AN1045: *Bluetooth* ® Over-the-Air Device Firmware Update for EFR32xG1 and BGM11x Series Products

This application note describes the legacy OTA (Over-the-Air) firmware update mechanism used in the Silicon Labs Blue Gecko Bluetooth SDK (Software Development Kit) for EFR32xG1 SoCs and BGM11x and BGM121/BGM123 modules. OTA enables deployment of firmware updates to devices in the field, making it possible to introduce new features or other changes after a product has been launched.

KEY POINTS

- OTA firmware update basic procedure
- Different types of OTA updates
- How to enable OTA in user applications
- Creation of OTA update images
- OTA host example

The Silicon Labs Bluetooth SDK allows designers to easily add OTA capability in their products. Most of the OTA functionality is built in to the Bluetooth stack, which greatly simplifies the user application development. This documents explains how OTA is implemented in the stack and what needs to be done in the user application code to enable OTA.

NOTICE: The legacy OTA update method was deprecated in version 2.6.0 and removed from version 2.7.0 of the Bluetooth SDK (December 2017). For information on the current OTA update method, see *UG266: Silicon Labs Gecko Bootloader User's Guide* and *AN1086: Using the Gecko Bootloader with Silicon Labs Bluetooth Applications*.

1. Introduction

The ability to update the firmware of units already deployed in the field is a common requirement for many products. For example, it may be necessary to add new features into products after the first version has been launched. If a software bug or some unanticipated compatibility issue is identified after the product has been shipped, it is essential to provide a firmware update that fixes the problem, without the need to recall units or for the customer to take them to service for reprogramming.

Products that use Bluetooth Low Energy technology are often designed to work with smartphones, tablets, or similar consumer electronic devices that are connected to the Internet. This makes it possible to implement OTA (Over-the-Air) firmware update capability without adding any extra cost or significant increase in the software complexity. The Internet of Things (IoT) also poses a challenge, because new types of devices and use cases are continually introduced, which can create unanticipated interoperability issues. For this reason, the ability to do OTA firmware updates is essential for any IoT device.

In Bluetooth SDK version 2.3.0.0, Silicon Labs introduced a new Gecko Bootloader, which is required for the new EFR32xG12 platform and future parts. It adds many improvements and features, including a configurable base framework, security options, support for multiple update images, and in-field updates for the bootloader itself. While it is required for the new EFR32xG12 parts, the Gecko Bootloader can also be used on some earlier parts. The following section details which bootloaders can be used with which parts.

Silicon Labs continues to support the legacy OTA bootloader. This document describes the details and usage of OTA with the legacy bootloaders. For more information on the new Gecko Bootloader, see *UG266: Silicon Labs Gecko Bootloader User's Guide*. For background on bootloader categories and the various Silicon Labs bootloaders, see *UG103.6: Application Development Fundamentals: Bootloading*.

1.1 Bootloaders and Supported Parts

If you are designing for the EFR32xG12 or later parts, you must use the Gecko Bootloader. For some other parts, you have the option of using the Gecko Bootloader if you want to take advantage of its new functions. The following table shows bootloader support by part category.

Table 1.1. Bootloader Part Support

	Legacy OTA Bootloader	Gecko Bootloader
EFR32xG1 256 kB Flash parts	yes	yes
EFR32xG1 128 kB Flash parts	Limited (app OTA only)	Limited
BGM11x	Yes	Yes
BGM121 / BGM123	Yes	Yes
EFR32xG12	No	Yes
Future releases SoC and Module	No	Yes

Please note that, although the Gecko Bootloader supports field-upgradable bootloader configuration, the Bluetooth legacy bootloaders are not field-upgradable. If a legacy bootloader needs to be updated, it requires a flashing the image using the SWD (Serial Wire Debug).

1.2 Reasons for Firmware Updates

The software in a Bluetooth-enabled device can be roughly divided into two parts:

- The protocol stack, link layer implementation, low-level radio drivers, and so on, that are supplied by the hardware vendor
- User application code that uses the features provided by the protocol stack

The user application may also use other software components, for example an operating system or libraries providing cryptographic functions.

A firmware update may be needed for reasons not under the control of the application designer. For example, the Bluetooth specification is under constant development, with different revisions of the protocol stack released over time. A revision may include new features, performance optimizations, and patches to bugs or interoperability issues. It may be desirable to use the OTA update procedures to change to the latest available stack revision, even if the user application itself remains unchanged.

In other cases, the changes to be applied with OTA update are limited to the user application. Examples include fixing a security hole, or making a minor configuration change that results in improved battery lifetime. In such cases, it is desirable to update only the user application, without touching the protocol stack or other lower software layers. The Silicon Labs Bluetooth SDK supports both full and minimal updates. The difference between these two is discussed in the next section.

1.3 Different Types of OTA

The Silicon Labs Bluetooth SDK supports two ways of performing OTA updates:

- **Full OTA**: Updates both the Bluetooth stack and the user application
- **Minimal OTA**: Updates only the user application

Full OTA updates everything except the bootloader (as described in more detail in chapter 4. Internal OTA Processes). It provides almost the same level of flexibility as when programming the device using a cable during the development phase.

Minimal OTA updates only the user application, leaving the Bluetooth stack and the other low-level firmware layers intact. The benefit of minimal OTA is the smaller size of the firmware update image, allowing a faster update procedure with lower energy consumption.

To further explain the difference between full and minimal OTA, consider the following (theoretical) example:

- The total size of the firmware image is 130 kB (stack and application)
- The stack size is 115 kB; the application size is15 kB
- The average transfer speed when uploading the firmware image is 25 kbit/s

With these parameters, transferring the full firmware image takes (130*8)/25 = 42 seconds to complete, while the minimal OTA image can be transferred in just (15*8)/25 = 5 seconds. The shorter duration also reduces energy consumption.

Because the SDK OTA solution supports both forms of update, either can be used depending on the nature of the firmware update.

1.4 OTA Firmware Update Requirements

The following table summarizes some high level requirements that are typically required for OTA firmware update.

