



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



This application note includes detailed information on using the Silicon Labs Gecko Bootloader with Silicon Labs Bluetooth applications. It supplements the general Gecko Bootloader implementation information provided in [UG266: Silicon Labs Gecko Bootloader User's Guide](#). If you are not familiar with the basic principles of performing a firmware upgrade or want more information about upgrade image files, refer to [UG103.6: Application Development Fundamentals: Bootloading](#).

KEY POINTS

- Gecko Bootloader overview
- Using Gecko bootloader for BGAPI UART DFU
- Using Gecko bootloader for Bluetooth OTA upgrade

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 bootload 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: Application Development Fundamentals: Bootloading*.

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 plugin configuration. Plugins can be enabled and configured through the Simplicity Studio IDE.

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

Unlike the legacy Bluetooth bootloaders, 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 Upgrade (DFU)

This is the firmware upgrade used in NCP-mode Bluetooth applications. A GBL image containing the new firmware is written to target device using UART as the physical interface and BGAPI protocol. 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.

2.1 UART DFU Options

The target device must be programmed with the Gecko Bootloader configured as **BGAPI UART DFU Bootloader**. The default configuration is as follows:

UART Options

- Selected USART: USART0
- Baudrate 115200
- TX pin: PA0
- RX pin: PA1
- HW flow control: disabled
- UART enable port: PA5 (VCOM_ENABLE on WSTK)

GPIO Activation

- GPIO activation is enabled and mapped to pin PF6 (active low).

The default settings are suitable for testing with a WSTK (Wireless Starter Kit). These settings can be easily changed by editing the Plugin parameters. This is done by going to the Plugins tab of the AppBuilder project, and selecting the UART driver plugin. Here, Hardware Flow Control can be enabled or disabled, and the baud rate and pinout can be configured.

The GPIO activation plugin is enabled by default, allowing bootloader entry by activating a GPIO through reset. This plugin can be disabled if this functionality is not desired, or the GPIO pin used for this can be configured under the GPIO Activation plugin on the Plugins tab.

2.2 UART DFU Process

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

1. Boot the target device into DFU mode (by sending `dfu_reset(1)`).
2. Wait for the `DFU boot` event.
3. Send the command `DFU Flash Set Address` to start the firmware upgrade
4. Send the entire contents of the GBL upgrade image (using the command `DFU flash upload`).
5. After sending all data, the host sends the command `DFU flash upload finish`.
6. To finalize the upgrade, the host resets the target device into normal mode (by sending `dfu_reset(0)`).

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

At the beginning of the upgrade, the NCP host uses the command `Flash Set Address` to define the start address. The start address shall be always 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.

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 project's root folder. Two scripts are provided in the SDK examples:

- **create_ebl_files.bat** (for Windows)
- **create_ebl_files.sh** (for Linux / Mac)

Running the **create_ebl_files** script creates three GBL files in a subfolder named `output_gbl`. The file named **full.gbl** is the upgrade image used for UART DFU. The other two files (**app.gbl**, **stack.gbl**) 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.

Note: The script also generates EBL files. EBL is the file format that is used in Bluetooth SDK versions 2.1.1 and earlier. The generated EBL files are stored in the subfolder `output_ebl`. These files can be ignored when working with Gecko Bootloader.

2.4 UART DFU Host Example

The UART DFU host example is a C program that is located under the SDK examples in following directory:

```
C:\SiliconLabs\SimplicityStudio\v4\developer\sdk\gecko_sdk_suite\v1.0\app\bluetooth_2.x\  
examples_ncp_host\uart_dfu
```

In Windows this program can be built using, for example, MinGW or Cygwin. 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 named **uart-dfu.exe** is created in subfolder **exe**.

Before running the example you need to check the COM port number associated with your NCP target. For more details, see *AN1042: Using the Silicon Labs Bluetooth® Stack in Network Co-Processor Mode*.

