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 button PB0 (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 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_bl_files.bat** (for Windows)
- **create_bl_files.sh** (for Linux / Mac)

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.

Note: Depending on the SDK version, the script may also generate 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 (the exact path depends on the installed SDK version):

```
C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\<version>\app\bluetooth\
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.

3.1 AppLoader

A Bluetooth application developed with Silicon Labs Bluetooth SDK comprises two parts: AppLoader and the user application. AppLoader is a small standalone application that is required to support in-place OTA updates. AppLoader can run independently of the user application. It contains a minimal version of the Bluetooth stack, including only those features that are necessary to perform the OTA update. 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.
- The AppLoader itself can also be updated.
- Only one Bluetooth connection is supported, GATT server role only.
- Encryption and other security features such as bonding are not supported.

The user application is placed in code flash after AppLoader. The default linker script provided in the SDK places the user application so that it starts at the next flash sector following 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 does not need to be supported then the AppLoader can be removed completely to free up flash for other use (code space or data storage). Section 4. Implementing Device Firmware Update in the User Application describes how OTA can be implemented in application code, without any involvement of AppLoader.

For more details on AppLoader and the overall structure of a Bluetooth application see *UG136: Silicon Labs Bluetooth® C Application Developers Guide*.

3.2 Gecko Bootloader Configuration

The Gecko Bootloader must be configured as an application bootloader. The OTA functionality is implemented almost entirely in the AppLoader, or alternatively in the user application. The Gecko Bootloader takes care of copying data from the download area to the final destination in flash. Additionally, AppLoader takes advantage of some features supported by Gecko Bootloader, for example passing the incoming GBL image.

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.

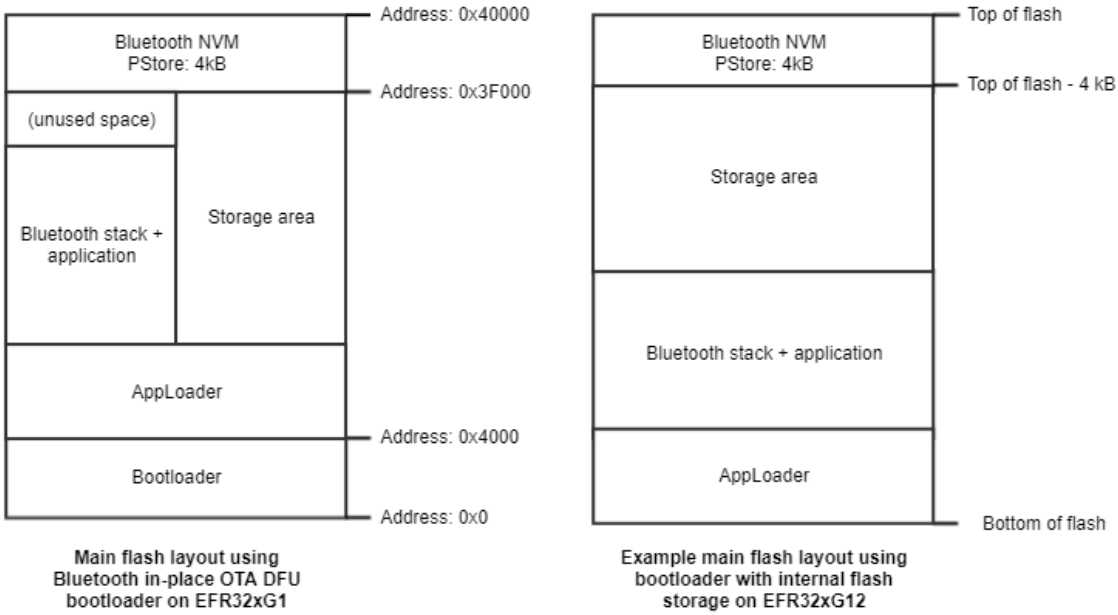


Figure 3.1. Examples of Main Flash Layout when Using Gecko Bootloader with Bluetooth OTA DFU

3.3 In-Place OTA Process

Most of the OTA functionality is handled autonomously by the AppLoader, which greatly simplifies application development. The minimum requirement for the user application is for a way to trigger a reboot into DFU mode. Rebooting into DFU mode in this context means that after the device is reset, the AppLoader is run instead of the user application. 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. The following sections explain in detail how this is done in the user application.

AppLoader supports two types of update:

- Full update: both AppLoader and the user application are updated
- Partial update: only the user application is updated

Note: In earlier stack versions (SDK 2.6.x and earlier), the meaning of partial and full update is different compared to the current OTA implementation. To avoid confusion, the main differences between the old and new OTA are summarized below.

	SDK 2.6.x and older	SDK 2.7.x and later
OTA update files generated	stack.gbl: Bluetooth stack and OTA update part (*supervisor*) app.gbl: User application	apploader.gbl: AppLoader (including minimal Bluetooth stack) application.gbl: user application (including full Bluetooth stack)
Partial update	Only user application is updated. Bluetooth stack remains the same. Application must be built with same SDK version that is currently installed in the target device.	User application is updated. The Bluetooth stack is part of user application, therefore the stack is also updated. The user application and AppLoader do not need to be built from same SDK.
Full update	Both the Bluetooth stack and the user application are updated in two phases (first stack, then application)	Both the AppLoader and user application (including Bluetooth stack) are updated in two phases (first AppLoader, then user application)

From the OTA client viewpoint, the overall OTA process is the same in both old and new versions. Full update is performed by uploading two GBL files into the target device. Partial update requires only one file. Because the mechanism of uploading GBL files over the air is identical, the new OTA solution introduced in SDK 2.7.0 is backwards compatible:

- Device running application from SDK 2.6.x (or older, down to 2.0.x) can be upgraded to 2.7.x using OTA
- Device running 2.7.x firmware can be downgraded to 2.6.x or older using OTA

The partial update process using AppLoader consists of 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 (application.gbl) 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. Full update is done in two steps. Updating the AppLoader always erases the user application and therefore AppLoader update must always be followed by application update.

The first phase of full update updates the AppLoader and it consists of 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 AppLoader image (apploader.gbl) is uploaded to the target.

6. AppLoader copies the image into the download area (specified in Gecko bootloader configuration).
7. When upload is finished and 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.

The SDK includes an example OTA client implementation that can be used to perform both full and partial updates. This example app is described in section 3.9 OTA DFU Host Example. Full and partial OTA can also be performed using the Blue Gecko smartphone app.

3.4 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	Properties
OTA Control Attribute	F7BF3564-FB6D-4E53-88A4-5E37E0326063	Hex	1 byte	Mandatory	Write
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 AppLoader version starting from SDK 2.7.0; was stack version in earlier versions.

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.
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. 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 (unless the OTA update is implemented fully in application code, as described in section 4. Implementing Device Firmware Update in the User Application.

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.

It is also possible to use an application-specific trigger to enter OTA mode, and therefore it is not absolutely 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 Blue Gecko smartphone app the OTA trigger must be implemented as described above.

Note: AppLoader has its own GATT database that is independent of the user 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 (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. The OTA version number in SDK 2.7.0 is 3. 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 (file btl_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 the file application_properties.c. The encoding of this value is application-specific. In the SDK example projects, the application_properties.c source is included but the application version is set to zero. In real applications it is highly recommended to use some meaningful application version so that it can be read over-the-air when the device is in OTA mode. The application properties file is discussed in more detail in section 3.7 OTA-Related Configurations in the Bluetooth Stack.

AppLoader does not include support for encryption or bonding and therefore there are no access restrictions on any of the characteristics listed in Table 3.1 Silicon Labs OTA Service Characteristics on page 9. 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.

3.5 OTA GATT Database and 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 4.2, 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 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

3.6 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. 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) {                                    2.
    /* 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;                                                                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 Bluetooth 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 AppLoader and no further actions are needed from the user application.

3.7 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),
  .gattdb=&bg_gattdb_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:

Flag	Description
GECKO_OTA_FLAGS_RANDOM_ADDRESS	If set, AppLoader will use the static random address during OTA mode.

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

- Use 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 `cmd_system_set_device_name`) then this address is also used during OTA (starting with SDK version 2.8.0).
- The default Bluetooth address (programmed into the device in production) is used if neither static random address nor custom address is defined.

`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 API call **cmd_system_set_device_name**. This method allows the name to be changed after the stack has been initialized.

The source file **application_properties.c** needs to 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 application_properties.c:

```
/// Version number for this application (uint32_t)
#define APP_PROPERTIES_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.

Note: Earlier SDK versions required that header file aat.h must be include by user application. Beginning with SDK version 2.7.0 this file is no longer needed and it **must not be included**.

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

Running the **create_bl_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.

3.9 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.

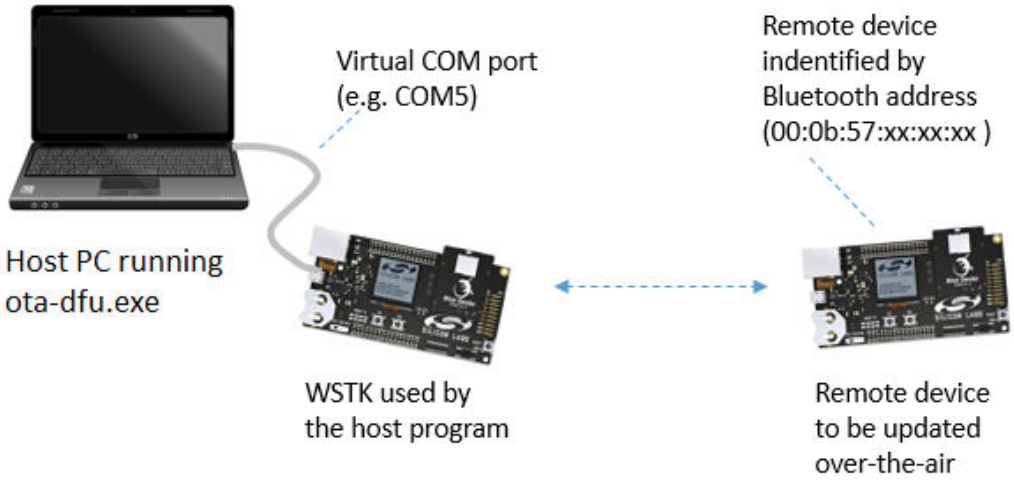


Figure 3.5. OTA test setup

3.9.1 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.

3.9.2 Building the OTA Host Example Application

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

```
C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\<version>\app\bluetooth\
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.

3.9.3 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 AppLoader and another for the user application. Full OTA requires the host example program to be invoked twice. An example usage is shown below:

```
./ota-dfu.exe COM49 115200 apploader.gbl 00:0B:57:0B:49:23
./ota-dfu.exe COM49 115200 application.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 **application.gbl** file passed as parameter. In other words, only the second of the two commands listed above is run.

Note: Starting from SDK 2.7.0, the user application also includes the Bluetooth stack and therefore the Bluetooth stack can be updated without full update. Full update is needed only if the AppLoader needs to be updated.

3.9.4 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 (partial OTA or full OTA).

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

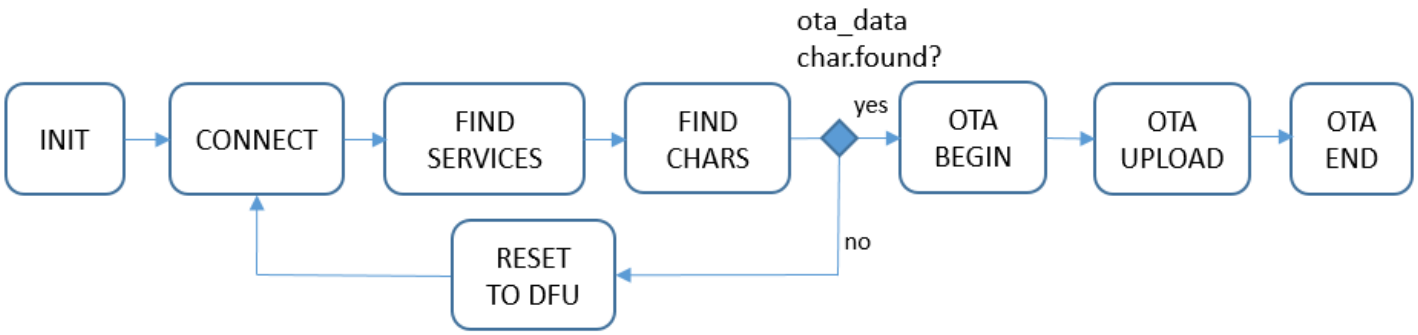


Figure 3.6. OTA Host Example State Transitions

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 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.

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. The download area must not overlap with the user application and therefore this DFU solution is not applicable to devices or modules based on the EFR32xG1 (see Figure 3.1 Examples of Main Flash Layout when Using Gecko Bootloader with Bluetooth OTA DFU on page 6).

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 are 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 (exact path depends on installed SDK version):

\gecko_sdk_suite\v2.1\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 strings to the include paths:

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

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 the appendix 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 (AppLoader is not involved in the update).
2. Only full update is supported and the update is performed using one single GBL file (application.gbl).
3. In-place OTA is not supported and the download area must not overlap with the user application.
4. The OTA_data characteristic must be added to the user application's GATT database.