Table 1.2. OTA High Level Requirements

Category	Requirements
OTA client	• Can perform OTA update using normal BLE-enabled smartphone/tablet/PC (no special hardware needed)
Robustness	• Target device can verify that uploaded firmware is valid • Procedure can recover from interrupted OTA due, for example, to sudden power loss or a dropped connection
Security	• Use encrypted update images • Prevent OTA update from unauthorized clients
Non-functional requirements	• Ease of use (from end user viewpoint) • Small overhead in terms of hardware cost / software development effort

1.5 Document Organization and Scope

Chapter 2. Application Requirements for OTA Update explains the general requirements to enable OTA updates in the user application. These requirements are the same regardless whether the application is implemented using C.

Chapter 3. Enabling OTA Updates for C-Based Applications explains in more detail the configurations in application code that are needed for projects based on C.

Chapter 4. Internal OTA Processes goes into more detail about how the OTA update is implemented in the Bluetooth stack.

Chapter 5. OTA DFU Host Example explains the OTA host example application that is provided in the SDK. The OTA host example can be used as a reference by customers developing their own OTA host application using their preferred host platform.

Chapter 6. Error Handling in OTA describes some typical error conditions and how to handle them.

Most of the high-level requirements in Table 1.2 OTA High Level Requirements on page 3 are covered in this document, except for security. At the time of writing, the default OTA implementation in Bluetooth SDK version 2.0.1.0 does not use any security features. Some built-in features to prevent OTA updates from unauthorized clients are discussed in chapter 2. Application Requirements for OTA Update, but generally it is up to the designer to implement the necessary security mechanisms at the application level.

2. Application Requirements for OTA Update

This chapter explains user application requirements to support OTA. You should already be familiar with the following topics:

- Basic flow of developing applications in C with Simplicity Studio and Silicon Labs Bluetooth SDK
- Bluetooth Low Energy GATT services and basic data transfer mechanisms

For an introduction to the Silicon Labs' Bluetooth Stack and development tools, please refer to *QSG108: Getting Started with Silicon Labs' Bluetooth® Software*. A documentation map at the end guides you to more in-depth documentation on different topics.

2.1 Requirements for the User Application

Most of the OTA functionality is built in to the Silicon Labs Bluetooth stack. The minimum application requirement is to provide a mechanism that allows the device to be rebooted into Device Firmware Update (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 in 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 C-based applications.

Note that there are other possible ways to trigger the DFU reboot. For example, user application code may have one or more conditions that need to be fulfilled before reboot into DFU mode is allowed. The following lists a few examples of such conditions:

- Battery voltage must be above certain threshold
- The client that is attempting to start the OTA procedure must be bonded
- Some user input (for example a button press) is required before OTA is enabled

Regardless of how the DFU reboot is triggered, after the device has been rebooted into DFU mode the rest of the OTA procedure is handled autonomously by the stack, without any intervention from the user application. How OTA is implemented in the stack is discussed in section 4. Internal OTA Processes.

2.2 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.

```
<service uuid="1d14d6ee-fd63-4fa1-bfa4-8f47b42119f0">
    <description>Silicon Labs OTA</description>
    <characteristic uuid="f7bf3564-fb6d-4e53-88a4-5e37e0326063" id="ota_control">
        <properties write="true" />
        <value length="1" type="user" />
    </characteristic>
    <characteristic uuid="984227f3-34fc-4045-a5d0-2c581f81a153" id="ota_data">
        <properties write_no_response="true" />
        <value length="20" />
    </characteristic>
</service>
```

Figure 2.1. Silicon Labs OTA Service

The OTA service exposes two characteristics, described in the following table. The UUID value of the service itself is `1d14d6ee-fd63-4fa1-bfa4-8f47b42119f0`.

Table 2.1. 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	20 bytes[2]	Mandatory	Configurable	Write without response, Write[3]
BLE stack version[4]	4f4a2368-8cca-451e-bfff-cf0e2ee23e9f	Hex	8	Optional		Read
OTA version[4]	4cc07bcf-0868-4b32-9dad-ba4cc41e5316	Hex	1	Optional		Read

Note:

1. This characteristic is excluded from the user application GATT database starting with SDK version 2.3.0.
2. The maximum length of the OTA data attribute was increased to 55 bytes beginning with SDK version 2.1.0, and up to 244 bytes with SDK version 2.3.0.
3. Support for write operations was added in SDK version 2.1.0.
4. The stack version number and OTA version number are automatically added by the stack when running in DFU mode. These are optional in the application GATT database. These features were added in SDK version 2.3.0.

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.

3. Enabling OTA Updates for C-Based Applications includes source code examples of how to process writes to the OTA control attribute in C-based applications.

The security settings for the OTA characteristics are configurable. This chapter describes in detail how the configurations are set in C-based applications. 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.

Table 2.2. 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*.

2.3 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. The Generic Attribute Service can be enabled by setting the `generic_attribute_service` parameter in the `<gatt>` XML definition, as in the following example.

```
<gatt out="gatt_db.c" header="gatt_db.h" generic_attribute_service="true">
```

The user application needs only enable the Generic Attribute Service as shown above. 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 (that same as in the OTA GATT database).

More details on the Generic Attribute Service can be found on the Bluetooth SIG website, please refer to https://www.bluetooth.com/specifications/gatt/services .

2.4 Creating OTA Update Images

One essential part of the OTA flow is creation of the update images. How the update images are generated depends on whether the application is written using C. These details are covered in chapter 3. Enabling OTA Updates for C-Based Applications.

The file format of the update images in C-based development flows is .EBL, a proprietary Silicon Labs file format used by several radio protocol stacks as well as Bluetooth. EBL files include CRC checksums and other metadata that can be used to validate the integrity of the update image.

The stack processes the EBL images autonomously, without any intervention from the user application. Therefore the user application does not need to know the internal details of the EBL file format.

EBL images are generated during the user application build process. The stack and application are separated into two different image files, so that either a full update (both stack and application) or a minimal update (application only) can be performed.

2.5 Summary of User Application Requirements to Enable OTA

As a summary, the following steps are needed to enable OTA updates:

- Implement a way to reboot the device into DFU mode
- Enable Generic Attribute Service, to avoid any issues with GATT caching
- Configure the build settings to generate two update images (stack.ebl, app.ebl)

The following chapters explain in detail how these steps are implemented in C-based applications.

3. Enabling OTA Updates for C-Based Applications

This chapter explains in detail how to enable OTA updates in C-based applications.

3.1 Handling a Write Request to the OTA Control Characteristic

The minimum functional requirement to enable OTA in a C-based application is to implement a 'hook' that allows the device to be re-booted into DFU mode. By default, this is done through the Silicon Labs OTA service that was first described in section 2.2 Silicon Labs OTA GATT service.