The `uart-dfu.exe` program requires three command line arguments:

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

Example usage and expected output:

```
./uart-dfu.exe COM42 115200 full.gbl  
Syncing..DFU OK  
Bootloader version: ***  
.....  
.....  
finish
```

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

To enable Bluetooth OTA upgrade, the target device must be programmed with Gecko Bootloader that is configured as **Internal Storage Bootloader**. This is an application bootloader and it requires that the new firmware image acquisition is managed by application.

A Bluetooth application developed with Silicon Labs Bluetooth SDK comprises two parts:

- The Bluetooth stack, provided as precompiled library
- The user application that uses services provided by the Bluetooth stack

Most of the OTA functionality is built in to the Bluetooth stack, which greatly simplifies application development. OTA upgrade is managed by *supervisor*, a maintenance application that is part of the stack. The Bluetooth stack can be started in DFU mode, which means that the supervisor is run instead of the user application. In DFU mode, the supervisor application temporarily overrides the user application. This makes it possible to perform OTA upgrade without any involvement from the user application.

The only requirement for the user application is for a way to trigger a reboot into DFU 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. The following sections explain in detail how this is done in the user application.

3.1 Gecko Bootloader Configuration

The Gecko Bootloader must be configured as an application bootloader. Most of the OTA functionality is built into the Bluetooth stack and the Gecko Bootloader is only involved in copying data from download area to the final destination in flash.

For EFR32xG1, the **Bluetooth in-place OTA DFU Bootloader** configuration is used as a default. In this configuration, the upper half of the main flash, normally used to hold the Bluetooth application, is repurposed as a storage area while a Bluetooth stack upgrade is downloaded.

For EFR32xG12 and later, any application bootloader configuration may be used, using internal or external storage. The default example application configurations are suitable for Bluetooth OTA upgrades, and may be modified to fit the needs of the application.

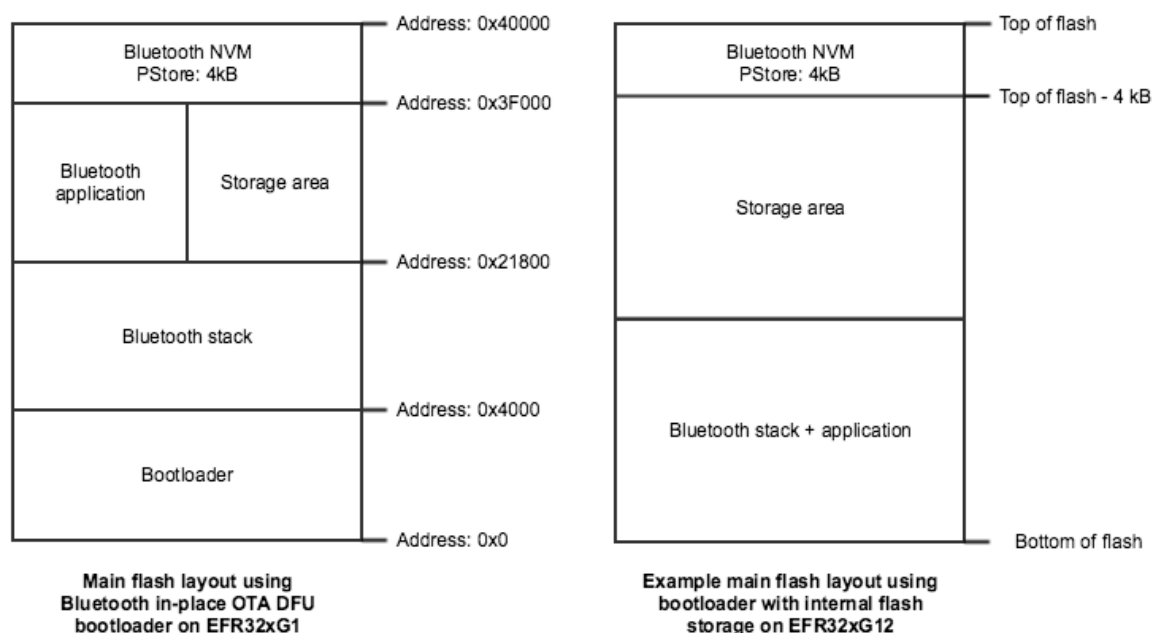


Figure 3.1. Examples of Main Flash Layout when Using Gecko Bootloader with Bluetooth OTA DFU

