTA service that is included as part of the application's GATT database. An example of a user defined trigger can be found in section 3.2.4.

Apploader supports two types of updates:

- Partial update: only the user application is updated
- Full update: both Apploader and the user application are updated (1)

The partial update process using Apploader consists of the following steps:

1. OTA client connects to target device.
2. Client requests target device to reboot into DFU mode.
3. After reboot, client connects again.
4. During the 2nd connection, target device is running Apploader (not the user application).
5. New firmware image is uploaded to the target.
6. Apploader copies the new application on top of the existing application.
7. When upload is finished and connection closed, Apploader reboots back to normal mode.
8. Update complete.

With partial update, it is possible to update the Bluetooth stack and user application. Apploader is not modified during partial update.

Full update enables updating both the Apploader and the user application. A full update is always recommended when moving from one SDK version to another. The size of Apploader can vary depending on the SDK version, so this may prevent a partial OTA update if the new application image overlaps with the old Apploader version.

Full update is done in two steps. Updating the Apploader always erases the user application and therefore the Apploader update must be followed by application update.

The first phase of full update updates the Apploader, and it consists of following steps for Series 1 devices (1):

1. OTA client connects to the target device.
2. The Client requests target device to reboot into DFU mode.
3. After reboot, the client reconnects.
4. During the 2nd connection, the target device is running Apploader (not the user application).
5. New Apploader image is uploaded to the target.
6. Apploader copies the image into the download area (specified in Gecko bootloader configuration).
7. When the upload is finished and the connection closed, Apploader reboots and requests Gecko Bootloader to install the downloaded image.
8. Gecko Bootloader updates Apploader using the downloaded image and reboots.
9. After reboot, the new Apploader is started.

At the end of the Apploader update, the device does not contain a valid user application and therefore Apploader will remain in DFU mode. To complete the update, a new user application is uploaded following the same sequence of operations that were described for the partial update.

Notes:

(1) For Series 2 devices, the same steps can be followed but the Apploader is updated as part of the bootloader, after which the application must be updated. For this, the bootloader must be converted into .gbl format by using Simplicity Commander:

```
commander gbl create bootloader-apploader.gbl --bootloader bootloader-apploader.s37
```

3.2.3 Configuring the Apploader

In order to use OTA for firmware upgrades, the user must first configure the bootloader (only for Series 2) and the application to support the Apploader. Once this is done, the user is ready to generate the .gbl file, as described in section 3.3.2.

Bootloader

This step is only required for series 2. If you are using a series 1 device, you can move on to the application

Configuring the bootloader for OTA can be done in two ways. The most straightforward method is to include the sample Bootloader project, **SoC Bluetooth Apploader OTA DFU**, where all necessary configurations are already set.

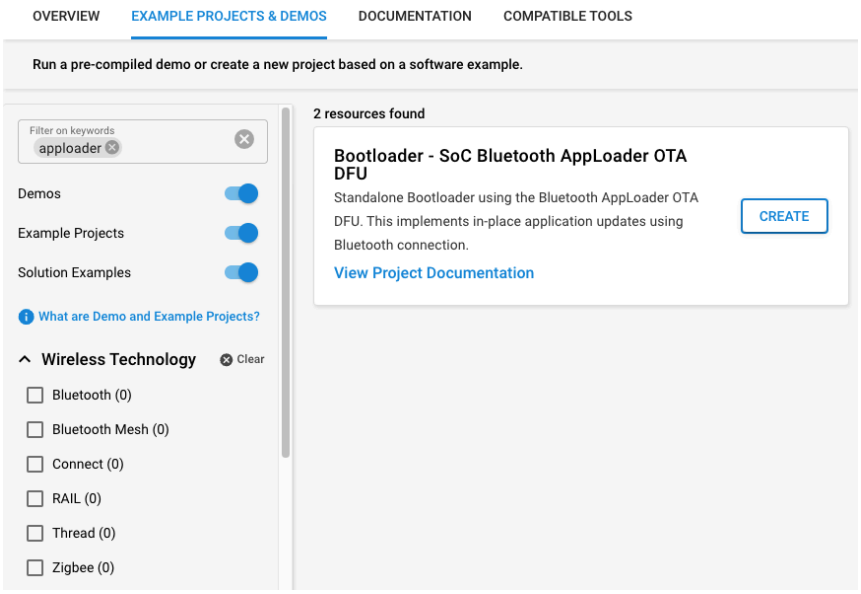


Figure 6 Sample bootloader project containing the Apploader

Alternatively, you can add the **bootloader_apploader** software component to your already existing bootloader. After this component is added, the base address of the bootloader image must be updated in the Bootloader Core component. This is done to prevent the new bootloader from overwriting itself during the OTA update. Any appropriate value can be used as long as the base address is larger than the size of the bootloader itself. The recommended value is 0x18000.