The following figure includes a code snippet is from the SoC Thermometer example supplied with the SDK. The code to enter DFU mode is similar in the other C-based examples.

```
/* Events related to OTA upgrading
----------------------------------------------------------------------------- */

/* 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)
   {
   gecko_cmd_system_reset(1);
   }
  break;
```

Figure 3.1. Handling Write to OTA Control Characteristic in C Code

The code shown above is found in file main.c, at the end of the case-statement that handles any events raised by the Bluetooth stack.

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, reboots the device into DFU mode. The API command `gecko_cmd_system_reset` is used to trigger the reboot. Parameter value 1 indicates that the device is to be rebooted into DFU mode instead of normal reboot.

The GATT database definition for the SoC Thermometer example is found in the source file named gatt.xml.

Note that normally, when a **user** type characteristic is written, the application is expected to acknowledge that request by calling the API function `gatt_server_send_user_write_response`. The OTA control characteristic is an exception. In this case, the application only performs the DFU reboot by calling `gecko_cmd_system_reset(2)*`. This causes the remote BLE client to see the connection dropped, and the client then reconnects. During the second connection, the target device is in DFU mode, and the subsequent accesses to the OTA service are handled by the stack. The whole OTA sequence is described in more detail in chapter 4. Internal OTA Processes.

* The API call parameter to boot to OTA DFU was changed in SDK version 2.3.0. Check the *Bluetooth Software API Reference Manual* for the latest information in your preferred SDK version.

3.2 Other OTA-Related Configuration Requirements

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 in the following figure 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 3.2. 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 in the following figure.

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

Figure 3.3. 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 2.2 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.

`device_name_len` and `device_name_ptr` specify the Bluetooth device name that is used when the device has been rebooted into DFU mode. The DFU mode GATT database is completely independent of the user application's GATT database. The DFU mode GATT database is created by the stack, and it includes the Silicon Labs OTA service. Therefore, the user application does not necessarily have to expose the OTA service in the application GATT database.

Normally the device name is defined in the application GATT database. To allow identification of the device in DFU mode, the application specifies the device name string to be used during the OTA procedure. The name can be set to the same value as in the application GATT database, or it can be set to any arbitrary value. In the example shown in Figure 3.3 OTA Configuration Struct on page 10, the device name during OTA has been configured simply as `OTA`.

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()`). It cannot be changed afterwards.

Finally, 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 ATT content is used internally by the stack during OTA update. The application need only include the att.h header in the main.c source file.

3.3 Creating OTA Update Images with a C Application

Starting with SDK version 2.1.0, all C projects include a script for generating the update images in EBL format. The script is placed in the project root folder and there are two versions: create_ebl_files.bat (for Windows) and create_ebl_files.sh (for Mac or Linux). Note that the script is not run automatically when you build the project. You must run the script manually to create the EBL files. The files are placed in a subdirectory named output_ebl.

SDK versions 2.0.0 and 2.0.1 do not include any automation for EBL file creation. The script needs to be manually added by user. An example of a minimal Windows batch file for generating the stack.ebl and app.ebl files is shown in the following figure. For more details, refer to this knowledgebase article: http://community.silabs.com/t5/Bluetooth-Wi-Fi-Knowledge-Base/Creating-EBL-files-for-OTA-update-in-Bluetooth-SDK-2-0/ta-p/182756.

```
:: modify the filename to match your *.out file...
set FILENAME_OUT=soc-thermometer.out

:: full path to required utilities
set OBJCOPY=c:\SiliconLabs\SimplicityStudio\v4\developer\toolchains\gnu_arm\4.9_2015q3\bin\arm-none-eabi-objcopy.exe
set COMMANDER=c:\SiliconLabs\SimplicityStudio\v4\developer\adapter_packs\commander\commander.exe

:: create the EBL files
%OBJCOPY% -O srec -j .text_stack* %FILENAME_OUT% stack.srec
%COMMANDER% convert stack.srec -o stack.ebl -d EFR32F256
%OBJCOPY% -O srec -j .text_app* %FILENAME_OUT% app.srec
%COMMANDER% convert app.srec -o app.ebl -d EFR32F256

pause
```

Figure 3.4. Example Script for Creating OTA Images

4. Internal OTA Processes

The previous chapters discussed the configuration settings needed in application code to enable OTA updates. The key points are:

- The application provides a way to reboot into DFU mode.
- As part of the build process two update images (EBL files) are generated.
- After rebooting into DFU mode, the rest of the OTA procedure is handled by the stack.

This chapter describes how the OTA update works internally, in other words what happens after the device has been booted into DFU mode. This information is provided for developers who may want a more detailed understanding of how the process works. No actions are required from the user application during these procedures.

4.1 Introduction to In-Place OTA

In many OTA solutions the update requires a lot of free flash memory to be used as a temporary working area during the update. A simplified OTA update sequence could be summarized as follows:

- Upload a new complete firmware image into temporary storage.
- Reboot the target device into bootloader mode.
- Bootloader copies the new firmware image on top of the old image.
- Bootloader resets the device and the execution continues from the new application.

While this kind of OTA procedure is simple to implement, there is some significant overhead. Typically the amount of available flash must be doubled to make enough space for the temporary storage. The extra flash can be provided by using an SoC variant with very large flash memory or possibly even an external flash memory chip. Both of these options add extra cost to the system.

The Silicon Labs OTA implementation eliminates the need for extra flash storage by using an in-place update. In summary, the software is divided into two parts that reside in separate areas of flash memory. The protocol stack and user application have their own memory spaces and the update is done in two phases. A full OTA can be summarized in three steps as follows:

- Upload the new stack image to the application flash area.
- Copy the new stack image to the stack flash area.
- Upload the new application to the application flash area.

The flash area that is reserved for application use is used as the temporary storage during the two-phase update. This has two benefits. First, it eliminates the need for extra flash for temporary storage. Second, it supports minimal updates that only update the application without touching the stack.

4.2 Flash Memory Organization

The following figure illustrates how the flash memory is organized. It assumes a total flash size of 256 kB (top address 0x40000).