3.2 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 3.1. Silicon Labs OTA Service Characteristics

Characteristic	UUID	Type	Length	Support	Security	Properties
OTA Control Attribute	F7BF3564-FB6D-4E53-88A4-5E37E0326063	Hex	1 byte	Mandatory	Configurable	Write
OTA Data Attribute ¹	984227F3-34FC-4045-A5D0-2C581F81A153	Hex	Variable, max 244 bytes	Mandatory	Configurable	Write without response, Write
BLE stack version ²	4F4A2368-8CCA-451E-BFFF-CF0E2EE23E9F	Hex	8	Optional	-	Read
OTA version ²	4CC07BCF-0868-4B32-9DAD-BA4CC41E5316	Hex	1	Optional	-	Read

Notes:

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

² Stack version and OTA version are automatically added by the stack when running in DFU mode. These are optional in the application GATT database.

Table 3.2. 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
Other values	Other values are reserved for future use and must not be used by application

In DFU mode, the supervisor uses the full OTA service described above. This allows a remote Bluetooth device to upload a new firmware image, as described later in this chapter. 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 characteristic in its GATT database.

From the user application viewpoint, only the OTA control attribute is relevant. In the OTA host example reference implementation that is 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 any data transfers related to OTA upgrades and therefore the OTA Data Attribute is excluded from the user application's GATT.

Note: The supervisor has its own GATT database that is independent of the user application's GATT database. When the device is booted into DFU mode the supervisor GATT database temporarily overrides the application's GATT database.

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 (supervisor). 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 security settings for the OTA characteristics are configurable. By default, the security setting is `none`, meaning that any remote client can access these characteristics without any restrictions. The possible security options are listed below.

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

Table 3.3. OTA Characteristics Security Options

Setting	Description
None (default)	No access restrictions.
authenticated_write	Remote device must be bonded using MITM and the connection must be encrypted.
encrypted_write	Connection must be encrypted but bonding is not required.
bonded_write	Remote device must be bonded (MITM or Just Works) and the connection must be encrypted.

For more information on GATT characteristic properties, please refer to [UG118: Blue Gecko Bluetooth® Profile Toolkit Developer's Guide](#).

3.3 OTA GATT Database and Generic Attribute Service

When booted into DFU mode, the stack uses a GATT database that is different than the normal GATT used by the application. This GATT temporarily overrides the user-defined GATT during OTA.

The OTA DFU GATT database that is automatically created by the stack contains following services:

- Generic Attribute (UUID 0x1801)
- Generic Access (UUID 0x1800)
- Device Information (UUID 0x180A)
- 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 4.2, Vol. 3, Part G, 7 DEFINED GENERIC ATTRIBUTE PROFILE SERVICE.

The Generic Attribute service is automatically included in the temporary 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.

The user application needs only enable the Generic Attribute Service. The stack automatically generates an indication to the remote client when the GATT database content is changed, that is when switching into OTA DFU mode (OTA GATT is taken into use) and when returning back to normal mode (application GATT is restored).

Note: The automatic service changed indication requires that the client is bonded and has enabled the indication for this characteristic.

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 Attribute 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>

3.4 Triggering Reboot into DFU Mode from the User Application

The minimum functional requirement to enable OTA in a C-based application is to implement a ‘hook’ that allows the device to be rebooted into DFU mode. By default, this is done through the Silicon Labs OTA service.

The following code snippet is from the SoC Thermometer example supplied with the SDK. The code to enter DFU mode is similar in the other examples.

```
case gecko_evt_le_connection_closed_id:
    /* Check if need to boot to dfu mode */
    if (boot_to_dfu) {
        /* Enter to DFU OTA mode */
        gecko_cmd_system_reset(2);
    }
    else {
        ....

/* Checks if the user-type OTA Control Characteristic was written.
 * If written, boots the device into Device Firmware Upgrade (DFU) mode. */
case gecko_evt_gatt_server_user_write_request_id:
    if (evt->data.evt_gatt_server_user_write_request.characteristic==gattdb_ota_control) {
        /* Set flag to enter to OTA mode */
        boot_to_dfu = 1;
        /* Send response to Write Request */
        gecko_cmd_gatt_server_send_user_write_response(
            evt->data.evt_gatt_server_user_write_request.connection,
            gattdb_ota_control,
            bg_err_success);

        /* Close connection to enter to DFU OTA mode */
        gecko_cmd_endpoint_close(evt->data.evt_gatt_server_user_write_request.connection);
    }
    break;
```