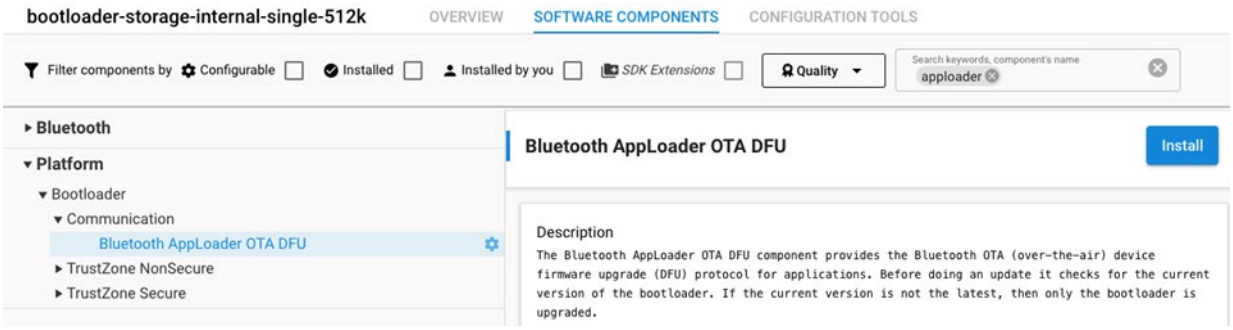


Figure 7 The software component that adds Apploader to the bootloader

Application

In your application project, only the **in_place_ota_dfu** component needs to be added. If you are using a sample project, this component might be included by default.

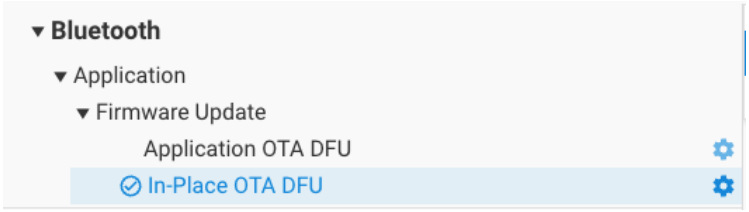
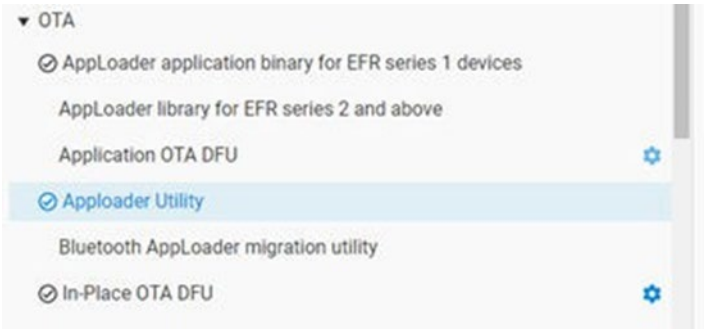


Figure 8 The required software component for adding OTA support to the application

The **in_place_ota_dfu** component requires the following components, which are automatically included:

- **apploader** which on:
  - Series-1 devices includes the Apploader binary as an additional prebuilt library.
  - Series-2 devices moves the application start address to give space for an Apploader OTA DFU Bootloader. It also requires a Gecko Bootloader with an Apploader OTA DFU plugin to be present on the device.
- **apploader_util**, which provides utility functions related to OTA DFU, such as a unified API for resetting the device to DFU mode.



3.2.4 Triggering Reboot into DFU Mode from the User Application

The minimum functional requirement to enable OTA in the user application is to implement a 'hook' that allows the device to be rebooted into DFU mode.

1. The code checks if the OTA control characteristic was written, and if so, sets `boot_to_dfu` to true. After this, the connection is closed.
2. The device is rebooted into DFU mode with `sl_bt_system_reset(2)`. Parameter value 2 indicates that the device is to be rebooted into OTA DFU mode. The rest of the OTA upgrade is managed by Apploader and no further actions are needed from the user application.

```
///////////////////////////////////////////////////////////////////////////////
// This event indicates that a remote GATT client is attempting to write //
// a value of a user type attribute in to the local GATT database.       //
///////////////////////////////////////////////////////////////////////////////
case sl_bt_evt_gatt_server_user_write_request_id:
  // If user-type OTA Control Characteristic was written, boot the device
  // into Device Firmware Upgrade (DFU) mode.
  if (evt->data.evt_gatt_server_user_write_request.characteristic == gattdb_ota_control) {
    // Set flag to enter OTA mode.
    boot_to_dfu = true;
    // Send response to user write request.                                    1
    sc = sl_bt_gatt_server_send_user_write_response(
      evt->data.evt_gatt_server_user_write_request.connection,
      gattdb_ota_control,
      SL_STATUS_OK);
    app_assert(sc == SL_STATUS_OK,
               "[E: 0x%04x] Failed to send response to user write request\n",
               (int)sc);
    // Close connection to enter to DFU OTA mode
    sc = sl_bt_connection_close(
      evt->data.evt_gatt_server_user_write_request.connection);
    app_assert(sc == SL_STATUS_OK,
               "[E: 0x%04x] Failed to close connection to enter to DFU OTA mode\n",
               (int)sc);
  }
  break;

///////////////////////////////////////////////////////////////////////////////
// This event indicates that a connection was closed.                    //
///////////////////////////////////////////////////////////////////////////////
case sl_bt_evt_connection_closed_id:
  // Check if need to boot to OTA DFU mode.
  if (boot_to_dfu) {
    // Reset MCU and enter OTA DFU mode.                  2
    sl_bt_system_reset(2);
  }
  break;
```

Figure 9 Handling the Write to OTA Control Characteristic in C Code

3.3 OTA DFU Example

The Bluetooth SDK includes an OTA host reference implementation that can be used for testing OTA. The example is written in C and uses a WSTK in Network Co-Processor (NCP) mode. The OTA host application itself runs on the host computer. For more information on the NCP mode of operation, see *AN1259: Using the Silicon Labs Bluetooth® Stack v3.x and Higher in Network Co-Processor Mode*.

The target device to be upgraded over-the-air is identified by its Bluetooth address.