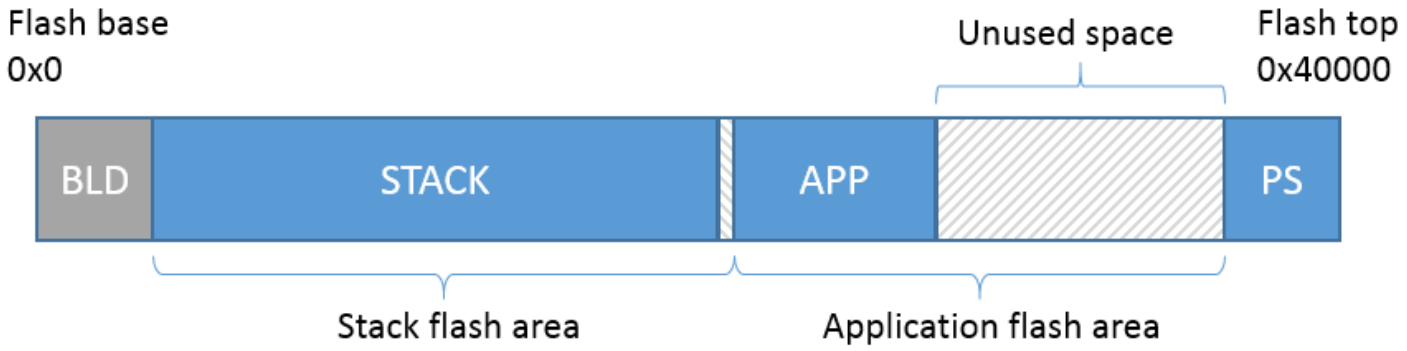


Figure 4.1. Flash Memory Organization

The first 16 kB of flash is reserved for the bootloader. This part is fixed and is not touched during OTA updates.

The available flash memory is split roughly in half between the stack (and bootloader) and the user application. The stack resides on the lower half of flash and user application is located on the upper half. At the end a fixed area is located for Persistent Storage (PS). The size of this area is 4 kB (two flash sectors, 2 kB each).

Note that the point where the stack area ends and the application area starts is not necessarily fixed across different SDK versions. For example, in SDK version 2.0.0 the application area starts at offset 0x0001F800 (126 kB). The application area is aligned in flash memory so that it begins at the next free flash sector following the stack. For this reason there is a small gap between stack and application code, as shown in the figure above.

There is some unused space between the application and the Persistent Storage located at the last flash sectors. The size of this free area depends on the application. The user application can use this free space for any purpose, for example logging measurement data. However, it should be noted that part or all of this area will be overwritten during full OTA update. In other words, any non-volatile data that the application stores in this area is not guaranteed to be preserved across OTA updates.

4.3 Full OTA Sequence

The following figure illustrates how the full OTA update proceeds in terms of flash memory content.

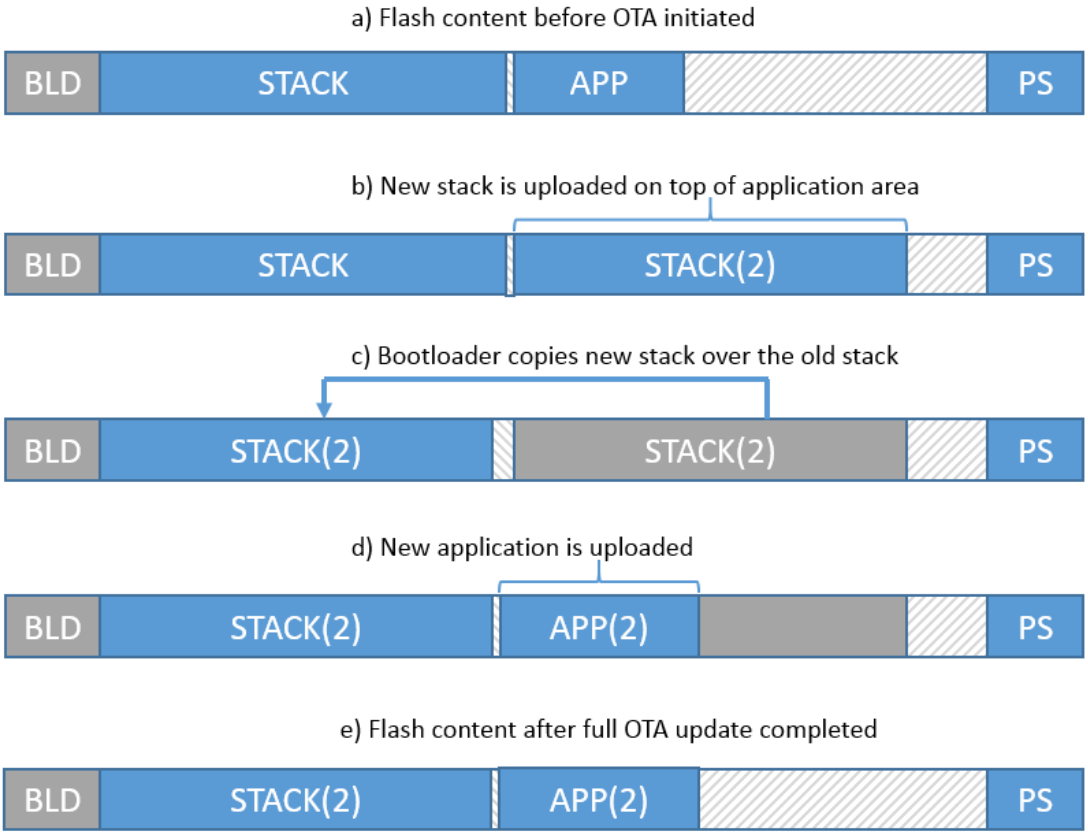


Figure 4.2. Full OTA Sequence

In the figure above, (a) shows the memory content before update. The current stack and application images are located in the lower and upper half of flash, respectively. The application provides a means for the remote BLE client to boot the device into DFU mode.

(b) shows the device is in DFU mode. The OTA functionality built into the stack allows the remote client to upload a new stack image (stack.ebl) on top of the application area.

After the new stack image is uploaded, the device is rebooted and the bootloader copies the new stack on top of the old one, as shown in (c) above. Then the device reboots into DFU mode again. At this point there is no application in the upper half of flash and the only option is to stay in DFU mode.

In (d), the remote client uploads the new application (app.ebl), which is placed in the upper half of flash. Note that the area between the application and Persistent Storage has been partially overwritten by the stack image that was temporarily stored in the upper half of flash. For this reason, any application-specific data that is stored in this area is not preserved across full OTA updates.