Figure 3.2. Handling Write to OTA Control Characteristic in C Code

The event with ID `gecko_evt_gatt_server_user_write_request_id` indicates that one of the characteristics (of type user) has been written by the remote BLE client.

In this example, the code simply checks if the OTA control characteristic was written and, if so, triggers a reboot into DFU mode. Before rebooting, the application closes the Bluetooth connection. The variable `boot_to_dfu` is set so indicate that DFU reboot has been requested. When the connection closed event is raised by the stack, the application checks the variable `boot_to_dfu` and if set, performs the DFU reboot by calling `gecko_cmd_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 the stack and no further actions are needed from the user application.

3.5 OTA-Related Configurations in the Bluetooth Stack

Besides implementing the hook to enter DFU mode, the user application must implement some additional OTA-related configurations.

The user application initializes the Bluetooth stack by calling `gecko_init()`. This function takes one parameter, a pointer to a struct (of type `gecko_configuration_t`) containing various configuration parameters. The code snippet shown below is taken from the SoC Thermometer example from the Bluetooth C SDK. The three OTA-related configuration parameters are highlighted.

```
static const gecko_configuration_t config = {
    .config_flags=0,
    .sleep.flags=SLEEP_FLAGS_DEEP_SLEEP_ENABLE,
    .bluetooth.max_connections=MAX_CONNECTIONS,
    .bluetooth.heap=bluetooth_stack_heap,
    .bluetooth.heap_size=sizeof(bluetooth_stack_heap),
    .gattddb=&bg_gattddb_data,
    .ota.flags=0,
    .ota.device_name_len=3,
    .ota.device_name_ptr="OTA",
#ifdef FEATURE_PTI_SUPPORT
    .pti = &ptiInit,
#endif
};
```

Figure 3.3. OTA Configuration Parameters Passed to the Stack

The OTA parameters are collected in a smaller struct named `gecko_ota_config_t` that is part of `gecko_configuration_t`. The definition of `gecko_ota_config_t` is shown below.

```
typedef struct
{
    uint32_t flags;
    uint8_t device_name_len;
    char *device_name_ptr;
}gecko_ota_config_t;
```

Figure 3.4. OTA Configuration Struct

`flags` is a set of configuration flags. The following flag values, defined in `gecko_configuration.h`, are possible:

```
#define GECKO_OTA_FLAGS_AUTHENTICATED_WRITE    0x200
#define GECKO_OTA_FLAGS_ENCRYPTED_WRITE       0x100
#define GECKO_OTA_FLAGS_BONDED_WRITE         0x400
```

The parameter can be used to configure the security settings of the `ota_control` and `ota_data` characteristics that are part of the Silicon Labs OTA service (see [Table 3.3 OTA Characteristics Security Options on page 7](#)). If the parameter is set to zero (the default value) then there are no access restrictions. If access control is required then one of the three above values can be assigned to the `flags` parameter. Only one value can be used. Note that this setting affects only the GATT database that is used by the stack (supervisor) during DFU mode and not the GATT database of the user application.

`device_name_len` and `device_name_ptr` specify the Bluetooth device name that is used when the device has been rebooted into DFU mode. Note that, in addition to specifying the name string, the application must also specify the exact number of characters in that string in the `device_name_len` parameter.

The device name used during OTA does not have to be static. The string can be dynamically generated, for example based on the serial number of the device or some other value that uniquely identifies the device. However, the name must be set when the stack is initialized (by calling `gecko_init()`).

An alternative way to define the OTA device name is to use the command `gecko_cmd_system_set_device_name`. This method allows the name to be changed after the stack has been initialized.