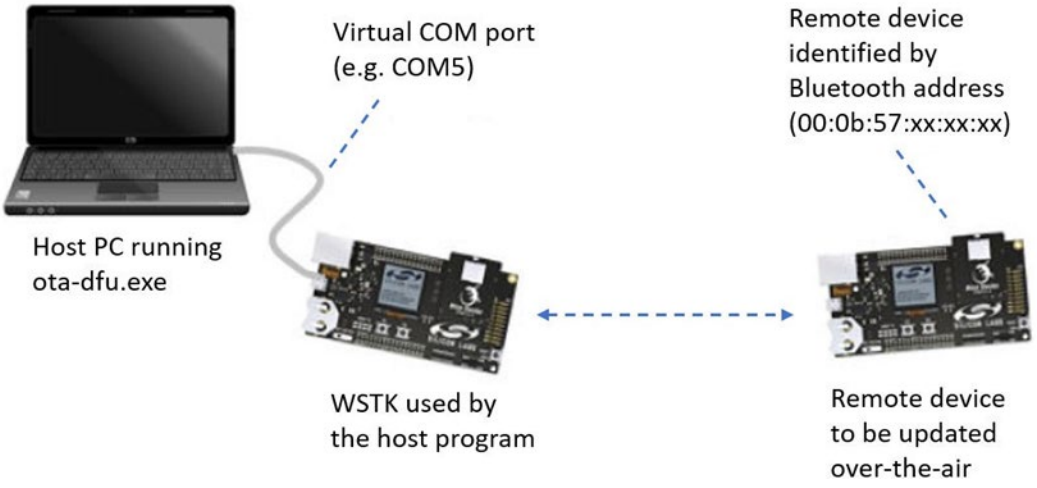
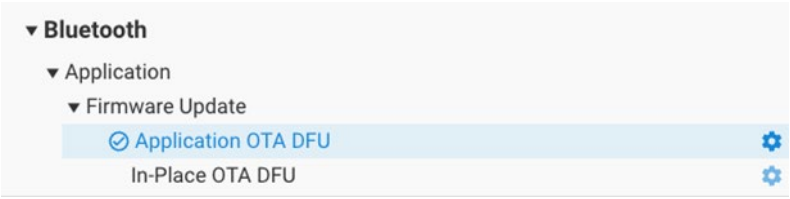


Figure 10 OTA host example setup

In this example, we will use the application-based OTA, but Apploader can also be used.

3.3.1 Preparing the SoC Application

In this example, we will use the "**Bluetooth – SoC Empty"** sample-application, together with one of the internal-storage bootloaders.



As the **in_place_ota_dfu** is not used, it can be removed from the project. The source of **app_ota_dfu** can be found at:

```
<sdk_suite>\<version>\app\bluetooth\common\app_ota_dfu
```

3.3.2 Creating the GBL files

Once the application-based OTA has been added, the project can be compiled. After compiling, the GBL-files must be created separately. This is done by running one of the following scripts from the project root folder:

- create_bl_files.bat (for Windows)
- create_bl_files.sh (for Linux / Mac)

Before running the scripts, two environment variables must be defined, PATH_SCMD and PATH_GCCARM, as seen in Table 2.

Table 2 Environment variables required for creating .gbl files

Variable Name	Example Variable Value
PATH_SCMD	C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\commander
PATH_GCCARM	C:\SiliconLabs\SimplicityStudio\v5\developer\toolchains\gnu_arm\10.3_2021.10

Running the **create_bl_files** script creates GBL files in a subfolder named `output_gbl`. The **application.gbl** file is the relevant file for OTA.

If signing and/or encryption keys (named **app-sign-key.pem**, **app-encrypt-key.txt**) are present in the project directory then the script also creates secure variants of the GBL files. More information on creating secure firmware for DFU can be found in our training document: Secure Firmware Upgrade using OTA.

3.3.3 Preparing the NCP Application

The development kit that is used on the host side should be programmed with firmware that is suitable for NCP mode. The Bluetooth SDK includes an example project called "**Bluetooth NCP**" that can be used for this purpose.

The WSTK features an on-board USB-to-UART converter. The board will be seen as a virtual COM port by the host computer.

3.3.4 Preparing the Host Application

The OTA host example is found in the following directory under the SDK installation tree (the exact path depends on the installed SDK version), but for example:

```
<sdk_suite>\<version>\app\bluetooth\example_host\bt_host_ota_dfu
```

The project folder contains a makefile that allows the program to be built using for example MinGW (by running `mingw32-make`) or Cygwin (by running `make`). an executable file is created in subfolder named **exe**. The executable filename is **bt_host_ota_dfu.exe**.

3.3.5 Running OTA with the NCP Host Example

The OTA host program expects the following command-line arguments:

1. The COM port associated with the development kit used in NCP mode
2. Baud rate (use fixed value 115200)
3. Name of the GBL file to be uploaded into target device
4. Bluetooth address of the target device
5. (optional) force write without response (possible values 0 / 1, default is 0)

A full OTA upgrade for is done in two parts, and it requires two separate GBL files, one for the bootloader and another for the user application. If the bootloader and application has been combined into one image, as described in section 3.3.2, the upgrade can be done in one part.

For upgrading the bootloader and application separately, the host application must be invoked twice:

```
./bt_host_ota_dfu.exe COM49 115200 bootloader-second_stage.gbl 00:0B:57:0B:49:23
./bt_host_ota_dfu.exe COM49 115200 application.gbl 00:0B:57:0B:49:23
```

Please note, the GBL-files can also be uploaded using the Simplicity Connect mobile application, as described in Using Simplicity Connect for OTA DFU.