Finally, (e) shows the flash content after the full OTA update is completed. Both the stack and application have been updated. Bootloader and Persistent Storage have not been touched during update.

4.4 Minimal OTA Sequence

As discussed earlier, it is also possible to do a minimal OTA update that updates only the application. This can be considered a subset of the full OTA procedure described above. The starting point is the same as shown in Figure 4.2 Full OTA Sequence on page 14, frame (a). The process begins by a reboot into DFU mode. Next, the remote BLE client uploads a new application image that is stored into the upper half of flash (see Figure 4.2 Full OTA Sequence on page 14, frame (d)). The client does not need inform the target device whether it is going to perform a full or a minimal update. The OTA DFU code in the stack automatically detects the incoming file type and treats it accordingly.

After the new application is uploaded, the device is rebooted and the new application is ready to run.

Note: When updating only the application, it is important that the application is built with the same stack / SDK version that was used to create the original application.

4.5 OTA Size Limitations

Due to the flash memory organization illustrated in Figure 4.1 Flash Memory Organization on page 13 and Figure 4.2 Full OTA Sequence on page 14, there are some restrictions for the stack and application size.

The application size is limited by the Persistent Storage area that is placed at the end of flash. The application must fit into the available space without overwriting any of the area reserved for PS.

The exact size limits need to be checked for each stack version, because the flash memory map is not fixed. The size of the stack may grow or shrink, and this affects the placement of the application code area in flash. The following example calculation is valid for SDK ver sion 2.0.0.

In SDK version 2.0.0 the application code starts at offset 0x0001F800. The last 4 kB of flash are reserved for PS, meaning that the PS area starts at offset 0x0003F000. The flash area located in range 0x1F800…0x3F000 is available for user application. The maximum application size is therefore 126 kB (129024 bytes).

A theoretical upper limit for the stack size can also be calculated. Assuming a total flash size of 256 kB and taking into account the 20 kB required for bootloader and Persistent Storage, the amount of flash that is left for application and stack is 236 kB. The stack can consume at most half of this space (otherwise in-place OTA would not be possible) and therefore the maximum size for the stack can be calculated as 236 kB / 2 = 118 kB. This is also the minimum size that is available for application code.

4.6 Uploading EBL Files

During OTA update, the remote client that performs the update must first connect to the target device and reboot it into DFU mode. The different ways to boot into DFU mode in C-based applications were covered earlier in chapter 3. Enabling OTA Updates for C-Based Applications.

This section explains in more detail how the update image is uploaded into the target device.

The data to be uploaded into the target device is split into two files, stack.ebl and app.ebl. In minimal OTA, only the latter one is used. The files are uploaded by writing to the ota_data characteristic that is part of the Silicon Labs OTA service. This characteristic has a fixed length of 20 bytes, and supports only non-acknowledged writes (***write_no_response***).

The OTA client does not need to parse the EBL file content. It simply sends the content of the file to the target device by performing the necessary number of Write Command (write without response) operations in a loop. At the receiving side, the stack that is running in DFU mode decodes the data and copies it to the correct location based on the information stored in the header.

The entire content of the EBL file is sent to the OTA target device. The OTA client is expected to determine the data size by checking the file size, and not by decoding the file content.

Note: Starting with SDK version 2.1.0, the ota_data characteristic supports both write and write_no_response operations. Additionally, the maximum size of that characteristic has been increased to 55 bytes. The OTA host application example included in the SDK uses write_no_response, which results in the best throughput. However, a custom OTA host implementation can also use the acknowledged write for uploading EBL data.

4.7 Using the OTA Control Characteristic

Triggering a reboot into DFU mode has been discussed earlier. The default way to do this is to write value 0 to the OTA control characteristic that is part of the Silicon Labs OTA service.

Another use for the OTA control characteristic is to signal to the target device that the complete update image has been sent. The OTA client indicates this by writing value 3 to the OTA control characteristic.

Table 4.1. Possible Control Words Written to the OTA Control Characteristic

Hex value	Description
0x00	OTA client initiates the update procedure by writing value 0
0x03	After whole EBL file has been uploaded the client will write this value to indicate that upload is finished
Other values	Other values are reserved for future use and must not be used by application

5. OTA DFU Host Example

This chapter discusses the OTA host reference implementation provided as part of the Bluetooth SDK. 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 updated over the air is shown on the right hand side. It is identified by its Bluetooth address.

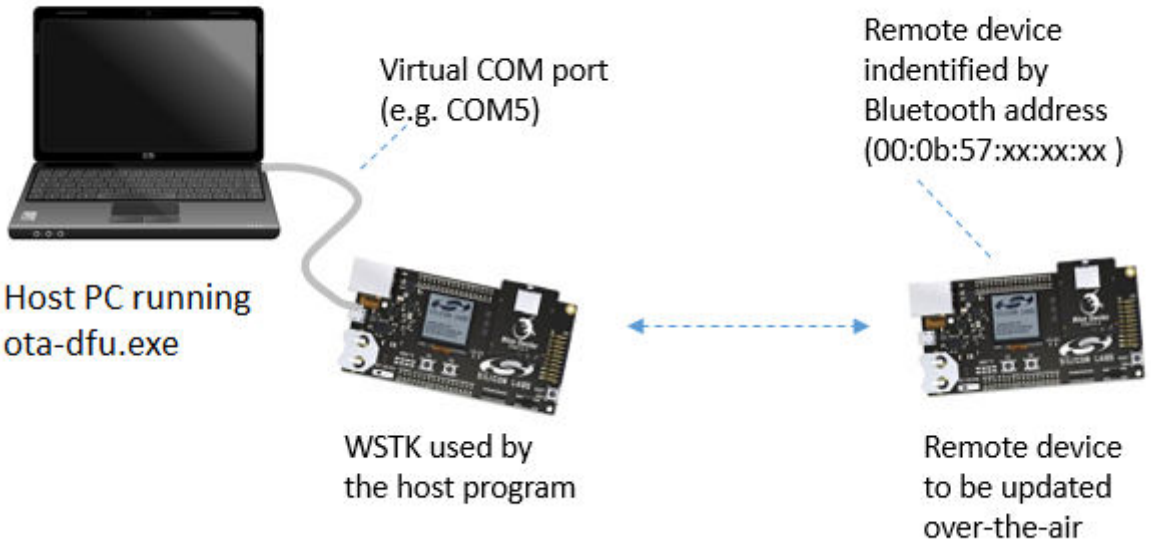


Figure 5.1. Full OTA Sequence

5.1 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.