Because AppLoader is not involved in this type of DFU implementation, it can be completely removed from the project to minimize flash usage. AppLoader can be removed from the SDK examples by deleting binapploader.o from the project linker settings.

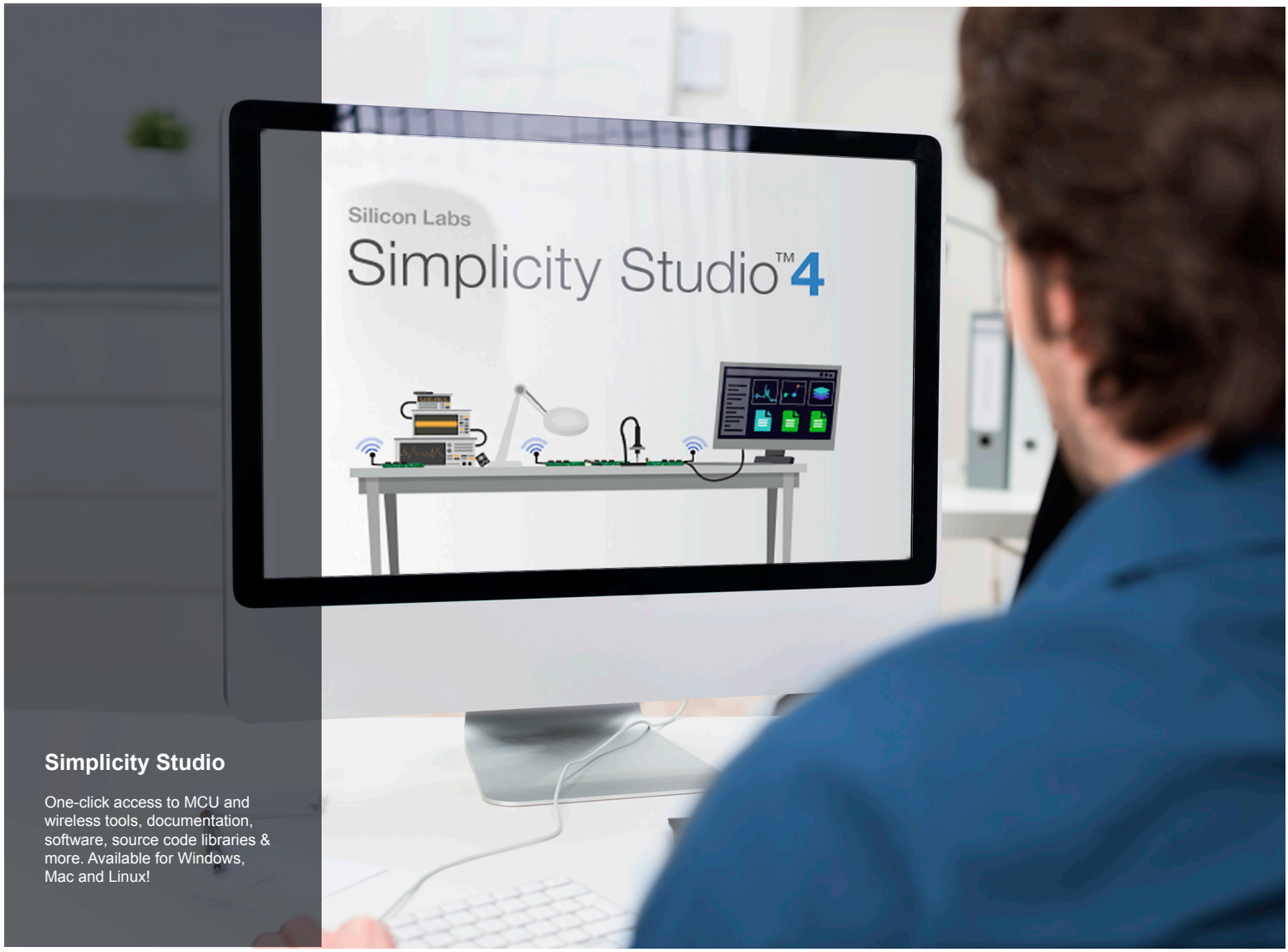
5. Appendix: Bluetooth OTA Update under Application Control

```
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;
}
```



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 and Community
community.silabs.com

SILICON LABS

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

http://www.silabs.com