If the application alone is going to be upgraded, then the host program is run once, with the **application.gbl** file passed as parameter.

3.3.6 OTA Host Example Internal Operation

The OTA host example is implemented as a state machine. The key steps in the OTA sequence are summarized below. Note that the program execution is independent of the type of upgrade image that is used. The program simply uploads one GBL file into the target device. It is up to the user to invoke the program either once or twice, depending on the upgrade type (partial OTA or full OTA).

The following diagram illustrates the state transitions in the OTA host example program in a slightly simplified form.

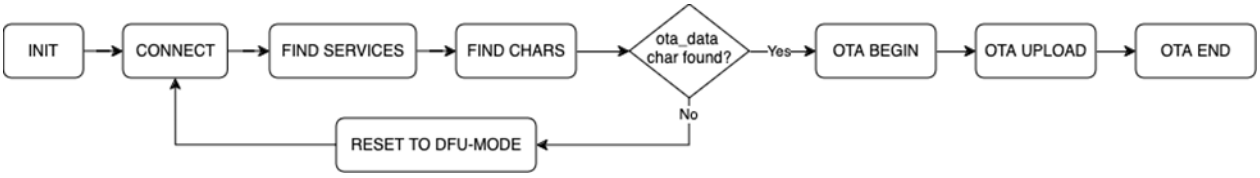


Figure 11 OTA Host Application States

INIT - The program checks the total size of the GBL file that is passed as a command-line parameter. The GBL file content is not parsed. It is enough to know the file size so that the entire content can be uploaded to target device.

CONNECT - The program tries to open a connection to the target device whose Bluetooth address is given as a command line parameter. The host program does not scan for devices. If the target device is not advertising, then the connection open attempt cau- ses the program to be blocked.

FIND SERVICES - After a connection has been established, the program performs a service discovery. In this case only the OTA service is of interest, and therefore the program performs discovery of services with that specific UUID using the API call `sl_bt_gatt_discover_primary_services_by_uuid`. More information on the OTA service can be found in section 3.6.

FIND CHARACTERISTICS - After the service has been found, the characteristic of the OTA service are queried using API call `sl_bat_gatt_discover_characteristics`. The handle value for the **ota_control** needs to be discovered in order to proceed with the OTA procedure.

RESET TO DFU - If the target is not already in the DFU-mode, the host program requests reboot into DFU mode by writing value 0x00 to the ota_control characteristic. The execution then jumps back to the **CONNECT** state.

OTA BEGIN - If both ota_data and ota_control characteristic handles have been detected, the host program initiates OTA by writing value 0x00 to the ota_control characteristic. This does not cause reboot or any other side effects because the target device is already in DFU mode.

OTA UPLOAD - This is where the GBL file is uploaded to the target device. The whole content of the GBL file is uploaded to the target device, by performing multiple write operations into the ota_data characteristic. The host pro- gram uses the write-without-response transfer type to optimize throughput. Note that even if the write-without-response operations are not acknowledged at the application level, error checking (and retransmission when needed) at the lower protocol layers ensures that all packets are delivered reliably to the target device.

OTA END - Once the OTA upload is completed, the host program ends the OTA procedure by writing value 0x03 to the ota_control characteristic. Finally, the program terminates.

Some error cases have been omitted from the state diagram for simplicity. For example, the program exits with an error code if the OTA service is not found when performing service discovery or if the ota_control characteristic is not discovered in **FIND CHARACTERIS- TICS** state.

Note: When the target device reboots into DFU mode, the host program must perform full service and characteristic discovery again. It is not possible to store the ota_control and ota_data characteristic handles in memory and use those cached values during the second connection. This is because the target device has two GATT databases that are independent of each other: one that is used by the application in normal mode and the other that is used by Apploader in OTA DFU mode. While both of these GATT databases might include the Silicon Labs OTA service, the characteristic handles are likely to have different values. Therefore any kind of GATT caching cannot be used.

3.4 OTA GATT Database amd Generic Attribute Service

When booted into DFU mode, the Apploader uses a GATT database that is different than the normal GATT used by the application. The OTA DFU GATT database used by Apploader contains following services:

- Generic Attribute (UUID 0x1801)
- Generic Access (UUID 0x1800)
- Silicon Labs OTA service (UUID 0x1d14d6ee-fd63-4fa1-bfa4-8f47b42119f0)

The Bluetooth specification requires that, if GATT-based services can change in the lifetime of the device, then the **Generic Attribute Service** (UUID 0x1801) and the **Service Changed** characteristic (UUID 0x2A05) shall exist in the GATT database. For details, please see Bluetooth Core specification, Version 5.2, Vol. 3, Part G, 7 DEFINED GENERIC ATTRIBUTE PROFILE SERVICE.

The Generic Attribute service is automatically included in the Apploader GATT database used during OTA. To avoid any interoperability issues due to GATT caching, it is strongly recommended that the application GATT database used in normal mode also enables this service. Generic Attribute service is enabled by default in the SDK example applications.

Note: Apploader does not generate a service changed indication when rebooting to DFU mode or rebooting back to normal mode.

Automatic service changed indication requires that the client is bonded and has enabled the indication for this characteristic. Apploader does not support bonding and therefore the service changed indication is not generated.

The Generic Attribute Service can also be explicitly defined in the application's GATT database using the same XML notation that is used for other services. The Generic Attribute service must be the first service in the list, to ensure it is aligned with the Generic Attrib- ute Service that is used during OTA. The Bluetooth specification requires that the attribute handle of the Service Changed characteristic shall not change and therefore this service must be first on the list (the same as in the OTA GATT database).