5.2 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\stacks\ble\v2.0.1.0\app\bluetooth_2.0\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".

5.3 Running OTA with the NCP Host Example

The OTA host program expects four command-line arguments:

- COM port number associated with the development kit used in NCP mode
- Baud rate (use fixed value 115200)
- Name of the EBL file to be uploaded into target device
- Bluetooth address of the target device

Note that when using the on-board USB-to-UART converter on the development kit, the baud rate must be fixed at 115200. This does not have a significant impact on the OTA execution speed, because the OTA transfer speed is limited by the Bluetooth connection. A typical throughput that can be expected with this host example is about 25 kbit/s, which enables the full OTA procedure to be completed in about 40-60 seconds, depending on the application size.

A full OTA update is done in two parts, and it requires two separate EBL 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.ebl 00:0B:57:0B:49:23
./ota-dfu.exe COM49 115200 app.ebl 00:0B:57:0B:49:23
```

Example output is provided in section 7. Appendix: Example Output of a Full OTA Update.

The following section describes in more detail how the OTA example program works internally.

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

5.4 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 update image that is used. The program simply uploads one EBL file into the target device. It is up to the user to invoke the program either once or twice, depending on the update type (minimal OTA or full OTA).

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

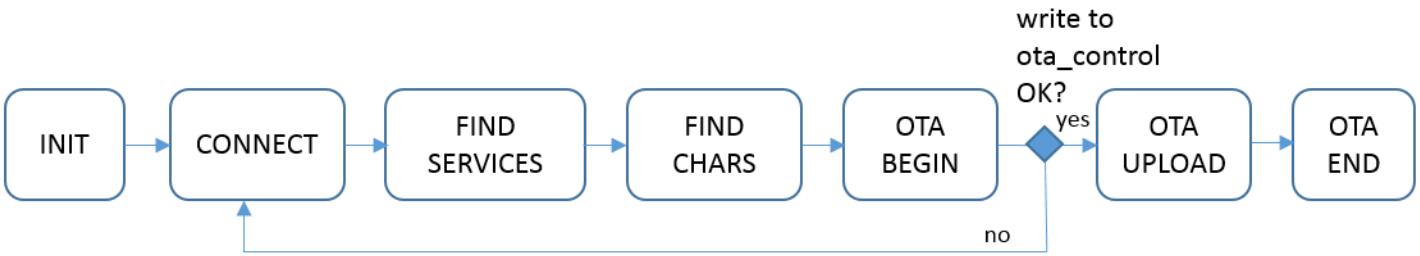


Figure 5.2. OTA Host Example State Transitions

In **INIT** state, the program checks the total size of the EBL file that is passed as a command-line parameter. The EBL 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 values for the ota_control and ota_data characteristics need to be discovered in order to proceed with the OTA procedure.

When the characteristic handles have been detected, the next state is **OTA BEGIN**. In this state the host program initiates OTA by writing value 0x00 to the ota_control characteristic. What happens next depends on the current state of the target device.

If the target device is not already in DFU mode (which is the typical case), then writing to the OTA control characteristic triggers a reboot into DFU mode. This causes the connection to be dropped and the host program returns to **CONNECT** state so that it will open a new connection as soon as the target device comes back online and starts advertising.

If the target device responds normally to the ota_control write operation, then the execution proceeds into **OTA UPLOAD** state. This is what is normally expected when the state machine arrives at the **OTA BEGIN** state for the second time.

In **OTA UPLOAD** state the whole content of the EBL 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 EBL 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 either the ota_control or ota_data characteristic is not discovered in **FIND CHARACTERISTICS** state.