In addition to configuring the OTA parameters, the application code must include the header file `att.h` in the `main.c` source file. This is needed so that an Application Address Table (AAT) is included in the firmware image and linked to a specific address. The AAT content is used internally by the stack during OTA upgrade. The application need only include the `att.h` header in the `main.c` source file.

The source file **application_properties.c** needs to be included in projects that use OTA and the Gecko Bootloader. This file sets some application properties that are used during OTA to check that application version matches the stack version. This file is included in the SDK examples by default.

3.6 Creating OTA Upgrade Images

Building a C-based Bluetooth application in Simplicity Studio does not generate the OTA DFU upgrade images (GBL files) automatically. The GBL files need to be created separately by running a script located in the project's root folder. Two scripts are provided in the SDK examples:

- **create_ebl_files.bat** (for Windows)
- **create_ebl_files.sh** (for Linux / Mac)

Running the **create_ebl_files** script creates three GBL files in a subfolder named **output_gbl**. The files named **stack.gbl** and **app.gbl** are used for OTA DFU. The file **full.gbl** is related to UART DFU upgrade and 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.

Note: The script also generates EBL files. EBL is the file format that is used in Bluetooth SDK versions 2.1.1 and earlier. The generated EBL files are stored in the subfolder **output_ebl**. These files can be ignored when working with Gecko Bootloader.

3.7 OTA DFU Host Example

The Bluetooth SDK includes an OTA host reference implementation. The example is written in C language and uses a Bluetooth development kit as modem 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 *QSG108: Getting Started with Silicon Labs' Bluetooth® Software*.

The following figure shows an overview of an OTA test setup. The OTA host application is running on a laptop that is connected to one Bluetooth development kit. These two together form the **OTA client**. The host program uses the development kit in NCP mode and communicates with it via a virtual serial port connection using the BGAPI protocol.

The target device to be upgraded over-the-air is shown on the right hand side. It is identified by its Bluetooth address.

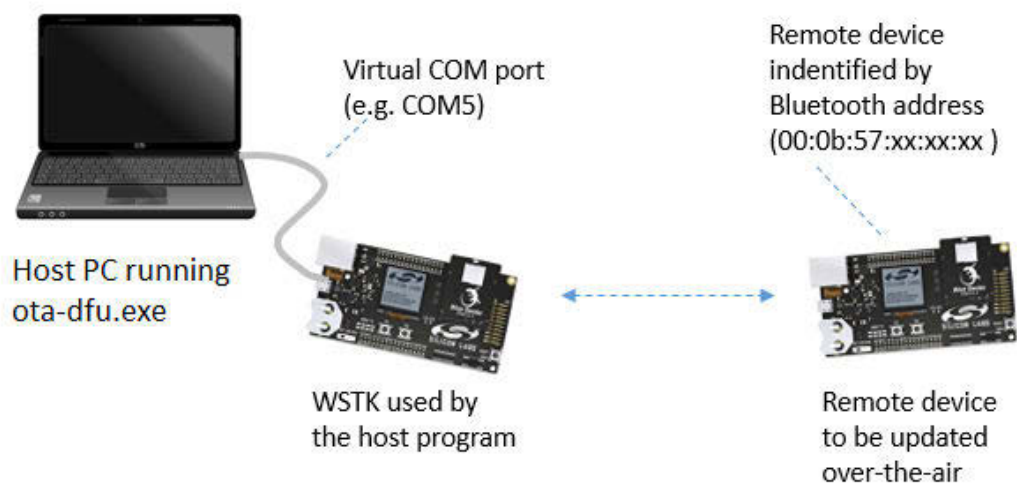


Figure 3.5. OTA test setup

3.7.1 Preparing the Development Kit for NCP Mode

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 named **NCP – Empty Target** that can be used for this purpose.