More details on the Generic Attribute Service can be found on the Bluetooth SIG website:

https://www.bluetooth.com/specifications/gatt/services

Note also that Apploader does not support the GATT caching enhancements that were introduced in the Bluetooth Core Specification 5.1 and Silicon Labs Bluetooth SDK 2.11.1.

3.5 Silicon Labs OTA GATT Service

The following XML representation defines the Silicon Labs OTA service. It is a custom service using 128-bit UUID values. The service content and the UUID values are fixed and must not be changed.

The OTA service characteristics are described in the following table. The UUID value of the service itself is `1d14d6eefd63-4fa1- bfa4-8f47b42119f0`.

Table 3 Silicon Labs OTA Service Characteristics

Characteristic	UUID	Type	Length	Support	Properties
OTA Control Attribute	F7BF3564-FB6D-4E53-88A4-5E37E0326063	Hex	1 byte	Mandatory	Write (4)
OTA Data Attribute (1)	984227F3-34FC-4045-A5D0-2C581F81A153	Hex	Variable; max 244 bytes	Mandatory	Write without response; Write
Apploader version (2) (Bluetooth stack version 2,3)	4F4A2368-8CCA-451E-BFFF-CF0E2EE23E9F	Hex	8	Optional	Read
OTA version (2)	4CC07BCF-0868-4B32-9DAD-BA4CC41E5316	Hex	1	Optional	Read
Gecko Bootloader version (2)	25F05C0A-E917-46E9-B2A5-AA2BE1245AFE	Hex	4	Optional	Read
Application version	0D77CC11-4AC1-49F2-BFA9-CD96AC7A92F8	Hex	4	Optional	Read

Notes:

1. This characteristic is excluded from the user application GATT database.

2. Version information is automatically added by Apploader when running in DFU mode. These are optional in the application GATT database.
3. This characteristic exposes the Apploader version.
4. Silicon Labs highly recommends that the default property (i.e, write) be used only over bonded connections to prevent uploading of new firmware by untrusted/unknown devices.

Table 4 Possible Control Words Written to the OTA Control Characteristic

Hex value	Description
0x00	OTA client initiates the upgrade procedure by writing value 0.
0x03	After the entire GBL file has been uploaded the client writes this value to indicate that upload is finished.
0x04	Request the target device to close connection. Typically the connection is closed by OTA client but using this control value it is possible to request that disconnection is initiated by the OTA target device.
Other values	Other values are reserved for future use and must not be used by application.

In DFU mode, Apploader uses the full OTA service described above. The GATT database of the user application includes only a subset of the full OTA service. The minimum application requirement is to include the OTA control characteristic. The application must not include the OTA data character- istic in its GATT database (unless the OTA update is implemented fully in application code, as described in section 3.1

From the user application viewpoint, only the OTA control attribute is relevant. In the OTA host example reference implementation included in the SDK, the OTA procedure is triggered when the client writes value 0 to the OTA control attribute. The user application does not handle data transfers related to OTA upgrades, so the OTA Data Attribute is excluded from the user applications GATT.

It is also possible to use an application-specific trigger to enter OTA mode, and therefore it is not necessary to include the OTA control attribute in the application's GATT database. If reboot into DFU mode is handled using some other mechanism, then it is possible to exclude the whole OTA service from the application GATT. However, it should be noted that to be compatible with the OTA host example from the SDK or the Si Connect smartphone app the OTA trigger must be implemented as described above

The presence of the OTA Data Attribute in the GATT database is used by the OTA host example application to check whether the target device is running in normal mode (user application) or DFU mode (Apploader). Therefore, the OTA Data Attribute must not be included in the user application's GATT. The OTA-enabled examples in the Bluetooth SDK only expose the OTA Control Attribute.

The four characteristics after the OTA data attribute are automatically added in the GATT database that is used by Apploader. These include version information that can be read by the OTA client before starting the firmware update. For example, by checking the AppLoader version, the OTA client may check if a full or partial update is needed.

The Apploader version is a 8-byte value that consists of four two-byte fields, indicating the Apploader version in the form <major>.<minor>.<patch>-<build>. For example, value 010000000000170b can be interpreted as version "1.0.0-2839".

The OTA version is a 1-byte value that indicates the OTA protocol version for compatibility checking. This version number is incremented only when needed, if there are some changes in the OTA implementation that may cause backward compatibility issues.

The Gecko Bootloader version is a 4-byte value that is configured in a Gecko Bootloader project in **config/btl_core_config.h**). The two most significant bytes are the major and minor numbers. The other two bytes are customer-specific, and they can be set to indicate certain Gecko Bootloader configuration options (for example, whether secure boot is required or not). As an example, value 00000401 indicates that the Gecko Bootloader version is "1.4" and the customer-specific part is 0x0000 (this is the default if no customer-specific version info has been configured in the Gecko Bootloader project).

The application version is a 4-byte value, and it is initialized to the same value that is defined in **config/app_properties_config.h**. The encoding of this value is application specific. The application properties file is discussed in more detail in section Application Properties in OTA Mode.

Apploader does not include support for encryption or bonding and therefore there are no access restrictions on any of the characteristics listed in Table 3. Because the user application has its own GATT database it is possible to include additional security requirements there as needed. For example, the user application can require that the OTA control attribute is writable only by a bonded client so that only bonded client can trigger reboot into DFU mode.

For additional security, it is recommended to configure the Gecko Bootloader to use secure boot and signed GBL images.