Note that, 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 second 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.

6. Error Handling in OTA

This chapter discusses some potential errors that can occur during OTA update and explains the suggested error recovery procedures to make sure that the firmware update is completed. In firmware updates done over a wireless connection there is always a chance of some unexpected disconnection, for example due to external interference.

6.1 Interrupted Stack Image Upload

A typical error scenario is an unexpected disconnection during OTA update. As explained earlier in this document, a full OTA update is done in two phases using two separate EBL files. The following figure illustrates the flash memory content (a) before OTA update is initiated, (b) after a successful stack upload into the application area, used as intermediate storage, and, in contrast, (c) after the connection is dropped during stack image upload into the upper half of flash.

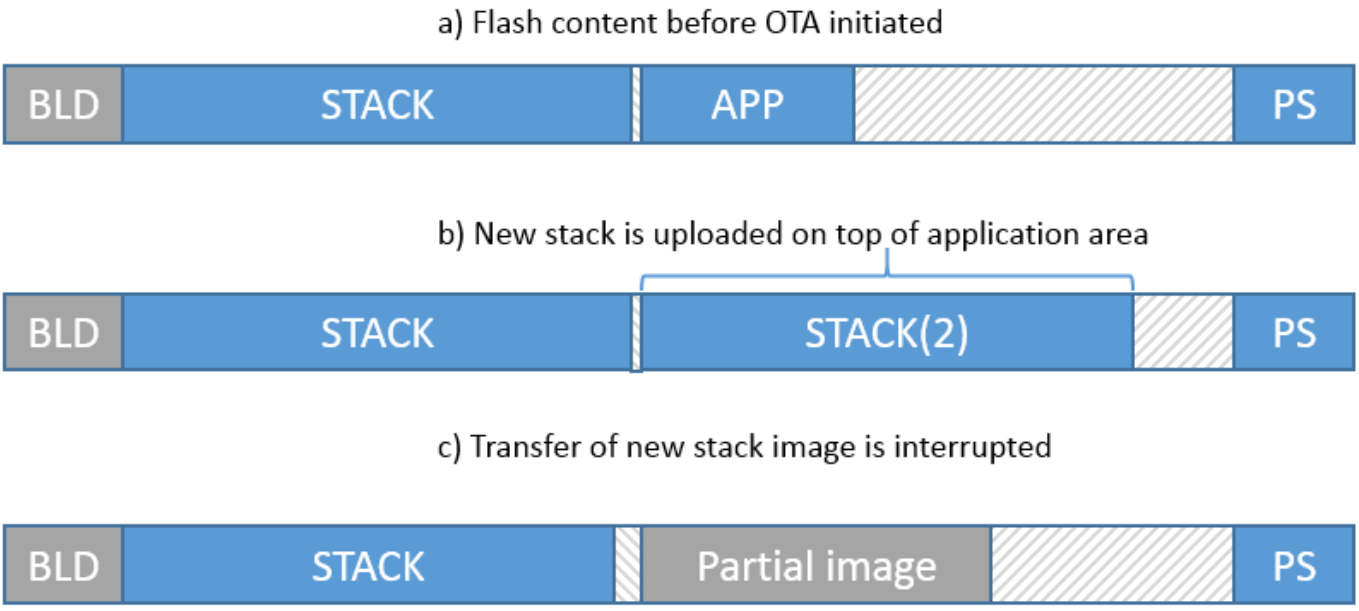


Figure 6.1. Flash Memory Content After Interrupted OTA Update

As shown above, the new stack image is only partially written to the upper half of flash. The user application has been overwritten, which means that the device is not fully operational. However, the original stack located in the lower half of flash is still intact. The existing stack area in flash is not overwritten until the new stack image has been fully received.

If the user application is missing, then the stack enters OTA DFU mode automatically so that the OTA update can be restarted. When using the OTA DFU host example provided in the Bluetooth SDK, it is enough to simply run the update procedure again to recover from an interrupted connection during stack image upload.

6.2 Interrupted Application Image Upload

It is also possible that the connection is dropped during application image upload. In that case, the target device ends up in a similar state to that illustrated in Figure 6.1 Flash Memory Content After Interrupted OTA Update on page 20; the stack area is valid but there is no valid application image in the upper half of flash. In this case, the error recovery is same as described above. The device enters DFU mode automatically because no valid user application is present, and the OTA update can be restarted.

Note that, if the application upload is interrupted in the second phase of a full update, there is no need to restart the whole procedure. If the first phase (stack update) was successfully completed, then it is enough to restart the second phase to upload the user application.

6.3 Corrupted EBL Data

The Bluetooth specification defines that each Link Layer packet must include a 24-bit CRC. Therefore the probability of corrupting the EBL data while it is being uploaded to the target device is small. Nevertheless, the OTA procedure uses an additional CRC check to detect the following errors that may result in a corrupted EBL file:

- Bit error that passes the 24-bit CRC check in the Bluetooth Link Layer
- EBL file is already corrupted before it is uploaded to the target device