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

3.7.2 Building the OTA Host Example Application

The OTA host example is found in the following directory under the Bluetooth SDK installation tree:

```
C:\SiliconLabs\SimplicityStudio\v4\developer\sdk\gecko_sdk_suite\v1.0\app\bluetooth_2.x\  
examples_ncp_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`). After successful compilation, the executable named **ota-dfu.exe** is placed in subfolder named `exe`.

3.7.3 Running OTA with the NCP Host Example

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

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

A full OTA upgrade is done in two parts, and it requires two separate GBL files, one for the stack and another for the application. Full OTA requires the host example program to be invoked twice. An example usage is shown below:

```
./ota-dfu.exe COM49 115200 stack.gbl 00:0B:57:0B:49:23  
./ota-dfu.exe COM49 115200 app.gbl 00:0B:57:0B:49:23
```

If the application alone is going to be upgraded, then the host program is run once, with the **app.gbl** file passed as parameter. In other words, only the second of the two commands listed above is run.

3.7.4 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 (minimal OTA or full OTA).

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

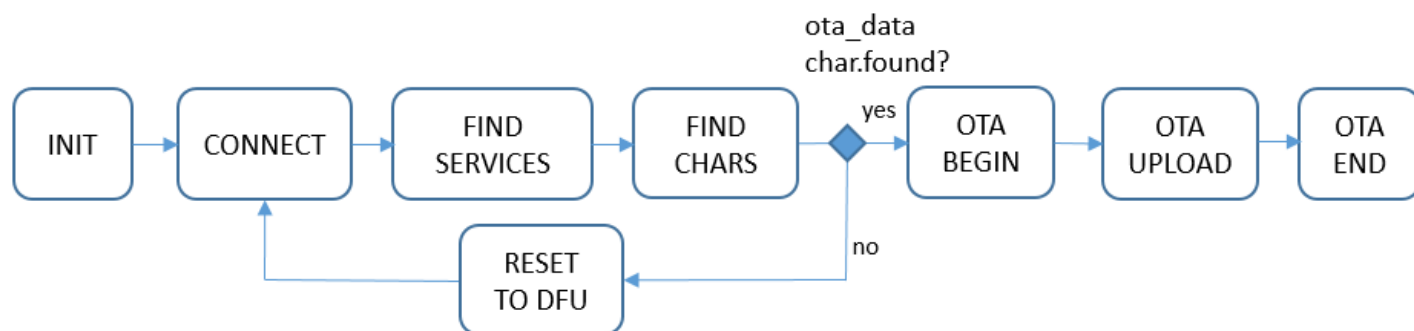


Figure 3.6. OTA Host Example State Transitions

In **INIT** state, 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.

In **CONNECT** state, 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 causes the program to be blocked.

After a connection has been established, the program moves to state **FIND SERVICES**, where it performs 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 `cmd_gatt_discover_primary_services_by_uuid`).

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

The `ota_data` characteristic may or may not be present, depending whether the target device is already in DFU mode or not. If the `ota_data` handle is not found, then the next state is **RESET TO DFU**. In this state 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.

If both `ota_data` and `ota_control` characteristic handles have been detected, the next state is **OTA BEGIN**. 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.

The state following **OTA BEGIN** is **OTA UPLOAD**. This is where the GBL file is uploaded to target device. The whole content of the GBL file is uploaded into the target device, by performing a number of write operations into the `ota_data` characteristic. The host program 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.

When the whole GBL file has been uploaded, the next state is **OTA END**. In this state 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 CHARACTERISTICS** 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 automatically created by the stack 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.

4. Implementing Device Firmware Update in the User Application

In addition to the basic UART and OTA DFU solutions discussed in previous chapters, it is possible to implement the firmware update functionality completely in the user application. This makes it possible to use a custom GATT service instead of the Silicon Labs OTA service. In case of UART DFU updates, the application can be designed to support some other protocol than BGAPI. The user application can be designed to support both OTA and UART DFU updates if needed and it is possible to support other interfaces such as SPI.

To use this update mechanism, any application bootloader configuration may be used, using internal or external storage. At least one download area must be defined and the area must be large enough to fit the full GBL file. Partial update is not supported.

4.1 Basic Steps to Update Firmware from the User Application

The general firmware upgrade sequence is explained in *UG266: Silicon Labs Gecko Bootloader User's Guide*. The basic steps are summarized below.

1. Application initializes the Gecko bootloader by calling `bootloader_init()`;
2. The download area 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, application writes the received bytes to the download area by calling `bootloader_writeStorage()`
4. (optional) Application can verify the integrity of the received GBL file by calling `bootloader_verifyImage()`
5. Before rebooting, call `bootloader_setAppImageToBootload(0)` to specify the slot ID where new image is stored
6. Reboot and instruct Gecko bootloader to perform the update by calling `bootloader_rebootAndInstall()`

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.

Note that the erase procedure in step 2) above takes several seconds to complete. If the new image is downloaded over a Bluetooth connection then 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 while the writing progresses. This can be done using `bootloader_eraseRawStorage()`.

4.2 Enabling Gecko Bootloader API

Gecko bootloader has an application interface exposed through a function table in the bootloader. To be able to call Gecko bootloader functions from your Bluetooth application, the following source files must be added into the project:

`btl_interface.c` (common interface)

`btl_interface_storage.c` (interface to storage functionality)

These files are found in the Gecko SDK suite in following directory:

```
\\gecko_sdk_suite\1.0\platform\bootloader\api\
```

The corresponding include files must be added to the source file(s) that call any Gecko bootloader functions:

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

Additionally, make sure that the following `\\platform\bootloader\` directory is added in the include paths. Using a symbolic path to the SDK installation, this can be configured by adding following string to the include paths:

```
"${StudioSdkPath}/platform/bootloader"
```

4.3 Example implementation of Bluetooth OTA update under application control

Silicon Labs OTA service is used here as a practical example on how to perform firmware update from user application code. The code sample included in Appendix 1 implements OTA update using the same service definition that was discussed in section 3. [Bluetooth OTA Upgrade](#). The main differences are:

1. The OTA is performed fully under application control and device is not rebooted into DFU mode (supervisor is not involved in the update)
2. Only full update is supported and the update is performed using one single GBL file

5. Appendix: Bluetooth OTA Update under Application Control

```
    evt=gecko_wait_event();

    switch(BGLIB_MSG_ID(evt->header))
    {
    case gecko_evt_le_connection_closed_id:
        if(ota_image_finished)
        {
            bootloader_setAppImageToBootload(0);
            bootloader_rebootAndInstall();
        }
        break;
    case gecko_evt_gatt_server_user_write_request_id:
        /*OTA support*/
        uint32_t connection=evt->data.evt_gatt_server_user_write_request.connection;
        uint32_t characteristic=evt->data.evt_gatt_server_user_write_request.characteristic;
        if (characteristic==gattdb_ota_control)
        {
            switch (evt->data.evt_gatt_server_user_write_request.value.data[0])
            {
            case 0://Erase and use slot 0
                bootloader_init();
                bootloader_eraseStorageSlot(0);
                ota_image_position=0;
                ota_in_progress=1;
                break;
            case 3://END OTA process
                //wait for connection close and then reboot
                ota_in_progress=0;
                ota_image_finished=1;
                break;
            default:
                break;
            }
        }
        else if (characteristic==gattdb_ota_data)
        {
            if(ota_in_progress)
            {
                bootloader_writeStorage(0,//use slot 0
                    ota_image_position,
                    evt->data.evt_gatt_server_user_write_request.value.data,
                    evt->data.evt_gatt_server_user_write_request.value.len);
                ota_image_position+=evt->data.evt_gatt_server_user_write_request.value.len;
            }
        }

        gecko_cmd_gatt_server_send_user_write_response(connection,characteristic,0);

        break;
    }
}
```


Silicon Labs

Simplicity Studio™4



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/loT



SW/HW
www.silabs.com/simplicity



Quality
www.silabs.com/quality



Support and Community
community.silabs.com

Disclaimer

Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Silicon Labs shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any Life Support System without the specific written consent of Silicon Labs. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons.

Trademark Information

Silicon Laboratories Inc.®, Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, Clockbuilder®, CMEMS®, DSPLL®, EFM®, EFM32®, EFR®, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Ember®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, ISOModem®, Micrium, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress®, Zentri and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. All other products or brand names mentioned herein are trademarks of their respective holders.



SILICON LABS

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

<http://www.silabs.com>