3.6 OTA Advertising Data

The default OTA advertising data includes the device name, TX power, advertising flags, and the Bluetooth device address. The following text snippet illustrates typical default raw OTA advertising data and how it is dissected into different advertising data elements.

```
0x02010604094F5441081B005B7728E20A68020A00: raw OTA advertising data
02:                         length = 2bytes
01:                         type = flags
06:                         value = 6 (General Discoverable Mode, BR/EDR Not Supported)
-----------------------------------------------------------------------------------------
04:                         length = 4 bytes
09:                         type = complete local name
4F5441:                     value = OTA
-----------------------------------------------------------------------------------------
08:                         length = 8bytes
1B:                         type = BL device address
005B7728E20A68:             value = 00(public) 5B:77:28:E2:0A:68
-----------------------------------------------------------------------------------------
02:                         length = 2 bytes
0A:                         type = Tx Power
00:                         value = 0dBm
```

For Series 1 devices, `sl_bt_ota_set_advertising_data(packet_type, adv_data_len, adv_data)` can be used to set custom OTA advertising data. The packet type identifies whether data is intended for advertising packets or scan response packets.

2: OTA advertising packets

4: OTA scan response packets

You can set a maximum of 31 bytes of data.

Note: The OTA configuration commands must be called after NVM3 (PS) initialization has been done—that is, after `sl_bt_init()`.

The Bluetooth device address that is used in OTA mode is determined as follows:

- Use a static random address if it has been enabled in the OTA configuration flags.
- If the user application has overridden the default Bluetooth address (using command `sl_bt_system_set_identity_address()`), then use this address.
- The default Bluetooth address (programmed into the device in production) is used if neither a static random address nor custom address has been defined.

For series-1 devices, in OTA mode, the TX power is hardcoded to 0 dBm. For series-2 devices, the TX power level can be configured using the Bluetooth Apploader OTA DFU component inside the bootloader-apploader project.

3.7 Application Properties in OTA Mode

The source file **app_properties.c** must be included in projects that use OTA and the Gecko Bootloader. This file is included in the SDK examples by default. Application properties are stored in a fixed location in code flash so that Apploader can access the data when the device is running in OTA mode. The properties include a 32-bit version number that is application-specific. It is up to the application designer to decide how this value is encoded. This value is exposed in the GATT database used by Apploader so that the OTA client can read it over the Bluetooth connection after the device has been rebooted into OTA mode.

The version information is set using following #define in **platform/bootloader/config/app_properties_config.h**:

```
// <o SL_APPLICATION_VERSION> Version number for this application
// <0-4294967295:1>
// <i> Default: 1 [0-4294967295]
#define SL_APPLICATION_VERSION                1
```

The default value is set to 1, but it is strongly recommended that meaningful version number information is added here so that the OTA client can check the exact version that is installed on the target device. This allows better management of OTA updates of units that are deployed in the field, especially in cases where units are running different versions of the application. If Apploader does not detect any valid application at all, then the application version in the GATT database is initialized to value zero.

3.8 OTA Error Codes

When a new GBL file is being uploaded, the Apploader performs various checks. Apploader can signal possible errors to the OTA client in two ways:

1. Response to the OTA termination code (0x03) that is written to the OTA control characteristic.
2. Response to writes to the OTA_data characteristic.

Option 2 is not available if the client uses unacknowledged writes. In that case, the possible error code is not available until the entire file has been uploaded and client finishes the upload by writing to the OTA control characteristic.

The OTA client must always check the response value to the last write to the OTA control characteristic. Any non-zero value indicates that the update was not successful. In that case, the device is not able to boot into the main program but rather stays in OTA mode. This makes it possible to try the update again.

The following table summarizes the possible result codes returned by Apploader.

Table 5 Apploader Result Codes

Result Code	Name	Description
0x0000	OK	Success / No errors found.
0x0480	CRC_ERROR	CRC check failed, or signature failure (if enabled).
0x0481	WRONG_STATE	This error is returned if the OTA has not been started (by writing value 0x0 to the control endpoint) and the client tries to send data or terminate the update.
0x0482	BUFFERS_FULL	Apploader has run out of buffer space.
0x0483	IMAGE_TOO_BIG	New firmware image is too large to fit into flash, or it overlaps with Apploader.
0x0484	NOT_SUPPORTED	GBL file parsing failed. Potential causes are for example:
"	"	1) Attempting a partial update from one SDK version to another (such as 2.3.0 to 2.4.0)
"	"	2) The file is not a valid GBL file (for example, client is sending an EBL file)
0x0485	BOOTLOADER	The Gecko bootloader cannot erase or write flash as requested by Apploader, for example if the download area is too small to fit the entire GBL image.
0x0486	INCORRECT_BOOTLOADER	Wrong type of bootloader. For example, target device has UART DFU bootloader instead of OTA bootloader installed.
0x0487	APPLICATION_OVERLAP_APPLOADER	New application image is rejected because it would overlap with the Apploader.
0x0488	INCOMPATIBLE_BOOTLOADER_VERSION	Apploader in requires Gecko Bootloader v1.11 or higher.
0x0489	ATT_ERROR_APPLICATION_VERSION_CHECK_FAIL	Apploader fails checking application version.

Note that the error codes listed above are applicable only when testing with the NCP host example. The upper half of the result code (0x04**) is generated by the BLE stack running on the NCP host device. The size of the ATT error code that is transmitted over the air is one octet. Values in the range 0x80-0x9F are reserved for application-specific errors in the Bluetooth specification.