The additional CRC check performed by the stack uses a 32-bit polynomial and it is calculated over the entire EBL file.

Note: The CRC check over EBL file content was added in SDK version 2.1.0.

As discussed earlier in Chapter 5. OTA DFU Host Example, the OTA host writes value 0x03 to the OTA control characteristic to indicate that the entire content of the EBL file has been transmitted. This triggers the CRC check over the entire EBL content at the receiving side. If the CRC check fails then the EBL file is rejected and the target device stays in OTA mode so that the update can be done again.

The OTA control characteristic uses a normal write transaction that is acknowledged. When the host writes value 0x03 to finish uploading the EBL file, the target device has two possible responses:

- 0x0000 (OK / no error)
- 0x0480 (application error)

In this context, a return value of 0x0480 means that the CRC check has failed.

6.4 Incompatible EBL Files

When minimal OTA is used, the application EBL image must be built using the same SDK version that was used to create the firmware currently running on the target device. For example, if the target device is running firmware created with 2.0.1.0 then it cannot be updated with an app.ebl file that was built using a different SDK version, such as 2.1.0.0.

The OTA procedure does not do any compatibility checking to ensure that the application image is compatible with the currently installed stack image. This needs to be done by the user application. One possibility is to add a custom service in the application's GATT that exposes enough details about the currently installed firmware version so that the OTA host can check for any compatibility issue before starting the OTA sequence.

7. Appendix: Example Output of a Full OTA Update

An example output of performing a full OTA update with the ota-dfu.exe host example is shown in the following figure.

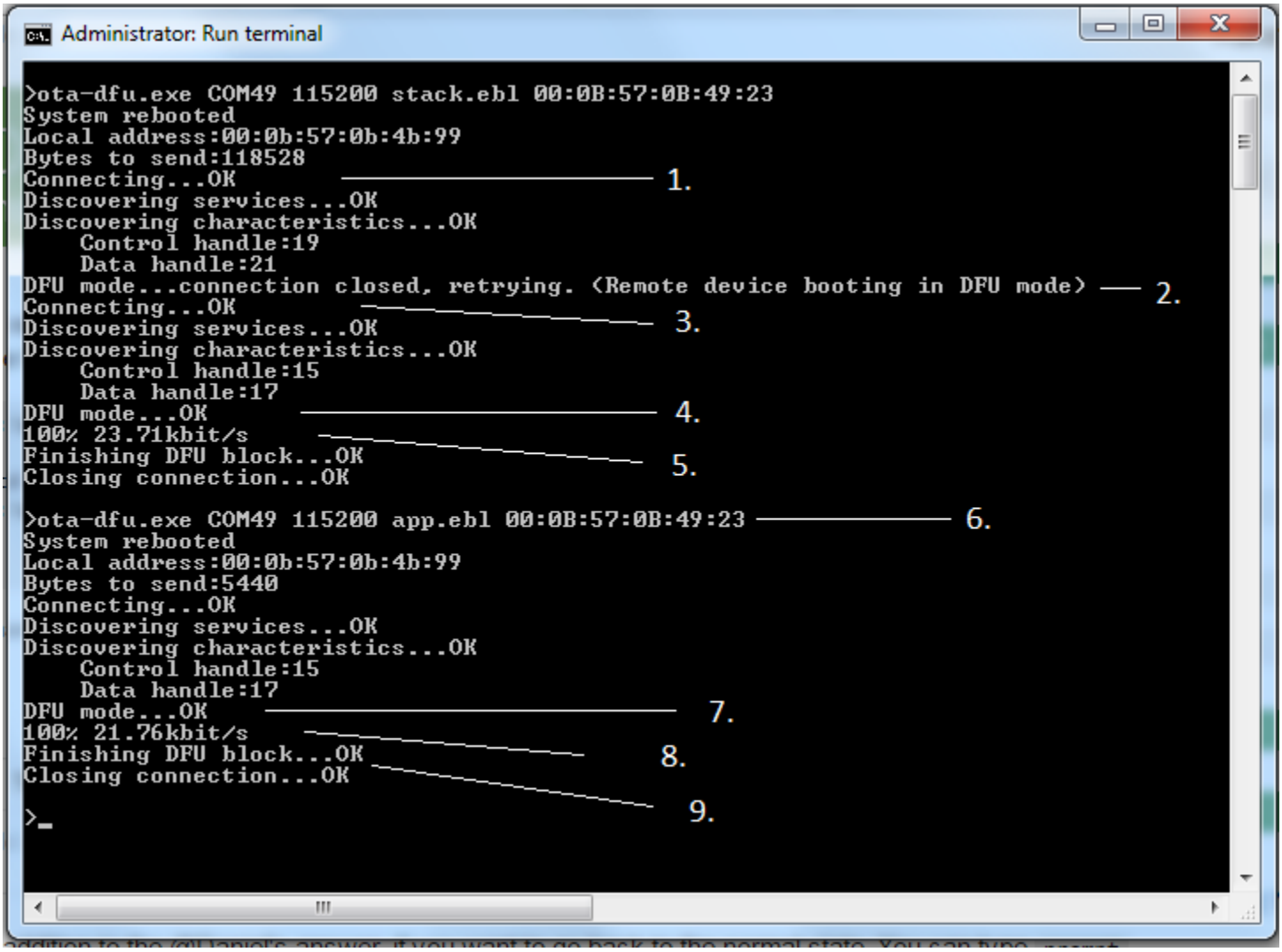
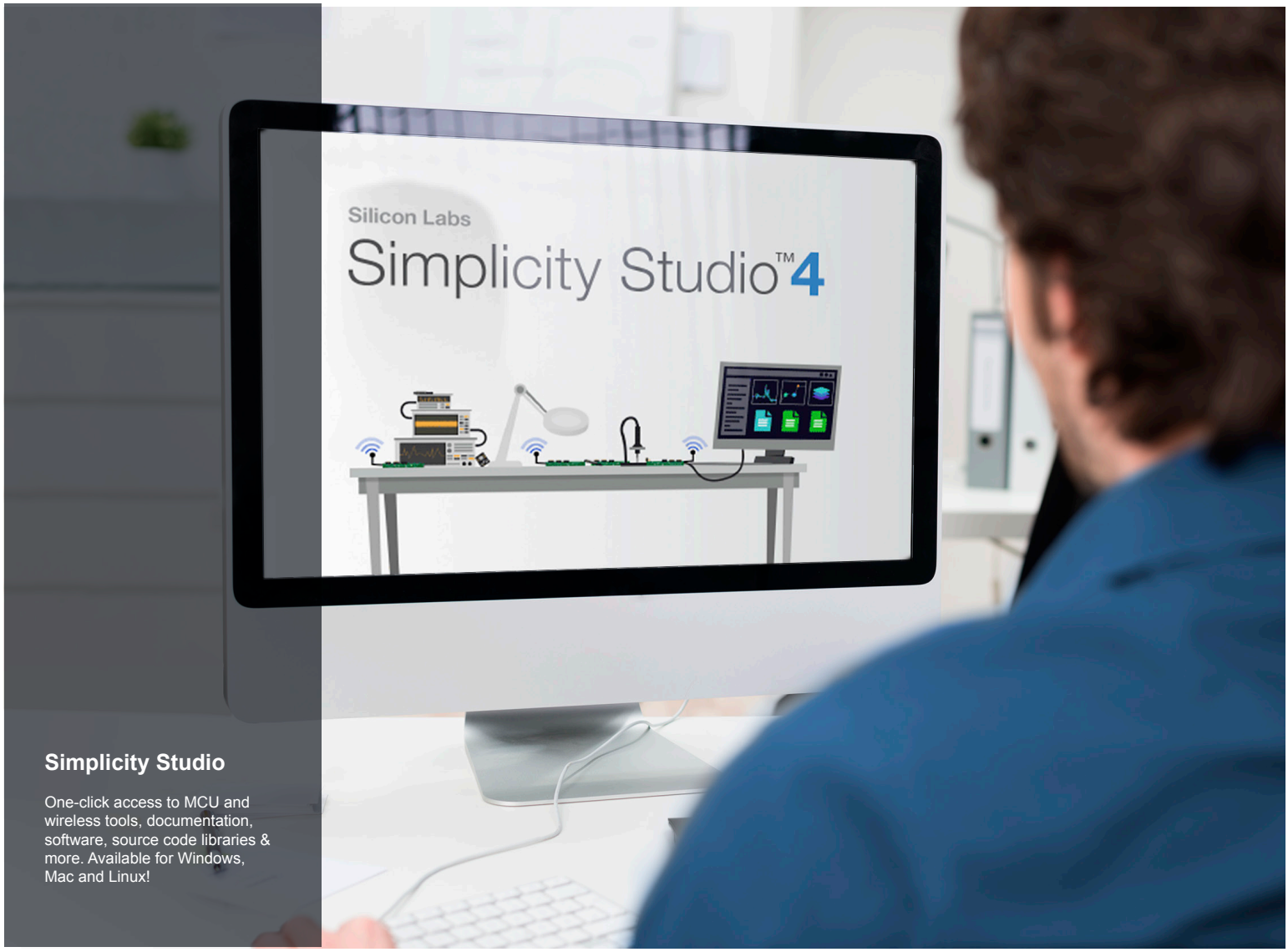


Figure 7.1. Example Output of a Full OTA Update

The full update is done in two parts. First, the stack is updated; the main steps in the procedure are:

1. Client connects to the target device for the first time (target is now in normal mode).
2. Client writes to ota_control to start the procedure. Target reboots in to DFU mode.
3. Client opens a new connection (this time target is in DFU mode).
4. Client writes to ota_control to start the procedure. Write is completed successfully.
5. New stack image is uploaded to target device (duration approximately 40 seconds).

   After updating the stack the ota-dfu.exe program needs to be run again to upload a new application. The main steps in this second part of the update are:
6. Client connects to target device (target is in DFU mode because it has no valid application).
7. Client writes to ota_control to start the procedure. Write completes successfully.
8. New application image is uploaded to target device (duration 4-40 seconds, depending on application size).
9. Update is finished, target device reboots and is running new stack and application.



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