3.9 Firmware Upgrade from PS Store to NVM3

If an application is already in the field using PS Store and should be upgraded to use NVM3, it can be upgraded using OTA DFU with new firmware that already uses NVM3.

However, in this case the data stored in the PS Store cannot be preserved. All bonding information and stored user data will be lost. Nevertheless, the new application can reinitialize the NVM area (at the end of the main flash) to use NVM3 instead of PS Store, and after the upgrade NVM3 will be functional.

Upgrading software from PS Store to NVM3 is challenging, mostly because the application provides information to the Apploader through non-volatile memory (PS Store / NVM3), which gets upgraded as well. The following are the detailed steps to perform an OTA upgrade from PS Store to NVM3, where the device doing the upgrade is bonded with the device to be upgraded.

Note: The procedure illustrates the situation where the upgrader and the device to be upgraded are bonded to highlight all the challenges. Bonding is not a condition for upgrading from PS Store to NVM3.

1. The device uses an application with PS Store.
2. The application sets `random address OTA flag` and `OTA device name` in PS Store 10.2_2020q4.
3. The Smartphone opens a connection to this device and gets bonded (if not bonded already).
4. The application stores bonding information in PS Store.
5. The Smartphone resets the device into OTA mode by writing 0x00 into the OTA control characteristic.
6. Apploader (with PS Store support) starts.
7. Apploader advertises with a random address and the OTA device name.
8. The Smartphone connects and uploads a new Apploader (with NVM3 support).
9. The device resets and applies the new Apploader image.
10. The new Apploader (with NVM3 support) starts.
11. The Apploader advertises with the public address and with the default name ("Apploader"), because it cannot read the random address flag and the OTA device name from PS Store.
12. The Smartphone sees the device as bonded because bonding information is associated with the public address, but Apploader does not support bonding.
13. The Smartphone removes bonding information for the device before re-connecting.
14. The Smartphone connects and uploads a new application (with NVM3 support).
15. The new application starts.
16. The application initializes NVM3 by reformatting the NVM area.
17. The application sets `random address OTA flag` and `OTA device name` in NVM3.
18. The Smartphone opens a connection and gets bonded (again).
19. The application stores bonding information in NVM3.

After this, NVM3 to NVM3 update will work normally:

1. The device uses an application with NVM3.
2. The application sets `random address OTA flag` and `OTA device name` in NVM3.
3. The Smartphone opens a connection to this device and gets bonded (if not bonded already).
4. The application stores bonding information in NVM3.
5. The Smartphone resets the device into OTA mode by writing 0x00 into the OTA control characteristic.
6. Apploader (with NVM3 support) starts.
7. Apploader advertises with a random address and the OTA device name.
8. The Smartphone connects and uploads a new Apploader (with NVM3 support).
9. The device resets and applies the new Apploader image.
10. The new Apploader (with NVM3 support) starts.
11. Apploader advertises with a random address and the OTA device name.
12. The Smartphone connects and uploads a new application (with NVM3 support).
13. The new application starts.
14. The Smartphone opens a connection and encrypts the connection with existing bonding information.
15. Bonding information is still stored in NVM3.

4 Working with Apploader and Secure Boot

When employing secure boot, the Apploader and application must be signed individually for the application to be allowed to run. This is because the Apploader authenticates the application before allowing it to run, just as the bootloader authenticates the Apploader be- fore allowing it to run.

This section describes how to work with this capability in a production environment.

4.1 Creating a Single Signed Image with a Batch File

As described previously, a signed GBL file can be produced by executing `create_bl_files.bat`/`create_bl_files.sh` with a private signing key file, `app_sign_key.pem`, in the same folder. The resulting GBL file, `full.gbl`, can be flashed directly to the target device for a successful boot. This method is convenient for testing secure boot during the development process, but is not secure for produc- tion since it requires the private signing key to be available in plain PEM format, rather than isolating it in a Hardware Security Module (HSM) and does not support the use of bootloader certificates.

4.2 Signing Firmware Images for Production

For Series One Devices:

To sign a firmware image using an HSM, the image must first be separated into the Apploader and application parts as follows.

1. Extract the Apploader portion with the following command: `objcopy -O srec -j .text_apploader* apploader.s37`.
2. Sign the Apploader for secure boot. For specific instructions on signing images with an HSM, see ‘Signing an Application for Secure Boot using a Hard Security Module’ in *UG162*. It is also possible to sign the Apploader with a certificate although direct signing is sufficient for most use cases. For instructions on signing with a certificate with an HSM, see “Signing an Application for Secure Boot using an Intermediary Certificate” in *UG162*.
3. Extract the application with the following command: `objcopy -O srec -R .text_apploader* -R .text_signature* application.s37`.
4. Sign the application for secure boot. The instructions for signing the apploader in step 2 above also apply to the application.
5. Combine the signed apploader and application into a single image as follows: `commander convert <signed apploader>.s37`
6. `<signed application>.s37 –outfile signed_fw_image.s37`.
7. Optionally, see “Creating a Partial Signed and Encrypted GBL Upgrade File for Use with a Hardware Security Module” and “Creating a Signed GBL File Using a Hardware Security Module” in *UG162*.

For series two devices (EFR32xG2x), the apploader can be included in the bootloader project as a software component. This makes it possible to sign the application and bootloader binaries without any need to perform steps 1 – 6 above.

For more information on secure boot, see *AN1218, Series 2 Secure Boot with RTSL*

5 OTA Delta DFU

SiSDK 2024.6.1 introduced a new feature called Delta DFU. This feature can be used to speedup OTA DFU, and is especially useful in large wireless networks when upgrading large amounts of devices. Delta DFU is accomplished by:

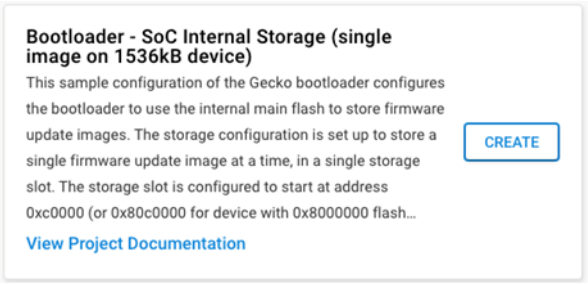
1. Comparing the current and new application firmwares
2. Creating a patch file based on the comparison
3. Only uploading the patch file to the target device

Delta DFU is not supported when using Apploader. Instead, OTA must be implemented in the application as described in section 3.3, with an internal-storage bootloader. The storage space must be large enough to hold the patch file and the reconstructed firmware.

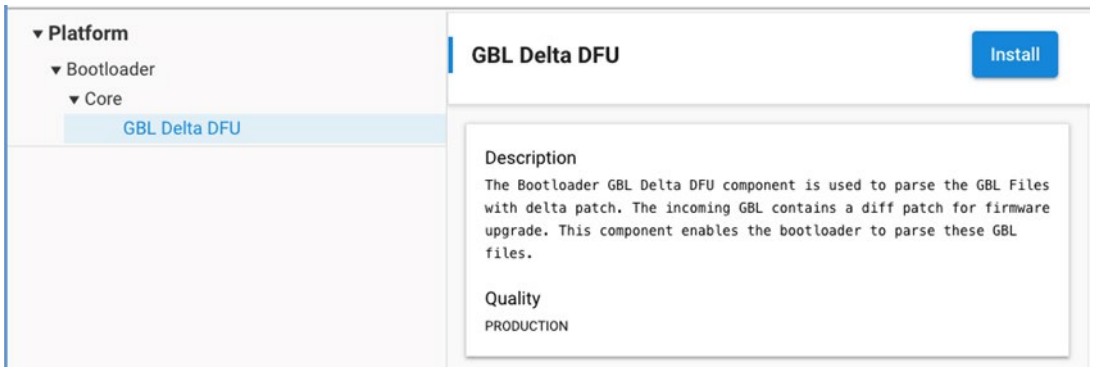
5.1 Preparing the bootloader

In Delta DFU, the bootloader takes care of applying the patch file to the current application firmware. For this, we need to prepare the bootloader:

1. Create an internal-storage bootloader project

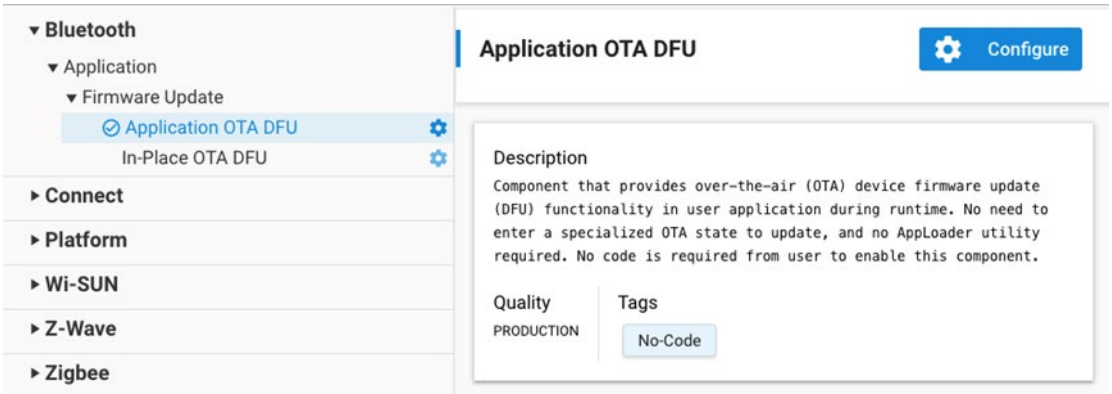


2. Add the bootloader_gbl_delta_dfu software component



5.2 Preparing the application

Both the current and the new application must support application-based OTA. Sample applications include the **in_place_ota_dfu** software component by default. However, OTA Delta DFU does not use the in-place method for DFU. Therefore, **in_place_ota_dfu** must be replaced by the **app_ota_dfu** software component.



5.3 Creating a patch file

The patch file can be created using Simplicity Commander. For this, we need both the current firmware file, and the new version.

```
commander gbl create patch.gbl --app app_v2.out --delta-app app_v1.out
```

For this to work, an ELF file with the same name must exist next to the application image. Once the patch file is created it can further be compressed using LZMA or LZ4. More information on creating a patch file can be found in *UG162: Simplicity Commander Reference Guide*.

5.4 Delta DFU process

Once a patch file has been created, it can be uploaded to the target device via OTA just as before in a partial update. When the patch file has been uploaded, the bootloader will reconstruct it into a full firmware image. This firmware image will then be written over the old image.

5.5 Delta DFU and Secure Boot

When using Delta DFU with Secure Boot, make sure both the current firmware and the new version (app_v1 and app_v2) are signed using the same private key before creating the patch file. When the patch file is created, as described in Creating a patch file, the signature of app_v2 will be included in the patch. Therefore, the reconstructed firmware will have the signature of app_v2, and the new application will boot as expected.

Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

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